# Control System Toolbox™
User's Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Control System Toolbox™ User's Guide

Revision History

June 2001	Online only	New for Version 5.1 (Release 12.1)
July 2002	Online only	Revised for Version 5.2 (Release 13)
June 2004	Online only	Revised for Version 6.0 (Release 14)
March 2005	Online only	Revised for Version 6.2 (Release 14SP2)
September 2005	Online only	Revised for Version 6.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 7.0 (Release 2006a)
September 2006	Online only	Revised for Version 7.1 (Release 2006b)
March 2007	Online only	Revised for Version 8.0 (Release 2007a)
September 2007	Online only	Revised for Version 8.0.1 (Release 2007b)
March 2008	Online only	Revised for Version 8.1 (Release 2008a)
October 2008	Online only	Revised for Version 8.2 (Release 2008b)
March 2009	Online only	Revised for Version 8.3 (Release 2009a)
September 2009	Online only	Revised for Version 8.4 (Release 2009b)
March 2010	Online only	Revised for Version 8.5 (Release 2010a)
September 2010	Online only	Revised for Version 9.0 (Release 2010b)
April 2011	Online only	Revised for Version 9.1 (Release 2011a)
September 2011	Online only	Revised for Version 9.2 (Release 2011b)
March 2012	Online only	Revised for Version 9.3 (Release 2012a)
September 2012	Online only	Revised for Version 9.4 (Release 2012b)
March 2013	Online only	Revised for Version 9.5 (Release 2013a)
September 2013	Online only	Revised for Version 9.6 (Release 2013b)
March 2014	Online only	Revised for Version 9.7 (Release 2014a)
October 2014	Online only	Revised for Version 9.8 (Release 2014b)
March 2015	Online only	Revised for Version 9.9 (Release 2015a)
September 2015	Online only	Revised for Version 9.10 (Release 2015b)
March 2016	Online only	Revised for Version 10.0 (Release 2016a)

Contents

Linear System Modeling

<div align="right">Model Creation</div>

2

Working with Linear Models

Data Manipulation

3

Model Interconnections

4

5

Model Simplification

6

Linear Analysis

Sensitivity Analysis

9

Passivity and Conic Sectors

10

Control Design

PID Controller Design

11

Classical Control Design

12

Control System Tuning

Loop-Shaping Design

14

15

<div align="right">

Gain-Scheduled Controllers

</div>

Customization

Customizing Response Plot Properties

19

20

<div align="right">

Design Case Studies

</div>

21

<div align="right">

Reliable Computations

</div>

22

Linear System Modeling

1

Linear System Model Objects

What Are Model Objects?

In this section...
"Model Objects Represent Linear Systems" on page 1-2
"About Model Data" on page 1-2

Model Objects Represent Linear Systems

In Control System Toolbox™, System Identification Toolbox™, and Robust Control Toolbox™ software, you represent linear systems as model objects. In System Identification Toolbox, you also represent nonlinear models as model objects. *Model objects* are specialized data containers that encapsulate model data and other attributes in a structured way. Model objects allow you to manipulate linear systems as single entities rather than keeping track of multiple data vectors, matrices, or cell arrays.

Model objects can represent single-input, single-output (SISO) systems or multiple-input, multiple-output (MIMO) systems. You can represent both continuous- and discrete-time linear systems.

The main families of model objects are:

- **Numeric Models** — Basic representation of linear systems with fixed numerical coefficients. This family also includes identified models that have coefficients estimated with System Identification Toolbox software.
- **Generalized Models** — Representations that combine numeric coefficients with tunable or uncertain coefficients. Generalized models support tasks such as parameter studies or compensator tuning.

About Model Data

The data encapsulated in your model object depends on the model type you use. For example:

- Transfer functions store the numerator and denominator coefficients
- State-space models store the A, B, C, and D matrices that describe the dynamics of the system
- PID controller models store the proportional, integral, and derivative gains

Other model attributes stored as model data include time units, names for the model inputs or outputs, and time delays. For more information about setting and retrieving model attributes, see "Model Attributes".

Note: All model objects are MATLAB® objects, but working with them does not require a background in object-oriented programming. To learn more about objects and object syntax, see "Role of Classes in MATLAB" in the MATLAB documentation.
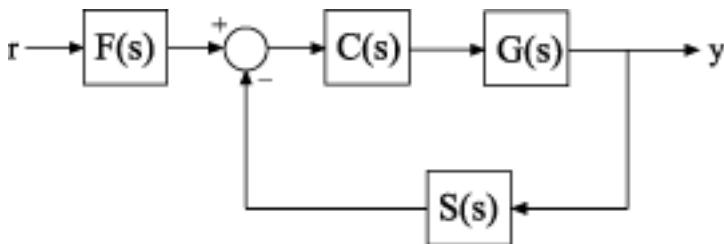
More About

·   "Control System Modeling with Model Objects" on page 1-4
·   "Types of Model Objects" on page 1-7

Control System Modeling with Model Objects

Model objects can represent individual components of a control architecture, such as the plant, actuators, sensors, or controllers. You can connect model objects to build aggregate models of block diagrams that represent the combined response of multiple elements.

For example, the following control system contains a prefilter *F*, a plant *G*, and a controller *C*, arranged in a single-loop configuration. The model also includes a representation of sensor dynamics, *S*.

You can represent each of the components as a model object. You do not need to use the same type of model object for each component. For example, represent the plant *G* as a zero-pole-gain (`zpk`) model with a double pole at `s` = -1; *C* as a PID controller, and *F* and *S* as transfer functions:

```
G = zpk([],[-1,-1],1);
C = pid(2,1.3,0.3,0.5);
S = tf(5,[1 4]);
F = tf(1,[1 1]);
```

You can then combine these elements build models that represent your control system or the control system as a whole. For example, create the open-loop response *SGC*:

```
open_loop = S*G*C;
```

To build a model of the unfiltered closed-loop response, use the `feedback` command:
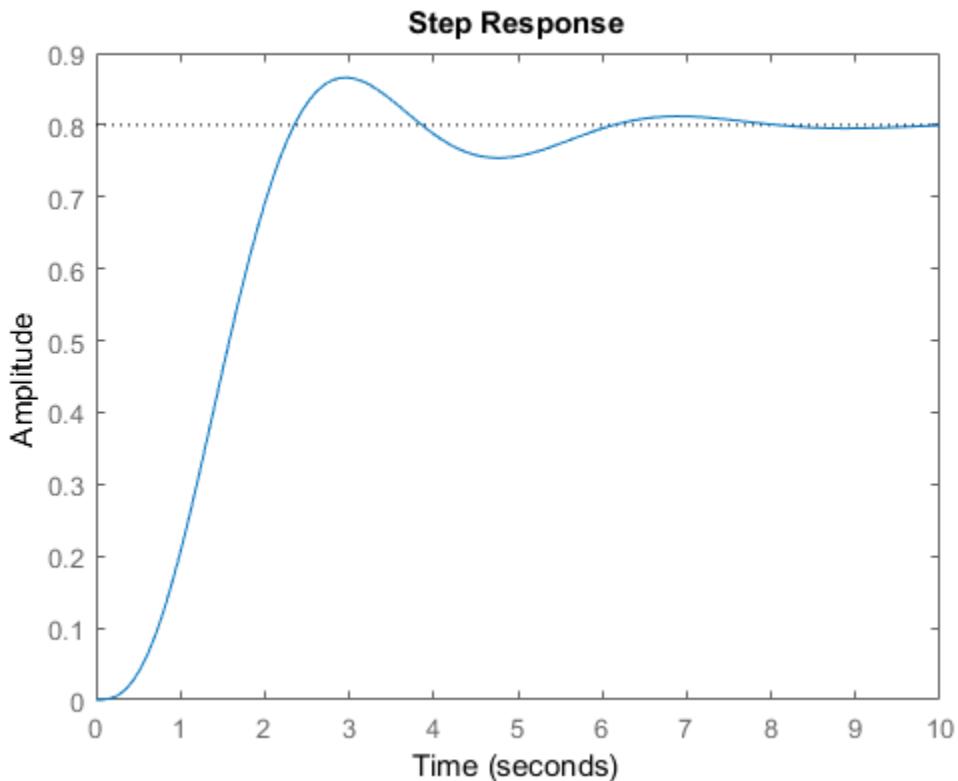
```
T = feedback(G*C,S);
```

To model the entire closed-loop system response from *r* to *y*, combine *T* with the filter transfer function:

```
Try = T*F;
```

The results `open_loop`, `T`, and `Try` are also linear model objects. You can operate on them with Control System Toolbox™ control design and analysis commands. For example, plot the step response of the entire system:

```
stepplot(Try)
```

Step Response

When you combine Numeric LTI models, the resulting Numeric LTI model represents the aggregate system. The resulting model does not retain the original data from the combined components. For example, `T` does not separately keep track of the dynamics of the components `G`, `C`, and `S` that are combined to create `T`.

See Also
`feedback`

Related Examples
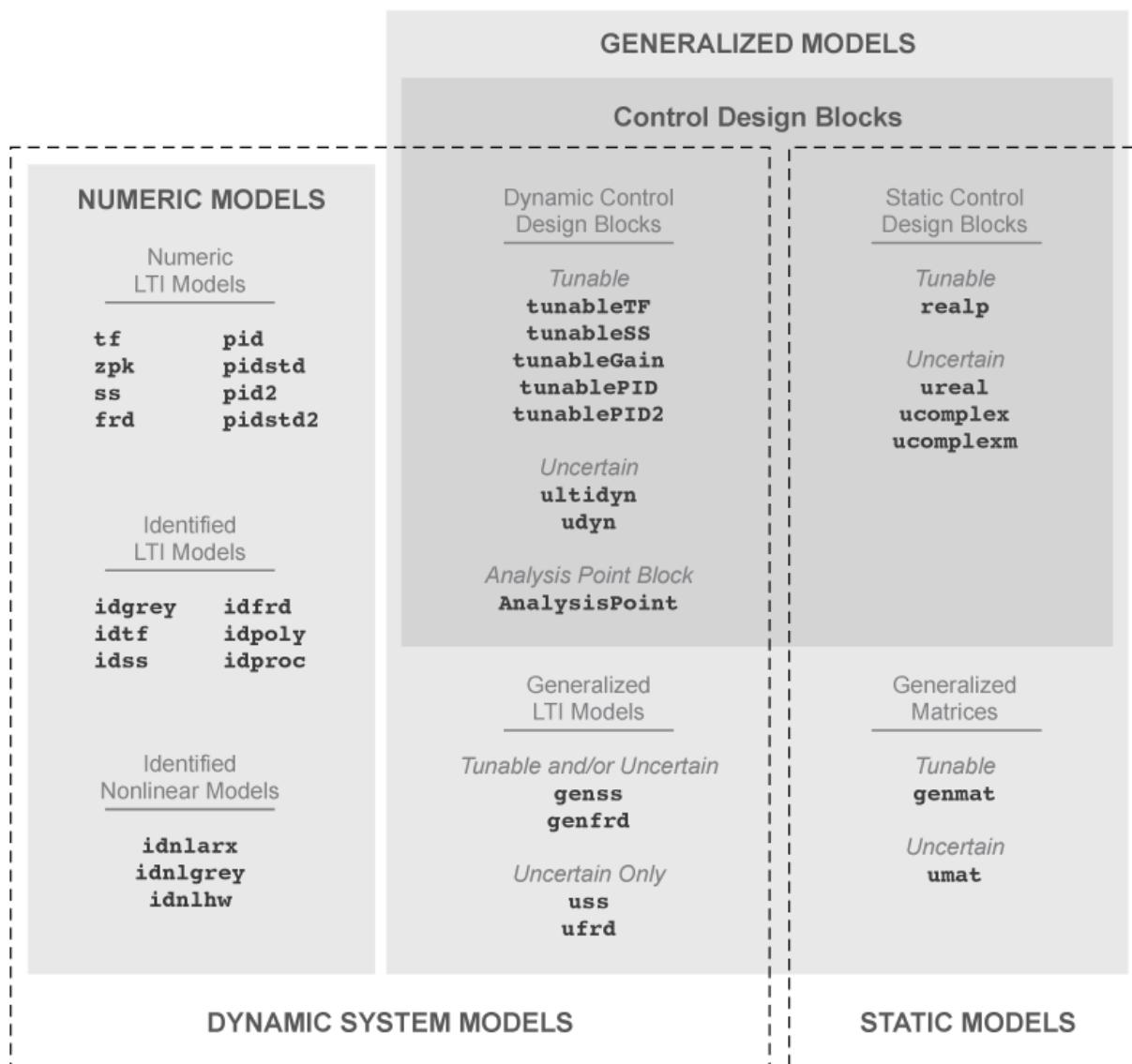
- "Numeric Model of SISO Feedback Loop" on page 4-6
- "Multi-Loop Control System" on page 4-10
- "MIMO Control System" on page 4-19

More About

- "Types of Model Objects" on page 1-7

Types of Model Objects

The following diagram illustrates the relationships between the types of model objects in Control System Toolbox, Robust Control Toolbox, and System Identification Toolbox software. Model types that begin with `id` require System Identification Toolbox software. Model types that begin with `u` require Robust Control Toolbox software. All other model types are available with Control System Toolbox software.

GENERALIZED MODELS

Control Design Blocks

NUMERIC MODELS

Numeric
LTI Models
_____

```
tf        pid
zpk       pidstd
ss        pid2
frd       pidstd2
```

Identified
LTI Models
_____

```
idgrey    idfrd
idtf      idpoly
idss      idproc
```

Identified
Nonlinear Models
_____

```
idnlarx
idnlgrey
idnlhw
```

Dynamic Control
Design Blocks
_____

Tunable
```
tunableTF
tunableSS
tunableGain
tunablePID
tunablePID2
```

Uncertain
```
ultidyn
udyn
```

Analysis Point Block
```
AnalysisPoint
```

Generalized
LTI Models
_____

Tunable and/or Uncertain
```
genss
genfrd
```

Uncertain Only
```
uss
ufrd
```

Static Control
Design Blocks
_____

Tunable
```
realp
```

Uncertain
```
ureal
ucomplex
ucomplexm
```

Generalized
Matrices
_____

Tunable
```
genmat
```

Uncertain
```
umat
```

DYNAMIC SYSTEM MODELS

STATIC MODELS

The diagram illustrates the following two overlapping broad classifications of model object types:

- **Dynamic System Models vs. Static Models** — In general, Dynamic System Models represent systems that have internal dynamics, while Static Models represent static input/output relationships.
- **Numeric Models vs. Generalized Models** — Numeric Models are the basic numeric representation of linear systems with fixed coefficients. Generalized Models represent systems with tunable or uncertain components.

More About

Dynamic System Models

Dynamic System Models generally represent systems that have internal dynamics or memory of past states such as integrators, delays, transfer functions, and state-space models.

Most commands for analyzing linear systems, such as `bode`, `margin`, and `linearSystemAnalyzer`, work on most Dynamic System Model objects. For Generalized Models, analysis commands use the current value of tunable parameters and the nominal value of uncertain parameters. Commands that generate response plots display random samples of uncertain models.

The following table lists the Dynamic System Models.

Model Family	Model Types
Numeric LTI models — Basic numeric representation of linear systems	tf
	zpk
	ss
	frd
	pid
	pidstd
	pid2
	pidstd2
Identified LTI models — Representations of linear systems with tunable coefficients, whose values can be identified using measured input/output data.	idtf
	idss
	idfrd
	idgrey
	idpoly
	idproc
Identified nonlinear models — Representations of nonlinear systems with tunable coefficients, whose values can be identified using input/output data. Limited	idnlarx
	idnlhw
	idnlgrey

Model Family	Model Types
support for commands that analyze linear systems.	
Generalized LTI models — Representations of systems that include tunable or uncertain coefficients	`genss`
	`genfrd`
	`uss`
	`ufrd`
Dynamic Control Design Blocks — Tunable, uncertain, or switch components for constructing models of control systems	`tunableGain`
	`tunableTF`
	`tunableSS`
	`tunablePID`
	`tunablePID2`
	`ultidyn`
	`udyn`
	`AnalysisPoint`

More About

Static Models

Static Models represent static input/output relationships and generalize the notions of matrix and numeric array to parametric or uncertain arrays. You can use static models to create parametric or uncertain expressions, and to construct Generalized LTI models whose coefficients are parametric or uncertain expressions. The Static Models family includes:

- Tunable parameters (`realp` objects)
- Generalized matrices (`genmat` objects)
- Uncertain parameters and matrices (`ureal`, `ucomplex`, `ucomplexm`) (requires Robust Control Toolbox software)
- Uncertain matrices (`umat`) objects (requires Robust Control Toolbox software)

For more information about using these objects to create parametric models, see "Models with Tunable Coefficients" on page 1-19. For information about creating uncertain static models, see "Uncertain Real Parameters" and "Uncertain Matrices" in the Robust Control Toolbox documentation.

Numeric Models

Numeric Linear Time Invariant (LTI) Models

Numeric LTI models are the basic numeric representation of linear systems or components of linear systems. Use numeric LTI models for modeling dynamic components, such as transfer functions or state-space models, whose coefficients are fixed, numeric values. You can use numeric LTI models for linear analysis or control design tasks.

The following table summarizes the available types of numeric LTI models.

Model Type	Description
tf	Transfer function model in polynomial form
zpk	Transfer function model in zero-pole-gain (factorized) form
ss	State-space model
frd	Frequency response data model
pid	Parallel-form PID controller
pidstd	Standard-form PID controller
pid2	Parallel-form two-degree-of-freedom (2-DOF) PID controller
pidstd2	Standard-form 2-DOF PID controller

Creating Numeric LTI Models

For information about creating numeric LTI models, see:

Applications of Numeric LTI Models

You can use Numeric LTI models to represent block diagram components such as plant or sensor dynamics. By connecting Numeric LTI models together, you can derive Numeric

LTI models of block diagrams. Use Numeric LTI models for most modeling, analysis, and control design tasks, including:

- Analyzing linear system dynamics using analysis commands such as `bode`, `step`, or `impulse`.
- Designing controllers for linear systems using the Control System Designer app or the PID Tuner GUI.
- Designing controllers using control design commands such as `pidtune`, `rlocus`, or `lqr/lqg`.

Identified LTI Models

Identified LTI Models represent linear systems with coefficients that are identified using measured input/output data (requires System Identification Toolbox software). You can specify initial values and constraints for the estimation of the coefficients.

The following table summarizes the available types of identified LTI models.

Model Type	Description
`idtf`	Transfer function model in polynomial form, with identifiable parameters
`idss`	State-space model, with identifiable parameters
`idpoly`	Polynomial input-output model, with identifiable parameters
`idproc`	Continuous-time process model, with identifiable parameters
`idfrd`	Frequency-response model, with identifiable parameters
`idgrey`	Linear ODE (grey-box) model, with identifiable parameters

Identified Nonlinear Models

Identified Nonlinear Models represent nonlinear systems with coefficients that are identified using measured input/output data (requires System Identification Toolbox software). You can specify initial values and constraints for the estimation of the coefficients.

The following table summarizes the available types of identified nonlinear models.

Model Type	Description
`idnlarx`	Nonlinear ARX model, with identifiable parameters
`idnlgrey`	Nonlinear ODE (grey-box) model, with identifiable parameters
`idnlhw`	Hammerstein-Wiener model, with identifiable parameters

Generalized Models

Generalized and Uncertain LTI Models

Generalized LTI Models represent systems having a mixture of fixed coefficients and tunable or uncertain coefficients. Generalized LTI models arise from combining numeric LTI models with Control Design Blocks. For more information about tunable Generalized LTI models and their applications, see "Models with Tunable Coefficients" on page 1-19.

Uncertain LTI Models are a special type of Generalized LTI model that include uncertain coefficients but not tunable coefficients. For more information about using uncertain models, see "Uncertain State-Space Models (uss)" and "Create Uncertain Frequency Response Data Models" in the Robust Control Toolbox documentation.

Family	Model Type	Description
Generalized LTI Models	genss	Generalized LTI model arising from combination of Numeric LTI models (except frd models) with Control Design Blocks
	genfrd	Generalized LTI model arising from combination frd models with Control Design Blocks
Uncertain LTI Models (requires Robust Control Toolbox software)	uss	Generalized LTI model arising from combination of Numeric LTI models (except frd models) with uncertain Control Design Blocks
	ufrd	Generalized LTI model arising from combination frd models with uncertain Control Design Blocks

Control Design Blocks

Control Design Blocks are building blocks for constructing tunable or uncertain models of control systems. Combine tunable Control Design Blocks with numeric arrays or Numeric LTI models to create Generalized Matrices or Generalized LTI models that include both fixed and tunable components.

Tunable Control Design Blocks include tunable parameter objects as well as tunable linear models with predefined structure. For more information about using tunable Control Design Blocks, see "Models with Tunable Coefficients" on page 1-19.

If you have Robust Control Toolbox software, you can use uncertain Control Design Blocks to model uncertain parameters or uncertain system dynamics. For more information about using uncertain blocks, see "Uncertain LTI Dynamics Elements", "Uncertain Real Parameters", and "Uncertain Complex Parameters and Matrices" in the Robust Control Toolbox documentation.

The following tables summarize the available types of Control Design Blocks.

Dynamic System Model Control Design Blocks

Family	Model Type	Description
Tunable Linear Components	tunableGain	Tunable gain block
	tunableTF	SISO fixed-order transfer function with tunable coefficients
	tunableSS	Fixed-order state-space model with tunable coefficients
	tunablePID	One-degree-of-freedom PID controller with tunable coefficients
	tunablePID2	Two-degree-of-freedom PID controller with tunable coefficients
Uncertain Dynamics (requires Robust Control Toolbox software)	ultidyn	Uncertain linear time-invariant dynamics
	udyn	Unstructured uncertain dynamics
Analysis Point Block	AnalysisPoint	Points of interest for linear analysis or control system tuning

Static Model Control Design Blocks

Family	Model Type	Description
Tunable Parameter	`realp`	Tunable scalar parameter or matrix
Uncertain Parameters (requires Robust Control Toolbox software)	`ureal`	Uncertain real scalar
	`ucomplex`	Uncertain complex scalar
	`ucomplexm`	Uncertain complex matrix

Generalized Matrices

Generalized Matrices extend the notion of numeric matrices to matrices that include tunable or uncertain values.

Create tunable generalized matrices by building rational expressions involving `realp` parameters. You can use generalized matrices as inputs to `tf` or `ss` to create tunable linear models with structures other than the predefined structures of the Control Design Blocks. Use such models for parameter studies or some compensator tuning tasks.

If you have Robust Control Toolbox software, you can create uncertain matrices by building rational expressions involving uncertain parameters such as `ureal` or `ucomplex`.

Model Type	Description
`genmat`	Generalized matrix that includes parametric or tunable entries
`umat` (requires Robust Control Toolbox software)	Generalized matrix that includes uncertain entries

For more information about generalized matrices and their applications, see "Models with Tunable Coefficients" on page 1-19.

Models with Tunable Coefficients

In this section...
"Tunable Generalized LTI Models" on page 1-19
"Modeling Tunable Components" on page 1-19
"Modeling Control Systems with Tunable Components" on page 1-20
"Internal Structure of Generalized Models" on page 1-20

Tunable Generalized LTI Models

Tunable Generalized LTI models represent systems having both fixed and tunable (or parametric) coefficients.

You can use tunable Generalized LTI models to:

- Model a tunable (or parametric) component of a control system, such as a tunable low-pass filter.
- Model a control system that contains both:

    - Fixed components, such as plant dynamics and sensor dynamics
    - Tunable components, such as filters and compensators

You can use tunable Generalized LTI models for parameter studies. For an example, see "Study Parameter Variation by Sampling Tunable Model" on page 2-111. You can also use tunable Generalized LTI models for tuning fixed control structures using tuning commands such as `systune` or the Control System Tuner app. See "Multiloop, Multiobjective Tuning".

Modeling Tunable Components

Control System Toolbox includes tunable components with predefined structure called "Control Design Blocks" on page 1-16. You can use tunable Control Design Blocks to model any tunable component that fits one of the predefined structures.

To create tunable components with a specific custom structure that is not covered by the Control Design Blocks:

1. Use the tunable real parameter `realp` or the generalized matrix `genmat` to represent the tunable coefficients of your component.

2. Use the resulting `realp` or `genmat` objects as inputs to `tf` or `ss` to model the component. The result is a generalized state-space (`genss`) model of the component.

For examples of creating such custom tunable components, see:

- "Tunable Low-Pass Filter" on page 2-77
- "Tunable Second-Order Filter" on page 2-78
- "State-Space Model with Both Fixed and Tunable Parameters" on page 2-80

Modeling Control Systems with Tunable Components

To construct a tunable Generalized LTI model representing a control system with both fixed and tunable components:

1. Model the nontunable components of your system using "Numeric Models" on page 1-13.

2. Model each tunable component using Control Design Blocks or expressions involving such blocks. See "Modeling Tunable Components" on page 1-19.

3. Use model interconnection commands such as `series`, `parallel` or `connect`, or the arithmetic operators `+`, `-`, `*`, `/`, `\`, and `^`, to combine all the components of your system.
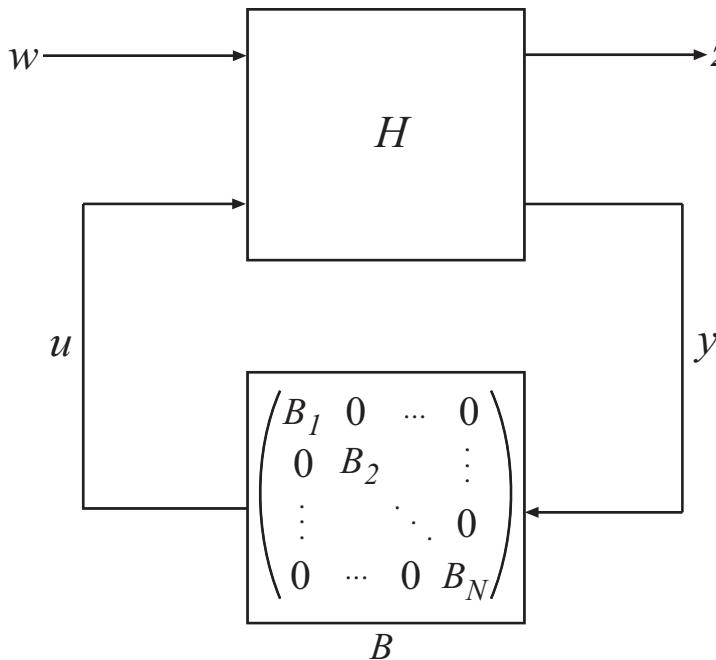
The resulting model is:

- A `genss` model, if none of the nontunable components is a frequency response data model (for example, `frd`)
- A `genfrd` model, if the nontunable component is a `frd` model

For an example of constructing a `genss` model of a control system with both fixed and tunable components, see "Control System with Tunable Components" on page 2-82.

Internal Structure of Generalized Models

A Generalized model separately stores the numeric and parametric portions of the model by structuring the model in *Standard Form*, as shown in the following illustration.

w and z represent the inputs and outputs of the Generalized model.
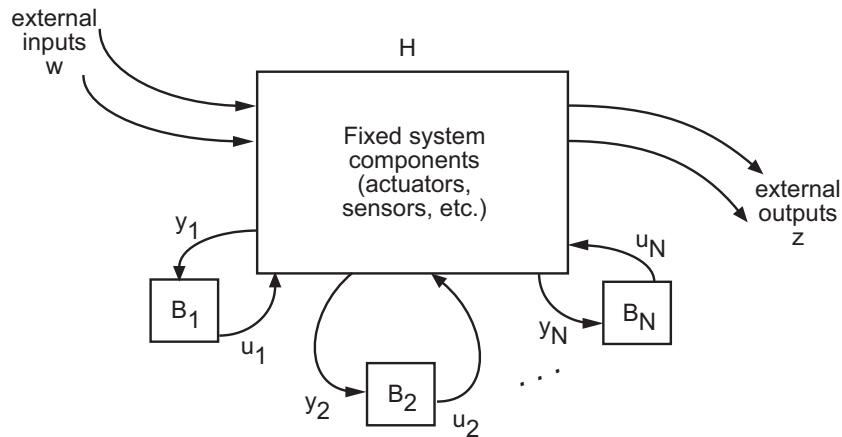
H represents all portions of the Generalized model that have fixed (non-parametric) coefficients. H is:

- A state-space (`ss`) model, for `genss` models
- A frequency response data (`frd`) model, for `genfrd` models
- A matrix, for `genmat` models

B represents the parametric components of the Generalized model, which are the Control Design Blocks $B_1, \ldots, B_N$. The `Blocks` property of the Generalized model stores a list of the names of these blocks. If the Generalized model has blocks that occur multiple times in $B_1, \ldots, B_N$, these are only listed once in the `Blocks` property.

To access the internal representation of a Generalized model, including `H` and `B`, use the `getLFTModel` command.

This Standard Form can represent any control structure. To understand why, consider the control structure as an aggregation of fixed-coefficient elements interacting with the parametric elements:

To rewrite this in Standard Form, define

$$u := [u_1, \ldots, u_N]$$
$$y := [y_1, \ldots, y_N],$$

and group the tunable control elements $B_1, \ldots, B_N$ into the block-diagonal configuration C. P includes all the fixed components of the control architecture—actuators, sensors, and other nontunable elements—and their interconnections.
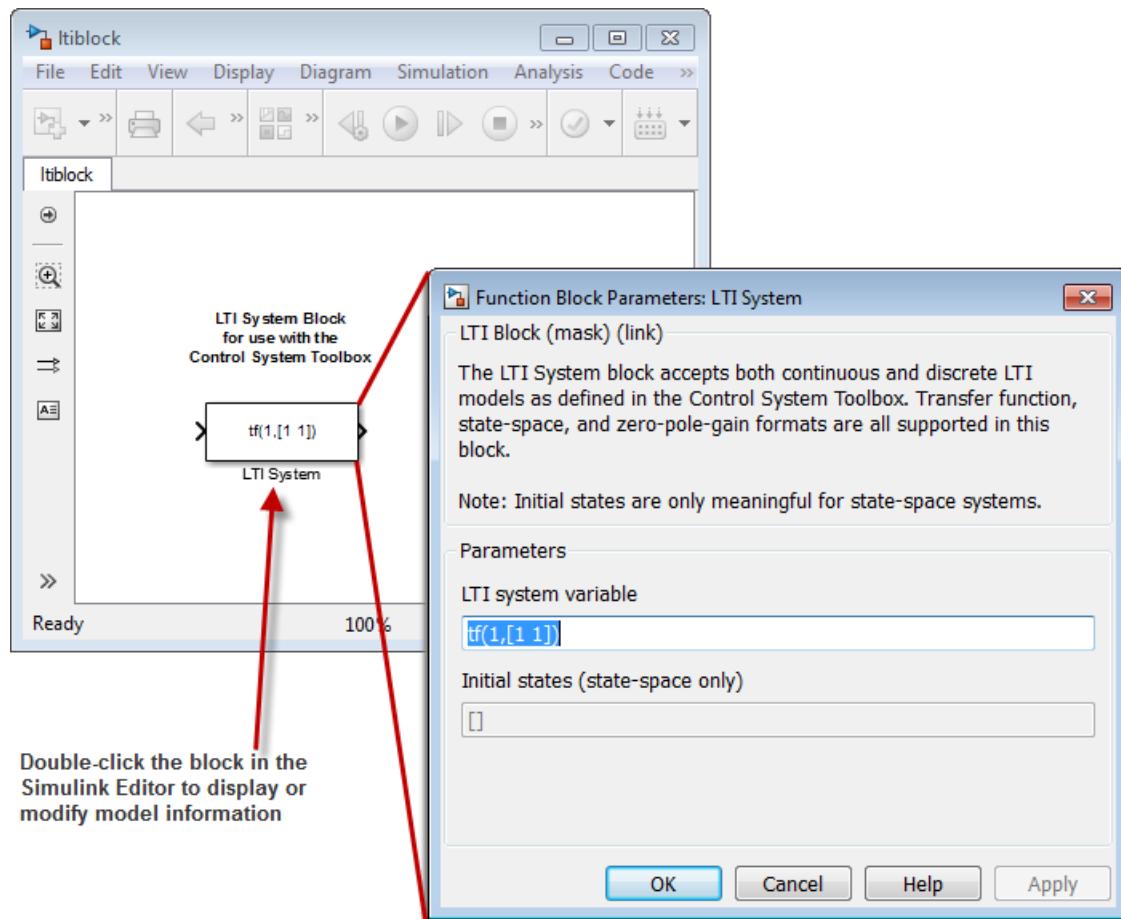
Using Model Objects

After you represent your dynamic system as a model object, you can:

- Attach additional information to the model using model attributes (properties). See "Model Attributes".
- Manipulate the model using arithmetic and model interconnection operations. See "Model Interconnection".
- Analyze the model response using commands such as `bode` and `step`. See "Linear Analysis".
- Perform parameter studies using model arrays. See "Model Arrays".
- Design compensators. You can:

  - Design compensators for systems specified as numeric LTI models. Available compensator design techniques include PID tuning, root locus analysis, pole placement, LQG optimal control, and frequency domain loop-shaping. See "PID Controller Tuning", "Classical Control Design", or "Linear-Quadratic-Gaussian Control".
  - Manually tune many control architectures using the SISO Design Tool. See "Classical Control Design".
  - Use tuning commands such as `systune` or the Control System Tuner app to automatically tune a control system that you represent as a `genss` model with tunable blocks. See "Multiloop, Multiobjective Tuning".

Simulink Block for LTI Systems

You can incorporate model objects into Simulink diagrams using the LTI System block shown below.

Double-click the block in the Simulink Editor to display or modify model information

The LTI System block can be accessed either by typing

```
ltiblock
```

at the MATLAB prompt or from the **Control System Toolbox** section of the main Simulink library.

The LTI System block consists of the dialog box shown on the right in the figure above. In the editable text box labeled **LTI system variable**, enter either the variable name of an LTI object located in the MATLAB workspace (for example, `sys`) or a MATLAB expression that evaluates to an LTI object (for example, `tf(1,[1 1])`). The LTI System block accepts both continuous and discrete LTI objects in either transfer function, zero-pole-gain, or state-space form. All types of delays are supported in the LTI block. Simulink converts the model to its state-space equivalent prior to initializing the simulation.

Use the editable text box labeled **Initial states** to enter an initial state vector for state-space models. The concept of "initial state" is not well-defined for transfer functions or zero-pole-gain models, as it depends on the choice of state coordinates used by the realization algorithm. As a result, you cannot enter nonzero initial states when you supply TF or ZPK models to LTI blocks in a Simulink diagram.

References

[1] Dorf, R.C. and R.H. Bishop, *Modern Control Systems*, Addison-Wesley, Menlo Park, CA, 1998.

2

Model Creation

Transfer Functions

In this section...
"Transfer Function Representations" on page 2-3
"Commands for Creating Transfer Functions" on page 2-4
"Create Transfer Function Using Numerator and Denominator Coefficients" on page 2-4
"Create Transfer Function Model Using Zeros, Poles, and Gain" on page 2-5

Transfer Function Representations

Control System Toolbox software supports transfer functions that are continuous-time or discrete-time, and SISO or MIMO. You can also have time delays in your transfer function representation.

A SISO continuous-time transfer function is expressed as the ratio:

$$G(s) = \frac{N(s)}{D(s)},$$

of polynomials $N(s)$ and $D(s)$, called the numerator and denominator polynomials, respectively.

You can represent linear systems as transfer functions in polynomial or factorized (zero-pole-gain) form. For example, the polynomial-form transfer function:

$$G(s) = \frac{s^2 - 3s - 4}{s^2 + 5s + 6}$$

can be rewritten in factorized form as:

$$G(s) = \frac{(s+1)(s-4)}{(s+2)(s+3)}.$$

The `tf` model object represents transfer functions in polynomial form. The `zpk` model object represents transfer functions in factorized form.

MIMO transfer functions are arrays of SISO transfer functions. For example:

$$G(s) = \begin{bmatrix} \dfrac{s-3}{s+4} \\ \dfrac{s+1}{s+2} \end{bmatrix}$$

is a one-input, two output transfer function.

Commands for Creating Transfer Functions

Use the commands described in the following table to create transfer functions.

Command	Description
tf	Create tf objects representing continuous-time or discrete-time transfer functions in polynomial form.
zpk	Create zpk objects representing continuous-time or discrete-time transfer functions in zero-pole-gain (factorized) form.
filt	Create tf objects representing discrete-time transfer functions using digital signal processing (DSP) convention.

Create Transfer Function Using Numerator and Denominator Coefficients

This example shows how to create continuous-time single-input, single-output (SISO) transfer functions from their numerator and denominator coefficients using tf.

Create the transfer function $G(s) = \dfrac{s}{s^2 + 3s + 2}$:

```
num = [1 0];
den = [1 3 2];
G = tf(num,den);
```

num and den are the numerator and denominator polynomial coefficients in descending powers of *s*. For example, den = [1 3 2] represents the denominator polynomial $s^2 + 3s + 2$.

G is a tf model object, which is a data container for representing transfer functions in polynomial form.

Tip Alternatively, you can specify the transfer function $G(s)$ as an expression in s:

1   Create a transfer function model for the variable s.

```
s = tf('s');
```

2   Specify $G(s)$ as a ratio of polynomials in s.

```
G = s/(s^2 + 3*s + 2);
```

Create Transfer Function Model Using Zeros, Poles, and Gain

This example shows how to create single-input, single-output (SISO) transfer functions in factored form using `zpk`.

Create the factored transfer function $G(s) = 5\dfrac{s}{(s+1+i)(s+1-i)(s+2)}$:

```
Z = [0];
P = [-1-1i -1+1i -2];
K = 5;
G = zpk(Z,P,K);
```

`Z` and `P` are the zeros and poles (the roots of the numerator and denominator, respectively). `K` is the gain of the factored form. For example, $G(s)$ has a real pole at $s = -2$ and a pair of complex poles at $s = -1 \pm i$. The vector `P = [-1-1i -1+1i -2]` specifies these pole locations.

`G` is a `zpk` model object, which is a data container for representing transfer functions in zero-pole-gain (factorized) form.

More About

See Also
`filt` | `tf` | `zpk`

Related Examples

More About

State-Space Models

State-Space Model Representations

State-space models rely on linear differential equations or difference equations to describe system dynamics. Control System Toolbox software supports SISO or MIMO state-space models in continuous or discrete time. State-space models can include time delays. You can represent state-space models in either explicit or descriptor (implicit) form.

State-space models can result from:

- Linearizing a set of ordinary differential equations that represent a physical model of the system.
- State-space model identification using System Identification Toolbox software.
- State-space realization of transfer functions. (See "Conversion Between Model Types" on page 5-2 for more information.)

Use `ss` model objects to represent state-space models.

Explicit State-Space Models

Explicit continuous-time state-space models have the following form:

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

where x is the state vector. u is the input vector, and y is the output vector. A, B, C, and D are the state-space matrices that express the system dynamics.

A discrete-time explicit state-space model takes the following form:

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

where the vectors $x[n]$, $u[n]$, and $y[n]$ are the state, input, and output vectors for the nth sample.

Descriptor (Implicit) State-Space Models

A *descriptor state-space model* is a generalized form of state-space model. In continuous time, a descriptor state-space model takes the following form:

$$E\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

where x is the state vector. u is the input vector, and y is the output vector. A, B, C, D, and E are the state-space matrices.

Commands for Creating State-Space Models

Use the commands described in the following table to create state-space models.

Command	Description
ss	Create explicit state-space model.
dss	Create descriptor (implicit) state-space model.
delayss	Create state-space models with specified time delays.

Create State-Space Model From Matrices

This example shows how to create a continuous-time single-input, single-output (SISO) state-space model from state-space matrices using ss.

Create a model of an electric motor where the state-space equations are:

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

where the state variables are the angular position θ and angular velocity $d\theta/dt$:

$$x = \begin{bmatrix} \theta \\ \dfrac{d\theta}{dt} \end{bmatrix},$$

u is the electric current, the output y is the angular velocity, and the state-space matrices are:

$$A = \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 3 \end{bmatrix}, \quad C = \begin{bmatrix} 0 & 1 \end{bmatrix}, \quad D = \begin{bmatrix} 0 \end{bmatrix}.$$

To create this model, enter:

```
A = [0 1;-5 -2];
B = [0;3];
C = [0 1];
D = 0;
sys = ss(A,B,C,D);
```

sys is an ss model object, which is a data container for representing state-space models.

Tip To represent a system of the form:

$$E\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

use dss. This command creates a ss model with a nonempty E matrix, also called a descriptor state-space model. See "MIMO Descriptor State-Space Models" on page 2-33 for an example.

See Also
delayss | dss | ss

Related Examples

More About

Frequency Response Data (FRD) Models
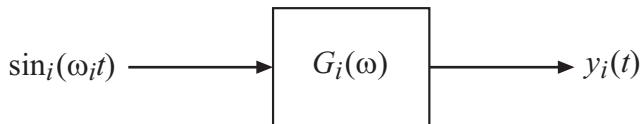
In this section...
"Frequency Response Data" on page 2-11
"Commands for Creating FRD Models" on page 2-12
"Create Frequency-Response Model from Data" on page 2-12

Frequency Response Data

In the Control System Toolbox software, you can use `frd` models to store, manipulate, and analyze frequency response data. An `frd` model stores a vector of frequency points with the corresponding complex frequency response data you obtain either through simulations or experimentally.

For example, suppose you measure frequency response data for the SISO system you want to model. You can measure such data by driving the system with a sine wave at a set of frequencies ω_1, ω_2, ,...,ω_n, as shown:

$$\sin_i(\omega_i t) \longrightarrow \boxed{\quad G_i(\omega) \quad} \longrightarrow y_i(t)$$

At steady state, the measured response $y_i(t)$ to the driving signal at each frequency ω_i takes the following form:

$$y_i(t) = a \sin(\omega_i t + b), \quad i = 1, \ldots, n.$$

The measurement yields the complex frequency response G at each input frequency:

$$G(j\omega_i) = ae^{jb}, \quad i = 1, \ldots, n.$$

You can do most frequency-domain analysis tasks on `frd` models, but you cannot perform time-domain simulations with them. For information on frequency response analysis of linear systems, see Chapter 8 of [1].

Commands for Creating FRD Models

Use the following commands to create FRD models.

Command	Description
`frd`	Create `frd` objects from frequency response data.
`frestimate`	Create `frd` objects by estimating the frequency response of a Simulink model. This approach requires Simulink Control Design™ software. See "Frequency Response Estimation" in the Simulink Control Design documentation for more information.

Create Frequency-Response Model from Data

This example shows how to create a single-input, single-output (SISO) frequency-response model using `frd`.

A frequency-response model stores a vector of frequency points with corresponding complex frequency response data you obtain either through simulations or experimentally. Thus, if you measure the frequency response of your system at a set of test frequencies, you can use the data to create a frequency response model:

1   Load the frequency response data in `AnalyzerData.mat`.

    ```
    load AnalyzerData
    ```

    This command loads the data into the MATLAB workspace as the column vectors `freq` and `resp`. The variables `freq` and `resp` contain 256 test frequencies and the corresponding complex-valued frequency response points, respectively.

    Tip To inspect these variables, enter:

    ```
    whos freq resp
    ```
    ---

2   Create a frequency response model.

    ```
    sys = frd(resp,freq);
    ```

`sys` is an `frd` model object, which is a data container for representing frequency response data.

You can use `frd` models with many frequency-domain analysis commands. For example, visualize the frequency response data using `bode`.

Tip By default, the `frd` command assumes that the frequencies are in radians/second. To specify different frequency units, use the `TimeUnit` and `FrequencyUnit` properties of the `frd` model object. For example:

```
sys = frd(resp,freq,'TimeUnit','min','FrequencyUnit','rad/TimeUnit')
```

sets the frequency units to radians/minute.

See Also
`frd` | `frestimate`

Related Examples
- "MIMO Frequency Response Data Models" on page 2-38
- "Discrete-Time Numeric Models" on page 2-24

More About
- "What Are Model Objects?" on page 1-2
- "Store and Retrieve Model Data" on page 3-2

Proportional-Integral-Derivative (PID) Controllers

You can represent PID controllers using the specialized model objects `pid` and `pidstd`. This topic describes the representation of PID controllers in MATLAB. For information about automatic PID controller tuning, see "PID Controller Tuning".

In this section...
"Continuous-Time PID Controller Representations" on page 2-14
"Create Continuous-Time Parallel-Form PID Controller" on page 2-15
"Create Continuous-Time Standard-Form PID Controller" on page 2-15

Continuous-Time PID Controller Representations

You can represent continuous-time Proportional-Integral-Derivative (PID) controllers in either parallel or standard form. The two forms differ in the parameters used to express the proportional, integral, and derivative actions and the filter on the derivative term, as shown in the following table.

Form	Formula
Parallel (`pid` object)	$$C = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1},$$ where: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time
Standard (`pidstd` object)	$$C = K_p \left(1 + \frac{1}{T_i s} + \frac{T_d s}{\frac{T_d}{N} s + 1} \right),$$ where: • K_p = proportional gain

Form	Formula
	• T_i = integrator time
	• T_d = derivative time
	• N = derivative filter divisor

Use a controller form that is convenient for your application. For example, if you want to express the integrator and derivative actions in terms of time constants, use standard form.

For information on representing PID Controllers in discrete time, see "Discrete-Time Proportional-Integral-Derivative (PID) Controllers" on page 2-25

Create Continuous-Time Parallel-Form PID Controller

This example shows how to create a continuous-time Proportional-Integral-Derivative (PID) controller in parallel form using `pid`.

Create the following parallel-form PID controller: $C = 29.5 + \dfrac{26.2}{s} - \dfrac{4.3s}{0.06s + 1}$.

```
Kp = 29.5;
Ki = 26.2;
Kd = 4.3;
Tf = 0.06;
C = pid(Kp,Ki,Kd,Tf)
```

C is a `pid` model object, which is a data container for representing parallel-form PID controllers. For more examples of how to create PID controllers, see the `pid` reference page.

Create Continuous-Time Standard-Form PID Controller

This example shows how to create a continuous-time Proportional-Integral-Derivative (PID) controller in standard form using `pidstd`.

Create the following standard-form PID controller: $C = 29.5\left(1 + \dfrac{1}{1.13s} + \dfrac{0.15s}{\dfrac{0.15}{2.3}s + 1} \right)$.

```
Kp = 29.5;
Ti = 1.13;
Td = 0.15;
N = 2.3;
C = pidstd(Kp,Ti,Td,N)
```

C is a `pidstd` model object, which is a data container for representing standard-form PID controllers. For more examples of how to create standard-form PID controllers, see the `pidstd` reference page.

See Also
pid | pidstd | pidtune | pidTuner

Related Examples
- "Transfer Functions" on page 2-3
- "Discrete-Time Proportional-Integral-Derivative (PID) Controllers" on page 2-25
- "Two-Degree-of-Freedom PID Controllers" on page 2-17

More About
- "What Are Model Objects?" on page 1-2
- "Store and Retrieve Model Data" on page 3-2

Two-Degree-of-Freedom PID Controllers

Two-degree-of-freedom (2-DOF) PID controllers include setpoint weighting on the proportional and derivative terms. A 2-DOF PID controller is capable of fast disturbance rejection without significant increase of overshoot in setpoint tracking. 2-DOF PID controllers are also useful to mitigate the influence of changes in the reference signal on the control signal.
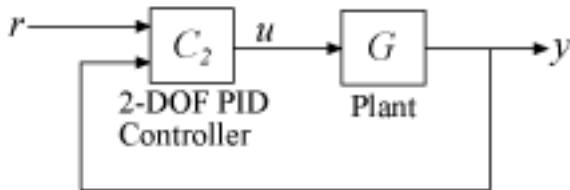
You can represent PID controllers using the specialized model objects `pid2` and `pidstd2`. This topic describes the representation of 2-DOF PID controllers in MATLAB. For information about automatic PID controller tuning, see "PID Controller Tuning".

In this section...
"Continuous-Time 2-DOF PID Controller Representations" on page 2-17
"2-DOF Control Architectures" on page 2-19

Continuous-Time 2-DOF PID Controller Representations

This illustration shows a typical control architecture using a 2-DOF PID controller.

The relationship between the 2-DOF controller's output (u) and its two inputs (r and y) can be represented in either parallel or standard form. The two forms differ in the parameters used to express the proportional, integral, and derivative actions of the controller, as expressed in the following table.

Form	Formula
Parallel (`pid2` object)	$u = K_p\left(br - y\right) + \dfrac{K_i}{s}\left(r - y\right) + \dfrac{K_d s}{T_f s + 1}\left(cr - y\right).$
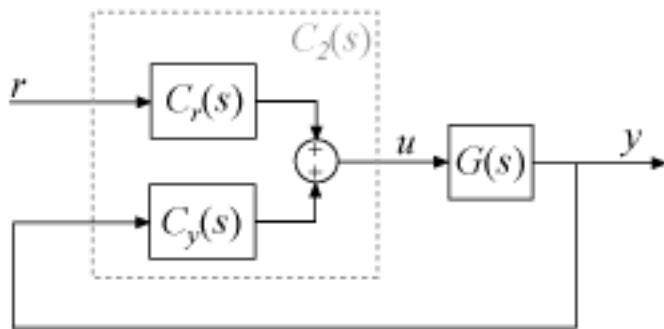
Form	Formula
	In this representation:
	• K_p = proportional gain
	• K_i = integrator gain
	• K_d = derivative gain
	• T_f = derivative filter time
	• b = setpoint weight on proportional term
	• c = setpoint weight on derivative term
Standard (`pidstd2` object)	$$u = K_p \left[(br - y) + \frac{1}{T_i s}(r - y) + \frac{T_d s}{\frac{T_d}{N} s + 1}(cr - y) \right].$$ In this representation: • K_p = proportional gain • T_i = integrator time • T_d = derivative time • N = derivative filter divisor • b = setpoint weight on proportional term • c = setpoint weight on derivative term

Use a controller form that is convenient for your application. For instance, if you want to express the integrator and derivative actions in terms of time constants, use standard form. For examples showing how to create parallel-form and standard-form controllers, see the `pid2` and `pidstd2` reference pages, respectively.

For information on representing PID Controllers in discrete time, see "Discrete-Time Proportional-Integral-Derivative (PID) Controllers" on page 2-25.

2-DOF Control Architectures

The 2-DOF PID controller is a two-input, one output controller of the form $C_2(s)$, as shown in the following figure. The transfer function from each input to the output is itself a PID controller.

Each of the components $C_r(s)$ and $C_y(s)$ is a PID controller, with different weights on the proportional and derivative terms. For example, in continuous time, these components are given by:

$$C_r(s) = bK_p + \frac{K_i}{s} + \frac{cK_d s}{T_f s + 1},$$

$$C_y(s) = -\left[K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1} \right].$$

You can access these components by converting the PID controller into a two-input, one-output transfer function. For example, suppose that `C2` is a 2-DOF PID controller, stored as a `pid2` object.

```
C2tf = tf(C2);
Cr = C2tf(1);
Cy = C2tf(2);
```

$C_r(s)$ is the transfer function from the first input of `C2` to the output. Similarly, $C_y(s)$ is the transfer function from the second input of `C2` to the output.

Suppose that G is a dynamic system model, such as a zpk model, representing the plant. Build the closed-loop transfer function from *r* to *y*. Note that the $C_y(s)$ loop has positive feedback, by the definition of $C_y(s)$.

```
T = Cr*feedback(G,Cy,+1)
```
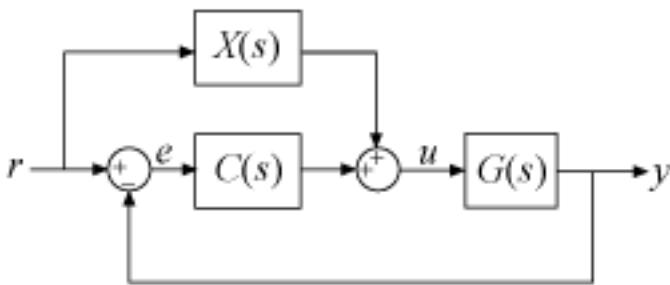
Alternatively, use the connect command to build an equivalent closed-loop system directly with the 2-DOF controller C2. To do so, set the InputName and OutputName properties of G and C2.

```
G.InputName = 'u';
G.OutputName = 'y';
C2.Inputname = {'r','y'};
C2.OutputName = 'u';
T = connect(G,C2,'r','y');
```

There are other configurations in which you can decompose a 2-DOF PID controller into SISO components. For particular choices of *C*(*s*) and *X*(*s*), each of the following configurations is equivalent to the 2-DOF architecture with $C_2(s)$. You can obtain *C*(*s*) and *X*(*s*) for each of these configurations using the getComponents command.

Feedforward

In the feedforward configuration, the 2-DOF PID controller is decomposed into a conventional SISO PID controller that takes the error signal as its input, and a feedforward controller.

For a continuous-time, parallel-form 2-DOF PID controller, the components are given by:

$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1},$$

$$X(s) = (b - 1) K_p + \frac{(c - 1) K_d s}{T_f s + 1}.$$

Access these components using `getComponents`.
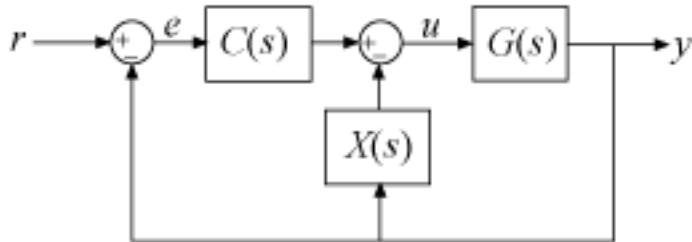
```
[C,X] = getComponents(C2,'feedforward');
```

The following command constructs the closed-loop system from *r* to *y* for the feedforward configuration.

```
T = G*(C+X)*feedback(1,G*C);
```

Feedback

In the feedback configuration, the 2-DOF PID controller is decomposed into a conventional SISO PID controller and a feedback controller.

For a continuous-time, parallel-form 2-DOF PID controller, the components are given by:

$$C(s) = bK_p + \frac{K_i}{s} + \frac{cK_d s}{T_f s + 1},$$

$$X(s) = (1 - b) K_p + \frac{(1 - c) K_d s}{T_f s + 1}.$$

Access these components using `getComponents`.

2-21

```
[C,X] = getComponents(C2,'feedback');
```

The following command constructs the closed-loop system from *r* to *y* for the feedback configuration.

```
T = G*C*feedback(1,G*(C+X));
```

Filter

In the filter configuration, the 2-DOF PID controller is decomposed into a conventional SISO PID controller and a prefilter on the reference signal.

For a continuous-time, parallel-form 2-DOF PID controller, the components are given by:

$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1},$$

$$X(s) = \frac{\left(bK_p T_f + cK_d\right)s^2 + \left(bK_p + K_i T_f\right)s + K_i}{\left(K_p T_f + K_d\right)s^2 + \left(K_p + K_i T_f\right)s + K_i}.$$

The filter *X*(s) can also be expressed as the ratio: $-[C_r(s)/C_y(s)]$.

The following command constructs the closed-loop system from *r* to *y* for the filter configuration.

```
T = X*feedback(G*C,1);
```

For an example illustrating the decomposition of a 2-DOF PID controller into these configurations, see "Decompose a 2-DOF PID Controller into SISO Components" on page 5-8.

The formulas shown above pertain to continuous-time, parallel-form controllers. Standard-form controllers and controllers in discrete time can be decomposed into

analogous configurations. The `getComponents` command works on all 2-DOF PID controller objects.

See Also
`getComponents` | `pid2` | `pidstd2` | `pidtune` | `pidTuner`

Related Examples
- "Discrete-Time Proportional-Integral-Derivative (PID) Controllers" on page 2-25
- "Proportional-Integral-Derivative (PID) Controllers" on page 2-14

More About
- "What Are Model Objects?" on page 1-2
- "Store and Retrieve Model Data" on page 3-2

Discrete-Time Numeric Models

In this section...
"Create Discrete-Time Transfer Function Model" on page 2-24
"Other Model Types in Discrete Time Representations" on page 2-24

Create Discrete-Time Transfer Function Model

This example shows how to create a discrete-time transfer function model using `tf`.

Create the transfer function $G(z) = \dfrac{z}{z^2 - 2z - 6}$ with a sample time of 0.1 s.

```
num = [1 0];
den = [1 -2 -6];
Ts = 0.1;
G = tf(num,den,Ts)
```

`num` and `den` are the numerator and denominator polynomial coefficients in descending powers of z. `G` is a `tf` model object.

The sample time is stored in the `Ts` property of `G`. Access the sample time `Ts`, using dot notation:

```
G.Ts
```

Other Model Types in Discrete Time Representations

Create discrete-time `zpk`, `ss`, and `frd` models in a similar way to discrete-time transfer functions, by appending a sample time to the input arguments. For examples, see the reference pages for those commands.

See Also
`frd` | `ss` | `tf` | `zpk`

More About
*   "What Are Model Objects?" on page 1-2

Discrete-Time Proportional-Integral-Derivative (PID) Controllers

All the PID controller object types, `pid`, `pidstd`, `pid2`, and `pidstd2`, can represent PID controllers in discrete time.

Discrete-Time PID Controller Representations

Discrete-time PID controllers are expressed by the following formulas.

Form	Formula
Parallel (`pid`)	$$C = K_p + K_i IF(z) + \frac{K_d}{T_f + DF(z)},$$ where: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time
Standard (`pidstd`)	$$C = K_p\left(1 + \frac{1}{T_i}IF(z) + \frac{T_d}{\dfrac{T_d}{N} + DF(z)}\right),$$ where: • K_p = proportional gain • T_i = integrator time • T_d = derivative time • N = derivative filter divisor
2-DOF Parallel (`pid2`)	The relationship between the 2-DOF controller's output (u) and its two inputs (r and y) is:

Form	Formula
	$$u = K_p(br - y) + K_i IF(z)(r - y) + \frac{K_d}{T_f + DF(z)}(cr - y).$$ In this representation: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time • b = setpoint weight on proportional term • c = setpoint weight on derivative term
2-DOF Standard (`pidstd2` object)	$$u = K_p\left[(br - y) + \frac{1}{T_i}IF(z)(r - y) + \frac{T_d}{\frac{T_d}{N} + DF(z)}(cr - y)\right].$$ In this representation: • K_p = proportional gain • T_i = integrator time • T_d = derivative time • N = derivative filter divisor • b = setpoint weight on proportional term • c = setpoint weight on derivative term

In all of these expressions, $IF(z)$ and $DF(z)$ are the discrete integrator formulas for the integrator and derivative filter, respectively. Use the `IFormula` and `DFormula` properties of the controller objects to set the $IF(z)$ and $DF(z)$ formulas. The next table shows available formulas for $IF(z)$ and $DF(z)$. T_s is the sample time.

IFormula or DFormula	IF(z) or DF(z)
ForwardEuler (default)	$\dfrac{T_s}{z-1}$
BackwardEuler	$\dfrac{T_s z}{z-1}$
Trapezoidal	$\dfrac{T_s}{2}\dfrac{z+1}{z-1}$

If you do not specify a value for IFormula, DFormula, or both when you create the controller object, ForwardEuler is used by default. For more information about setting and changing the discrete integrator formulas, see the reference pages for the controller objects, pid, pidstd, pid2, and pidstd2.

Create Discrete-Time Standard-Form PID Controller

This example shows how to create a standard-form discrete-time Proportional-Integral-Derivative (PID) controller that has K_p = 29.5, T_i = 1.13, T_d = 0.15 N = 2.3, and sample time T_s 0.1 :

```
C = pidstd(29.5,1.13,0.15,2.3,0.1,...
              'IFormula','Trapezoidal','DFormula','BackwardEuler')
```

This command creates a pidstd model with $IF(z) = \dfrac{T_s}{2}\dfrac{z+1}{z-1}$ and $DF(z) = \dfrac{T_s z}{z-1}$.

You can set the discrete integrator formulas for a parallel-form controller in the same way, using pid.

Discrete-Time 2-DOF PI Controller in Standard Form

Create a discrete-time 2-DOF PI controller in standard form, using the trapezoidal discretization formula. Specify the formula using Name,Value syntax.

```
Kp = 1;
Ti = 2.4;
Td = 0;
```

```
N = Inf;
b = 0.5;
c = 0;
Ts = 0.1;
C2 = pidstd2(Kp,Ti,Td,N,b,c,Ts,'IFormula','Trapezoidal')


C2 =

                        1      Ts*(z+1)
  u = Kp * [(b*r-y) + ---- * -------- * (r-y)]
                        Ti    2*(z-1)

  with Kp = 1, Ti = 2.4, b = 0.5, Ts = 0.1

Sample time: 0.1 seconds
Discrete-time 2-DOF PI controller in standard form
```

Setting Td = 0 specifies a PI controller with no derivative term. As the display shows, the values of N and c are not used in this controller. The display also shows that the trapezoidal formula is used for the integrator.

See Also
pid | pid2 | pidstd | pidstd2

Related Examples
- "Proportional-Integral-Derivative (PID) Controllers" on page 2-14
- "Two-Degree-of-Freedom PID Controllers" on page 2-17

More About
- "What Are Model Objects?" on page 1-2
- "Store and Retrieve Model Data" on page 3-2

MIMO Transfer Functions

MIMO transfer functions are two-dimensional arrays of elementary SISO transfer functions. There are two ways to specify MIMO transfer function models:

- Concatenation of SISO transfer function models
- Using `tf` with cell array arguments

Concatenation of SISO Models

Consider the following single-input, two-output transfer function.

$$H(s) = \begin{bmatrix} \dfrac{s-1}{s+1} \\ \dfrac{s+2}{s^2+4s+5} \end{bmatrix}.$$

You can specify $H(s)$ by concatenation of its SISO entries. For instance,

```
h11 = tf([1 -1],[1 1]);
h21 = tf([1 2],[1 4 5]);
```

or, equivalently,

```
s = tf('s')
h11 = (s-1)/(s+1);
h21 = (s+2)/(s^2+4*s+5);
```

can be concatenated to form $H(s)$.

```
H = [h11; h21]
```

This syntax mimics standard matrix concatenation and tends to be easier and more readable for MIMO systems with many inputs and/or outputs.

Tip Use `zpk` instead of `tf` to create MIMO transfer functions in factorized form.

Using the tf Function with Cell Arrays

Alternatively, to define MIMO transfer functions using `tf`, you need two cell arrays (say, N and D) to represent the sets of numerator and denominator polynomials, respectively. See Cell Arrays in the MATLAB documentation for more details on cell arrays.

For example, for the rational transfer matrix $H(s)$, the two cell arrays N and D should contain the row-vector representations of the polynomial entries of

$$N(s) = \left[\frac{s-1}{s+2} \right], \quad D(s) = \left[\frac{s+1}{s^2+4s+5} \right].$$

You can specify this MIMO transfer matrix $H(s)$ by typing

```
N = {[1 -1];[1 2]};   % Cell array for N(s)
D = {[1 1];[1 4 5]};  % Cell array for D(s)
H = tf(N,D)

Transfer function from input to output...
       s - 1
 #1:   -----
       s + 1

          s + 2
 #2:   -------------
       s^2 + 4 s + 5
```

Notice that both N and D have the same dimensions as H. For a general MIMO transfer matrix $H(s)$, the cell array entries `N{i,j}` and `D{i,j}` should be row-vector representations of the numerator and denominator of $H_{ij}(s)$, the ijth entry of the transfer matrix $H(s)$.

See Also
tf | zpk

Related Examples
- "Transfer Functions" on page 2-3

More About
- "What Are Model Objects?" on page 1-2
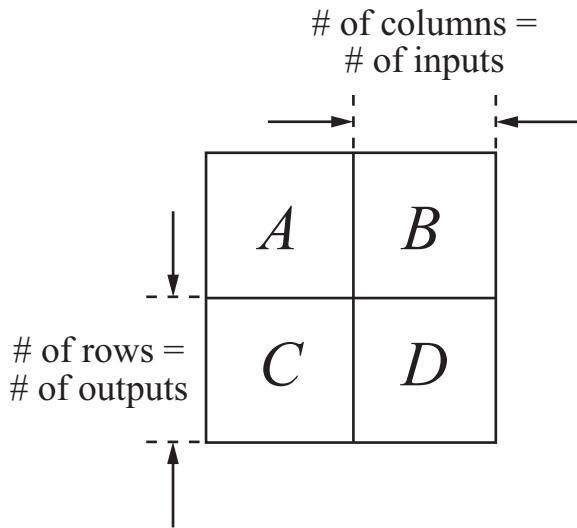
• "Store and Retrieve Model Data" on page 3-2

MIMO State-Space Models

MIMO Explicit State-Space Models

You create a MIMO state-space model in the same way as you create a SISO state-space model. The only difference between the SISO and MIMO cases is the dimensions of the state-space matrices. The dimensions of the *B*, *C*, and *D* matrices increase with the numbers of inputs and outputs as shown in the following illustration.

In this example, you create a state-space model for a rotating body with inertia tensor *J*, damping force *F*, and three axes of rotation, related as:

$$J\frac{d\omega}{dt} + F\omega = T$$

$$y = \omega.$$

The system input *T* is the driving torque. The output *y* is the vector of angular velocities of the rotating body.

To express this system in state-space form:

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

rewrite it as:

$$\frac{d\omega}{dt} = -J^{-1}F\omega + J^{-1}T$$
$$y = \omega.$$

Then the state-space matrices are:

$$A = -J^{-1}F, \quad B = J^{-1}, \quad C = I, \quad D = 0.$$

To create this model, enter the following commands:

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
```

These commands assume that J is the inertia tensor of a cube rotating about its corner, and the damping force has magnitude 0.2.

`sys_mimo` is an `ss` model.

MIMO Descriptor State-Space Models

This example shows how to create a continuous-time descriptor (implicit) state-space model using `dss`.

This example uses the same rotating-body system shown in "MIMO Explicit State-Space Models" on page 2-32, where you inverted the inertia matrix J to obtain the value of the B matrix. If J is poorly-conditioned for inversion, you can instead use a descriptor (implicit) state-space model. A *descriptor (implicit) state-space model* is of the form:

$$E\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

Create a state-space model for a rotating body with inertia tensor J, damping force F, and three axes of rotation, related as:

$$J\frac{d\omega}{dt} + F\omega = T$$
$$y = \omega.$$

The system input T is the driving torque. The output y is the vector of angular velocities of the rotating body. You can write this system as a descriptor state-space model having the following state-space matrices:

$$A = -F, \quad B = I, \quad C = I, \quad D = 0, \quad E = J.$$

To create this system, enter:

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -F;
B = eye(3);
C = eye(3);
D = 0;
E = J;
sys_mimo = dss(A,B,C,D,E)
```

These commands assume that J is the inertia tensor of a cube rotating about its corner, and the damping force has magnitude 0.2.

`sys` is an `ss` model with a nonempty E matrix.

State-Space Model of Jet Transport Aircraft

This example shows how to build a MIMO model of a jet transport. Because the development of a physical model for a jet aircraft is lengthy, only the state-space

equations are presented here. See any standard text in aviation for a more complete discussion of the physics behind aircraft flight.

The jet model during cruise flight at MACH = 0.8 and H = 40,000 ft. is

```
A = [-0.0558    -0.9968     0.0802     0.0415
      0.5980    -0.1150    -0.0318         0
     -3.0500     0.3880    -0.4650         0
          0      0.0805     1.0000        0];

B = [ 0.0073         0
     -0.4750     0.0077
      0.1530     0.1430
          0         0];

C = [0     1     0     0
     0     0     0     1];

D = [0     0
     0     0];
```

Use the following commands to specify this state-space model as an LTI object and attach names to the states, inputs, and outputs.

```
states = {'beta' 'yaw' 'roll' 'phi'};
inputs = {'rudder' 'aileron'};
outputs = {'yaw rate' 'bank angle'};

sys_mimo = ss(A,B,C,D,'statename',states,...
'inputname',inputs,...
'outputname',outputs);
```

You can display the LTI model by typing `sys_mimo`.

```
sys_mimo
```

```
a =
                    beta         yaw        roll         phi
       beta      -0.0558     -0.9968      0.0802      0.0415
        yaw        0.598      -0.115     -0.0318           0
       roll        -3.05       0.388      -0.465           0
        phi            0      0.0805           1           0
```

```
b =
                  rudder      aileron
        beta      0.0073            0
         yaw      -0.475       0.0077
        roll       0.153        0.143
         phi           0            0


c =
                    beta          yaw         roll          phi
    yaw rate           0            1            0            0
  bank angle           0            0            0            1


d =
                  rudder      aileron
    yaw rate           0            0
  bank angle           0            0
```

Continuous-time model.

The model has two inputs and two outputs. The units are radians for `beta` (sideslip angle) and `phi` (bank angle) and radians/sec for `yaw` (yaw rate) and `roll` (roll rate). The rudder and aileron deflections are in degrees.

As in the SISO case, use `tf` to derive the transfer function representation.

```
tf(sys_mimo)
```

```
Transfer function from input "rudder" to output...
               -0.475 s^3 - 0.2479 s^2 - 0.1187 s - 0.05633
 yaw rate:   -------------------------------------------------
             s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674

                      0.1148 s^2 - 0.2004 s - 1.373
 bank angle:  -------------------------------------------------
             s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674

Transfer function from input "aileron" to output...
          0.0077 s^3 - 0.0005372 s^2 + 0.008688 s + 0.004523
 yaw rate:   -------------------------------------------------
             s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674

                      0.1436 s^2 + 0.02737 s + 0.1104
```

```
bank angle:  --------------------------------------------------
             s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674
```

See Also
```
ss
```

Related Examples
·   "State-Space Models" on page 2-7

More About
·   "What Are Model Objects?" on page 1-2
·   "Store and Retrieve Model Data" on page 3-2

MIMO Frequency Response Data Models

This example shows how to create a MIMO frequency-response model using `frd`.

Frequency response data for a MIMO system includes a vector of complex response data for each of the input/output (I/O) pair of the system. Thus, if you measure the frequency response of each I/O pair your system at a set of test frequencies, you can use the data to create a frequency response model:

1 Load frequency response data in `AnalyzerDataMIMO.mat`.

```
load AnalyzerDataMIMO H11 H12 H21 H22 freq
```

This command loads the data into the MATLAB workspace as five column vectors `H11`, `H12`, `H21`, `H22`, and `freq`. The vector `freq` contains 100 test frequencies. The other four vectors contain the corresponding complex-valued frequency response of each I/O pair of a two-input, two-output system.

Tip To inspect these variables, enter:

```
whos H11 H12 H21 H22 freq
```

---

2 Organize the data into a three-dimensional array.

```
Hresp = zeros(2,2,length(freq));
Hresp(1,1,:) = H11;
Hresp(1,2,:) = H12;
Hresp(2,1,:) = H21;
Hresp(2,2,:) = H22;
```

The dimensions of `Hresp` are the number of outputs, number of inputs, and the number of frequencies for which there is response data. `Hresp(i,j,:)` contains the frequency response from input `j` to output `i`.

3 Create a frequency-response model.

```
H = frd(Hresp,freq);
```

`H` is an `frd` model object, which is a data container for representing frequency response data.

You can use `frd` models with many frequency-domain analysis commands. For example, visualize the response of this two-input, two-output system using `bode`.

Tip By default, the `frd` command assumes that the frequencies are in radians/second. To specify different frequency units, use the `TimeUnit` and `FrequencyUnit` properties of the `frd` model object. For example:

```
H = frd(Hresp,freq,'TimeUnit','min','FrequencyUnit','rad/TimeUnit')
```

sets the frequency units to in radians/minute.

See Also
`frd`

Related Examples
- "Frequency Response Data (FRD) Models" on page 2-11

More About
- "What Are Model Objects?" on page 1-2
- "Store and Retrieve Model Data" on page 3-2

Select Input/Output Pairs in MIMO Models

This example shows how to select the response from the first input to the second output of a MIMO model.

1 Create a two-input, one-output transfer function.

```
N = {[1 -1],[1];[1 2],[3 1 4]};
D = [1 1 10];
H = tf(N,D)
```

Note: For more information about using cell arrays to create MIMO transfer functions, see the `tf` reference page.

2 Select the response from the second input to the output of H.

To do this, use MATLAB array indexing.

```
H12 = H(1,2)
```

For any MIMO system H, the index notation `H(i,j)` selects the response from the *j*th input to the *i*th output.

Related Examples
- "MIMO Transfer Functions" on page 2-29
- "MIMO State-Space Models" on page 2-32

More About
- "Store and Retrieve Model Data" on page 3-2

Time Delays in Linear Systems

Use the following model properties to represent time delays in linear systems.

- `InputDelay`, `OutputDelay` — Time delays at system inputs or outputs
- `ioDelay`, `InternalDelay` — Time delays that are internal to the system

In discrete-time models, these properties are constrained to integer values that represent delays expressed as integer multiples of the sample time. To approximate discrete-time models with delays that are a fractional multiple of the sample time, use `thiran`.

The following examples illustrate the creation of models with different types of time delays.

In this section...
"First Order Plus Dead Time Model" on page 2-41
"Input and Output Delay in State-Space Model" on page 2-42
"Transport Delay in MIMO Transfer Function" on page 2-44
"Discrete-Time Transfer Function with Time Delay" on page 2-45

First Order Plus Dead Time Model

This example shows how to create a first order plus dead time model using the `InputDelay` or `OutputDelay` properties of `tf`.

To create the following first-order transfer function with a 2.1 s time delay:

$$G(s) = e^{-2.1s} \frac{1}{s+10},$$

enter:

```
G = tf(1,[1 10],'InputDelay',2.1)
```

where `InputDelay` specifies the delay at the input of the transfer function.

Tip You can use `InputDelay` with `zpk` the same way as with `tf`:

```
G = zpk([],-10,1,'InputDelay',2.1)
```

For SISO transfer functions, a delay at the input is equivalent to a delay at the output. Therefore, the following command creates the same transfer function:

```
G = tf(1,[1 10],'OutputDelay',2.1)
```

Use dot notation to examine or change the value of a time delay. For example, change the time delay to 3.2 as follows:

```
 G.OutputDelay = 3.2;
```

To see the current value, enter:

```
G.OutputDelay
```

```
ans =

    3.2000
```

Tip An alternative way to create a model with a time delay is to specify the transfer function with the delay as an expression in *s*:

1 Create a transfer function model for the variable *s*.

```
    s = tf('s');
```

2 Specify *G*(*s*) as an expression in *s*.

```
    G = exp(-2.1*s)/(s+10);
```

Input and Output Delay in State-Space Model

This example shows how to create state-space models with delays at the inputs and outputs, using the `InputDelay` or `OutputDelay` properties of `ss`.

Create a state-space model describing the following one-input, two-output system:

$$\frac{dx(t)}{dt} = -2x(t) + 3u(t-1.5)$$
$$y(t) = \begin{bmatrix} x(t-0.7) \\ -x(t) \end{bmatrix}.$$

This system has an input delay of 1.5. The first output has an output delay of 0.7, and the second output is not delayed.

Note: In contrast to SISO transfer functions, input delays are not equivalent to output delays for state-space models. Shifting a delay from input to output in a state-space model requires introducing a time shift in the model states. For example, in the model of this example, defining $T = t - 1.5$ and $X(T) = x(T + 1.5)$ results in the following equivalent system:

$$\frac{dX(T)}{dT} = -2X(T) + 3u(T)$$
$$y(T) = \begin{bmatrix} X(T-2.2) \\ -X(T-1.5) \end{bmatrix}.$$

All of the time delays are on the outputs, but the new state variable X is time-shifted relative to the original state variable x. Therefore, if your states have physical meaning, or if you have known state initial conditions, consider carefully before shifting time delays between inputs and outputs.

To create this system:

1 Define the state-space matrices.

```
A = -2;
B = 3;
C = [1;-1];
D = 0;
```

2 Create the model.

```
G = ss(A,B,C,D,'InputDelay',1.5,'OutputDelay',[0.7;0])
```

G is a `ss` model.

Tip Use `delayss` to create state-space models with more general combinations of input, output, and state delays, of the form:

$$\frac{dx}{dt} = Ax(t) + Bu(t) + \sum_{j=1}^{N} (A_j x(t - t_j) + B_j u(t - t_j))$$

$$y(t) = Cx(t) + Du(t) + \sum_{j=1}^{N} (C_j x(t - t_j) + D_j u(t - t_j))$$

Transport Delay in MIMO Transfer Function

This example shows how to create a MIMO transfer function with different transport delays for each input-output (I/O) pair.

Create the MIMO transfer function:

$$H(s) = \begin{bmatrix} e^{-0.1} \dfrac{2}{s} & e^{-0.3} \dfrac{s+1}{s+10} \\ 10 & e^{-0.2} \dfrac{s-1}{s+5} \end{bmatrix}.$$

Time delays in MIMO systems can be specific to each I/O pair, as in this example. You cannot use `InputDelay` and `OutputDelay` to model I/O-specific transport delays. Instead, use `ioDelay` to specify the transport delay across each I/O pair.

To create this MIMO transfer function:

1 Create a transfer function model for the variable `s`.

```
s = tf('s');
```

2 Use the variable `s` to specify the transfer functions of `H` without the time delays.

```
H = [2/s (s+1)/(s+10); 10 (s-1)/(s+5)];
```

3 Specify the `ioDelay` property of `H` as an array of values corresponding to the transport delay for each I/O pair.

```
H.ioDelay = [0.1 0.3; 0 0.2];
```

`H` is a two-input, two-output `tf` model. Each I/O pair in `H` has the time delay specified by the corresponding entry in `tau`.

Discrete-Time Transfer Function with Time Delay

This example shows how to create a discrete-time transfer function with a time delay.

In discrete-time models, a delay of one sampling period corresponds to a factor of z^{-1} in the transfer function. For example, the following transfer function represents a discrete-time SISO system with a delay of 25 sampling periods.

$$H\left(z\right) = z^{-25}\frac{2}{z - 0.95}.$$

To represent integer delays in discrete-time systems in MATLAB, set the `'InputDelay'` property of the model object to an integer value. For example, the following command creates a `tf` model representing $H\left(z\right)$ with a sampling time of 0.1 s.

```
H = tf(2,[1 -0.95],0.1,'InputDelay',25)
```

```
H =

              2
  z^(-25) * --------
            z - 0.95

Sample time: 0.1 seconds
Discrete-time transfer function.
```

If system has a time delay that is not an integer multiple of the sampling time, you can use the thiran command to approximate the fractional portion of the time delay with an all-pass filter. See "Time-Delay Approximation".

Related Examples
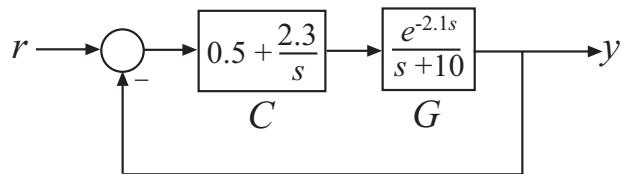
More About

Closing Feedback Loops with Time Delays

This example shows how internal delays arise when you interconnect models that have input, output, or transport time delays.

Create a model of the following control architecture:

G is the plant model, which has an input delay. C is a proportional-integral (PI) controller.

To create a model representing the closed-loop response of this system:

1   Create the plant G and the controller C.

```
G = tf(1,[1 10],'InputDelay',2.1);
C = pid(0.5,2.3);
```

C has a proportional gain of 0.5 and an integral gain of 2.3.

2   Use `feedback` to compute the closed-loop response from r to y.

```
T = feedback(C*G,1);
```

The time delay in T is not an input delay as it is in G. Because the time delay is internal to the closed-loop system, the software returns T as an `ss` model with an *internal time delay* of 2.1 seconds.
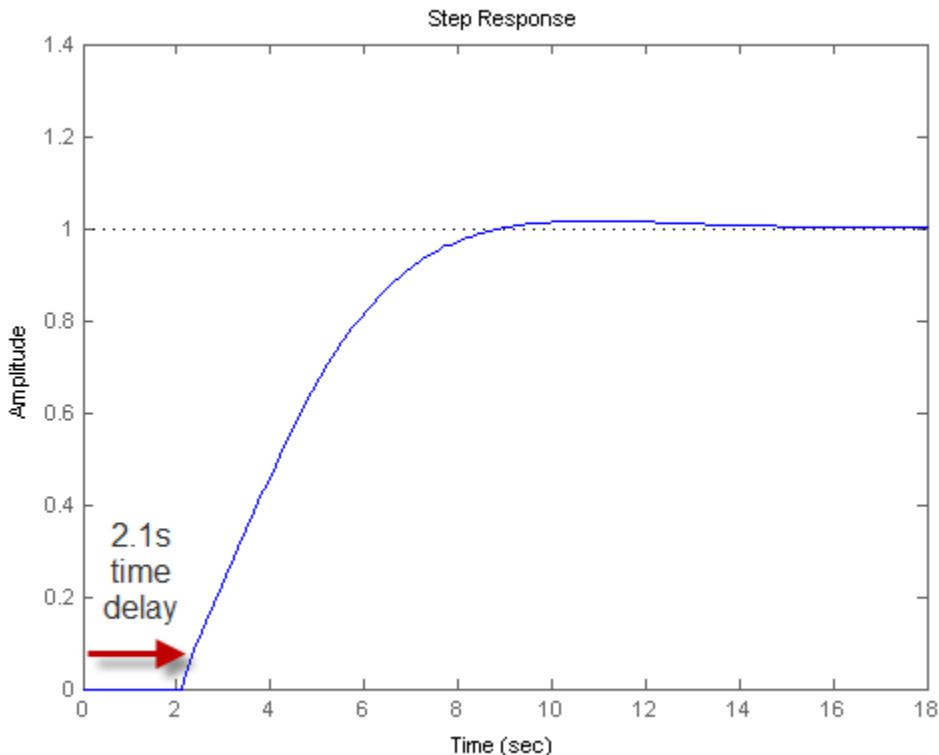
Note: In addition to `feedback`, any system interconnection function (including `parallel` and `series`) can give rise to internal delays.

T is an exact representation of the closed-loop response, not an approximation. To access the internal delay value, enter:

```
T.InternalDelay
```

A step plot of *T* confirms the presence of the time delay:

```
step(T)
```


Note: Most analysis commands, such as `step`, `bode` and `margin`, support models with internal delays.

The internal time delay is stored in the `InternalDelay` property of `T`. Use dot notation to access `InternalDelay`. For example, to change the internal delay to 3.5 seconds, enter:

```
T.InternalDelay = 3.5
```

You cannot modify the number of internal delays because they are structural properties of the model.

Related Examples

- "Convert Time Delay in Discrete-Time Model to Factors of 1/z" on page 2-65

More About

- "Internal Delays" on page 2-72

Time-Delay Approximation

Many control design algorithms cannot handle time delays directly. For example, techniques such as root locus, LQG, and pole placement do not work properly if time delays are present. A common technique is to replace delays with all-pass filters that approximate the delays.

To approximate time delays in continuous-time LTI models, use the `pade` command to compute a Padé approximation. The Padé approximation is valid only at low frequencies, and provides better frequency-domain approximation than time-domain approximation. It is therefore important to compare the true and approximate responses to choose the right approximation order and check the approximation validity.

Time-Delay Approximation in Discrete-Time Models

For discrete-time models, use `absorbDelay` to convert a time delay to factors of $1/z$ where the time delay is an integer multiple of the sample time.
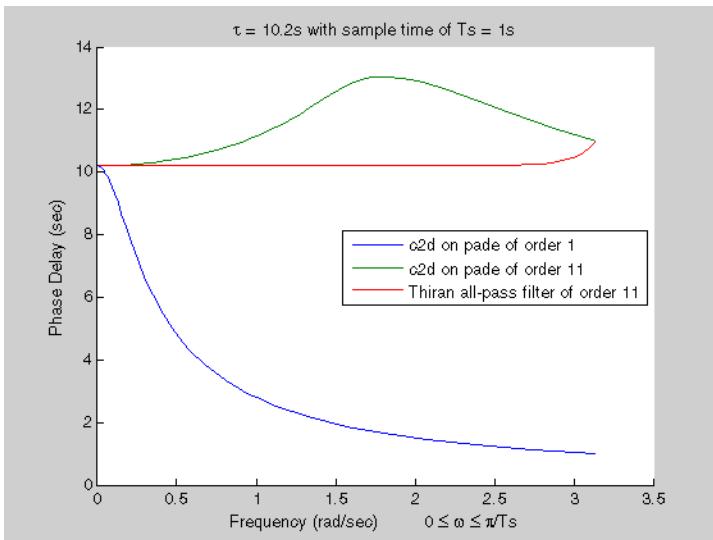
Use the `thiran` command to approximate a time delay that is a fractional multiple of the sample time as a Thiran all-pass filter.

For a time delay of `tau` and a sample time of `Ts`, the syntax `thiran(tau,Ts)` creates a discrete-time transfer function that is the product of two terms:

- A term representing the integer portion of the time delay as a pure line delay, $(1/z)^N$, where `N = ceil(tau/Ts)`.
- A term approximating the fractional portion of the time delay (`tau - NTs`) as a Thiran all-pass filter.

Discretizing a Padé approximation does not guarantee good phase matching between the continuous-time delay and its discrete approximation. Using `thiran` to generate a discrete-time approximation of a continuous-time delay can yield much better phase matching. For example, the following figure shows the phase delay of a 10.2-second time delay discretized with a sample time of 1 s, approximated in three ways:

- a first-order Padé approximation, discretized using the `tustin` method of `c2d`
- an 11th-order Padé approximation, discretized using the `tustin` method of `c2d`
- an 11th-order Thiran filter

The Thiran filter yields the closest approximation of the 10.2-second delay.

See the `thiran` reference page for more information about Thiran filters.

See Also
absorbDelay | pade | thiran
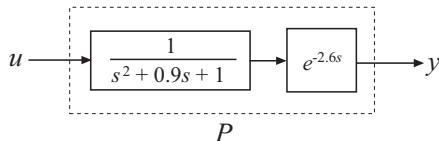
Related Examples
- "Time-Delay Approximation in Continuous-Time Open-Loop Model" on page 2-51
- "Convert Time Delay in Discrete-Time Model to Factors of 1/z" on page 2-65
- "Approximate Different Delays with Different Approximation Orders" on page 2-61

Time-Delay Approximation in Continuous-Time Open-Loop Model

This example shows how to approximate delays in a continuous-time open-loop system using `pade`.

Padé approximation is helpful when using analysis or design tools that do not support time delays.

1   Create sample open-loop system with an output delay.


```
s = tf('s');
P = exp(-2.6*s)/(s^2+0.9*s+1);
```

P is a second-order transfer function (`tf`) object with a time delay.

2   Compute the first-order Padé approximation of P.

```
Pnd1 = pade(P,1)


Pnd1 =

              -s + 0.7692
  ---------------------------------
  s^3 + 1.669 s^2 + 1.692 s + 0.7692

Continuous-time transfer function.
```
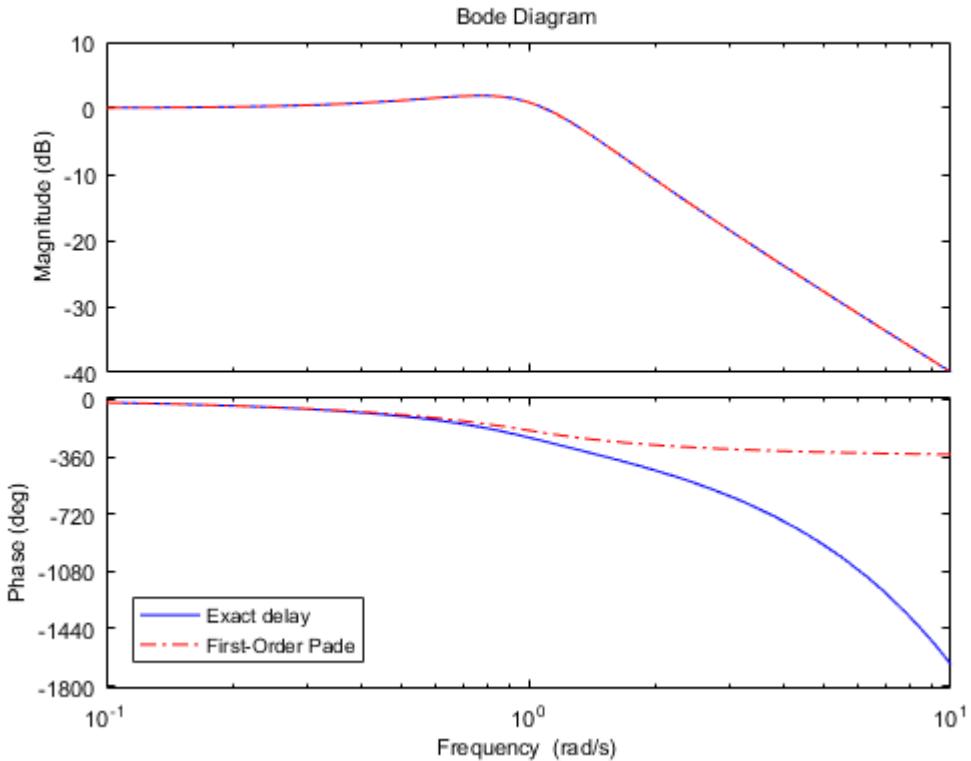
This command replaces all time delays in P with a first-order approximation. Therefore, Pnd1 is a third-order transfer function with no delays.

3   Compare the frequency response of the original and approximate models using `bodeplot`.

```
h = bodeoptions;
h.PhaseMatching = 'on';
bodeplot(P,'-b',Pnd1,'-.r',{0.1,10},h)
```

```
legend('Exact delay','First-Order Pade','Location','SouthWest')
```


Bode Diagram
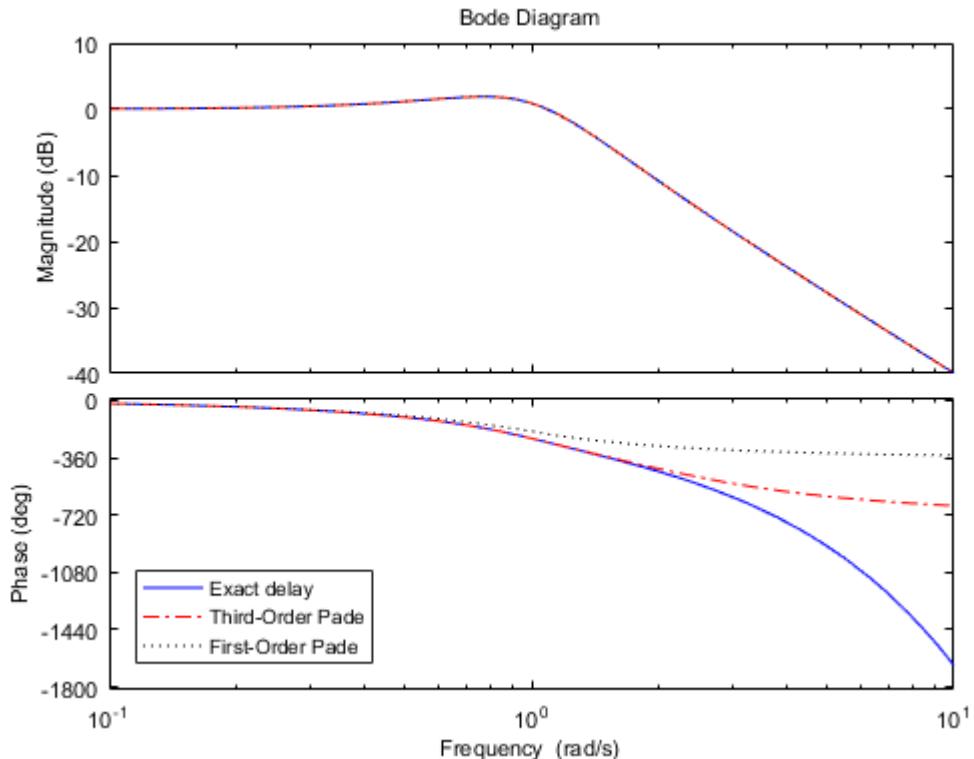
The magnitude of P and Pnd1 match exactly. However, the phase of Pnd1 deviates from the phase of P beyond approximately 1 rad/s.

4 Increase the Padé approximation order to extend the frequency band in which the phase approximation is good.

```
Pnd3 = pade(P,3);
```

5 Compare the frequency response of P, Pnd1 and Pnd3.
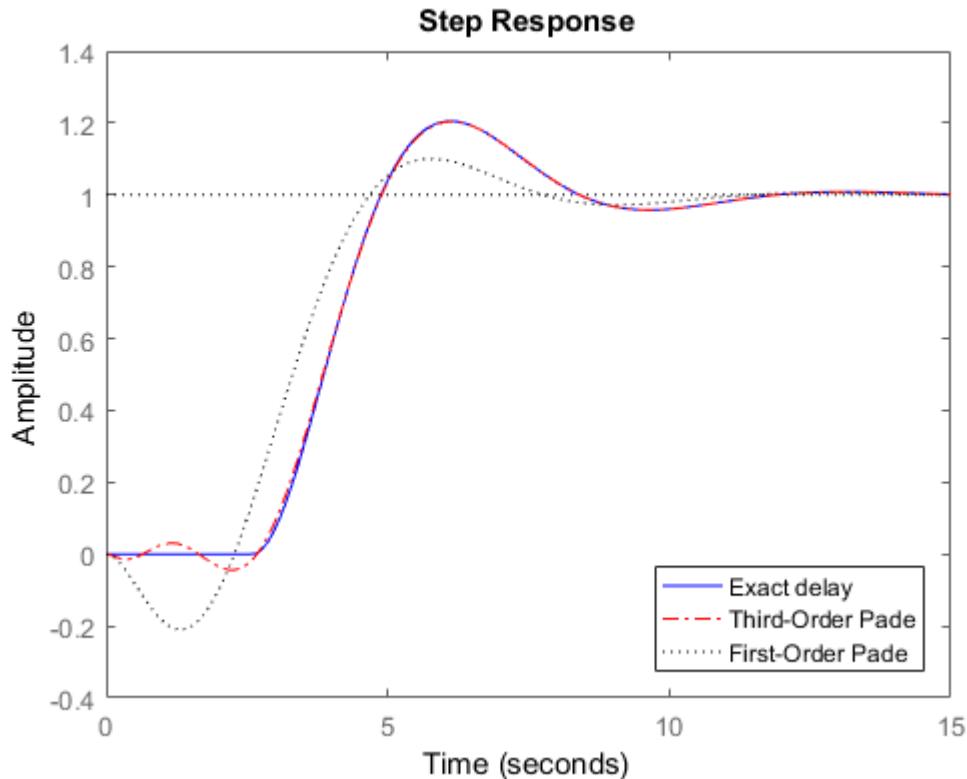
```
bodeplot(P,'-b',Pnd3,'-.r',Pnd1,':k',{0.1 10},h)
legend('Exact delay','Third-Order Pade','First-Order Pade',...
        'Location','SouthWest')
```

The phase approximation error is reduced by using a third-order Padé approximation.

6 Compare the time domain responses of the original and approximated systems using `stepplot`.

```
stepplot(P,'-b',Pnd3,'-.r',Pnd1,':k')
legend('Exact delay','Third-Order Pade','First-Order Pade',...
       'Location','Southeast')
```

Using the Padé approximation introduces a nonminimum phase artifact ("wrong way" effect) in the initial transient response. The effect is quite pronounced in the first-order approximation, which dips significantly below zero before changing direction. The effect is reduced in the higher-order approximation, which far more closely matches the exact system's response.

Note: Using too high an approximation order may result in numerical issues and possibly unstable poles. Therefore, avoid Padé approximations with order N>10.

See Also

pade

Related Examples

• "Time-Delay Approximation in Continuous-Time Closed-Loop Model" on page 2-56

More About
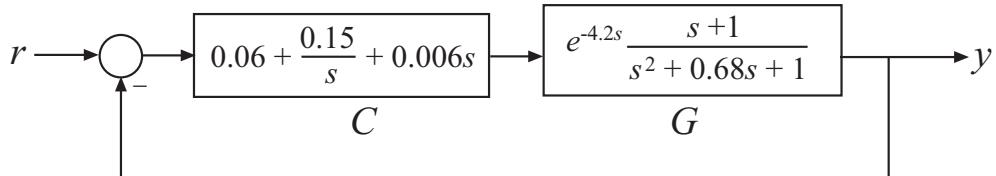
• "Time-Delay Approximation" on page 2-49
• "Internal Delays" on page 2-72

Time-Delay Approximation in Continuous-Time Closed-Loop Model

This example shows how to approximate delays in a continuous-time closed-loop system with internal delays, using `pade`.

Padé approximation is helpful when using analysis or design tools that do not support time delays.

1 Create sample continuous-time closed-loop system with an internal delay.

Construct a model `Tcl` of the closed-loop transfer function from `r` to `y`.

```
s = tf('s');
G = (s+1)/(s^2+.68*s+1)*exp(-4.2*s);
C = pid(0.06,0.15,0.006);
Tcl = feedback(G*C,1);
```

Examine the internal delay of `Tcl`.

```
Tcl.InternalDelay
```

```
ans =

    4.2000
```

2 Compute the first-order Padé approximation of `Tcl`.
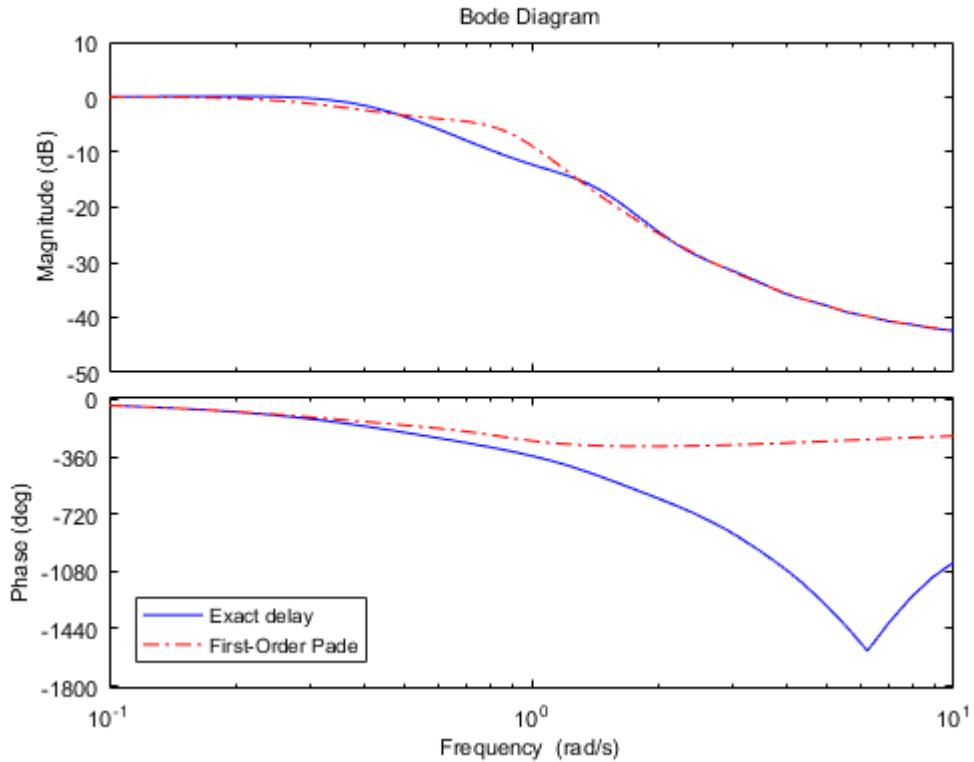
```
Tnd1 = pade(Tcl,1);
```

`Tnd1` is a state-space (`ss`) model with no delays.

3 Compare the frequency response of the original and approximate models.

```
h = bodeoptions;
h.PhaseMatching = 'on';
bodeplot(Tcl,'-b',Tnd1,'-.r',{.1,10},h);
```
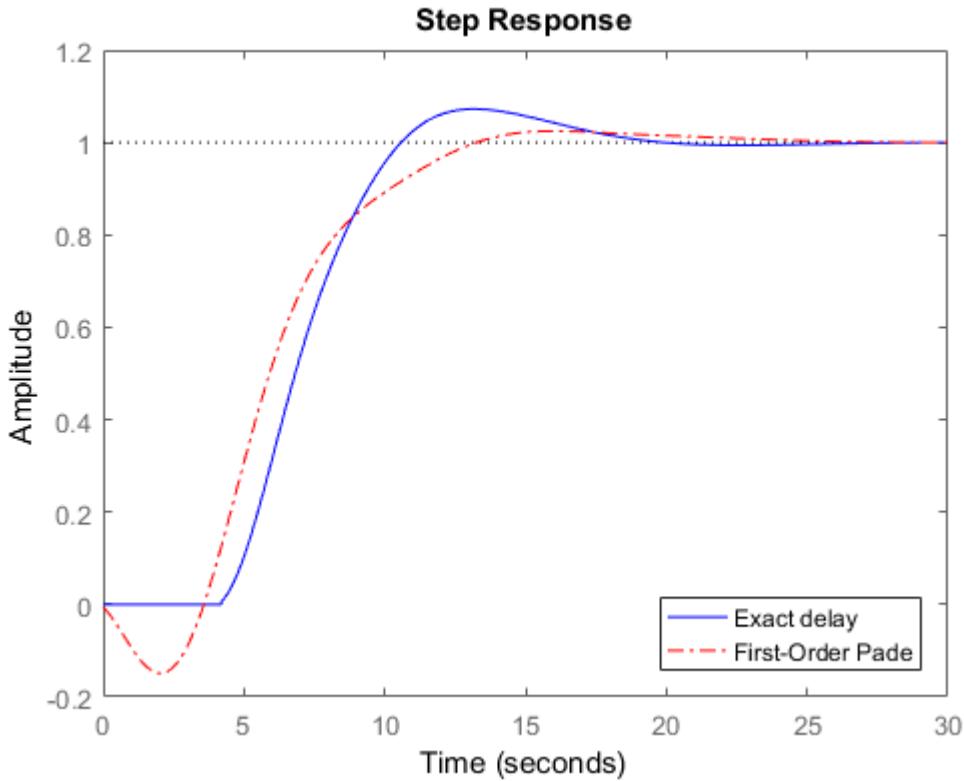
```matlab
legend('Exact delay','First-Order Pade','Location','SouthWest');
```


The magnitude and phase approximation errors are significant beyond 1 rad/s.

4 Compare the time domain response of `Tcl` and `Tnd1` using `stepplot`.

```matlab
stepplot(Tcl,'-b',Tnd1,'-.r');
legend('Exact delay','First-Order Pade','Location','SouthEast');
```
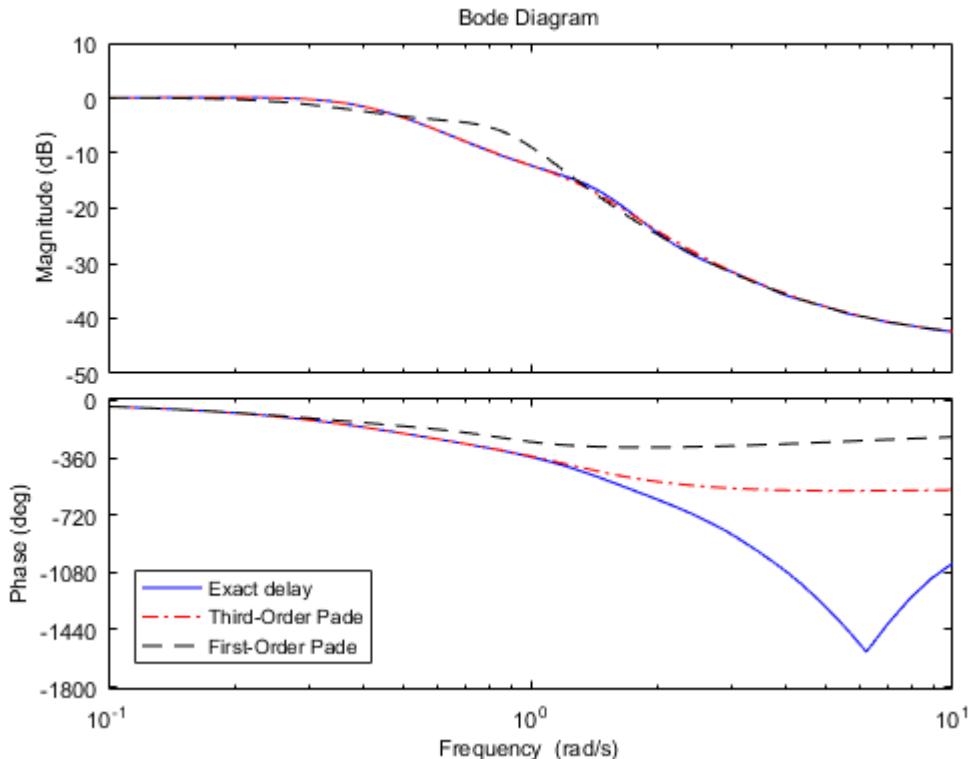
Using the Padé approximation introduces a nonminimum phase artifact ("wrong way" effect) in the initial transient response.

5 Increase the Padé approximation order to see if this will extend the frequency with good phase and magnitude approximation.

```
Tnd3 = pade(Tcl,3);
```

6 Observe the behavior of the third-order Padé approximation of `Tcl`. Compare the frequency response of `Tcl` and `Tnd3`.

```
bodeplot(Tcl,'-b',Tnd3,'-.r',Tnd1,'--k',{.1,10},h);
legend('Exact delay','Third-Order Pade','First-Order Pade',...
       'Location','SouthWest');
```

The magnitude and phase approximation errors are reduced when a third-order Padé approximation is used.

Increasing the Padé approximation order extends the frequency band where the approximation is good. However, too high an approximation order may result in numerical issues and possibly unstable poles. Therefore, avoid Padé approximations with order N>10.

See Also
pade

Related Examples

More About

Approximate Different Delays with Different Approximation Orders

This example shows how to specify different Padé approximation orders to approximate internal and output delays in a continuous-time open-loop system.

Load a sample continuous-time open-loop system that contains internal and output time delays.

```
load(fullfile(matlabroot,'examples','control','PadeApproximation1.mat'),'sys')
sys
```

```
sys =

  A =
        x1    x2
   x1  -1.5  -0.1
   x2    1     0

  B =
       u1
   x1   1
   x2   0

  C =
        x1    x2
   y1  0.5   0.1

  D =
       u1
   y1   0

  (values computed with all internal delays set to zero)

  Output delays (seconds): 1.5
  Internal delays (seconds): 3.4

Continuous-time state-space model.
```

sys is a second-order continuous-time ss model with internal delay 3.4 s and output delay 1.5 s.

Use the `pade` function to compute a third-order approximation of the internal delay and a first-order approximation of the output delay.

```
P13 = pade(sys,inf,1,3);
size(P13)
```

```
State-space model with 1 outputs, 1 inputs, and 6 states.
```

The three input arguments following `sys` specify the approximation orders of any input, output, and internal delays of `sys`, respectively. `inf` specifies that a delay is not to be approximated. The approximation orders for the output and internal delays are one and three respectively.

Approximating the time delays with `pade` absorbs delays into the dynamics, adding as many states to the model as orders in the approximation. Thus, `P13` is a sixth-order model with no delays.

For comparison, approximate only the internal delay of `sys`, leaving the output delay intact.

```
P3 = pade(sys,inf,inf,3);
size(P3)
```

```
State-space model with 1 outputs, 1 inputs, and 5 states.
```

```
P3.OutputDelay
```

```
ans =

    1.5000
```

```
P3.InternalDelay
```

```
ans =

  Empty matrix: 0-by-1
```
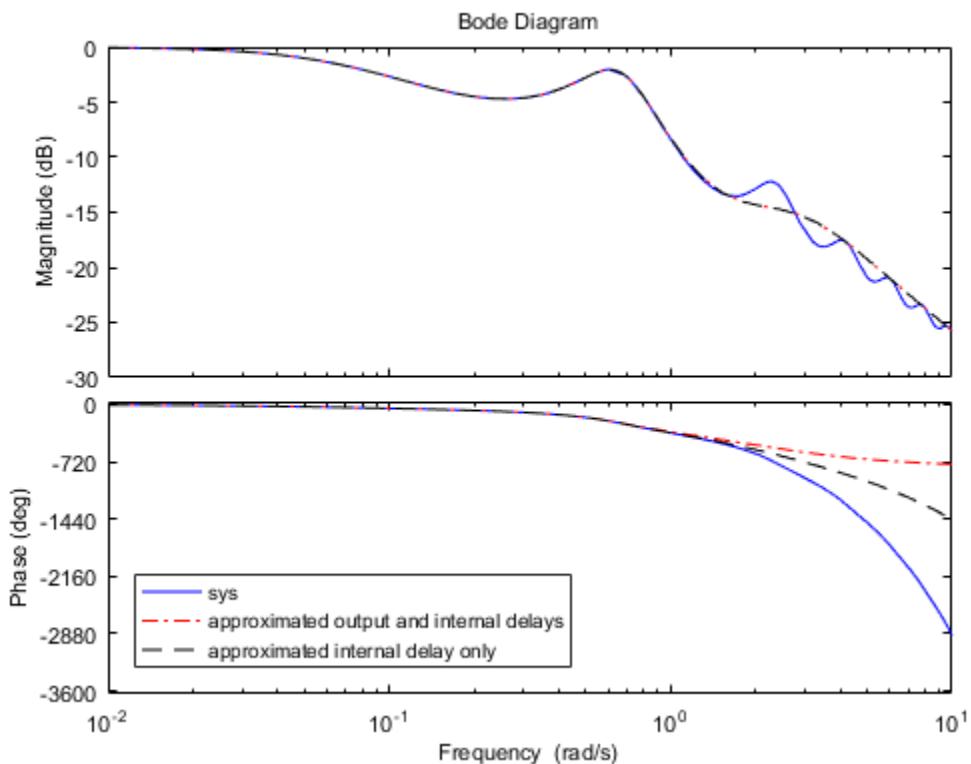
`P3` retains the output delay, but the internal delay is approximated and absorbed into the state-space matrices, resulting in a fifth-order model without internal delays.

Compare the frequency response of the exact and approximated systems `sys`, `P13`, `P3`.

```
h = bodeoptions;
h.PhaseMatching = 'on';
bode(sys,'b-',P13,'r-.',P3,'k--',h,{.01,10});
legend('sys','approximated output and internal delays','approximated internal delay on
    'location','SouthWest')
```


Notice that approximating the internal delay loses the gain ripple displayed in the exact system.

See Also
pade

Related Examples

More About
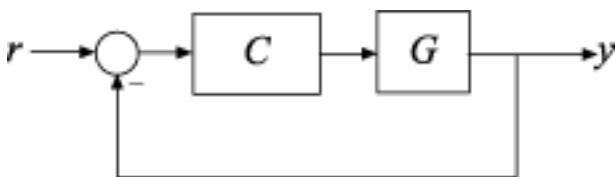
Convert Time Delay in Discrete-Time Model to Factors of 1/z

This example shows how to convert a time delay in a discrete-time model to factors of 1/_z_.

In a discrete-time model, a time delay of one sampling interval is equivalent to a factor of 1/_z_ (a pole at $z = 0$) in the model. Therefore, time delays stored in the `InputDelay`, `OutputDelay`, or `ioDelay` properties of a discrete-time model can be rewritten in the model dynamics by rewriting them as poles at $z = 0$. However, the additional poles increase the order of the system. Particularly for large time delays, this can yield systems of very high order, leading to long computation times or numerical inaccuracies.

To illustrate how to eliminate time delays in a discrete-time closed-loop model, and to observe the effects of doing so, create the following closed-loop system:

G is a first-order discrete-time system with an input delay, and C is a PI controller.

```
G = ss(0.9,0.125,0.08,0,'Ts',0.01,'InputDelay',7);
C = pid(6,90,0,0,'Ts',0.01);
T = feedback(C*G,1);
```

Closing the feedback loop on a plant with input delays gives rise to internal delays in the closed-loop system. Examine the order and internal delay of T.

```
order(T)
```

```
ans =

     2
```

```
T.InternalDelay
```

```
ans =

     7
```

T is a second-order state-space model. One state is contributed by the first-order plant, and the other by the one pole of the PI controller. The delays do not increase the order of T. Instead, they are represented as an internal delay of seven time steps.

Replace the internal delay by z^{-7}.

```
Tnd = absorbDelay(T);
```

This command converts the internal delay to seven poles at $z = 0$. To confirm this, examine the order and internal delay of Tnd.

```
order(Tnd)
```

```
ans =

     9
```

```
Tnd.InternalDelay
```
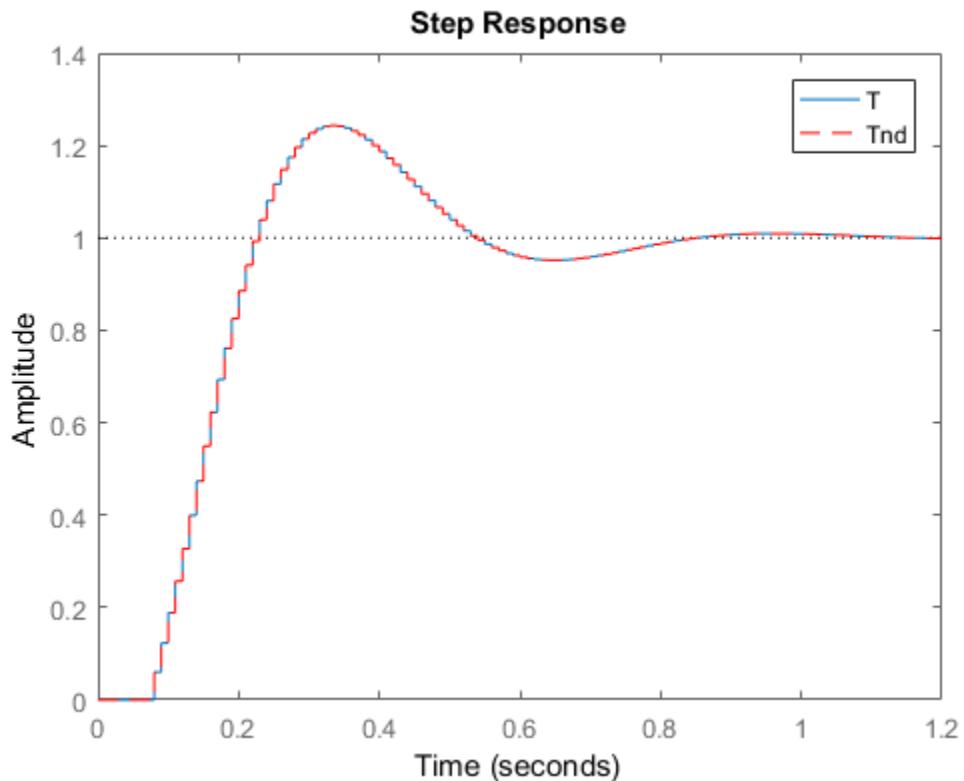
```
ans =

   Empty matrix: 0-by-1
```
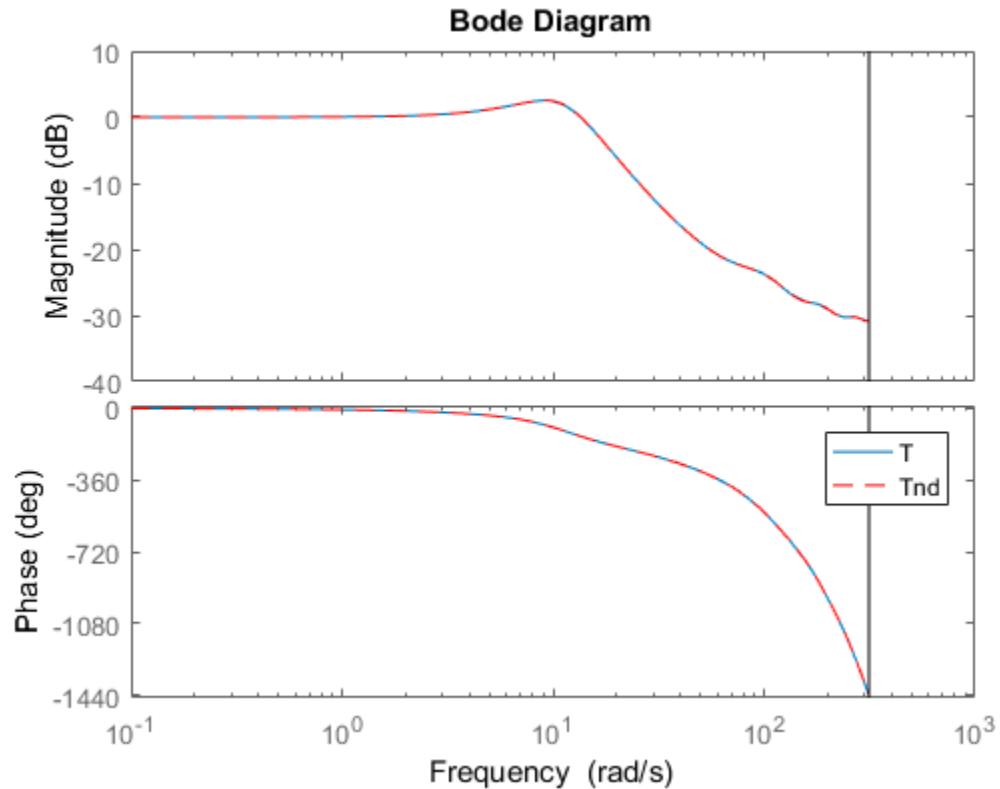
Tnd has no internal delay, but it is a ninth-order model, due to the seven extra poles introduced by absorbing the seven-unit delay into the model dynamics.

Despite this difference in representation, the responses of Tnd exactly match those of T.

```
stepplot(T,Tnd,'r--')
legend('T','Tnd')
```

Step Response


```
bodeplot(T,Tnd,'r--')
legend('T','Tnd')
```

See Also

pade

Related Examples

- "Time-Delay Approximation in Continuous-Time Open-Loop Model" on page 2-51

More About

- "Time-Delay Approximation" on page 2-49
- "Internal Delays" on page 2-72

Frequency Response Data (FRD) Model with Time Delay

This example shows that absorbing time delays into frequency response data can cause undesirable phase wrapping at high frequencies.

When you collect frequency response data for a system that includes time delays, you can absorb the time delay into the frequency response as a phase shift. Alternatively, if you are able to separate time delays from your measured frequency response, you can represent the delays using the InputDelay, OutputDelay, or ioDelay properties of the frd model object. The latter approach can give better numerical results, as this example illustrates.
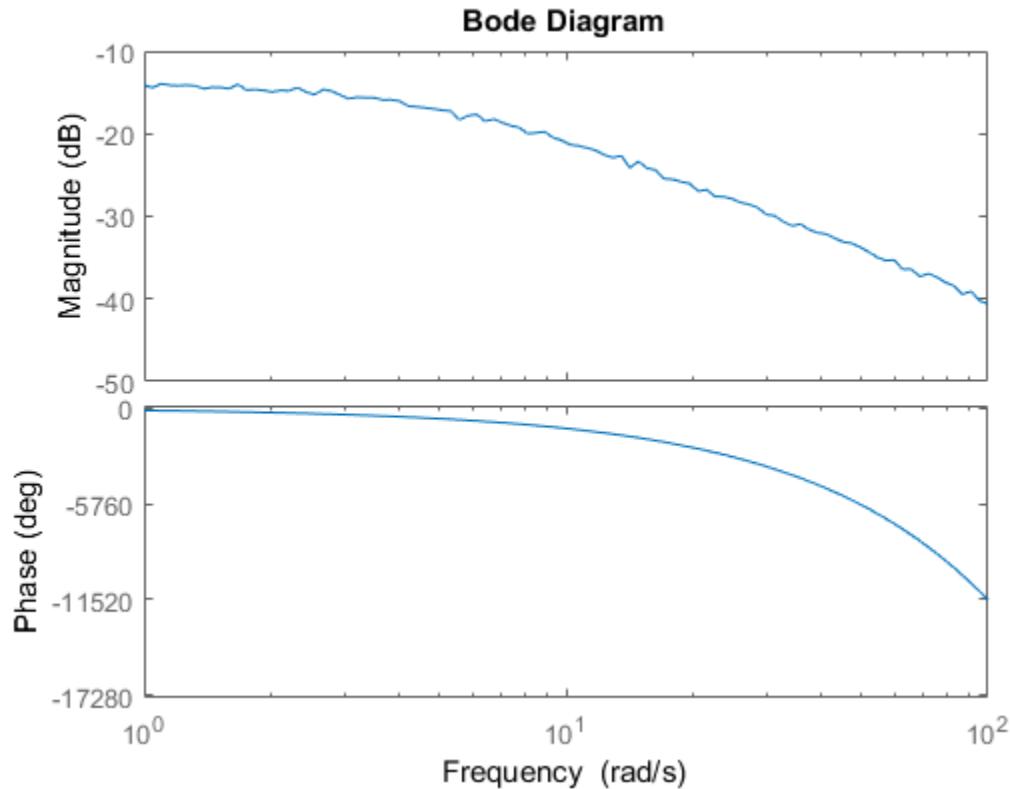
The frd model fsys includes a transport delay of 2 s. Load the model into the MATLAB® workspace and inspect the time delay.

```
load(fullfile(matlabroot,'examples','control','frddelayexample.mat'),'fsys')
fsys.ioDelay
```

```
ans =

     2
```

A Bode plot of fsys shows the effect of the transport delay, causing the accumulation of phase as frequency increases.

```
bodeplot(fsys)
```

The `absorbDelay` command absorbs all time delays directly into the frequency response, resulting in an `frd` model with `ioDelay = 0`.
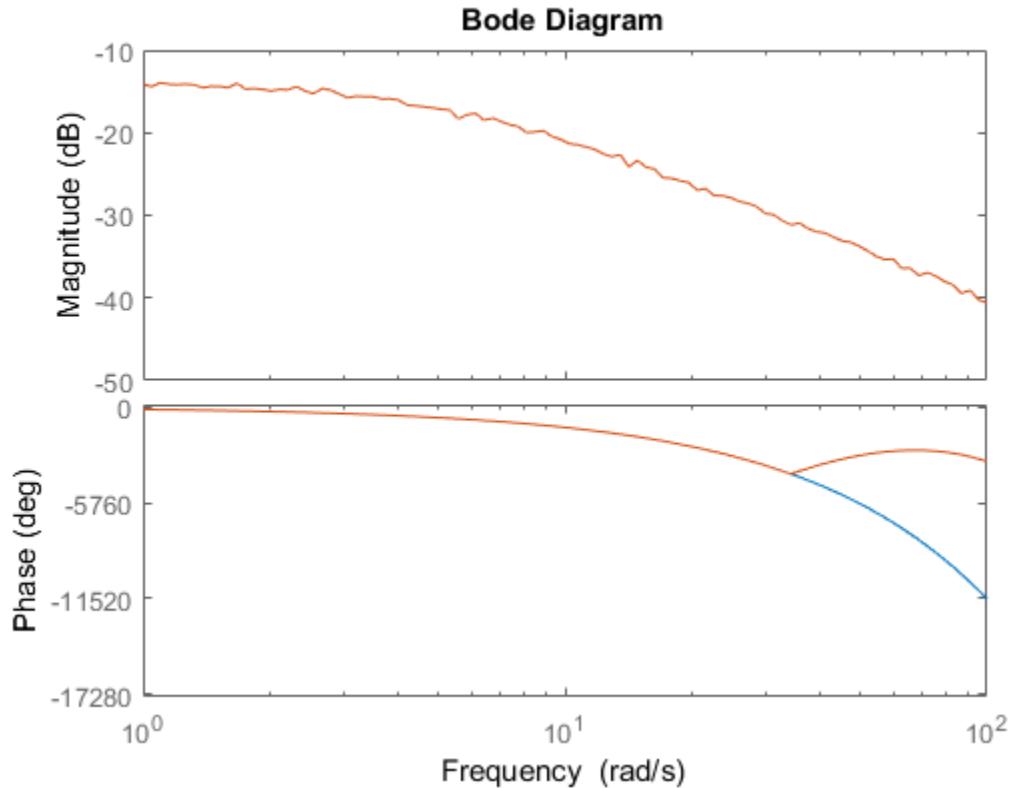
```
fsys2 = absorbDelay(fsys);
fsys2.ioDelay
```

```
ans =

     0
```

Comparing the two ways of representing the delay shows that absorbing the delay into the frequency response causes phase-wrapping.

```
bode(fsys,fsys2)
```


Phase wrapping can introduce numerical inaccuracy at high frequencies or where the frequency grid is sparse. For that reason, if your system takes the form $e^{-\tau s}G(s)$, you might get better results by measuring frequency response data for $G(s)$ and using `InputDelay`, `OutputDelay`, or `ioDelay` to model the time delay τ.

See Also

`absorbDelay`

More About

- "Time-Delay Approximation" on page 2-49

Internal Delays

Using the `InputDelay`, `OutputDelay`, and `ioDelay` properties, you can model simple processes with transport delays. However, these properties cannot model more complex situations, such as feedback loops with delays. In addition to the `InputDelay` and `OutputDelay` properties, state-space (`ss`) models have an `InternalDelay` property. This property lets you model the interconnection of systems with input, output, or transport delays, including feedback loops with delays. You can use `InternalDelay` property to accurately model and analyze arbitrary linear systems with delays. Internal delays can arise from the following:
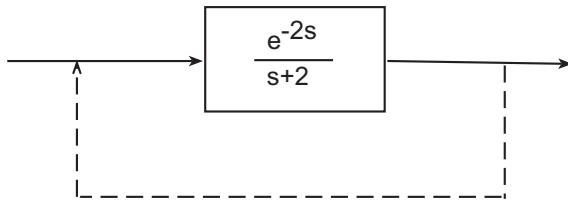
- Concatenating state-space models with input and output delays
- Feeding back a delayed signal
- Converting MIMO `tf` or `zpk` models with transport delays to state-space form

Using internal time delays, you can do the following:

- In continuous time, generate approximate-free time and frequency simulations, because delays do not have to be replaced by a Padé approximation. In continuous time, this allows for more accurate analysis of systems with long delays.
- In discrete time, keep delays separate from other system dynamics, because delays are not replaced with poles at $z = 0$, which boosts efficiency of time and frequency simulations for discrete-time systems with long delays.
- Use most Control System Toolbox functions.
- Test advanced control strategies for delayed systems. For example, you can implement and test an accurate model of a Smith predictor. See the example Control of Processes with Long Dead Time: The Smith Predictor .

Why Internal Delays Are Necessary

This example illustrates why input, output, and transport delays not enough to model all types of delays that can arise in dynamic systems. Consider the simple feedback loop with a 2 s. delay:

The closed-loop transfer function is

$$\frac{e^{-2s}}{s+2+e^{-2s}}$$

The delay term in the numerator can be represented as an output delay. However, the delay term in the denominator cannot. In order to model the effect of the delay on the feedback loop, the `InternalDelay` property is needed to keep track of internal coupling between delays and ordinary dynamics.

Typically, you do not create state-space models with internal delays directly, by specifying the *A*, *B*, *C*, and *D* matrices together with a set of internal delays. Rather, such models arise when you interconnect models having delays. There is no limitation on how many delays are involved and how the models are connected. For an example of creating an internal delay by closing a feedback loop, see "Closing Feedback Loops with Time Delays" on page 2-46.

Behavior of Models With Internal Delays

When you work with models having internal delays, be aware of the following behavior:

- When a model interconnection gives rise to internal delays, the software returns an `ss` model regardless of the interconnected model types. This occurs because only `ss` supports internal delays.
- The software fully supports feedback loops. You can wrap a feedback loop around any system with delays.
- When displaying the `A`, `B`, `C`, and `D` matrices, the software sets all delays to zero (creating a zero-order Padé approximation). This approximation occurs for the display only, and not for calculations using the model.

  For some systems, setting delays to zero creates singular algebraic loops, which result in either improper or ill-defined, zero-delay approximations. For these systems:
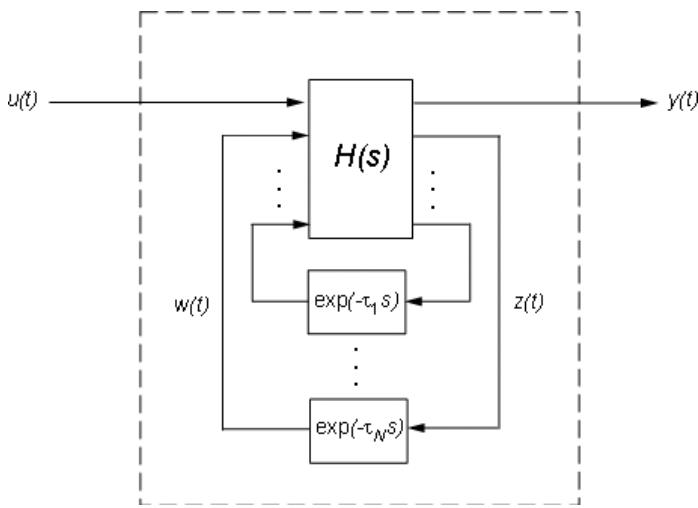
- Entering `sys` returns only sizes for the matrices of a system named `sys`.
- Entering `sys.a` produces an error.

  The limited display and the error do not imply a problem with the model `sys` itself.

Inside Time Delay Models

State-space objects use generalized state-space equations to keep track of internal delays. Conceptually, such models consist of two interconnected parts:

- An ordinary state-space model *H(s)* with an augmented I/O set
- A bank of internal delays.

The corresponding state-space equations are:

$$\dot{x} = Ax(t) + B_1 u(t) + B_2 w(t)$$
$$y(t) = C_1 x(t) + D_{11} u(t) + D_{12} w(t)$$
$$z(t) = C_2 x(t) + D_{21} u(t) + D_{22} w(t)$$
$$w_j(t) = z(t - \tau_j), \quad j = 1,...,N$$

You need not bother with this internal representation to use the tools. If, however, you want to extract H or the matrices A, B1, B2, ... , you can use `getDelayModel`, For the example:

```
P = 5*exp(-3.4*s)/(s+1);
C = 0.1 * (1 + 1/(5*s));
T = feedback(ss(P*C),1);
[H,tau] = getDelayModel(T,'lft');
size(H)
```

Note that H is a two-input, two-output model whereas T is SISO. The inverse operation (combining H and `tau` to construct T) is performed by `setDelayModel`.

See [1], [2] for details.

Functions That Support Internal Time Delays

The following commands support internal delays for both continuous- and discrete-time systems:

- All interconnection functions
- Time domain response functions—except for `impulse` and `initial`
- Frequency domain functions—except for `norm`

Limitations on Functions that Support Internal Time Delays

The following commands support internal delays for both continuous- and discrete-time systems and have certain limitations:

- `allmargin`, `margin`—Uses interpolation, therefore these commands are only as precise as the fineness of the specified grid.
- `pole`, `zero`—Returns poles and zeros of the system with all delays set to zero.
- `ssdata`, `get`—If an SS model has internal delays, these commands return the A, B, C, and D matrices of the system with all internal delays set to zero. Use `getDelayModel` to access the internal state-space representation of models with internal delays.

Functions That Do Not Support Internal Time Delays

The following commands do not support internal time delays:

- System dynamics—`norm` and `isstable`
- Time-domain analysis—`initial` and `initialplot`
- Model simplification—`balreal`, `balred`, and `modred`
- Compensator design—`rlocus`, `lqg`, `lqry`, `lqrd`, `kalman`, `kalmd`, `lqgreg`, `lqgtrack`, `lqi`, and `augstate`.

  To use these functions on a system with internal delays, use `pade` to approximate the internal delays. See "Time-Delay Approximation" on page 2-49.

References

[1] P. Gahinet and L.F. Shampine, "Software for Modeling and Analysis of Linear Systems with Delays," *Proc. American Control Conf.*, Boston, 2004, pp. 5600-5605

[2] L.F. Shampine and P. Gahinet, Delay-differential-algebraic Equations in Control Theory, *Applied Numerical Mathematics*, 56 (2006), pp. 574-588

Related Examples
- "Closing Feedback Loops with Time Delays" on page 2-46

Tunable Low-Pass Filter

This example shows how to create the low-pass filter $F = a/(s + a)$ with one tunable parameter a.

You cannot use `tunableTF` to represent F, because the numerator and denominator coefficients of a `tunableTF` block are independent. Instead, construct F using the tunable real parameter object `realp`.

1 Create a tunable real parameter.

```
a = realp('a',10);
```

The `realp` object `a` is a tunable parameter with initial value 10.

2 Use `tf` to create the tunable filter `F`:

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

F is a `genss` object which has the tunable parameter `a` in its `Blocks` property. You can connect F with other tunable or numeric models to create more complex models of control systems. For an example, see "Control System with Tunable Components" on page 2-82.

Related Examples

More About

Tunable Second-Order Filter

This example shows how to create a parametric model of the second-order filter:

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

where the damping ζ and the natural frequency ω_n are tunable parameters.

1   Define the tunable parameters using `realp`.

```
wn = realp('wn',3);
zeta = realp('zeta',0.8);
```

wn and `zeta` are `realp` parameter objects, with initial values 3 and 0.8, respectively.

2   Create a model of the filter using the tunable parameters.

```
F = tf(wn^2,[1 2*zeta*wn wn^2])
```

The inputs to `tf` are the vectors of numerator and denominator coefficients expressed in terms of `wn` and `zeta`.

F is a `genss`. The property `F.Blocks` lists the two tunable parameters `wn` and `zeta`.

3   (Optional) Examine the number of tunable blocks in the model using `nblocks`.

```
nblocks(F)
```

This command returns the result:

```
ans =

     6
```

F has two tunable parameters, but the parameter `wn` appears five times — twice in the numerator and three times in the denominator.

4   (Optional) Rewrite F for fewer occurrences of `wn`.

The second-order filter transfer function can be expressed as follows:

$$F(s) = \frac{1}{\left(\dfrac{s}{\omega_n}\right)^2 + 2\zeta\left(\dfrac{s}{\omega_n}\right) + 1}.$$

Use this expression to create the tunable filter:

```
F = tf(1,[(1/wn)^2 2*zeta*(1/wn) 1])
```

5 (Optional) Examine the number of tunable blocks in the new filter model.

```
nblocks(F)
```

This command returns the result:

```
ans =

     4
```

In the new formulation, there are only three occurrences of the tunable parameter wn. Reducing the number of occurrences of a block in a model can improve performance time of calculations involving the model. However, the number of occurrences does not affect the results of tuning the model or sampling it for parameter studies.

Related Examples

- "Tunable Low-Pass Filter" on page 2-77
- "State-Space Model with Both Fixed and Tunable Parameters" on page 2-80
- "Control System with Tunable Components" on page 2-82

More About

- "Models with Tunable Coefficients" on page 1-19

State-Space Model with Both Fixed and Tunable Parameters

This example shows how to create a state-space (`genss`) model having both fixed and tunable parameters.

Create a state-space model having the following state-space matrices:

$$A = \begin{bmatrix} 1 & a+b \\ 0 & ab \end{bmatrix}, \quad B = \begin{bmatrix} -3.0 \\ 1.5 \end{bmatrix}, \quad C = \begin{bmatrix} 0.3 & 0 \end{bmatrix}, \quad D = 0,$$

where a and b are tunable parameters, whose initial values are -1 and 3, respectively.

1 Create the tunable parameters using `realp`.

```
a = realp('a',-1);
b = realp('b',3);
```

2 Define a generalized matrix using algebraic expressions of `a` and `b`.

```
A = [1 a+b;0 a*b]
```

A is a generalized matrix whose `Blocks` property contains `a` and `b`. The initial value of A is `M = [1 2;0 -3]`, from the initial values of `a` and `b`.

3 Create the fixed-value state-space matrices.

```
B = [-3.0;1.5];
C = [0.3 0];
D = 0;
```

4 Use `ss` to create the state-space model.

```
sys = ss(A,B,C,D)
```

`sys` is a generalized LTI model (`genss`) with tunable parameters `a` and `b`.

Related Examples
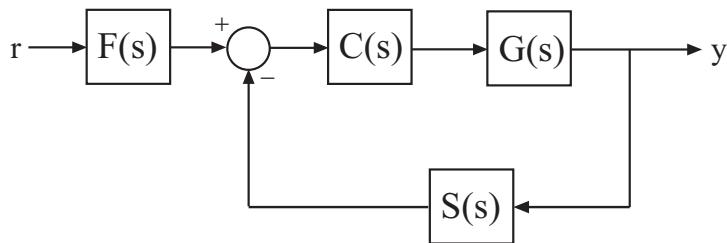
- "Tunable Low-Pass Filter" on page 2-77
- "Tunable Second-Order Filter" on page 2-78
- "Control System with Tunable Components" on page 2-82

More About

*   "Models with Tunable Coefficients" on page 1-19

Control System with Tunable Components

This example shows how to create a tunable model of the control system in the following illustration.

The plant response $G(s) = 1/(s + 1)^2$. The model of sensor dynamics is $S(s) = 5/(s + 4)$. The controller C is a tunable PID controller, and the prefilter $F = a/(s + a)$ is a low-pass filter with one tunable parameter, a.

Create models representing the plant and sensor dynamics.

Because the plant and sensor dynamics are fixed, represent them using numeric LTI models `zpk` and `tf`.

```
G = zpk([],[-1,-1],1);
S = tf(5,[1 4]);
```

Create a tunable representation of the controller C.

```
C = tunablePID('C','PID');

C =

  Parametric continuous-time PID controller "C" with formula:

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

  and tunable parameters Kp, Ki, Kd, Tf.

Type "pid(C)" to see the current value and "get(C)" to see all properties.
```

`C` is a `tunablePID` object, which is a Control Design Block with a predefined proportional-integral-derivative (PID) structure.

Create a model of the filter $F = a/(s + a)$ with one tunable parameter.

```
a = realp('a',10);
F = tf(a,[1 a]);
```

a is a `realp` (real tunable parameter) object with initial value 10. Using a as a coefficient in `tf` creates the tunable `genss` model object F.

Connect the models together to construct a model of the closed-loop response from *r* to *y*.

```
T = feedback(G*C,S)*F
```

T is a `genss` model object. In contrast to an aggregate model formed by connecting only Numeric LTI models, T keeps track of the tunable elements of the control system. The tunable elements are stored in the `Blocks` property of the `genss` model object.

Display the tunable elements of T.

```
T.Blocks

ans =

    C: [1x1 tunablePID]
    a: [1x1 realp]
```

You can use tuning commands such as `systune` to tune the free parameters of T to meet design requirements you specify.

Related Examples
*   "Tunable Low-Pass Filter" on page 2-77
*   "Tunable Second-Order Filter" on page 2-78
*   "State-Space Model with Both Fixed and Tunable Parameters" on page 2-80
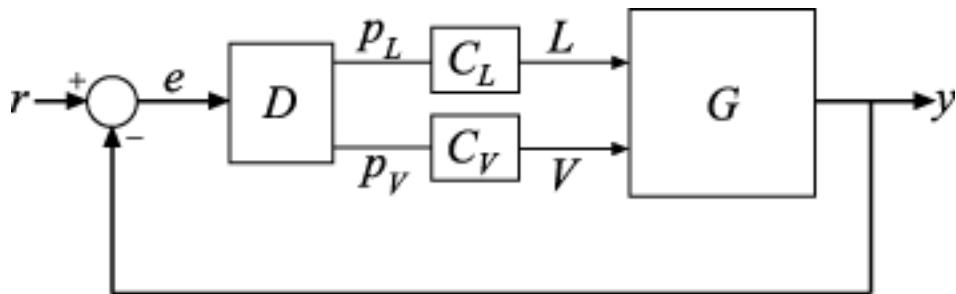
More About
*   "Models with Tunable Coefficients" on page 1-19

Control System with Multichannel Analysis Points
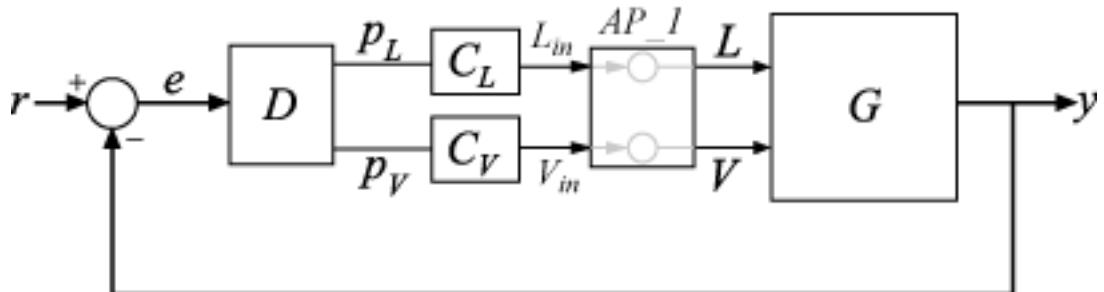
This example shows how to insert multichannel analysis points in a generalized state-space model of a MIMO control system.

Consider the following two-input, two-output control system.

The plant G has two inputs and two outputs. Therefore, the line marked y in the block diagram represents two signals, y(1) and y(2). Similarly, r and e each represent two signals.

Suppose you want to create tuning requirements or extract responses that require injecting or measuring signals at the locations L and V. To do so, create an AnalysisPoint block and include it in the closed-loop model of the control system as shown in the following illustration.

To create a model of this system, first create the numeric LTI models and control design blocks that represent the plant and controller elements. D is a tunable gain block, and C_L and C_V are tunable PI controllers. Suppose the plant model is the following:

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

```
s = tf('s');
G = [87.8 -86.4 ; 108.2 -109.6]/(75*s+1);

D = tunableGain('Decoupler',eye(2));
C_L = tunablePID('C_L','pi');
C_V = tunablePID('C_V','pi');
```

Create an `AnalysisPoint` block that bundles together the L and V channels.

```
AP_1 = AnalysisPoint('AP_1',2)


AP_1 =

Multi-channel analysis point at locations:
   AP_1(1)
   AP_1(2)

Type "ss(AP_1)" to see the current value and "get(AP_1)" to see all properties.
```
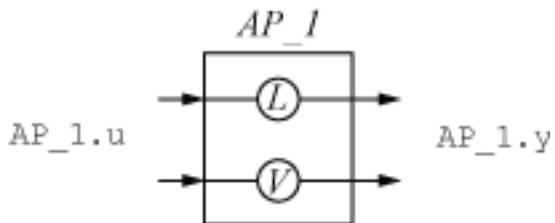
For convenience, rename the channels to match the corresponding signals.

```
AP_1.Location = {'L';'V'}


AP_1 =

Multi-channel analysis point at locations:
   L
   V

Type "ss(AP_1)" to see the current value and "get(AP_1)" to see all properties.
```

The following diagram illustrates the input names, output names, and channel names (locations) in the block AP_1.

The input and output names of the `AnalysisPoint` block are distinct from the channel names. Use the channel names to refer to the analysis-point locations when extracting responses or defining design goals for tuning. You can use the input and output names `AP_1.u` and `AP_1.y`, for example, when interconnecting blocks using the `connect` command.

You can now build the closed-loop model of the control system. First, join all the plant and controller blocks along with the first `AnalysisPoint` block.
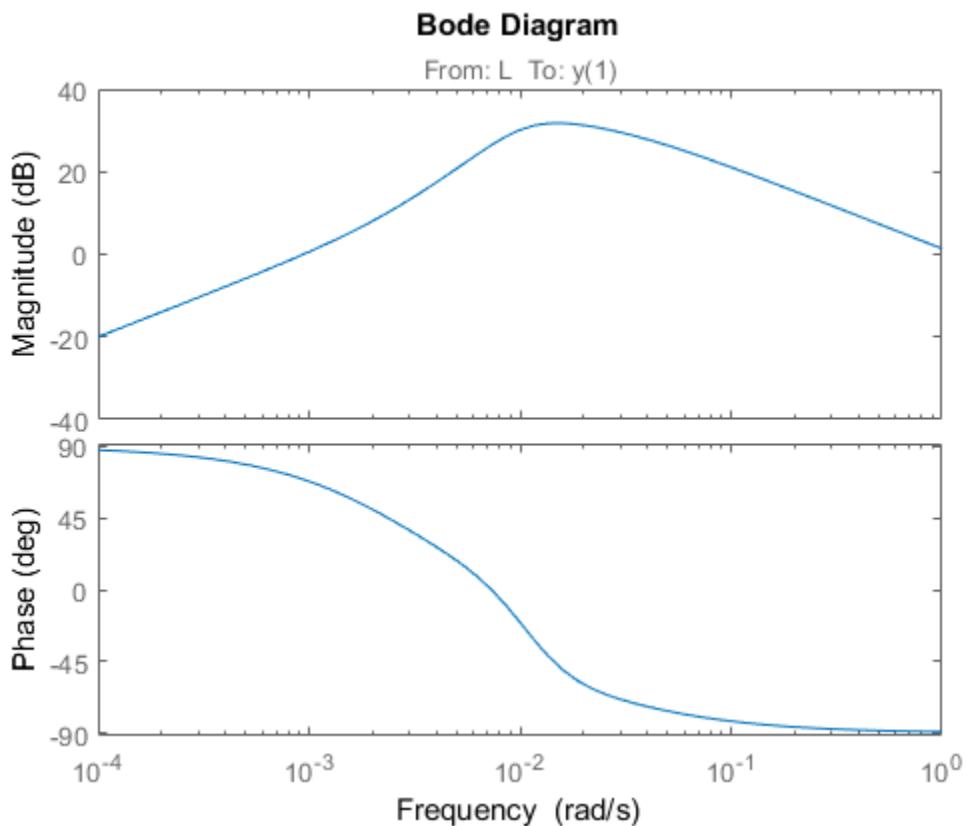
```
GC = G*AP_1*append(C_L,C_V)*D;
```

Then, close the feedback loop. Recall that `GC` has two inputs and outputs.

```
CL = feedback(GC,eye(2));
```

You can now use the analysis points for analysis or tuning. For example, extract the SISO closed-loop transfer function from `'L'` to the first output. Assign a name to the output so you can reference it in analysis functions. The software automatically expands the assigned name `'y'` to the vector-valued output signals `{y(1),y(2)}`.

```
CL.OutputName = 'y';
TLy1 = getIOTransfer(CL,'L','y(1)');
bodeplot(TLy1);
```

See Also

AnalysisPoint

More About

- "Marking Signals of Interest for Control System Analysis and Design" on page 2-88

Marking Signals of Interest for Control System Analysis and Design

In this section...
"Analysis Points" on page 2-88
"Specifying Analysis Points for MATLAB Models" on page 2-90
"Specifying Analysis Points for Simulink Models" on page 2-90
"Referring to Analysis Points for Analysis and Tuning" on page 2-93

Analysis Points

Whether you model your control system in MATLAB or Simulink, use *analysis points* to mark points of interest in the model. Analysis points give you access to internal signals, perform open-loop analysis, or specify requirements for controller tuning. In the block diagram representation, an analysis point can be thought of as an access port to a signal flowing from one block to another. In Simulink, analysis points are attached to the outports of Simulink blocks. For example, the reference signal, r, and the control signal, u, are analysis points of the following simple feedback loop model, ex_scd_analysis_pts1:
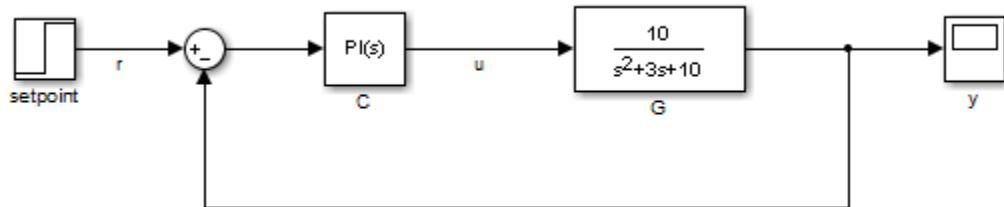
Figure 1: Simple Feedback Loop

Analysis points serve three purposes:

- **Input:** The software injects an additive input signal at an analysis point, for example, to model a disturbance at the plant input, u.

- **Output:** The software measures the signal value at a point, for example, to study the impact of this disturbance on the plant output, y.

- **Loop Opening:** The software interprets a break in the signal flow at a point, for example, to study the open-loop response at the plant input, u.

You can apply these purposes concurrently. For example, to compute the open-loop response from u to y, you can treat u as both a loop opening and an input. When you use an analysis point for more than one purpose, the software always applies the purposes in a specific sequence: output (measurement), then loop opening, then input, as shown in the following diagram.
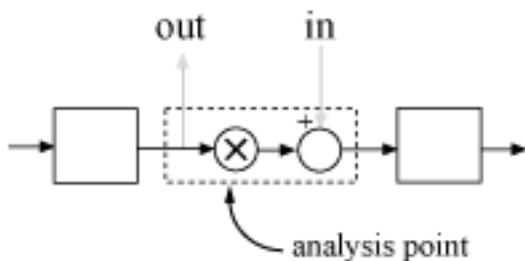
Figure 2: Analysis Point as Input, Output, and Loop Opening

Analysis points enable you to extract open-loop and closed-loop responses from a control system model. For example, suppose T represents the closed-loop system in the model ex_scd_analysis_pts1, and u and y are marked as analysis points. T can be either a generalized state-space model or an slLinearizer or slTuner interface to a Simulink model. You can plot the closed-loop response to a step disturbance at the plant input with the following commands:

```
Tuy = getIOTransfer(T,'u','y');
stepplot(Tuy)
```

Analysis points are also useful to specify design requirements when tuning control systems with the systune command. For example, you can create a requirement that attenuates disturbances at the plant input by a factor of 10 (20 dB) or more.

```
Req = TuningGoal.Rejection('u',10);
```

2-89

Specifying Analysis Points for MATLAB Models

Consider an LTI model of the block diagram in Figure 1.

```
G = tf(10,[1 3 10]);
C = pid(0.2,1.5);
T = feedback(G*C,1);
```

With this model, you can obtain the closed-loop response from r to y. However, you cannot analyze the open-loop response at the plant input or simulate the rejection of a step disturbance at the plant input. To enable such analysis, mark the signal u as an analysis point by inserting an `AnalysisPoint` block between the plant and controller.

```
AP = AnalysisPoint('u');
T = feedback(G*AP*C,1);
```

The plant input, u, is now available for analysis. For instance, you can plot the open-loop response at u.

```
bodeplot(getLoopTransfer(T,'u',-1))
```

Recall that the `AnalysisPoint` block includes an implied loop-opening switch that behaves as shown in Figure 2 for analysis purposes. By default, this switch is closed when computing closed-loop responses. For example, plot the closed-loop response to a step disturbance at the plant input.

```
T.OutputName = 'y';
stepplot(getIOTransfer(T,'u','y'))
```

In creating the model T, you manually created the analysis point block AP and explicitly included it in the feedback loop. When you combine models using the `connect` command, you can instruct the software to insert analysis points automatically at the locations you specify. For more information, see the `connect` reference page.

Specifying Analysis Points for Simulink Models

In Simulink, you can mark analysis points either explicitly in the block diagram, or programmatically using the `addPoint` command for `slLinearizer` or `slTuner` interfaces.

To mark an analysis point explicitly in the block diagram, right-click on the signal and use the **Linear Analysis Points** menu. Select one of the closed-loop analysis types, unless you also want to add a permanent opening at this location. The closed-loop analysis types are **Input Perturbation**, **Output Measurement**, **Sensitivity**, and

Complementary Sensitivity. The selected type does not affect analysis functions, such as `getIOTransfer`, and tuning goals, such as `TuningGoal.StepTracking`.
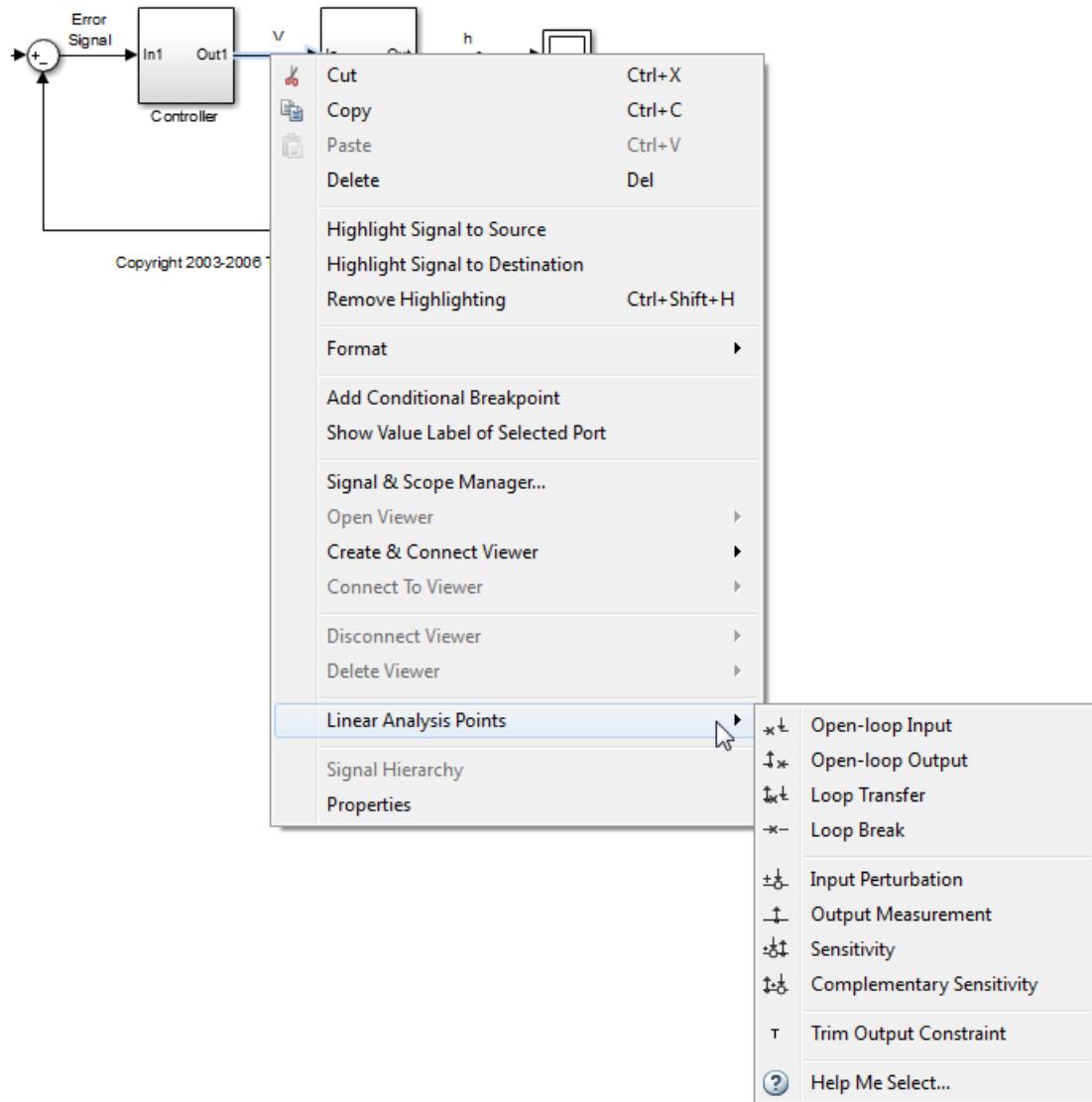
Figure 3: Marking Analysis Points in a Simulink Model

To mark analysis points programmatically, use `addPoint` for the `slLinearizer` or `slTuner` interfaces. Specify the point of interest using the block path, port number, and bus element, if applicable. For example, consider the `ex_scd_analysis_pts2` model, illustrated in the next two figures.
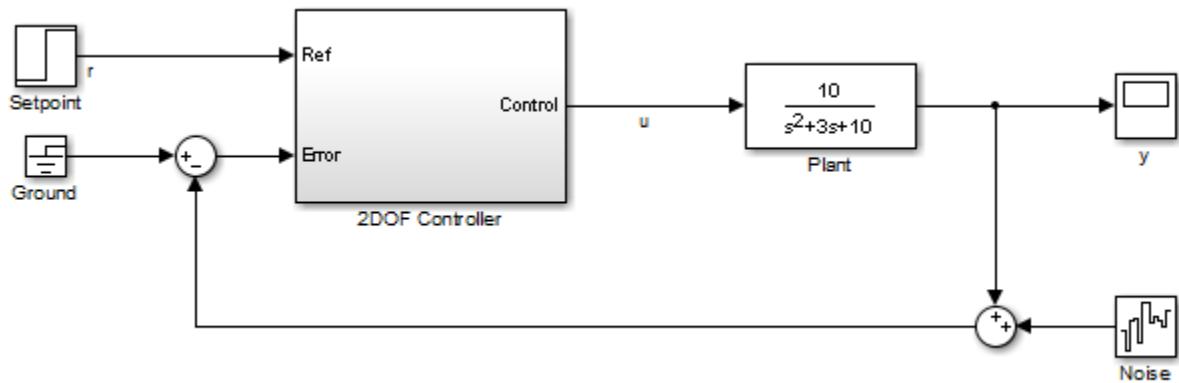
Figure 4: Simple Feedback Loop in Simulink
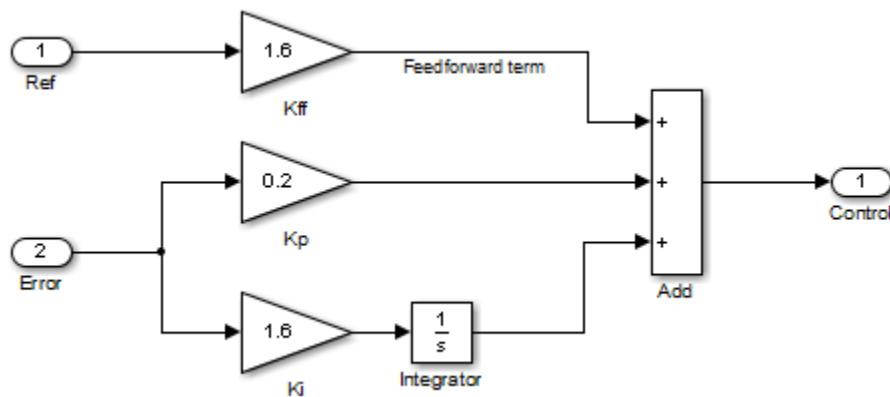
Figure 5: 2DOF Controller Subsystem

Mark the `u` and `Feedfordward term` signals as analysis points.

```
open_system('ex_scd_analysis_pts2');
```

```
ST = slLinearizer('ex_scd_analysis_pts2');
addPoint(ST,'ex_scd_analysis_pts2/2DOF Controller',1)
addPoint(ST,'ex_scd_analysis_pts2/2DOF Controller/Kff',1)
```

For convenience, you can also designate points of interest as analysis points using one of the abbreviations shown in the following examples:

- Signal name:

  ```
  addPoint(ST,{'u','r'})
  ```
- Block name and port number:

  ```
  addPoint(ST,'ex_scd_analysis_pts2/Plant/1')
  ```
- Block name and outport name:

  ```
  addPoint(ST,'ex_scd_analysis_pts2/2DOF Controller/Control')
  ```
- End of the full block name when unambiguous:

  ```
  addPoint(ST,'Controller/1')
  addPoint(ST,{'Setpoint','Noise'})
  ```

Finally, you can specify analysis points using linearization I/O objects (see `linio`):

```
ios = [...
   linio('ex_scd_analysis_pts2/Setpoint',1,'input'),...
   linio('ex_scd_analysis_pts2/Plant',1,'output')];
addPoint(ST,ios)
```

As when you use the **Linear Analysis Points** to mark analysis points, analysis functions such as `getIOTransfer` and tuning goals such as `TuningGoal.StepTracking` ignore the actual I/O type. However, an I/O type that implies a loop opening, for instance `loopbreak` or `openinput`, imposes a permanent loop opening at the point. This permanent opening remains in force throughout analysis and tuning.

When you specify response I/Os in a tool such as Linear Analysis Tool or Control System Tuner, the software creates analysis points as needed.

Referring to Analysis Points for Analysis and Tuning

Once you have marked analysis points, you can analyze the response at any of these points using functions such as `getIOTransfer` and `getLoopTransfer`. You can also

create tuning goals that constrain the system response at these points. The tools to perform these operations operate in a similar manner for models created at the command line and models created in Simulink.

Use the `getPoints` function to get a list of all available analysis points.

```
getPoints(T) % Model created at the command line
getPoints(ST) % Model created in Simulink®
```

For closed-loop models created at the command line, you can also use the model input and output names as inputs to functions such as `getIOTransfer`.

```
stepplot(getIOTransfer(T,'u','y'))
```

Similarly, you can use these names to compute open-loop responses or create tuning goals for `systune`.

```
L = getLoopTransfer(T,'u',-1);
```

```
R = TuningGoal.Margins('u',10,60);
```

Use the same method to refer to analysis points for models created in Simulink. In Simulink models, for convenience, that you can use any unambiguous abbreviation of the analysis point names returned by `getPoints`.

```
L = getLoopTransfer(ST,'u',-1);
```

```
stepplot(getIOTransfer(ST,'r','Plant'))
```

```
s = tf('s');
R = TuningGoal.Gain('Noise','Feedforw',1/(s+1));
```

Finally, if some analysis points are vector-valued signals or multichannel locations, you can use indices to select particular entries or channels. For example, suppose u is a two-entry vector in the model of Figure 2. You can compute the open-loop response of the second channel and measure the impact of a disturbance on the first channel, as shown here.

```
% Build closed-loop model of MIMO feedback loop
G = ss([-1 0.2;0 -2],[1 0;0.3 1],eye(2),0);
C = pid(0.2,0.5);
AP = AnalysisPoint('u',2);
T = feedback(G*AP*C,eye(2));
T.OutputName = 'y';
```

```
L = getLoopTransfer(T,'u(2)',-1);

stepplot(getIOTransfer(T,'u(1)','y'))
```

When you create tuning goals in Control System Tuner, the software creates analysis points as needed.

See Also
AnalysisPoint | getIOTransfer | getPoints

Related Examples
- "Control System with Multichannel Analysis Points" on page 2-84
- "Mark Analysis Points in Closed-Loop Models" on page 4-13

Model Arrays

What Are Model Arrays?

In many applications, it is useful to consider collections multiple model objects. For example, you may want to consider a model with a parameter that varies across a range of values, such as

```
sys1 = tf(1, [1 1 1]);
sys2 = tf(1, [1 1 2]);
sys3 = tf(1, [1 1 3]);
```

and so on. Model arrays are a convenient way to store and analyze such a collection. Model arrays are collections of multiple linear models, stored as elements in a single MATLAB array.

For all models collected in a single model array, the following attributes must be the same:

- The number of inputs and outputs
- The sample time `Ts`
- The time unit `TimeUnit`

Uses of Model Arrays

Uses of model arrays include:

- Representing and analyzing sensitivity to parameter variations
- Validating a controller design against several plant models
- Representing linear models arising from the linearization of a nonlinear system at several operating points
- Storing models obtained from several system identification experiments applied to one plant

Using model arrays, you can apply almost all of the basic model operations that work on single model objects to entire sets of models at once. Functions operate on arrays model by model, allowing you to manipulate an entire collection of models in a vectorized fashion. You can also use analysis functions such as `bode`, `nyquist`, and `step` to model

arrays to analyze multiple models simultaneously. You can access the individual models in the collection through MATLAB array indexing.

Visualizing Model Arrays
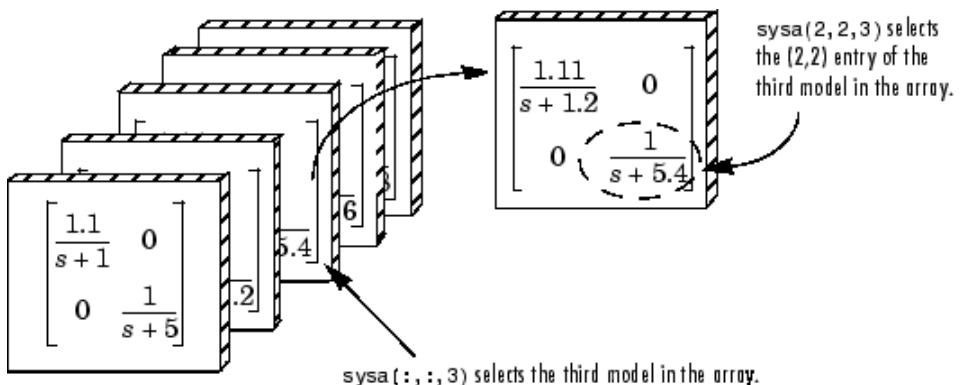
To visualize the concept of a model array, consider the set of five transfer function models shown below. In this example, each model has two inputs and two outputs. They differ by parameter variations in the individual model components.

$$\begin{bmatrix} \dfrac{1.1}{s+1} & 0 \\ 0 & \dfrac{1}{s+5} \end{bmatrix} \quad \begin{bmatrix} \dfrac{1.3}{s+1.1} & 0 \\ 0 & \dfrac{1}{s+5.2} \end{bmatrix} \quad \begin{bmatrix} \dfrac{1.11}{s+1.2} & 0 \\ 0 & \dfrac{1}{s+5.4} \end{bmatrix} \quad \begin{bmatrix} \dfrac{1.15}{s+1.3} & 0 \\ 0 & \dfrac{1}{s+5.6} \end{bmatrix} \quad \begin{bmatrix} \dfrac{1.09}{s+1.4} & 0 \\ 0 & \dfrac{1}{s+5.8} \end{bmatrix}$$
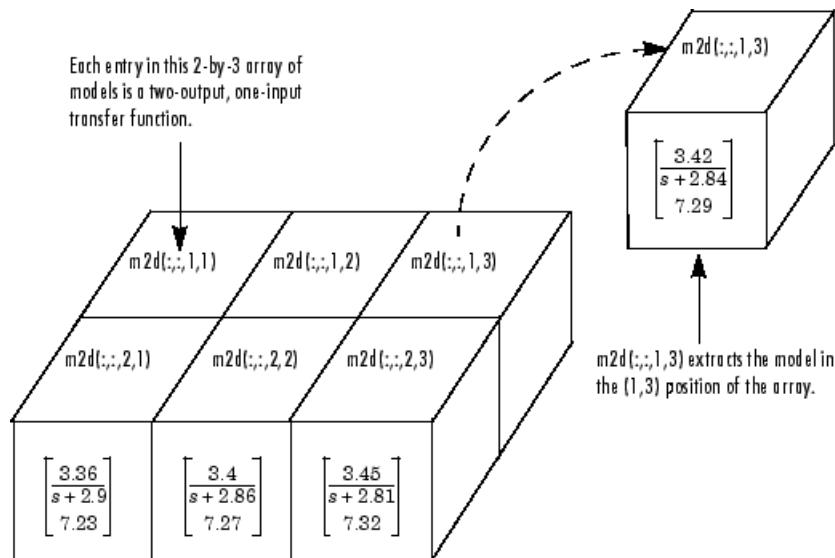
Just as you might collect a set of two-by-two matrices in a multidimensional array, you can collect this set of five transfer function models as a list in a model array under one variable name, say, `sys`. Each element of the model array is a single model object.

Visualizing Selection of Models From Model Arrays

The following illustration shows how indexing selects models from a one-dimensional model array. The illustration shows a 1-by-5 array `sysa` of 2-input, 2-output transfer functions.

sysa(2,2,3) selects the (2,2) entry of the third model in the array.

sysa(:,:,3) selects the third model in the array.

The following illustration shows selection of models from the two-dimensional model array m2d.

Related Examples

- "Query Array Size and Characteristics" on page 2-105
- "Select Models from Array" on page 2-102
- "Model Array with Variations in Two Parameters" on page 2-108

Model Array with Single Parameter Variation

This example shows how to create a one-dimensional array of transfer functions using the `stack` command. One parameter of the transfer function varies from model to model in the array. You can use such an array to investigate the effect of parameter variation on your model, such as for sensitivity analysis.

Create an array of transfer functions representing the following low-pass filter at three values of the roll-off frequency, a.

$$F(s) = \frac{a}{s+a}.$$

Create transfer function models representing the filter with roll-off frequency at $a = 3$, 5, and 7.

```
F1 = tf(3,[1 3]);
F2 = tf(5,[1 5]);
F3 = tf(7,[1 7]);
```

Use the `stack` command to build an array.

```
Farray = stack(1,F1,F2,F3);
```

The first argument to `stack` specifies the array dimension along which `stack` builds an array. The remaining arguments specify the models to arrange along that dimension. Thus, `Farray` is a 3-by-1 array of transfer functions.

Concatenating models with MATLAB® array concatenation commands, instead of with `stack`, creates multi-input, multi-output (MIMO) models rather than model arrays. For example:

```
G = [F1;F2;F3];
```

creates a one-input, three-output transfer function model, not a 3-by-1 array.

When working with a model array that represents parameter variations, You can associate the corresponding parameter value with each entry in the array. Set the `SamplingGrid` property to a data structure that contains the name of the parameter and the sampled parameter values corresponding with each model in the array. This assignment helps you keep track of which model corresponds to which parameter value.

```
Farray.SamplingGrid = struct('alpha',[3 5 7]);
Farray


Farray(:,:,1,1) [alpha=3] =

    3
  -----
  s + 3


Farray(:,:,2,1) [alpha=5] =

    5
  -----
  s + 5


Farray(:,:,3,1) [alpha=7] =

    7
  -----
  s + 7

3x1 array of continuous-time transfer functions.
```

The parameter values in Farray.SamplingGrid are displayed along with the each transfer function in the array.

Plot the frequency response of the array to examine the effect of parameter variation on the filter behavior.

```
bodeplot(Farray)
```

When you use analysis commands such as `bodeplot` on a model array, the resulting plot shows the response of each model in the array. Therefore, you can see the range of responses that results from the parameter variation.

Related Examples

- "Select Models from Array" on page 2-102

More About

- "Model Arrays" on page 2-96

Select Models from Array
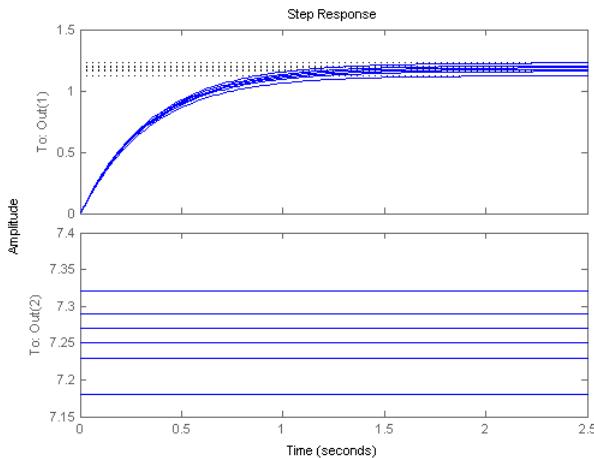
This example shows how to select individual models or sets of models from a model array using array indexing.

1 Load the transfer function array `m2d` into the MATLAB workspace.

```
load LTIexamples m2d
```

2 (Optional) Plot the step response of `m2d`.

```
step(m2d)
```


The step response shows that `m2d` contains six one-input, two-output models. The `step` command plots all of the models in an array on a single plot.

3 (Optional) Examine the dimensions of `m2d`.

```
arraydim = size(m2d)
```

This command produces the result:

```
arraydim =

     2     1     2     3
```

- The first entries of `arraydim`, 2 and 1, show that `m2d` is an array of two-output, one-input transfer functions.

- The remaining entries in `arraydim` give the array dimensions of `m2d`, 2-by-3.

In general, the dimensions of a model array are `[Ny,Nu,S1,...,Sk]`. `Ny` and `Nu` are the numbers of outputs and inputs of each model in the array. `S1,...,Sk` are the array dimensions. Thus, `Si` is the number of models along the ith array dimension.

4 Select the transfer function in the second row, first column of `m2d`.

To do so, use MATLAB array indexing.

```
sys = m2d(:,:,2,1)
```

Tip You can also access models using single index referencing of the array dimensions. For example,
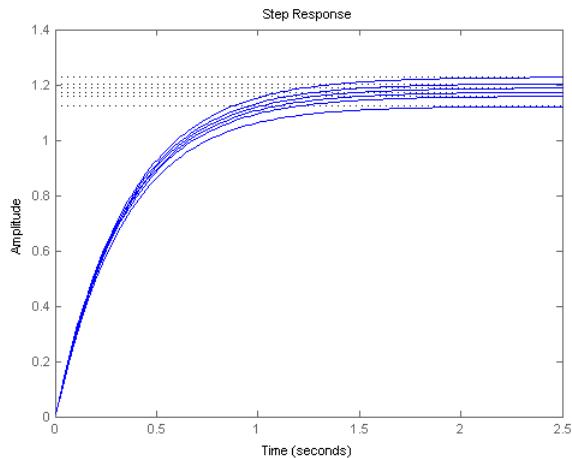
```
    sys = m2d(:,:,4)
```
selects the same model as `m2d(:,:,2,1)`.

---

5 Select the array of subsystems from the first input to the first output of each model in `m2d`.

```
m11 = m2d(1,1,:,:)
```

6 (Optional) Plot the step response of `m11`.

```
step(m11)
```

The step response shows that `m11` is an array of six single-input, single-output (SISO) models.

Note: For frequency response data (FRD) models, the array indices can be followed by the keyword `'frequency'` and some expression selecting a subset of the frequency points, as in:

```
sys(outputs,inputs,n1,...,nk,'frequency',SelectedFreqs)
```

Related Examples

- "Query Array Size and Characteristics" on page 2-105

More About

- "Model Arrays" on page 2-96

Query Array Size and Characteristics

This example shows how to query the size of model arrays, including the number of inputs and outputs of the models in the array, and the array dimensions. It also shows how to query characteristics of the models in the array, such as stability.

Array Size

Model arrays have two different sets of dimensions, the I/O dimensions and the array dimensions. The I/O dimensions are the numbers of inputs and outputs of the models in the array. (Each model in an array must have the same I/O dimensions.) The array dimensions are the dimensions of the array itself. Load a saved model array and query its dimensions.

```
load(fullfile(matlabroot,'examples','control','queryexample.mat'),'sysarr')
size(sysarr)

2x4 array of state-space models.
Each model has 3 outputs, 1 inputs, and 3 states.
```

When you use the `size` command on a model array with no output argument, the display shows the two sets of dimensions.

To obtain the array dimensions as a numeric array, use `size` with an output argument.

```
dims = size(sysarr)


dims =

     3     1     2     4
```

The first two entries in `dims` are the I/O dimensions of the models in `sysarr`, which each have three outputs and one input. The remaining entries in `dims` are the dimensions of the array itself. Thus, `sysarr` is a 2-by-4 array of models.

To query the number of dimensions in the array, rather than the values of those dimensions, use `ndims`.

```
dims = ndims(sysarr)


dims =
```

2-105

```
        4
```

In this case, `sysarr` has $4 = 2 + 2$ dimensions: The I/O dimensions (outputs and inputs), and the array dimensions. Query the I/O dimensions alone using the `iosize` command.

```
ios = iosize(sysarr)


ios =

     3     1
```

Query the total number of models in the array.

```
N = nmodels(sysarr)


N =

     8
```

Because `sysarr` is a 2-by-4 array of models, this command returns a value of $2 \times 4 = 8$.

Characteristics of Models in the Array

Query commands such as `isproper` and `isstable` work on model arrays. For example, query whether the models in `sysarr` are stable.

```
Bsiso = isstable(sysarr)


Bsiso =

     1
```

By default, `isstable` returns 1 (`true`) if all of the models in the array are stable. The commands returns 0 (`false`) if one or more of the models is not stable. To perform an element-by-element query of a model array, use the `'elem'` option.

```
Bsiso = isstable(sysarr,'elem')
```

```
Bsiso =

    1    1    1    1
    1    1    1    1
```

Now `isstable` returns an array of Boolean values. The dimensions of this array match the array dimensions of `sysarr`. Each entry in the array `Bsiso` indicates whether the corresponding model of `sysarr` is stable. The `'elem'` option works similarly for many query commands.

Related Examples

- "Select Models from Array" on page 2-102

More About

- "Model Arrays" on page 2-96

Model Array with Variations in Two Parameters

This example shows how to create a two-dimensional (2-D) array of transfer functions using `for` loops. One parameter of the transfer function varies in each dimension of the array.

You can use the technique of this example to create higher-dimensional arrays with variations of more parameters. Such arrays are useful for studying the effects of multiple-parameter variations on system response.

The second-order single-input, single-output (SISO) transfer function

$$H\left(s\right) = \frac{\omega^2}{s^2 + 2\zeta\omega s + \omega^2}.$$

depends on two parameters: the damping ratio, ζ, and the natural frequency, ω. If both ζ and ω vary, you obtain multiple transfer functions of the form:

$$H_{ij}\left(s\right) = \frac{\omega_j^2}{s^2 + 2\zeta_i\omega_j s + \omega_j^2},$$

where ζ_i and ω_j represent different measurements or sampled values of the variable parameters. You can collect all of these transfer functions in a single variable to create a two-dimensional model array.

Preallocate memory for the model array. Preallocating memory is an optional step that can enhance computation efficiency. To preallocate, create a model array of the required size and initialize its entries to zero.

```
H = tf(zeros(1,1,3,3));
```

In this example, there are three values for each parameter in the transfer function *H*. Therefore, this command creates a 3-by-3 array of single-input, single-output (SISO) zero transfer functions.

Create arrays containing the parameter values.

```
zeta = [0.66,0.71,0.75];
w = [1.0,1.2,1.5];
```

Build the array by looping through all combinations of parameter values.

```
for i = 1:length(zeta)
  for j = 1:length(w)
    H(:,:,i,j) = tf(w(j)^2,[1 2*zeta(i)*w(j) w(j)^2]);
  end
end
```

H is a 3-by-3 array of transfer functions. ζ varies as you move from model to model along a single column of H. The parameter ω varies as you move along a single row.

Plot the step response of H to see how the parameter variation affects the step response.

```
stepplot(H)
```

You can set the `SamplingGrid` property of the model array to help keep track of which set of parameter values corresponds to which entry in the array. To do so, create a grid of parameter values that matches the dimensions of the array. Then, assign these values to `H.SamplingGrid` with the parameter names.

```
[zetagrid,wgrid] = ndgrid(zeta,w);
H.SamplingGrid = struct('zeta',zetagrid,'w',wgrid);
```

When you display `H`, the parameter values in `H.SamplingGrid` are displayed along with the each transfer function in the array.

See Also
ndgrid

Related Examples
- "Study Parameter Variation by Sampling Tunable Model" on page 2-111

More About
- "Model Arrays" on page 2-96

Study Parameter Variation by Sampling Tunable Model

This example shows how to sample a parametric model of a second-order filter across a grid of parameter values using `sampleBlock`.

Consider the second-order filter represented by:

$$F\left(s\right) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}.$$

Sample this filter at varying values of the damping constant ζ and the natural frequency ω_n. Create a parametric model of the filter by using tunable elements for ζ and ω_n.

```
wn = realp('wn',3);
zeta = realp('zeta',0.8);
F = tf(wn^2,[1 2*zeta*wn wn^2])
```

```
F =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 2 states, and
    wn: Scalar parameter, 5 occurrences.
    zeta: Scalar parameter, 1 occurrences.

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

F is a `genss` model with two tunable Control Design Blocks, the `realp` blocks `wn` and `zeta`. The blocks `wn` and `zeta` have initial values of 3 and 0.8, respectively.

Sample F over a 2-by-3 grid of (`wn`, `zeta`) values.

```
wnvals = [3;5];
zetavals = [0.6 0.8 1.0];
Fsample = sampleBlock(F,'wn',wnvals,'zeta',zetavals);
```

Here, `sampleBlock` samples the model independently over the two ω_n values and three ζ values. Thus, Fsample is a 2-by-3 array of state-space models. Each entry in the array is a state-space model that represents F evaluated at the corresponding (`wn`, `zeta`) pair. For example, `Fsample(:,:,2,3)` has `wn` = 5 and `zeta` = 1.0.

Set the `SamplingGrid` property of the model array to help keep track of which set of parameter values corresponds to which entry in the array. To do so, create a grid of

parameter values that matches the dimensions of the array. Then, assign these values to `Fsample.SamplingGrid` in a structure with the parameter names.

```
[wngrid,zetagrid] = ndgrid(wnvals,zetavals);
Fsample.SamplingGrid = struct('wn',wngrid,'zeta',zetagrid);
```

The `ndgrid` command produces the full 2-by-3 grid of (`wn`, `zeta`) combinations. When you display `Fsample` in the command window, the parameter values in `Fsample.SamplingGrid` are displayed along with the each transfer function in the array. The parameter information is also available in response plots. For instance, examine the step response of `Fsample`.

```
stepplot(Fsample)
```

The step response plots show the variation in the natural frequency and damping constant across the six models in the array. When you click on one of the responses in the plot, the datatip includes the corresponding wn and zeta values as specified in Fsample.SamplingGrid.

Related Examples

·   "Select Models from Array" on page 2-102

More About

·   "Model Arrays" on page 2-96
·   "Models with Tunable Coefficients" on page 1-19

Linear Parameter-Varying Models

What are Linear Parameter-Varying Models?

A *linear parameter-varying* (LPV) system is a linear state-space model whose dynamics vary as a function of certain time-varying parameters called *scheduling parameters*. In MATLAB, an LPV model is represented in a state-space form using coefficients that are parameter dependent.

Mathematically, an LPV system is represented as:

$$dx(t) = A(p)x(t) + B(p)u(t)$$
$$y(t) = C(p)x(t) + D(p)u(t)$$
$$x(0) = x_0$$

where
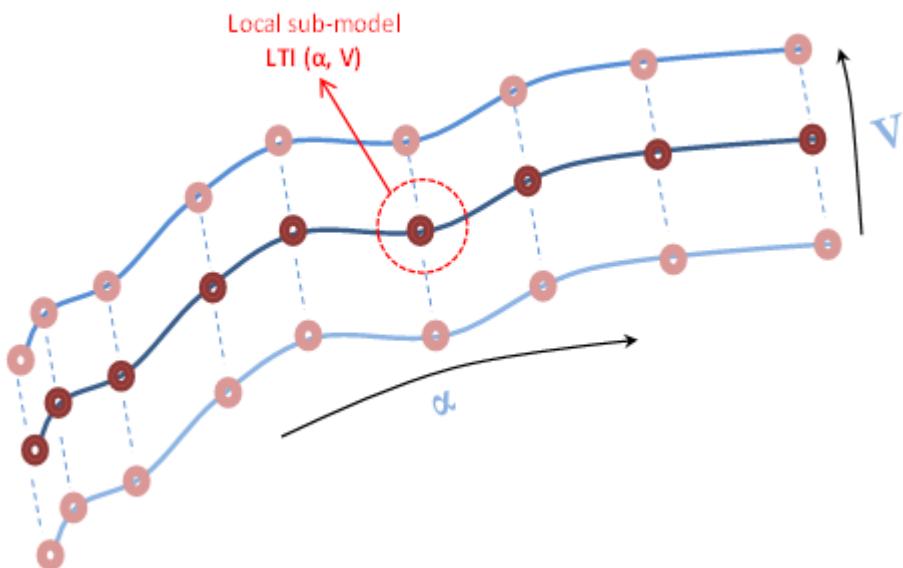
- $u(t)$ are the inputs
- $y(t)$ the outputs
- $x(t)$ are the model states with initial value $x0$
- $dx(t)$ is the state derivative vector $\dot{x}$ for continuous-time systems and the state update vector $x(t + \Delta T)$ for discrete-time systems. ΔT is the sample time.
- $A(p)$, $B(p)$, $C(p)$ and $D(p)$ are the state-space matrices parameterized by the scheduling parameter vector p.

- The parameters `p = p(t)` are measurable functions of the inputs and the states of the model. They can be a scalar quantity or a vector of several parameters. The set of scheduling parameters define the *scheduling space* over which the LPV model is defined.

Grid-Based LPV Model

A common way of representing LPV models is as an interpolated array of linear state-space models. A certain number of points in the scheduling space are selected, usually forming a regular grid. An LTI system is assigned to each point, representing the dynamics in the local vicinity of that point. The dynamics at scheduling locations in between the grid points is obtained by interpolation of LTI systems at neighboring points.

For example, the aerodynamic behavior of an aircraft is often scheduled over a grid of incidence angle (α) and wind speed (V) values. For each scheduling parameter, a range of values is chosen, such as α = 0:5:20 degrees, V = 700:100:1400 m/s. For each combination of (α,V) values, a linear approximation of the aircraft behavior is obtained. The local models are connected as shown in the following figure:

Each donut represents a local LTI model, and the connecting curves represent the interpolation rules. The abscissa and ordinate of the surface are the scheduling parameters (α, V).

This form is sometimes called the *grid-based LPV representation*. This is the form used by the `LPV System` block. For meaningful interpolations of system matrices, all the local models must use the same state basis.

Affine Form of LPV Model

The LPV system representation can be extended to allow offsets in `dx`, `x`, `u` and `y` variables. This form is known as *affine form* of the LPV model. Mathematically, the following represents an LPV system:

$$dx(t) = A(p)x(t) + B(p)u(t) + \left(\overline{dx}(p) - A(p)\overline{x}(p) - B(p)\overline{u}(p) \right)$$
$$y(t) = C(p)x(t) + D(p)u(t) + \left(\overline{y}(p) - C(p)\overline{x}(p) - D(p)\overline{u}(p) \right)$$
$$x(0) = x_0$$

$\overline{dx}(p),\ \overline{x}(p),\ \overline{u}(p),\ \overline{y}(p)$ are the offsets in the values of `dx(t)`, `x(t)`, `u(t)` and `y(t)` at a given parameter value `p = p(t)`.

You obtain such representations of the linear system array by linearizing a Simulink model over a batch of operating points (see "Batch Linearization" in Simulink Control Design documentation.) The offsets then correspond to the operating points at which you linearized the model.

In the affine representation, the linear model at a given point `p = p*` in the scheduling space is given by:

$$d\Delta x(t, p^*) = A\left(p^*\right)\Delta x\left(t, p^*\right) + B\left(p^*\right)\Delta u\left(t, p^*\right)$$
$$\Delta y\left(t, p^*\right) = C\left(p^*\right)\Delta x\left(t, p^*\right) + D\left(p^*\right)\Delta u\left(t, p^*\right)$$
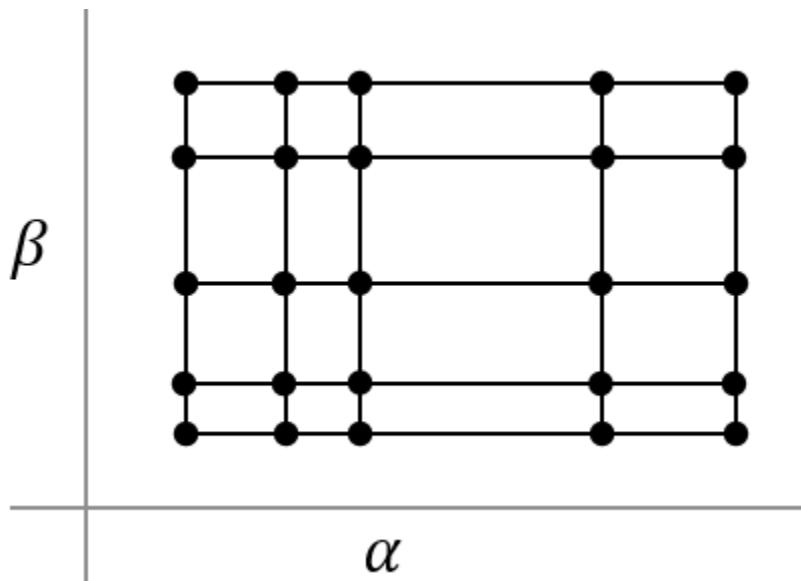
The states of this linear model are related to the states of the overall LPV model (Equation 2-2) by $\Delta x\left(t, p^*\right) = x(t) - \overline{x}\left(p^*\right)$. Similarly, $\Delta y\left(t, p^*\right) = y(t) - \overline{y}\left(p^*\right)$ and $\Delta u\left(t, p^*\right) = u(t) - \overline{u}\left(p^*\right)$.
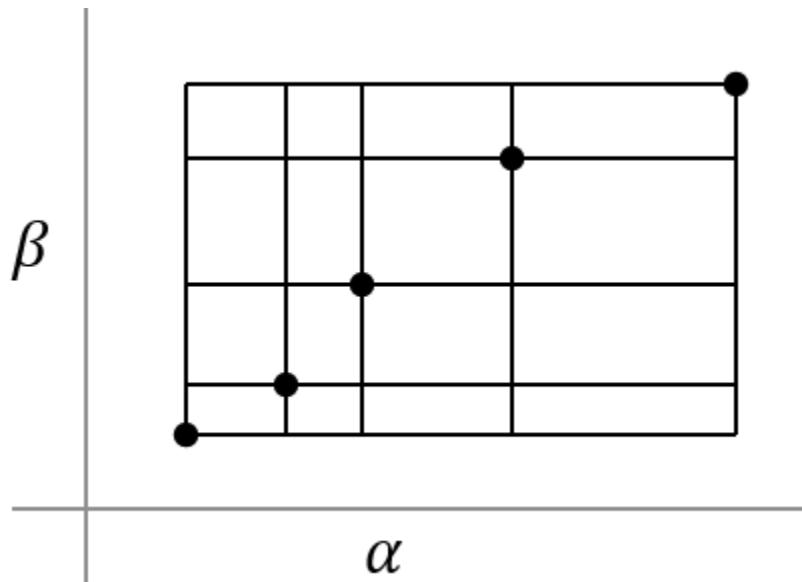
Regular vs. Irregular Grids
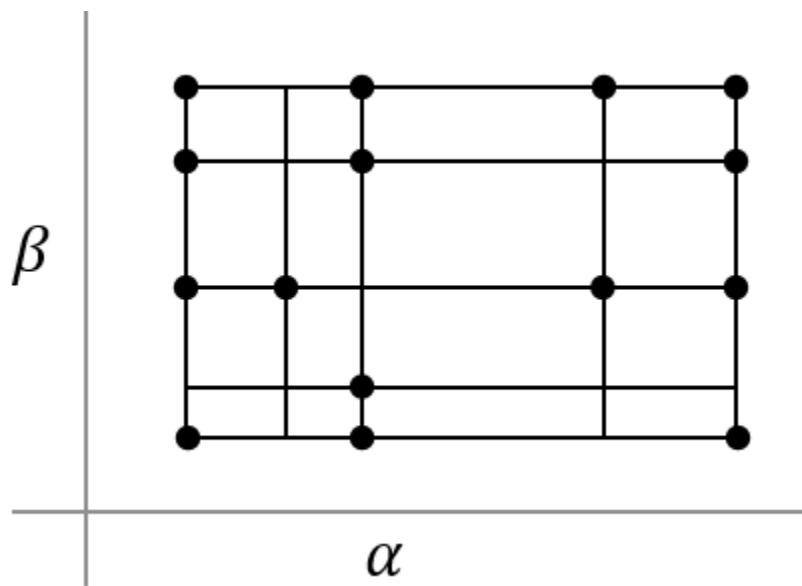
Consider a system that uses two scheduling parameters – α and β. When α and β vary monotonically, a regular grid is formed, as shown in the next figure. The state space array contains a value at every combination of α and β values. Regular grid does not imply uniform spacing between values.

When parameters co-vary, i.e., α and β increase together, an irregular grid is formed. The system array parameters are available only along the diagonal in the parameter plane.

If certain samples are missing from an otherwise regular grid, the grid is considered to be irregular.

Use Model Arrays to Create Linear Parameter-Varying Models

The array of state-consistent linear models that define an LPV model are represented by an array of state-space model objects. For more information on model arrays, see "Model Arrays".

The system array size is equal to the grid size in scheduling space. In the aircraft example, α takes 5 values in the 0–20 degrees range and V takes 8 values in the 700–1400 m/s range. If you define a linear model at every combination of (α,V) values (i.e., the grid is regular), the grid size is 5-by-8. Therefore, the model array size must be 5-by-8.

The information about scheduling parameters is attached to the linear model array using its `SamplingGrid` property. The value of the `SamplingGrid` property must be a structure with as many fields as there are scheduling parameters. For each field, the value must be set to all the values assumed by the corresponding variable in the scheduling space.

For the aircraft example, you can define the `SamplingGrid` property as:

```
Alpha = 0:5:20;
V = 700:100:1400;
[Alpha_Grid,V_Grid] = ndgrid(Alpha, V);
linsysArray.SamplingGrid = struct('Alpha',Alpha_Grid,'V',V_Grid);
```

Approximate Nonlinear Systems using LPV Models

In the same way as a linear model provides the approximation of system behavior at a given operating condition, an LPV model provides the approximation of the behavior over a span on operating conditions. A common approach for constructing the LPV model is by batch trimming and linearization, followed by stacking the local models in a state-space model array.

Note: When obtaining linear models by linearization, do not reduce or alter the state variables used by the models.

The operating region is usually of a high dimension because it consists of all the input and state variables. Generating or interpolating local models in such high-dimensional spaces is usually infeasible. A simpler approach is to use a small set of scheduling parameters as a proxy for the operating space variables. The scheduling parameters

are derived from the inputs and state variables of the original system. You must choose the values carefully so that for a fixed value of the scheduling parameters, the system behavior is approximately linear. This approach is not always possible.

Consider a nonlinear system described by the following equations:

$$\dot{x}_1 = x_1^2 + x_2^2$$
$$\dot{x}_2 = -2x_1 - 3x_2 + 2u$$
$$y = x_1 + 2$$

Suppose you use $p(t) = \dot{x}_1$ as a scheduling variable. At a given time instant t = t$_0$, you have:

$$\dot{x}_1 \approx 2x_1(t_0)x_1 + 2x_2(t_0)x_2 - \dot{x}_1(t_0)$$
$$\dot{x}_2 = -2x_1 - 3x_2 + 2u$$
$$y = x_1 + 2$$

Thus, the dynamics are linear (affine) in the neighborhood of a given value of $\dot{x}$. The approximation holds for all time spans and values of input u as long as of $\dot{x}$ does not deviate much from its nominal value at sampling point t$_0$. Note that scheduling on input u or states x$_1$ or x$_2$ does not help locally linearize the system. Therefore, they are not good candidates for scheduling parameters.

For an example of this approach, see "Approximating Nonlinear Behavior using an Array of LTI Systems".

Applications of Linear Parameter-Varying Models

Modeling Multimode Dynamics

You can use LPV models to represent systems that exhibit multiple modes (regimes) of operation. Examples of such systems include colliding bodies, systems controlled by operator switches, and approximations of systems affected by dry friction and hysteresis

effects. For an example, see "Using LTI Arrays for Simulating Multi-Mode Dynamics" on page 2-122.

Proxy Modeling for Faster Simulations

This approach is sometimes useful for generating surrogate models that you can use in place of the original system for enabling faster simulations, reducing memory footprint of target hardware code, and hardware-in-loop (HIL) simulations. You can also use surrogate models of this type for designing gain-scheduled controllers and for initializing the parameter estimation tasks in Simulink. For an example of approximating a general nonlinear system behavior by an LPV model, see "Approximating Nonlinear Behavior using an Array of LTI Systems".

LPV models can help speed up the simulation of physical component based systems, such as those built using Simscape™ Multibody™ and Simscape Power Systems™ software. For an example of this approach, see "LPV Approximation of a Boost Converter Model".

See Also
LPV System

Related Examples
- "Using LTI Arrays for Simulating Multi-Mode Dynamics" on page 2-122
- "Approximating Nonlinear Behavior using an Array of LTI Systems"
- "LPV Approximation of a Boost Converter Model"

Using LTI Arrays for Simulating Multi-Mode Dynamics

This example shows how to construct a Linear Parameter Varying (LPV) representation of a system that exhibits multi-mode dynamics.

Introduction

We often encounter situations where an elastic body collides with, or presses against, a possibly elastic surface. Examples of such situations are:

- An elastic ball bouncing on a hard surface.
- An engine throttle valve that is constrained to close to no more than 90° using a hard spring.
- A passenger sitting on a car seat made of polyurethane foam, a viscoelastic material.

In these situations, the motion of the moving body exhibits different dynamics when it is moving freely than when it is in contact with a surface. In the case of a bouncing ball, the motion of the mass can be described by rigid body dynamics when it is falling freely. When the ball collides and deforms while in contact with the surface, the dynamics have to take into account the elastic properties of the ball and of the surface. A simple way of modeling the impact dynamics is to use lumped mass spring-damper descriptions of the colliding bodies. By adjusting the relative stiffness and damping coefficients of the two bodies, we can model the various situations described above.

Modeling Bounce Dynamics

Figure 1 shows a mass-spring-damper model of the system. Mass 1 is falling freely under the influence of gravity. Its elastic properties are described by stiffness constant k_1 and damping coefficient c_1. When this mass hits the fixed surface, the impact causes Mass 1 and Mass 2 to move downwards together. After a certain "residence time" during which the Mass 1 deforms and recovers, it loses contact with Mass 2 completely to follow a projectile motion. The overall dynamics are thus broken into two distinct modes - when the masses are not in contact and when they are moving jointly.
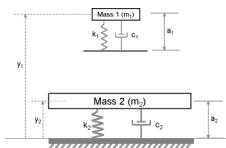
Figure 1: Elastic body bouncing on a fixed elastic surface.

The unstretched (load-free) length of spring attached to Mass 1 is a_1, while that of Mass 2 is a_2. The variables $y_1(t)$ and $y_2(t)$ denote the positions of the two masses. When the masses are not in contact ("Mode 1"), their motions are governed by the following equations:

$$\ddot{y}_1 = -g$$

$$m_2 \ddot{y}_2 + c_2 \dot{y}_2 + k_2(y_2 - a_2) = -m_2 g$$

with initial conditions $y_1(0) = h_1$, $\dot{y}_1(0) = 0$, $y_2(0) = h_2$, $\dot{y}_2(0) = 0$. h_1 is the height from which Mass 1 is originally dropped. $h_2 = a_2$ is the initial location of Mass 2 which corresponds to an unstretched state of its spring.

When Mass 1 touches Mass 2 ("Mode 2"), their displacements and velocities get interlinked. The governing equations in this mode are:

$$m_1 \ddot{y}_1 + c_1(\dot{y}_1 - \dot{y}_2) + k_1(y_1 - y_2 - a_1) = -m_1 g$$

$$m_2 \ddot{y}_2 + c_2 \dot{y}_2 + k_2(y_2 - a_2) - c_1(\dot{y}_1 - \dot{y}_2) - k_1(y_1 - y_2 - a_1) = -m_2 g$$

with $y_1(t_c) = y_2(t_c)$, where t_c is the time at which Mass 1 first touches Mass 2.

LPV Representation

The governing equations are linear and time invariant. However, there are two distinct behavioral modes corresponding to different equations of motion. Both modes are governed by sets of second order equations. If we pick the positions and velocities of the masses as state variables, we can represent each mode by a 4th order state-space equation.

In the state-space view, it becomes possible to treat the two modes as a single system whose coefficients change as a function of a certain condition which determines which mode is active. The condition is, of course, whether the two masses are moving freely or jointly. Such a representation, where the coefficients of a linear system are parameterized by an external but measurable parameter is called a Linear Parameter Varying (LPV) model. A common representation of an LPV model is by means of an array

of linear state-space models and a set of scheduling parameters that dictate the rules for choosing the correct model under a given condition. The array of linear models must all be defined using the same state variables.

For our example, we need two state-space models, one for each mode of operation. We also need to define a scheduling variable to switch between them. We begin by writing the above equations of motion in state-space form.

Define values of masses and their spring constants.

```matlab
m1 = 7;      % first mass (g)
k1 = 100;    % spring constant for first mass (g/s^2)
c1 = 2;      % damping coefficient associated with first mass (g/s)

m2 = 20;     % second mass (g)
k2 = 300;    % spring constant for second mass (g/s^2)
c2 = 5;      % damping coefficient associated with second mass (g/s)

g = 9.81;    % gravitational acceleration (m/s^2)

a1 = 12;     % uncompressed lengths of spring 1 (mm)
a2 = 20;     % uncompressed lengths of spring 2 (mm)

h1 = 100;    % initial height of mass m1 (mm)
h2 = a2;     % initial height of mass m2 (mm)
```

First mode: state-space representation of dynamics when the masses are not in contact.

```matlab
A11 = [0 1; 0 0];
B11 = [0; -g];
C11 = [1 0];
D11 = 0;

A12 = [0 1; -k2/m2, -c2/m2];
B12 = [0; -g+(k2*a2/m2)];
C12 = [1 0];
D12 = 0;

A1 = blkdiag(A11, A12);
B1 = [B11; B12];
C1 = blkdiag(C11, C12);
D1 = [D11; D12];

sys1 = ss(A1,B1,C1,D1);
```

Second mode: state-space representation of dynamics when the masses are in contact.

```
A2 = [ 0         1,        0,             0; ...
      -k1/m1,   -c1/m1,    k1/m1,         c1/m1;...
       0,        0,        0,             1; ...
       k1/m2,    c1/m2,    -(k1+k2)/m2,   -(c1+c2)/m2];

B2 = [0; -g+k1*a1/m1; 0; -g+(k2/m2*a2)-(k1/m2*a1)];
C2 = [1 0 0 0; 0 0 1 0];
D2 = [0;0];

sys2 = ss(A2,B2,C2,D2);
```

Now we stack the two models `sys1` and `sys2` together to create a state-space array.

```
sys = stack(1,sys1,sys2);
```

Use the information on whether the masses are moving freely or jointly for scheduling. Let us call this parameter "FreeMove" which takes the value of 1 when masses are moving freely and 0 when they are in contact and moving jointly. The scheduling parameter information is incorporated into the state-space array object (`sys`) by using its "SamplingGrid" property:

```
sys.SamplingGrid = struct('FreeMove',[1; 0]);
```
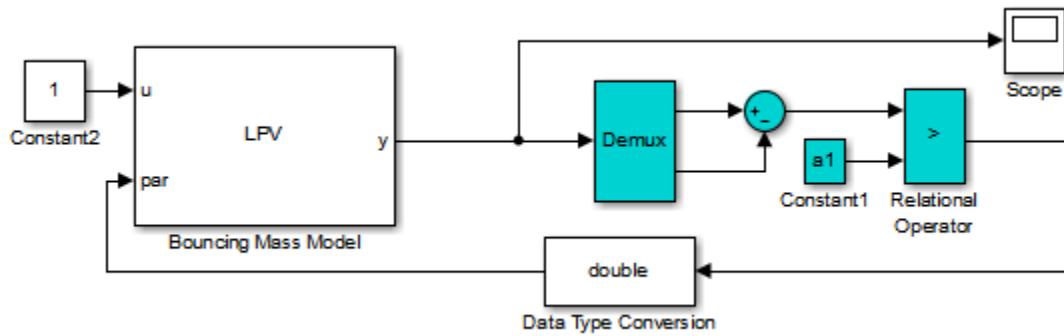
Whether the masses are in contact or not is decided by the relative positions of the two masses; when $y_1 - y_2 > a_1$, the masses are not in contact.

Simulation of LPV Model in Simulink

The state-space array `sys` has the necessary information to represent an LPV model. We can simulate this model in Simulink using the "LPV System" block from the Control System Toolbox™'s block library.

Open the preconfigured Simulink model `LPVBouncingMass.slx`

```
open_system('LPVBouncingMass')
open_system('LPVBouncingMass/Bouncing Mass Model','mask')
```

2-125

The block called "Bouncing Mass Model" is an LPV System block. Its parameters are specified as follows:
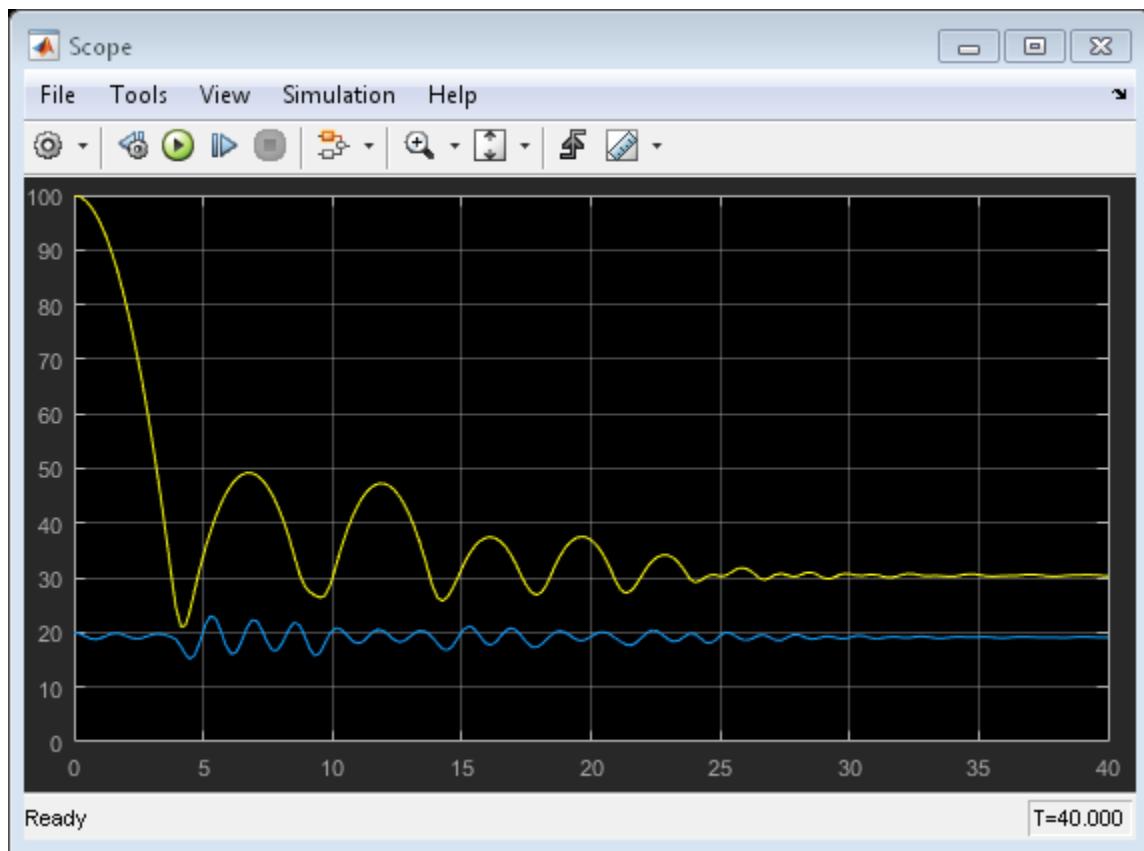
- For "State-space array" field, specify the state-space model array `sys` that was created above.
- For "Initial state" field, specify the initial positions and velocities of the two masses. Note that the state vector is: $[y_1, \dot{y}_1, y_2, \dot{y}_2]$. Specify its value as [h1 0 h2 0]'.
- Under the "Scheduling" tab, set the "Interpolation method" to "Nearest". This choice causes only one of the two models in the array to be active at any time. In our example, the behavior modes are mutually exclusive.
- Under the "Outputs" tab, uncheck all the checkboxes for optional output ports. We will be observing only the positions of the two masses.

The constant block outputs a unit value. This serves as the input to the model and is supplied from the first input port of the LPV block. The block has only one output port which outputs the positions of the two masses as a 2-by-1 vector.

The second input port of the LPV block is for specifying the scheduling signal. As discussed before, this signal represents the scheduling parameter "FreeMove" and takes discrete values 0 (masses in contact) or 1 (masses not in contact). The value of this parameter is computed as a function of the block's output signal. This computation is performed by the blocks with cyan background color. We take the difference between the two outputs (after demuxing) and compare the result to the unstretched length of spring attached to Mass 1. The resulting Boolean result is converted into a double signal which serves as the scheduling parameter value.

We are now ready to perform the simulation.

```
open_system('LPVBouncingMass/Scope')
sim('LPVBouncingMass')
```


The yellow curve shows the position of Mass 1 while the magenta curve shows the position of Mass 2. At the start of simulation, Mass 1 undergoes free fall until it hits Mass 2. The collision causes the Mass 2 to be displaced but it recoils quickly and bounces Mass 1 back. The two masses are in contact for the time duration where $y_1 - y_2 < a_1$. When the masses settle down, their equilibrium values are determined by the static settling due to gravity. For example, the absolute location of Mass 1 is $a_1 + a_2 - m1 * g/k1 - (m2 + m1) * g/k2 = 30.43mm.$

Conclusions

This example shows how a Linear Parameter Varying model can be constructed by using an array of state-space models and suitable scheduling variables. The example describes the case of mutually exclusive modes, although a similar approach can be used in cases where the dynamics behavior at a given value of scheduling parameters is influenced by several linear models.

The LPV System block facilitates the simulation of parameter varying systems. The block also supports code generation for various hardware targets.

Working with Linear Models

Data Manipulation

Store and Retrieve Model Data

In this section...

"Model Properties" on page 3-2

"Specify Model Properties at Model Creation" on page 3-2

"Examine and Change Properties of an Existing Model" on page 3-3

Model Properties

Model properties are the data fields that store all data about a dynamic system model. Data stored in model properties includes model dynamics, such as transfer-function coefficients, state-space matrices, and time delays. Model properties also let you specify other model attributes such as sample time, channel names, and state names.

For information about the properties associated with each model type, see the corresponding reference page, such as `tf`, `pid`, or `ss`.

Specify Model Properties at Model Creation

When you create a dynamic system model, the software sets all property values. Properties that contain model dynamics are automatically set with the appropriate values. Other properties are set to default values. (See model reference pages for information about default property values.)

You can specify other values for model properties at model creation using the `Name,Value` pair syntax of the model-creation command. In this syntax, you specify the name of of the property you want to set, followed by the value. You can set multiple property values in one command. For example, assign a transport delay and input and output names to a new transfer function model.

```
H = tf(1,[1 10],'ioDelay',6.5,'InputName','torque','OutputName','velocity')


H =

  From input "torque" to output "velocity":
                  1
  exp(-6.5*s) * ------
                s + 10

Continuous-time transfer function.
```

Some property values are reflected in the model display, such as the input and output names. You can use `Name,Value` pair syntax when creating any type of model.

Examine and Change Properties of an Existing Model

Load an existing state-space (ss) model.

```
load(fullfile(matlabroot,'examples','control','PadeApproximation1.mat'),'sys')
sys
```

```
sys =

  A =
         x1     x2
   x1  -1.5   -0.1
   x2     1      0

  B =
       u1
   x1   1
   x2   0

  C =
        x1    x2
   y1  0.5   0.1

  D =
       u1
   y1   0

  (values computed with all internal delays set to zero)

  Output delays (seconds): 1.5
  Internal delays (seconds): 3.4

Continuous-time state-space model.
```

The display shows that `sys` is a state-space model, and includes some of the property values of `sys`. To see all properties of `sys`, use the `get` command.

```
get(sys)
```

```
              A: [2x2 double]
              B: [2x1 double]
              C: [0.5000 0.1000]
              D: 0
              E: []
         Scaled: 0
      StateName: {2x1 cell}
      StateUnit: {2x1 cell}
  InternalDelay: 3.4000
      InputDelay: 0
    OutputDelay: 1.5000
             Ts: 0
       TimeUnit: 'seconds'
      InputName: {''}
      InputUnit: {''}
     InputGroup: [1x1 struct]
     OutputName: {''}
     OutputUnit: {''}
    OutputGroup: [1x1 struct]
           Name: ''
          Notes: {}
       UserData: []
   SamplingGrid: [1x1 struct]
```

Use dot notation to access the values of particular properties. For example, display the A matrix of `sys`.

```
Amat = sys.A
```

```
Amat =

   -1.5000   -0.1000
    1.0000         0
```

Dot notation also lets you change the value of individual model properties.

```
sys.InputDelay = 4.2;
sys.InputName = 'thrust';
sys.OutputName = 'velocity';
```

When you must change multiple property values at the same time to preserve the validity of the model, such as changing the dimensions of the state-space matrices, you

can use the set command. For example, create a 1-state state-space model, and then replace the matrices with new values representing a 2-state model.

```
sys2 = rss(1);
Anew = [-2, 1; 0.5 0];
Bnew = [1; -1];
Cnew = [0, -0.4];
set(sys2,'A',Anew,'B',Bnew,'C',Cnew)
sys2


sys2 =

  A =
        x1    x2
   x1   -2     1
   x2   0.5    0

  B =
        u1
   x1    1
   x2   -1

  C =
        x1     x2
   y1    0   -0.4

  D =
           u1
   y1   0.3426

Continuous-time state-space model.
```

Changing certain properties, such as Ts or TimeUnit, can cause undesirable changes in system behavior. See the property descriptions in the model reference pages for more information.

Related Examples

- "Attach Metadata to Models" on page 3-9
- "Extract Model Coefficients" on page 3-6

Extract Model Coefficients

In this section...

Functions for Extracting Model Coefficients

Control System Toolbox software includes several commands for extracting model coefficients such as transfer function numerator and denominator coefficients, state-space matrices, and proportional-integral-derivative (PID) gains.

The following commands are available for data extraction.

Command	Result
tfdata	Extract transfer function coefficients
zpkdata	Extract zero and pole locations and system gain
ssdata	Extract state-space matrices
dssdata	Extract descriptor state-space matrices
frdata	Extract frequency response data from frd model
piddata	Extract parallel-form PID data
pidstddata	Extract standard-form PID data
get	Access all model property values

Extracting Coefficients of Different Model Type

When you use a data extraction command on a model of a different type, the software computes the coefficients of the target model type. For example, if you use zpkdata on a ss model, the software converts the model to zpk form and returns the zero and pole locations and system gain.

Extract Numeric Model Data and Time Delay

This example shows how to extract transfer function numerator and denominator coefficients using `tfdata`.

1   Create a first-order plus dead time transfer function model.

```
H = exp(-2.5*s)/(s+12);
```

2   Extract the numerator and denominator coefficients.

```
[num,den] = tfdata(H,'v')
```

The variables `num` and `den` are numerical arrays. Without the `'v'` flag, `tfdata` returns cell arrays.

Note: For SISO transfer function models, you can also extract coefficients using:

```
num = H.num{1};
den = H.den{1};
```

---

3   Extract the time delay.

   a   Determine which property of `H` contains the time delay.

In a SISO `tf` model, you can express a time delay as an input delay, an output delay, or a transport delay (I/O delay).

```
get(H)

        num: {[O 1]}
        den: {[1 12]}
   Variable: 's'
    ioDelay: O
 InputDelay: O
OutputDelay: 2.5000
         Ts: O
   TimeUnit: 'seconds'
  InputName: {''}
  InputUnit: {''}
 InputGroup: [1x1 struct]
 OutputName: {''}
 OutputUnit: {''}
OutputGroup: [1x1 struct]
```

```
            Name: ''
           Notes: {}
        UserData: []
```

The time delay is stored in the `OutputDelay` property.

b Extract the output delay.

```
delay = H.OutputDelay;
```

Extract PID Gains from Transfer Function

This example shows how to extract PID (proportional-integral-derivative) gains from a transfer function using `piddata`. You can use the same steps to extract PID gains from a model of any type that represents a PID controller, using `piddata` or `pidstddata`.

1 Create a transfer function that represents a PID controller with a first-order filter on the derivative term.

```
Czpk = zpk([-6.6,-0.7],[0,-2],0.2)
```

2 Obtain the PID gains and filter constant.

```
[Kp,Ki,Kd,Tf] = piddata(Czpk)
```

This command returns the proportional gain `Kp`, integral gain `Ki`, derivative gain `Kd`, and derivative filter time constant `Tf`. Because `piddata` automatically computes the PID controller parameters, you can extract the PID coefficients without creating a `pid` model.

Related Examples

·   "Attach Metadata to Models" on page 3-9

More About

·   "Store and Retrieve Model Data" on page 3-2

Attach Metadata to Models

In this section...
"Specify Model Time Units" on page 3-9
"Interconnect Models with Different Time Units" on page 3-9
"Specify Frequency Units of Frequency-Response Data Model" on page 3-10
"Extract Subsystems of Multi-Input, Multi-Output (MIMO) Models" on page 3-11
"Specify and Select Input and Output Groups" on page 3-12

Specify Model Time Units

This example shows how to specify time units of a transfer function model.

The `TimeUnit` property of the `tf` model object specifies units of the time variable, time delays (for continuous-time models), and the sample time Ts (for discrete-time models). The default time units is `seconds`.

Create a SISO transfer function model $sys = \dfrac{4s + 2}{s^2 + 3s + 10}$ with time units in milliseconds:

```
num = [4 2];
den = [1 3 10];
sys = tf(num,den,'TimeUnit','milliseconds');
```

You can specify the time units of any dynamic system in a similar way.

The system time units appear on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system are used if the time and frequency units in the "Toolbox Preferences Editor" on page 17-2 are `auto`.

Note: Changing the `TimeUnit` property changes the system behavior. If you want to use different time units without modifying system behavior, use `chgTimeUnit`.

Interconnect Models with Different Time Units

This example shows how to interconnect transfer function models with different time units.

To interconnect models using arithmetic operations or interconnection commands, the time units of all models must match.

1 Create two transfer function models with time units of milliseconds and seconds, respectively.

```
sys1 = tf([1 2],[1 2 3],'TimeUnit','milliseconds');
sys2 = tf([4 2],[1 3 10]);
```

2 Change the time units of `sys2` to milliseconds.

```
sys2 = chgTimeUnit(sys2,'milliseconds');
```

3 Connect the systems in parallel.

```
sys = sys1+sys2;
```

Specify Frequency Units of Frequency-Response Data Model

This example shows how to specify units of the frequency points of a frequency-response data model.

The `FrequencyUnit` property specifies units of the frequency vector in the `Frequency` property of the `frd` model object. The default frequency units are `rad/TimeUnit`, where `TimeUnit` is the time unit specified in the `TimeUnit` property.

Create a SISO frequency-response data model with frequency data in GHz.

```
load AnalyzerData;
sys = frd(resp,freq,'FrequencyUnit','GHz');
```

You can independently specify the units in which you measure the frequency points and sample time in the `FrequencyUnit` and `TimeUnit` properties, respectively. You can also specify the frequency units of a `genfrd` in a similar way.

The frequency units appear on the frequency-domain plots. For multiple systems with different frequency units, the units of the first system are used if the frequency units in the "Toolbox Preferences Editor" on page 17-2 is `auto`.

Note: Changing the `FrequencyUnit` property changes the system behavior. If you want to use different frequency units without modifying system behavior, use `chgFreqUnit`.

Extract Subsystems of Multi-Input, Multi-Output (MIMO) Models

This example shows how to extract subsystems of a MIMO model using MATLAB indexing and using channel names.

Extracting subsystems is useful when, for example, you want to analyze a portion of a complex system.

Create a MIMO transfer function.

```
G1 = tf(3,[1 10]);
G2 = tf([1 2],[1 0]);
G = [G1,G2];
```

Extract the subsystem of G from the first input to all outputs.

```
Gsub = G(:,1);
```

This command uses MATLAB indexing to specify a subsystem as G(out,in), where out specifies the output indices and in specifies the input indices.

Using channel names, you can use MATLAB indexing to extract all the dynamics relating to a particular channel. By using this approach, you can avoid having to keep track of channel order in a complex MIMO model.

Assign names to the model inputs.

```
G.InputName = {'temperature';'pressure'};
```

Because G has two inputs, use a cell array of two strings to specify the channel names.

Extract the subsystem of G that contains all dynamics from the 'temperature' input to all outputs.

```
Gt = G(:,'temperature');
```

Gt is the same subsystem as Gsub.

Note: When you extract a subsystem from a state-space (ss) model, the resulting state-space model may not be minimal. Use sminreal to eliminate unnecessary states in the subsystem.

Specify and Select Input and Output Groups

This example shows how to specify groups of input and output channels in a model object and extract subsystems using the groups.

Input and output groups are useful for keeping track of inputs and outputs in complex MIMO models.

1 Create a state-space model with three inputs and four outputs.

```
H = rss(3,4,3);
```

2 Group the inputs as follows:

- Inputs 1 and 2 in a group named `controls`
- Outputs 1 and 3 to a group named `temperature`
- Outputs 1, 3, and 4 to a group named `measurements`

```
H.InputGroup.controls = [1 2];
H.OutputGroup.temperature = [1 3];
H.OutputGroup.measurements = [1 3 4];
```

`InputGroup` and `OutputGroup` are structures. The name of each field in the structure is the name of the input or output group. The value of each field is a vector that identifies the channels in that group.
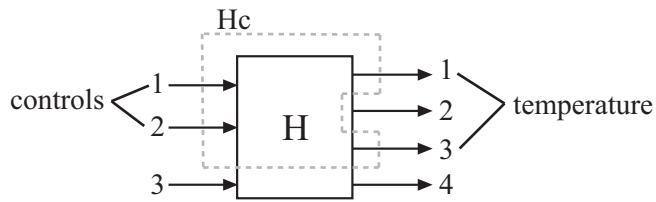
3 Extract the subsystem corresponding to the `controls` inputs and the `temperature` outputs.

You can use group names to index into subsystems.

```
Hc = H('temperature','controls')
```

`Hc` is a two-input, two-output `ss` model containing the I/O channels from the `'controls'` input to the `'temperature'` outputs.

You can see the relationship between `H` and the subsystem `Hc` in this illustration.

Related Examples

- "Store and Retrieve Model Data" on page 3-2
- "Extract Model Coefficients" on page 3-6
- "Query Model Characteristics" on page 3-14

Query Model Characteristics

This example shows how to query model characteristics such as stability, time domain, and number of inputs and outputs. You can use the techniques of this example on any type of dynamic system model.

Load a saved state-space (ss) model.

```
load(fullfile(matlabroot,'examples','control','queryexample.mat'),'T')
```

Query whether T has stable dynamics.

```
Bstab = isstable(T)
```

```
Bstab =

     1
```

The `isstable` command returns 1 (`true`) if all system poles are in the open left-half plane (for continuous-time models) or inside the open unit disk (for discrete-time models). Otherwise, `isstable` command returns 0 (`false`). Here, the result shows that the model is stable.

Query whether T has time delays.

```
Bdel = hasdelay(T)
```

```
Bdel =

     1
```

The returned value, 1, indicates that T has a time delay. For a state-space model, time delay can be stored as input delay, output delay, internal delay, or a combination. Use `get(T)` to determine which properties of T hold the time delay, and use dot notation to access the delay values. The `hasInternalDelay` command tells you whether there is any internal delay.

Query whether T is proper.

```
Bprop = isproper(T)
```

```
Bprop =

     1
```

The returned value indicates that the system has relative degree less than or equal to 0. This is true of a SISO system when it can be represented as a transfer function in which the degree of the numerator does not exceed the degree of the denominator.

Query the order of T.

```
N = order(T)
```

```
N =

     5
```

For a state-space model, `order` returns the number of states, which is 5 in this case. For a `tf` or `zpk` model, the order is the number of states required for a state-space realization of the system.

Query whether T is a discrete-time system.

```
Bdisc = isdt(T)
```

```
Bdisc =

     1
```

The returned value indicates that T is a discrete-time model. Similarly, use `isct` to query whether T is a continuous-time model.

Load a MIMO model and query the input/output dimensions.

```
load(fullfile(matlabroot,'examples','control','queryexample.mat'),'Tmimo')
ios = iosize(Tmimo)
```

```
ios =
```

              7   4

In the resulting array, the number of outputs is first. Therefore, `Tmimo` has 4 inputs and 7 outputs.

See Also
`isproper | isstable | size`

Related Examples
· "Select Models from Array" on page 2-102

More About
· "Store and Retrieve Model Data" on page 3-2

Customize Model Display

In this section...

Configure Transfer Function Display Variable

This example shows how to configure the MATLAB command-window display of transfer function (`tf`) models.

You can use the same steps to configure the display variable of transfer function models in factorized form (`zpk` models).

By default, `tf` and `zpk` models are displayed in terms of `s` in continuous time and `z` in discrete time. Use the `Variable` property change the display variable to `'p'` (equivalent to `'s'`), `'q'` (equivalent to `'z'`), `'z^-1'`, or `'q^-1'`.

1 Create the discrete-time transfer function $H(z) = \dfrac{z-1}{z^2 - 3z + 2}$

with a sample time of 1 s.

```
 H = tf([1 -1],[1 -3 2],0.1)

H =

      z - 1
  -------------
  z^2 - 3 z + 2

Sample time: 0.1 seconds
Discrete-time transfer function.
```

The default display variable is `z`.

2 Change the display variable to `q^-1`.

```
H.Variable = 'q^-1'

H =
```

```
      q^-1 - q^-2
   ------------------
   1 - 3 q^-1 + 2 q^-2

Sample time: 0.1 seconds
Discrete-time transfer function.
```

When you change the `Variable` property, the software computes new coefficients and displays the transfer function in terms of the new variable. The `num` and `den` properties are automatically updated with the new coefficients.

Tip Alternatively, you can directly create the same transfer function expressed in terms of `'q^-1'`.

```
H = tf([0 1 -1],[1 -3 2],0.1,'Variable','q^-1');
```

For the inverse variables `'z^-1'` and `'q^-1'`, `tf` interprets the numerator and denominator arrays as coefficients of ascending powers of `'z^-1'` or `'q^-1'`.

Configure Display Format of Transfer Function in Factorized Form

This example shows how to configure the display of transfer function models in factorized form (`zpk` models).

You can configure the display of the factorized numerator and denominator polynomials to highlight:

- The numerator and denominator roots
- The natural frequencies and damping ratios of each root
- The time constants and damping ratios of each root

See the `DisplayFormat` property on the `zpk` reference page for more information about these quantities.

1 Create a `zpk` model having a zero at $s = 5$, a pole at $s = -10$, and a pair of complex poles at $s = -3 \pm 5i$.

```
H = zpk(5,[-10,-3-5*i,-3+5*i],10)

H =
```

```
       10 (s-5)
  ---------------------
  (s+10) (s^2 + 6s + 34)

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

The default display format, `'roots'`, displays the standard factorization of the numerator and denominator polynomials.

2 Configure the display format to display the natural frequency of each polynomial root.

```
 H.DisplayFormat = 'frequency'

H =

              -0.14706 (1-s/5)
  -------------------------------------------
  (1+s/10) (1 + 1.029(s/5.831) + (s/5.831)^2)

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

You can read the natural frequencies and damping ratios for each pole and zero from the display as follows:

- Factors corresponding to real roots are displayed as $(1 - s/\omega_0)$. The variable ω_0 is the natural frequency of the root. For example, the natural frequency of the zero of H is 5.

- Factors corresponding to complex pairs of roots are displayed as $1 - 2\zeta(s/\omega_0) + (s/\omega_0)^2$. The variable ω_0 is the natural frequency, and ζ is the damping ratio of the root. For example, the natural frequency of the complex pole pair is 5.831, and the damping ratio is 1.029/2.

3 Configure the display format to display the time constant of each pole and zero.

```
H.DisplayFormat = 'time constant'

H =

              -0.14706 (1-0.2s)
  -------------------------------------------
  (1+0.1s) (1 + 1.029(0.1715s) + (0.1715s)^2)
```

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

You can read the time constants and damping ratios from the display as follows:

- Factors corresponding to real roots are displayed as (τs). The variable τ is the time constant of the root. For example, the time constant of the zero of H is 0.2.

- Factors corresponding to complex pairs of roots are displayed as $1 - 2\zeta(\tau s) + (\tau s)^2$. The variable τ is the time constant, and ζ is the damping ratio of the root. For example, the time constant of the complex pole pair is 0.1715, and the damping ratio is 1.029/2.

See Also
tf | zpk

Related Examples
- "Transfer Functions" on page 2-3

4
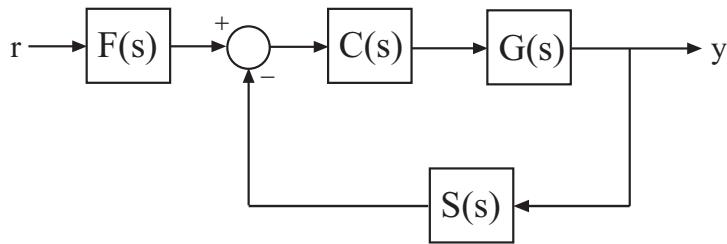
Model Interconnections

Why Interconnect Models?

Interconnecting models of components allows you to construct models of control systems. You can conceptualize your control system as a block diagram containing multiple interconnected components, such as a plant or a controller. Using model arithmetic or interconnection commands, you combine models of each of these components into a single model representing the entire block diagram.

For example, you can interconnect dynamic system models of a plant $G(s)$, a controller $C(s)$, sensor dynamics $S(s)$, and a filter $F(s)$ to construct a single model that represents the entire closed-loop control system in the following illustration:

More About

- "Catalog of Model Interconnections" on page 4-3

Catalog of Model Interconnections
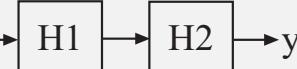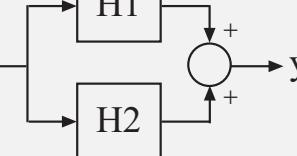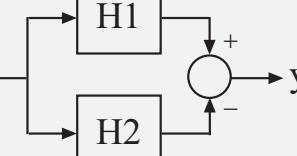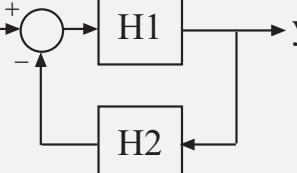
Each type of block diagram connection corresponds to a model interconnection command or arithmetic expression. The following tables summarize the block diagram connections with the corresponding interconnection command and arithmetic expression.
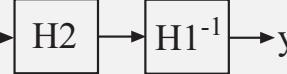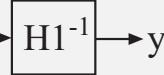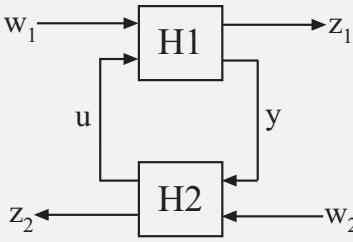
In this section...
"Model Interconnection Commands" on page 4-3
"Arithmetic Operations" on page 4-4

Model Interconnection Commands

Block Diagram Connection	Command	Arithmetic Expression
	`series(H1,H2)`	`H2*H1`
	`parallel(H1,H2)`	`H1+H2`
	`parallel(H1,-H2)`	`H1-H2`
	`feedback(H1,H2)`	`H1/(1+H2*H1)` (not recommended)

Block Diagram Connection	Command	Arithmetic Expression
	N/A	H1/H2 (division)
	N/A	H1\H2 (left division)
	inv(H1)	N/A
	lft(H1,H2,nu,ny)	N/A

Arithmetic Operations

You can apply almost all arithmetic operations to dynamic system models, including those shown below.

Operation	Description
+	Addition
-	Subtraction
*	Multiplication
.*	Element-by-element multiplication
/	Right matrix divide
\	Left matrix divide
inv	Matrix inversion
'	Conjugate system. For a system G, the transfer function of G' is:
	• $G(-s)^T$ in continuous time.

Operation	Description
	• $G(1/z)^T$ in discrete time.
.'	Transposition

^	Powers of a dynamic system model, as in the following syntax for creating transfer functions:
	
	`s = tf('s');`
	`G = 25/(s^2 + 10*s + 25);`

In some cases, you might obtain better results using model interconnection commands, such as `feedback` or `connect`, instead of model arithmetic. For example, the command `T = feedback(H1,H2)` returns better results than the algebraic expression `T = H1/(1+H2*H1)`. The latter expression duplicates the poles of H1, which inflates the model order and might lead to computational inaccuracy.

See Also
`connect` | `feedback` | `parallel` | `series`

Related Examples
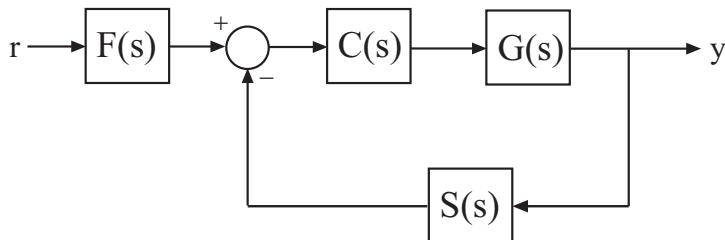
More About

Numeric Model of SISO Feedback Loop

This example shows how to interconnect "Numeric Models" on page 1-13 representing multiple system components to build a single numeric model of a closed-loop system, using model arithmetic and interconnection commands.

Construct a model of the following single-loop control system.

The feedback loop includes a plant $G(s)$, a controller $C(s)$, and a representation of sensor dynamics, $S(s)$. The system also includes a prefilter $F(s)$.

1 Create model objects representing each of the components.

```
G = zpk([],[-1,-1],1);
C = pid(2,1.3,0.3,0.5);
S = tf(5,[1 4]);
F = tf(1,[1 1]);
```

The plant G is a zero-pole-gain (zpk) model with a double pole at $s = -1$. Model object C is a PID controller. The models F and S are transfer functions.

2 Connect the controller and plant models.

```
H = G*C;
```

To combine models using the multiplication operator `*`, enter the models in reverse order compared to the block diagram.

Tip Alternatively, construct $H(s)$ using the `series` command:

```
H = series(C,G);
```

---

3 Construct the unfiltered closed-loop response $T(s) = \dfrac{H}{1 + HS}$.

```
T = feedback(H,S);
```

Caution Do not use model arithmetic to construct T algebraically:

```
T = H/(1+H*S)
```

This computation duplicates the poles of H, which inflates the model order and might lead to computational inaccuracy.

---

4 Construct the entire closed-loop system response from *r* to *y*.

```
T_ry = T*F;
```

T_ry is a Numeric LTI Model representing the aggregate closed-loop system. T_ry does not keep track of the coefficients of the components G, C, F, and S.

You can operate on T_ry with any Control System Toolbox control design or analysis commands.

See Also
connect | feedback | parallel | series

Related Examples
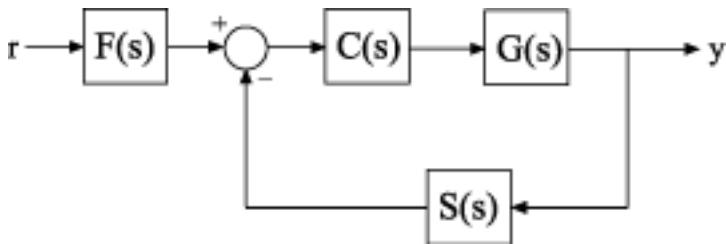
More About

Control System Model With Both Numeric and Tunable Components

This example shows how to create a tunable model of a control system that has both fixed plant and sensor dynamics and tunable control components.

Consider the the control system of the following illustration.

Suppose that the plant response is $G(s) = 1/(s+1)^2$, and that the model of the sensor dynamics is $S(s) = 5/(s+4)$. The controller C is a tunable PID controller, and the prefilter $F = a/(s+a)$ is a low-pass filter with one tunable parameter, a.

Create models representing the plant and sensor dynamics. Because the plant and sensor dynamics are fixed, represent them using numeric LTI models.

```
G = zpk([],[-1,-1],1);
S = tf(5,[1 4]);
```

To model the tunable components, use Control Design Blocks. Create a tunable representation of the controller C.

```
C = tunablePID('C','PID');
```

C is a `tunablePID` object, which is a Control Design Block with a predefined proportional-integral-derivative (PID) structure.

Create a model of the filter $F = a/(s+a)$ with one tunable parameter.

```
a = realp('a',10);
F = tf(a,[1 a]);
```

a is a `realp` (real tunable parameter) object with initial value 10. Using a as a coefficient in `tf` creates the tunable `genss` model object F.

Interconnect the models to construct a model of the complete closed-loop response from *r* to *y*.

```
T = feedback(G*C,S)*F


T =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 5 states, and
    C: Parametric PID controller, 1 occurrences.
    a: Scalar parameter, 2 occurrences.

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

T is a `genss` model object. In contrast to an aggregate model formed by connecting only numeric LTI models, T keeps track of the tunable elements of the control system. The tunable elements are stored in the `Blocks` property of the `genss` model object. Examine the tunable elements of T.

```
T.Blocks


ans =

    C: [1x1 tunablePID]
    a: [1x1 realp]
```

When you create a `genss` model of a control system that has tunable components, you can use tuning commands such as `systune` to tune the free parameters to meet design requirements you specify.
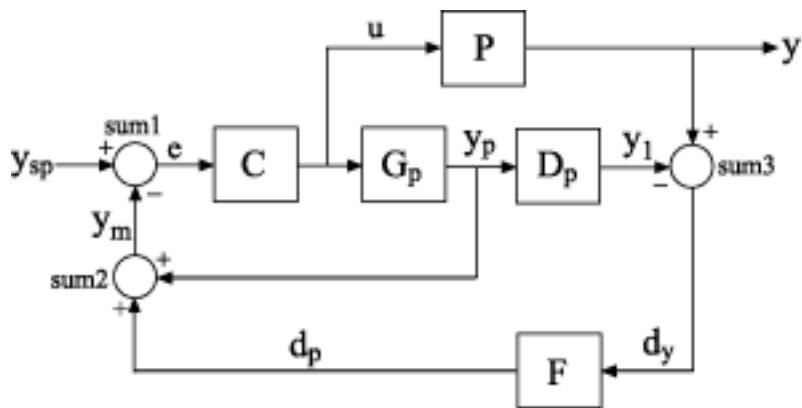
See Also
feedback | tunablePID

More About
- "Control Design Blocks" on page 1-16
- "Dynamic System Models" on page 1-10
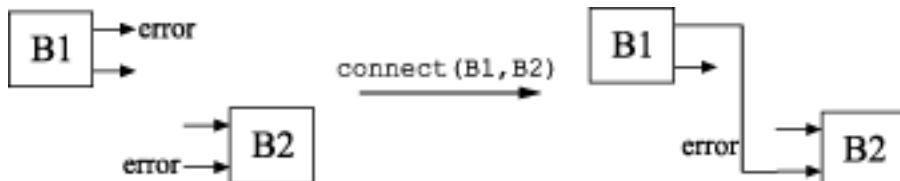
Multi-Loop Control System

This example shows how to build an arbitrary block diagram by connecting models using `connect`. The system is a Smith Predictor, the single-input, single-output (SISO) multi-loop control system shown in the following block diagram.

For more information about the Smith Predictor, see Control of Processes with Long Dead Time: The Smith Predictor.

The `connect` command lets you construct the overall transfer function from y_{sp} to y. To use `connect`, specify the input and output channel names of the components of the block diagram. `connect` automatically joins ports that have the same name, as shown in the following figure.

To build the closed loop model of the Smith Predictor system from y_{sp} to y:

1   Create the components of the block diagram: the process model P, the predictor model Gp, the delay model Dp, the filter F, and the PI controller C. Specify names for

the input and output channels of each model so that `connect` can automatically join them to build the block diagram.

```
s = tf('s');

P = exp(-93.9*s) * 5.6/(40.2*s+1);
P.InputName = 'u'; P.OutputName = 'y';

Gp = 5.6/(40.2*s+1);
Gp.InputName = 'u'; Gp.OutputName = 'yp';

Dp = exp(-93.9*s);
Dp.InputName = 'yp'; Dp.OutputName = 'y1';

F = 1/(20*s+1);
F.InputName = 'dy'; F.OutputName = 'dp';

C = pidstd(0.574,40.1);
C.Inputname = 'e'; C.OutputName = 'u';
```

2   Create the summing junctions needed to complete the block diagram.

```
sum1 = sumblk('e = ysp - ym');
sum2 = sumblk('ym = yp + dp');
sum3 = sumblk('dy = y - y1');
```

The argument to `sumblk` is a formula specified as a string. This string relates the input and output signals of the summing junction. `sumblk` creates a summing junction with the input and output signal names specified in the formula. For example, in `sum1`, the string `'e = ysp - ym'` specifies an output signal named `e`, which is the difference between input signals named `ysp` and `ym`.

3   Assemble the complete model from y_{sp} to y.

```
T = connect(P,Gp,Dp,C,F,sum1,sum2,sum3,'ysp','y');
```

You can list the models and summing junctions in any order because `connect` automatically interconnects them using their input and output channel names.

The last two arguments specify the input and output signals of the multi-loop control structure. Thus, `T` is a `ss` model with input `ysp` and output `y`.

See Also
connect | sumblk

Related Examples
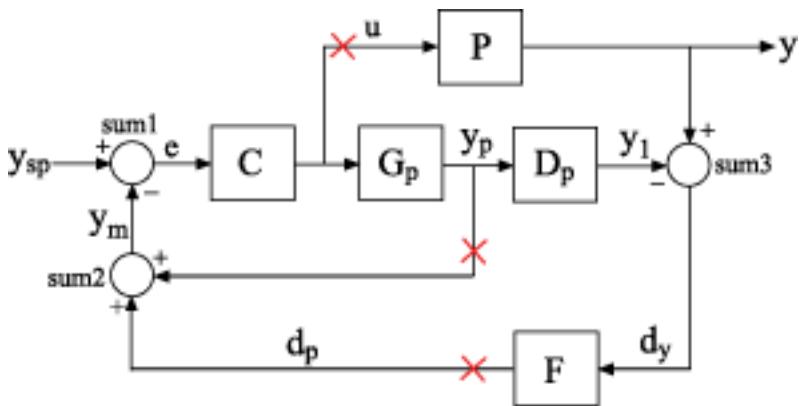
More About

Mark Analysis Points in Closed-Loop Models

This example shows how to build a block diagram and insert analysis points at locations of interest using the `connect` command. You can then use the analysis points to extract various system responses from the model.

For this example, create a model of a Smith predictor, the SISO multiloop control system shown in the following block diagram.

Points marked by x are analysis points to mark for this example. For instance, if you want to calculate the step response of the closed-loop system to a disturbance at the plant input, you can use an anlysis point at *u*. If you want to calculate the response of the system with one or both of the control loops open, you can use the analysis points at *yp* or *dp*.

To build this system, first create the dynamic components of the block diagram. Specify names for the input and output channels of each model so that `connect` can automatically join them to build the block diagram.

```
s = tf('s');

% Process model
P = exp(-93.9*s) * 5.6/(40.2*s+1);
P.InputName = 'u';
P.OutputName = 'y';

% Predictor model
```

```matlab
Gp = 5.6/(40.2*s+1);
Gp.InputName = 'u';
Gp.OutputName = 'yp';

% Delay model
Dp = exp(-93.9*s);
Dp.InputName = 'yp';
Dp.OutputName = 'y1';

% Filter
F = 1/(20*s+1);
F.InputName = 'dy';
F.OutputName = 'dp';

% PI controller
C = pidstd(0.574,40.1);
C.Inputname = 'e';
C.OutputName = 'u';
```

Create the summing junctions needed to complete the block diagram. (For more information about creating summing junctions, see sumblk).

```matlab
sum1 = sumblk('e = ysp - ym');
sum2 = sumblk('ym = yp + dp');
sum3 = sumblk('dy = y - y1');
```

Assemble the complete model.

```matlab
input = 'ysp';
output = 'y';
APs = {'u','dp','yp'};
T = connect(P,Gp,Dp,C,F,sum1,sum2,sum3,input,output,APs)
```

```
T =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 4 states, and
    AnalysisPoints_: Analysis point, 3 channels, 1 occurrences.

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

When you use the APs input argument, the connect command automatically inserts an AnalysisPoint block into the generalized state-space (genss) model, T. The

automatically generated block is named `AnalysisPoints_`. The three channels of `AnalysisPoints_` correspond to the three locations specified in the `APs` argument to the `connect` command. Use `getPoints` to see a list of the available analysis points in the model.
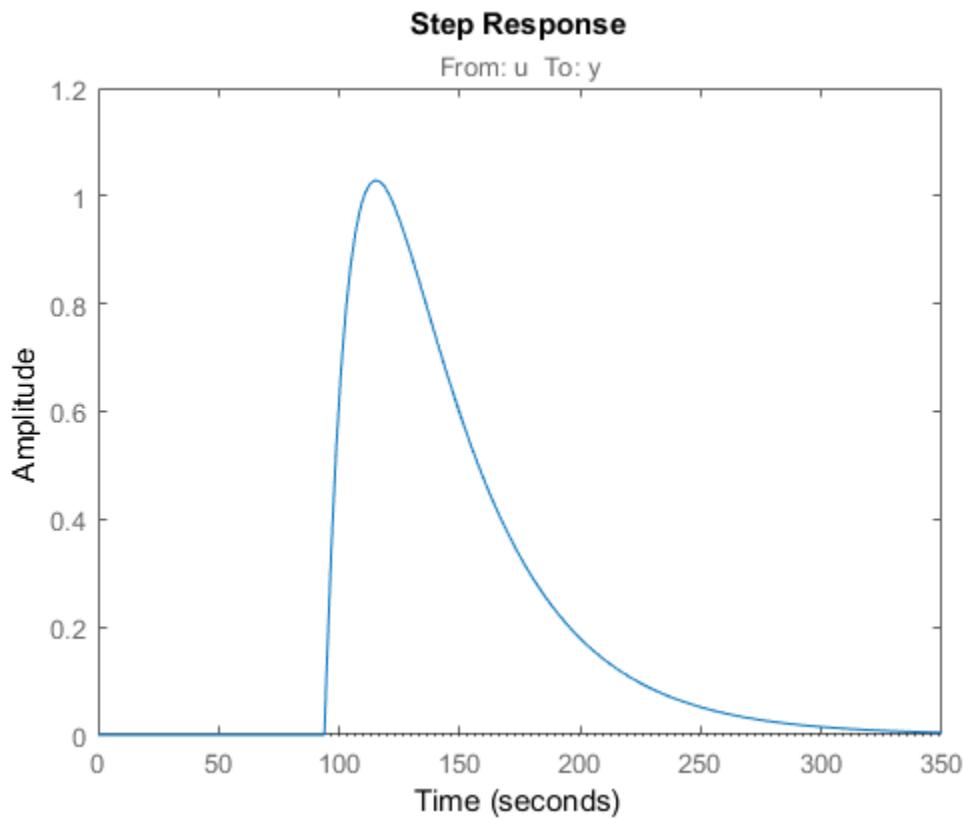
```
getPoints(T)


ans =

    'dp'
    'u'
    'yp'

```
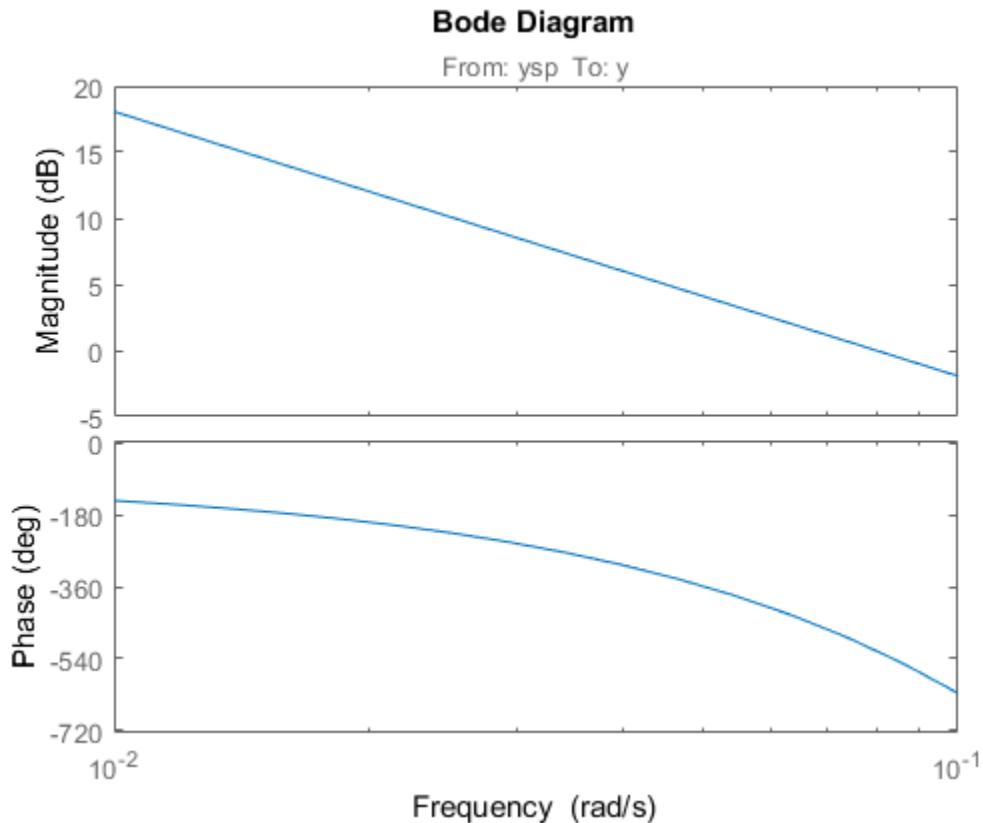
Use these locations as inputs, outputs, or loop openings when you extract responses from the model. For example, extract and plot the response at the system output to a step disturbance at the plant input, *u*.

```
Tp = getIOTransfer(T,'u','y');
stepplot(Tp)
```

Step Response

From: u   To: y

Similarly, calculate the open-loop response of the plant and controller by opening both feedback loops.

```
openings = {'dp','yp'};
L = getIOTransfer(T,'ysp','y',openings);
bodeplot(L)
```

Bode Diagram

From: ysp To: y

When you create a control system model, you can create an `AnalysisPoint` block explicitly and assign input and output names to it. You can then include it in the input arguments to `connect` as one of the blocks to combine. However, using the `APs` argument to `connect` as illustrated in this example is a simpler way to mark points of interest when building control system models.

See Also
`AnalysisPoint` | `connect` | `sumblk`

Related Examples
*   "Control System with Multichannel Analysis Points" on page 2-84
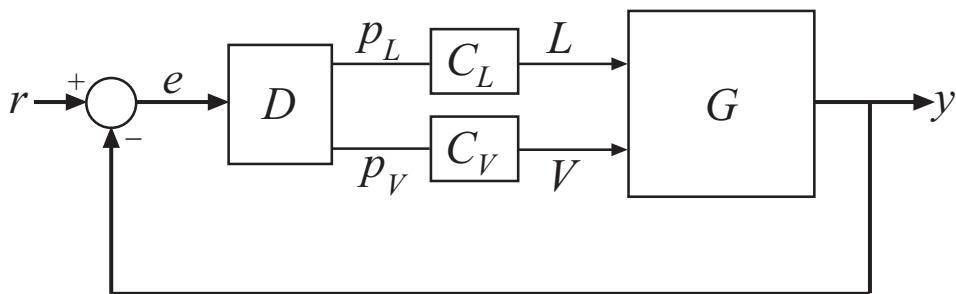
More About

- "Marking Signals of Interest for Control System Analysis and Design" on page 2-88

MIMO Control System

This example shows how to build a MIMO control system using `connect` to interconnect "Numeric Linear Time Invariant (LTI) Models" on page 1-13 and tunable "Control Design Blocks" on page 1-16.

Consider the following two-input, two-output control system.

The plant *G* is a distillation column with two inputs and two outputs. The two inputs are the reflux *L* and boilup *V*. The two outputs are the concentrations of two chemicals, represented by the vector signal $y = [y_1, y_2]$. You can represent this plant model as:

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

The vector setpoint signal $r = [r_1, r_2]$ specifies the desired concentrations of the two chemicals. The vector error signal *e* represents the input to *D*, a static 2-by-2 decoupling matrix. C_L and C_V represent independent PI controllers that control the two inputs of *G*.

To create a two-input, two-output model representing this closed-loop control system:

1 Create a Numeric LTI model representing the 2-by-2 plant *G*.

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

When you construct the closed-loop model, `connect` uses the input and output names to form connections between the block diagram components. Therefore, you

must assign names to the inputs and outputs of the transfer function G in either of the following ways: .

- You can assign input and output names to individual signals by specifying signal names in a cell array, as in G.InputName = {'L','V'}

- Alternatively, you can use vector signal naming, which the software automatically expands. For example, the command G.OutputName = 'y' assigns the names 'y(1)' and 'y(2)' to the outputs of G.

2 Create tunable Control Design Blocks representing the decoupling matrix *D* and the PI controllers C_L and C_V.

```
D = tunableGain('Decoupler',eye(2));
D.u = 'e';
D.y = {'pL','pV'};

C_L = tunablePID('C_L','pi');  C_L.TimeUnit = 'minutes';
C_L.u = 'pL'; C_L.y = 'L';

C_V = tunablePID('C_V','pi');  C_V.TimeUnit = 'minutes';
C_V.u = 'pV'; C_V.y = 'V';
```

Note: u and y are shorthand notations for the InputName and OutputName properties, respectively. Thus, for example, entering:

```
D.u = 'e';
D.y = {'pL','pV'};
```

is equivalent to entering:

```
D.InputName = 'e';
D.OutputName = {'pL','pV'};
```

3 Create the summing junction.

The summing junction produces the error signals *e* by taking the difference between *r* and *y*.

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

Sum represents the transfer function for the summing junction described by the formula string 'e = r - y'. The second argument to sumblk specifies that the inputs and outputs of Sum are each vector signals of length 2. The software therefore

automatically assigns the signal names {'r(1)','r(2)','y(1)','y(2)'} to Sum.InputName and {'e(1)','e(2)'} to Sum.OutputName.

4 Join all components to build the closed-loop system from *r* to *y*.

```
CLry = connect(G,D,C_L,C_V,Sum,'r','y');
```

The arguments to the connect function include all the components of the closed-loop system, in any order. connect automatically combines the components using the input and output names to join signals.

The last two arguments to connect specify the output and input signals of the closed-loop model, respectively. The resulting genss model CLry has two-inputs and two outputs.

See Also
connect | sumblk

Related Examples
- "Control System Model With Both Numeric and Tunable Components" on page 4-8
- "Multi-Loop Control System" on page 4-10
- "MIMO Control System" on page 4-19
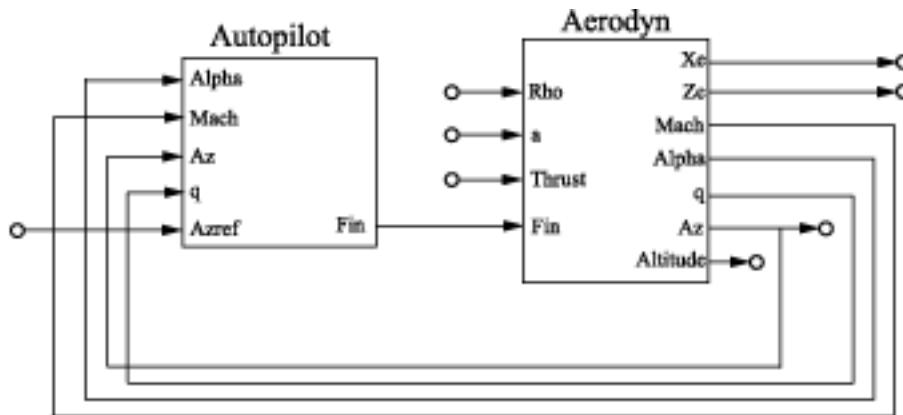
More About
- "Catalog of Model Interconnections" on page 4-3

MIMO Feedback Loop

This example shows how to obtain the closed-loop response of a MIMO feedback loop in three different ways.

In this example, you obtain the response from `Azref` to `Az` of the MIMO feedback loop of the following block diagram.

You can compute the closed-loop response using one of the following three approaches:

- Name-based interconnection with `connect`
- Name-based interconnection with `feedback`
- Index-based interconnection with `feedback`

You can use whichever of these approaches is most convenient for your application.

Load the plant `Aerodyn` and the controller `Autopilot` into the MATLAB® workspace. These models are stored in the datafile `MIMOfeedback.mat`.

```
load(fullfile(matlabroot,'examples','control','MIMOfeedback.mat'))
```

`Aerodyn` is a 4-input, 7-output state-space (`ss`) model. `Autopilot` is a 5-input, 1-output `ss` model. The inputs and outputs of both models names appear as shown in the block diagram.

Compute the closed-loop response from `Azref` to `Az` using `connect`.
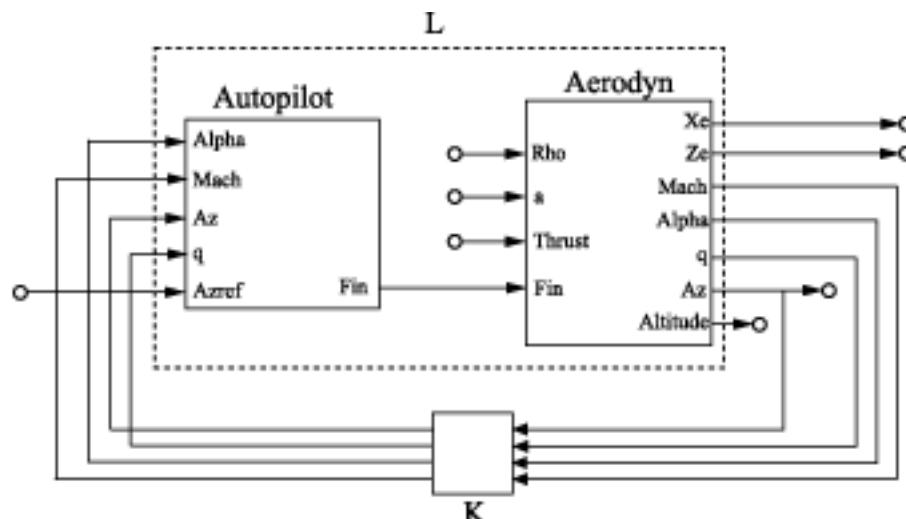
```
T1 = connect(Autopilot,Aerodyn,'Azref','Az');
```

```
Warning: The following block inputs are not used: Rho,a,Thrust.
Warning: The following block outputs are not used: Xe,Ze,Altitude.
```

The `connect` function combines the models by joining the inputs and outputs that have matching names. The last two arguments to `connect` specify the input and output signals of the resulting model. Therefore, `T1` is a state-space model with input `Azref` and output `Az`. The `connect` function ignores the other inputs and outputs in `Autopilot` and `Aerodyn`.

Compute the closed-loop response from `Azref` to `Az` using name-based interconnection with the `feedback` command. Use the model input and output names to specify the interconnections between `Aerodyn` and `Autopilot`.

When you use the `feedback` function, think of the closed-loop system as a feedback interconnection between an open-loop plant-controller combination `L` and a diagonal unity-gain feedback element `K`. The following block diagram shows this interconnection.


```
L = series(Autopilot,Aerodyn,'Fin');

FeedbackChannels = {'Alpha','Mach','Az','q'};
K = ss(eye(4),'InputName',FeedbackChannels,...
             'OutputName',FeedbackChannels);

T2 = feedback(L,K,'name',+1);
```

The closed-loop model T2 represents the positive feedback interconnection of L and K. The 'name' option causes feedback to connect L and K by matching their input and output names.

T2 is a 5-input, 7-output state-space model. The closed-loop response from Azref to Az is T2('Az','Azref').

Compute the closed-loop response from Azref to Az using feedback, using indices to specify the interconnections between Aerodyn and Autopilot.
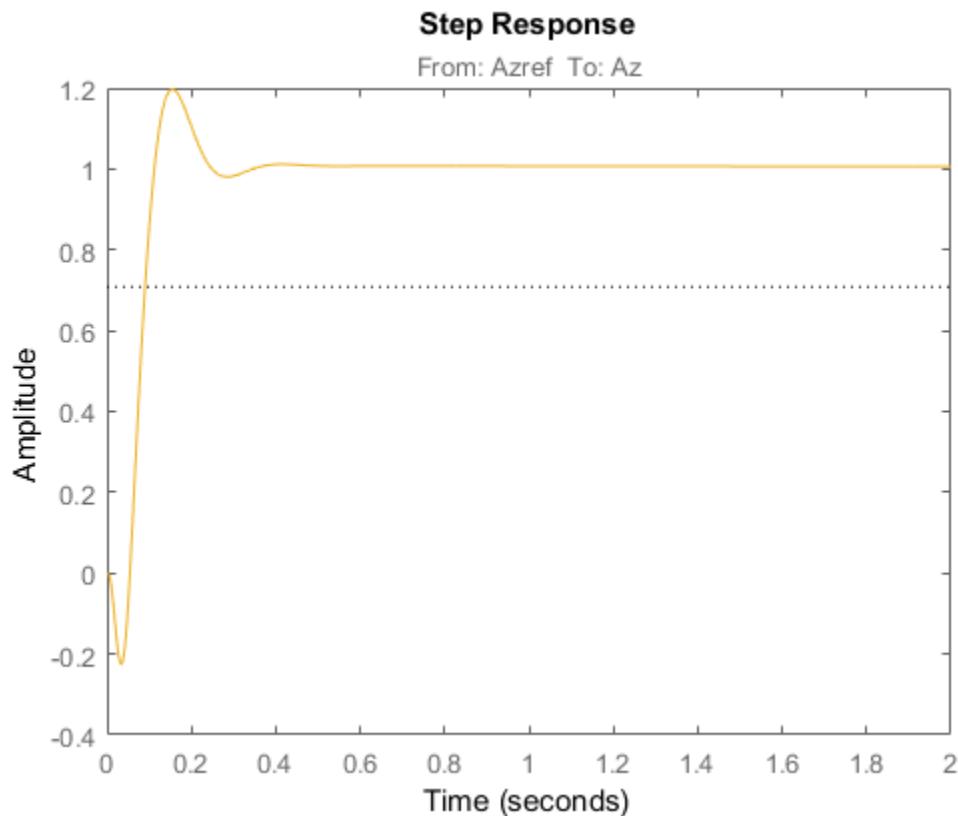
```
L = series(Autopilot,Aerodyn,1,4);
K = ss(eye(4));
T3 = feedback(L,K,[1 2 3 4],[4 3 6 5],+1);
```

The vectors [1 2 3 4] and [4 3 6 5] specify which inputs and outputs, respectively, complete the feedback interconnection. For example, feedback uses output 4 and input 1 of L to create the first feedback interconnection. The function uses output 3 and input 2 to create the second interconnection, and so on.

T3 is a 5-input, 7-output state-space model. The closed-loop response from Azref to Az is T3(6,5).

Compare the step response from Azref to Az to confirm that the three approaches yield the same results.

```
step(T1,T2('Az','Azref'),T3(6,5),2)
```

Step Response

From: Azref To: Az

See Also
connect | feedback

Related Examples
- "Multi-Loop Control System" on page 4-10
- "MIMO Control System" on page 4-19

More About
- "How the Software Determines Properties of Connected Models" on page 4-26

How the Software Determines Properties of Connected Models

When you interconnect models, the operation and the properties of the models you are connecting determine the resulting model's properties. The following table summarizes some general rules governing how resulting model property values are determined.

Property	Expected Behavior
`Ts`	When connecting discrete-time models, all models must have identical or unspecified (`sys.Ts = -1`) sample time. The resulting model inherits the sample time from the connected models.
`InputName` `OutputName` `InputGroup` `InputGroup`	In general, the resulting model inherits I/O names and I/O groups from connected models. However, conflicting I/O names or I/O groups are not inherited. For example, the `InputName` property for `sys1 + sys2` is left unspecified if `sys1` and `sys2` have different `InputName` property values.
`TimeUnit`	All connected models must have identical `TimeUnit` properties. The resulting model inherits its `TimeUnit` from the connected models.
`Variable`	A model resulting from operations on `tf` or `zpk` models inherits its `Variable` property value from the operands. Conflicts are resolved according the following rules: • For continuous-time models, `'p'` has precedence over `'s'`. • For discrete-time models, `'q^-1'` and `'z^-1'` have precedence over `'q'` and `'z'`, while `'q'` has precedence over `'z'`.
`Notes` `UserData`	Most operations ignore the `Notes` and `UserData` properties. These properties of the resulting model are empty.

More About

• "Rules That Determine Model Type" on page 4-27

Rules That Determine Model Type

This example explains therules that determine the type of model that results when you interconnect models of different types.

When you combine "Numeric Models" on page 1-13 other than `frd` models using `connect`, the resulting model is a state-space (`ss`) model. For other interconnection commands, the resulting model is determined by the following order of precedence:

`ss > zpk > tf > pid > pidstd`

For example, connect an `ss` model with a `pid` model.

```
P = ss([-0.8,0.4;0.4,-1.0],[-3.0;1.4],[0.3,0],0);
C = pid(-0.13,-0.61);
CL = feedback(P*C,1)
```

The `ss` model has the highest precedence among Numeric LTI models. Therefore, combining `P` and `C` with any model interconnection command returns an `ss` model.

Combining Numeric LTI models with "Generalized and Uncertain LTI Models" on page 1-16 or with "Control Design Blocks" on page 1-16 results in Generalized LTI models.

For example, connect the `ss` model `CL` with a Control Design Block.

```
F = tunableTF('F',0,1);
CLF = F*CL
```

`CLF` is a `genss` model.

Any connection that includes a frequency-response model (`frd` or `genfrd`) results in a frequency-response model.

Note: The software automatically converts all models to the resulting model type before performing the connection operation.

See Also
`connect` | `feedback` | `parallel` | `series`

Related Examples
· "Numeric Model of SISO Feedback Loop" on page 4-6

More About

Recommended Model Type for Building Block Diagrams

This example shows how choice of model type can affect numerical accuracy when interconnecting models.

You can represent block diagram components with any model type. However, certain connection operations yield better numerical accuracy for models in ss form.

For example, interconnect two models in series using different model types to see how different representations introduce numerical inaccuracies.

Load the models Pd and Cd. These models are ninth-order and second-order discrete-time transfer functions, respectively.

```
load numdemo Pd Cd
```
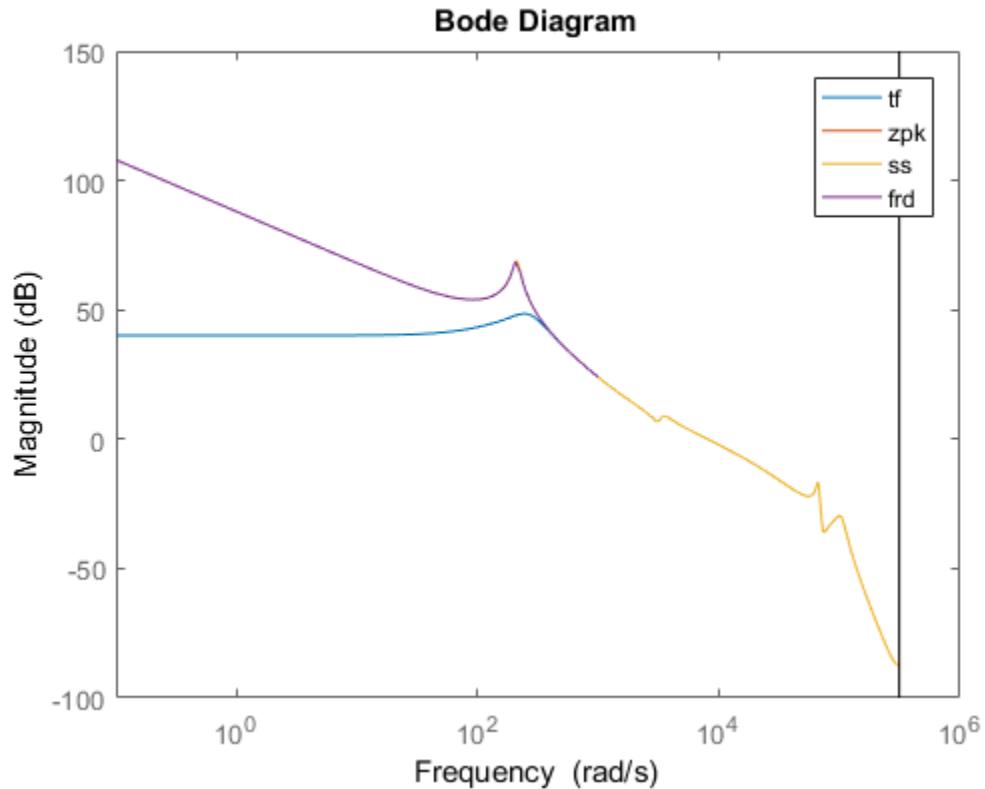
Compute the open-loop transfer function L = Pd*Cd using the tf, zpk, ss, and frd representations.

```
Ltf = Pd*Cd;
Lzp = zpk(Pd)*Cd;
Lss = ss(Pd)*Cd;

w = logspace(-1,3,100);
Lfrd = frd(Pd,w)*Cd;
```

Plot the magnitude of the frequency response to compare the four representations.

```
bodemag(Ltf,Lzp,Lss,Lfrd)
legend('tf','zpk','ss','frd')
```

The `tf` representation has lost low-frequency dynamics that other representations preserve.

More About

- "Rules That Determine Model Type" on page 4-27

5

Model Transformation

Conversion Between Model Types

In this section...
"Explicit Conversion Between Model Types" on page 5-2
"Automatic Conversion Between Model Types" on page 5-2
"Recommended Working Representation" on page 5-3

Explicit Conversion Between Model Types

You can explicitly convert a model from one representation to another using the model-creation command for the target model type. For example, convert to state-space representation using `ss`, and convert to parallel-form PID using `pid`. For information about converting to a particular model type, see the reference page for that model type.

In general, you can convert from any model type to any other. However, there are a few limitations. For example, you cannot convert:

- `frd` models to analytic model types such as `ss`, `tf`, or `zpk` (unless you perform system identification with System Identification Toolbox software).
- `ss` models with internal delays to `tf` or `zpk`.

You can convert between Numeric LTI models and Generalized LTI models.

- Converting a Generalized LTI model to a Numeric LTI model evaluates any Control Design Blocks at their current (nominal) value.
- Converting a Numeric LTI model to a Generalized LTI model creates a Generalized LTI model with an empty `Blocks` property.

Automatic Conversion Between Model Types

Some algorithms operate only on one type of model object. For example, the algorithm for zero-order-hold discretization with `c2d` can only be performed on state-space models. Similarly, commands such as `tfdata` or `piddata` expect a particular type of model (`tf` or `pid`, respectively). For convenience, such commands automatically convert input models to the appropriate or required model type. For example:

```
sys = ss(0,1,1,0)
[num,den] = tfdata(sys)
```

`tfdata` automatically converts the state-space model `sys` to transfer function form to return numerator and denominator data.

Conversions to state-space form are not uniquely defined. For this reason, automatic conversions to state space do not occur when the result depends on the choice of state coordinates. For example, the `initial` and `kalman` commands require state-space models.

Recommended Working Representation

You can represent numeric system components using any model type. However, Numeric LTI model types are not equally well-suited for numerical computations. In general, it is recommended that you work with state-space (`ss`) or frequency response data (`frd`) models, for the following reasons:

- The accuracy of computations using high-order transfer functions (`tf` or `zpk` models) is sometimes poor, particularly for MIMO or high-order systems. Conversions to a transfer function representation can incur a loss of accuracy.

- When you convert `tf` or `zpk` models to state space using `ss`, the software automatically performs balancing and scaling operations. Balancing and scaling improves the numeric accuracy of computations involving the model. For more information about balancing and scaling state-space models, see "Scaling State-Space Models" on page 21-2.

In addition, converting back and forth between model types can introduce additional states or orders, or introduce numeric inaccuracies. For example, conversions to state space are not uniquely defined, and are not guaranteed to produce a minimal realization for MIMO models. For a given state-space model `sys`,

```
ss(tf(sys))
```

can return a model with different state-space matrices, or even a different number of states in the MIMO case.

See Also
`frd` | `pid` | `ss` | `tf` | `zpk`

Related Examples
- "Convert From One Model Type to Another" on page 5-4

Convert From One Model Type to Another

This example shows how to convert a numeric LTI model from one type (`pid`) to another type (`tf`).

In general, you can convert a model from one type to another type using the model-creation command for the target type. Thus, for example, you can use the `tf` command to convert a `ss` model to transfer-function form, or use the `ss` command to convert a `zpk` model to state-space form.

Create a PID controller.

```
pid_sys = pid(1,1.5,3)


pid_sys =

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

  with Kp = 1, Ki = 1.5, Kd = 3

Continuous-time PID controller in parallel form.
```

Convert `pid_sys` to a transfer function model.

```
C = tf(pid_sys)


C =

  3 s^2 + s + 1.5
  ---------------
         s

Continuous-time transfer function.
```

`C` is a `tf` representation of `pid_sys`. `C` has the same dynamics as `pid_sys`, but stores the dynamic parameters as transfer-function numerator and denominator coefficients instead of proportional, integral, and derivative gains.

You can similarly convert transfer function models to `pid` models, provided the `tf` model object represents a parallel-form PID controller with $Tf \geq 0$.

In general, you can use the technique of this example to convert any type of model to another type of model. For more specific information about converting to a particular model type, see the reference page for that model type.

See Also

`frd` | `pid` | `ss` | `tf` | `zpk`

More About

• "Conversion Between Model Types" on page 5-2

Get Current Value of Generalized Model by Model Conversion

This example shows how to get the current value of a generalized model by converting it to a numeric model. This conversion is useful, for example, when you have tuned the parameters of the generalized model using a tuning command such as `systune`.

Create a Generalized Model

Represent the transfer function

$$F = \frac{a}{s+a}$$

containing a real, tunable parameter, `a`, which is initialized to 10.

```
a = realp('a',10);
F = tf(a,[1 a]);
```

`F` is a `genss` model parameterized by `a`.

Tune the Model

Typically, once of you have a generalized model, you tune the parameters of the model using a tuning command such as `systune`. For this example, instead of tuning the model, manually change the value of the tunable component of `F`.

```
F.Blocks.a.Value = 5;
```

Get the current value of the generalized model.

Get the current value of the generalized model by converting it to a numeric model.

```
F_cur_val = tf(F)

F_cur_val =

    5
  -----
  s + 5

Continuous-time transfer function.
```

`tf(F)` converts the generalized model, `F`, to a numeric transfer function, `F_cur_val`.

To view the state-space representation of the current value of F, type `ss(F)`.

To examine the current values of the individual tunable components in a generalized model, use `showBlockValue`.

See Also
`realp` | `showBlockValue` | `tf`

Related Examples
- "Convert From One Model Type to Another" on page 5-4

More About
- "Models with Tunable Coefficients" on page 1-19
- "Conversion Between Model Types" on page 5-2
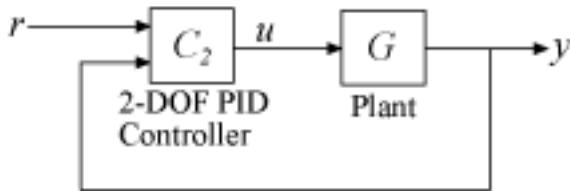
Decompose a 2-DOF PID Controller into SISO Components

This example shows how to extract SISO control components from a 2-DOF PID controller in each of the feedforward, feedback, and filter configurations. The example compares the closed-loop systems in all configurations to confirm that they are all equivalent.
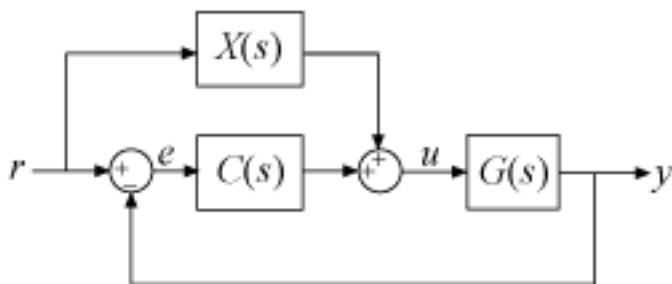
Obtain a 2-DOF PID controller. For this example, create a plant model, and tune a 2-DOF PID controller for it.

```
G = tf(1,[1 0.5 0.1]);
C2 = pidtune(G,'pidf2',1.5);
```

C2 is a `pid2` controller object. The control architecture for C2 is as shown in the following illustration.

This control system can be equivalently represented in several other architectures that use only SISO components. In the feedforward configuration, the 2-DOF controller is represented as a SISO PID controller and a feedforward compensator.

Decompose `C2` into SISO control components using the feedforward configuration.

```
[Cff,Xff] = getComponents(C2,'feedforward')


Cff =

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

  with Kp = 1.12, Ki = 0.23, Kd = 1.3, Tf = 0.122

Continuous-time PIDF controller in parallel form.


Xff =

  -10.898 (s+0.2838)
  -----------------
      (s+8.181)

Continuous-time zero/pole/gain model.
```
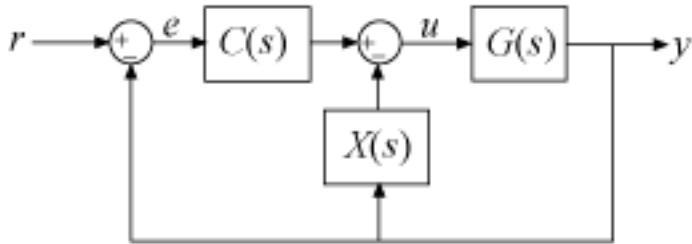
This command returns the SISO PID controller `Cff` as a `pid` object. The feedforward compensator `X` is returned as a `zpk` object.

Construct the closed-loop system for the feedforward configuration.

```
Tff = G*(Cff+Xff)*feedback(1,G*Cff);
```
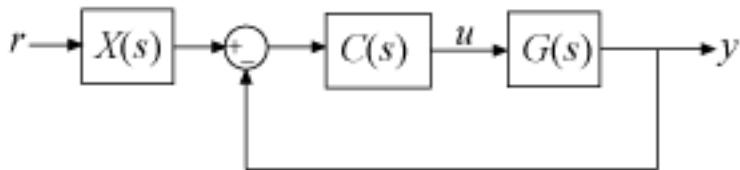
In the feedback configuration, the 2-DOF controller is represented as a SISO PID controller and an additional feedback compensator.

Decompose C2 using the feedback configuration and construct that closed-loop system.

```
[Cfb, Xfb] = getComponents(C2,'feedback');
Tfb = G*Cfb*feedback(1,G*(Cfb+Xfb));
```

In the filter configuration, the 2-DOF controller is represented as a SISO PID controller and prefilter on the reference signal.

Decompose C2 using the filter configuration. Construct that closed-loop system as well.
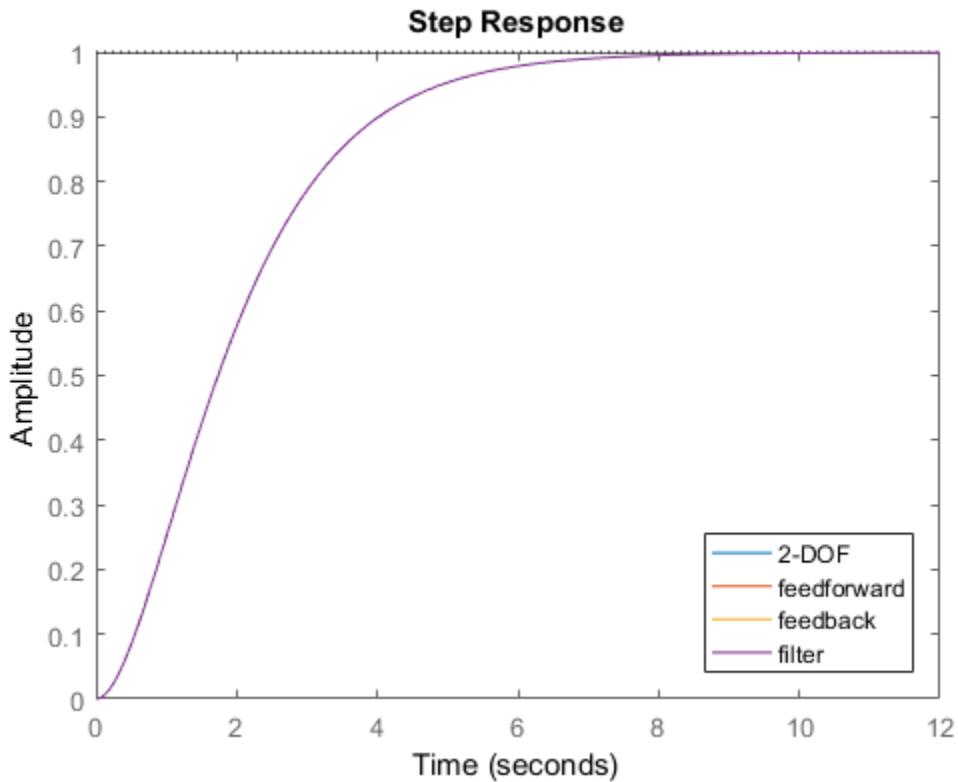
```
[Cfr, Xfr] = getComponents(C2,'filter');
Tfr = Xfr*feedback(G*Cfr,1);
```

Construct the closed-loop system for the original 2-DOF controller, C2. To do so, convert C2 to a two-input, one-output transfer function, and use array indexing to access the channels.

```
Ctf = tf(C2);
Cr = Ctf(1);
Cy = Ctf(2);
T = Cr*feedback(G,Cy,+1);
```

Compare the step responses of all the closed-loop systems. The traces coincide, demonstrating that all the systems are equivalent.

```
stepplot(T,Tff,Tfb,Tfr)
legend('2-DOF','feedforward','feedback','filter','Location','Southeast')
```


See Also
getComponents | pid2 | pidstd2
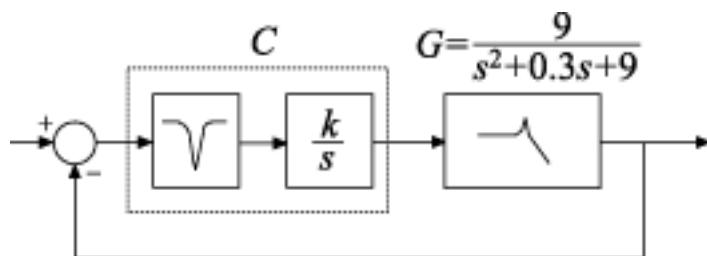
Related Examples
• "Two-Degree-of-Freedom PID Controllers" on page 2-17

Discretize a Compensator

This example shows how to convert a compensator from continuous to discrete time using several discretization methods, to identify a method that yields a good match in the frequency domain.
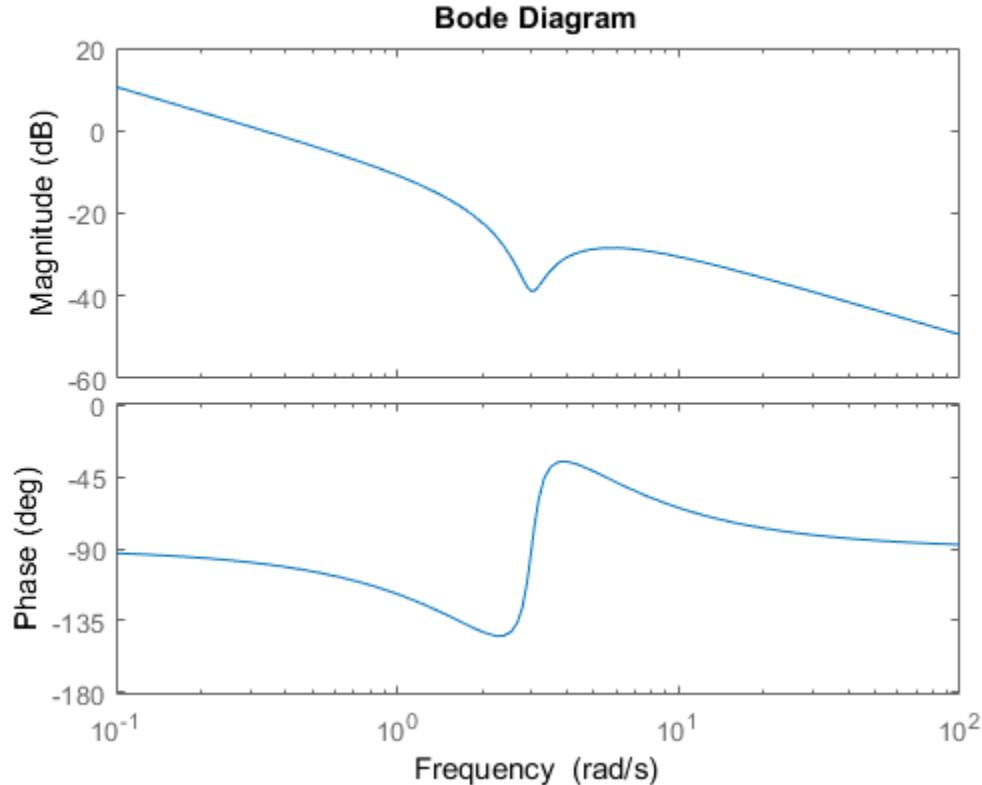
You might design a compensator in continuous time, and then need to convert it to discrete time for a digital implementation. When you do so, you want the discretization to preserve frequency-domain characteristics that are essential to your performance and stability requirements.

In the following control system, G is a continuous-time second-order system with a sharp resonance around 3 rad/s.

One valid controller for this system includes a notch filter in series with an integrator. Create a model of this controller.

```
notch = tf([1,0.5,9],[1,5,9]);
integ = pid(0,0.34);
C = integ*notch;
bodeplot(C)
```

Bode Diagram

The notch filter centered at 3 rad/s counteracts the effect of the resonance in G. This configuration allows higher loop gain for a faster overall response.

Discretize the compensator.
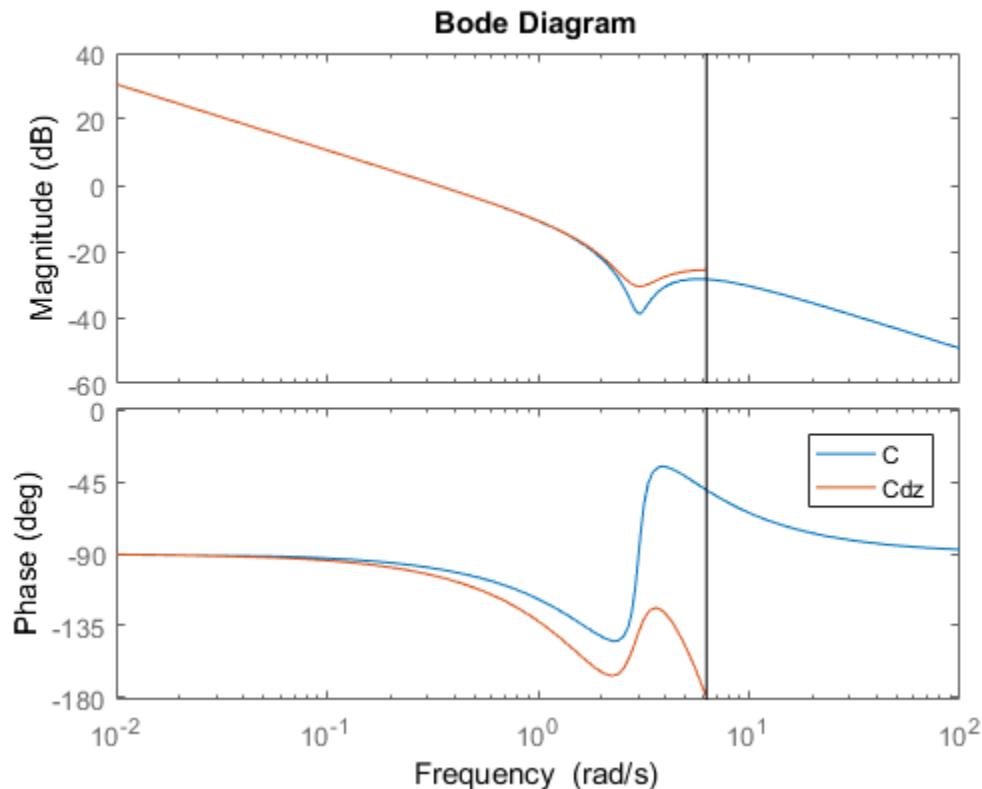
```
Cdz = c2d(C,0.5);
```

The c2d command supports several different discretization methods. Since this command does not specify a method, c2d uses the default method, Zero-Order Hold (ZOH). In the ZOH method, the time-domain response of the discretized compensator matches the continuous-time response at each time step.

The discretized controller `Cdz` has a sample time of 0.5 s. In practice, the sample time you choose might be constrained by the system in which you implement your controller, or by the bandwidth of your control system.

Compare the frequency-domain response of `C` and `Cdz`.
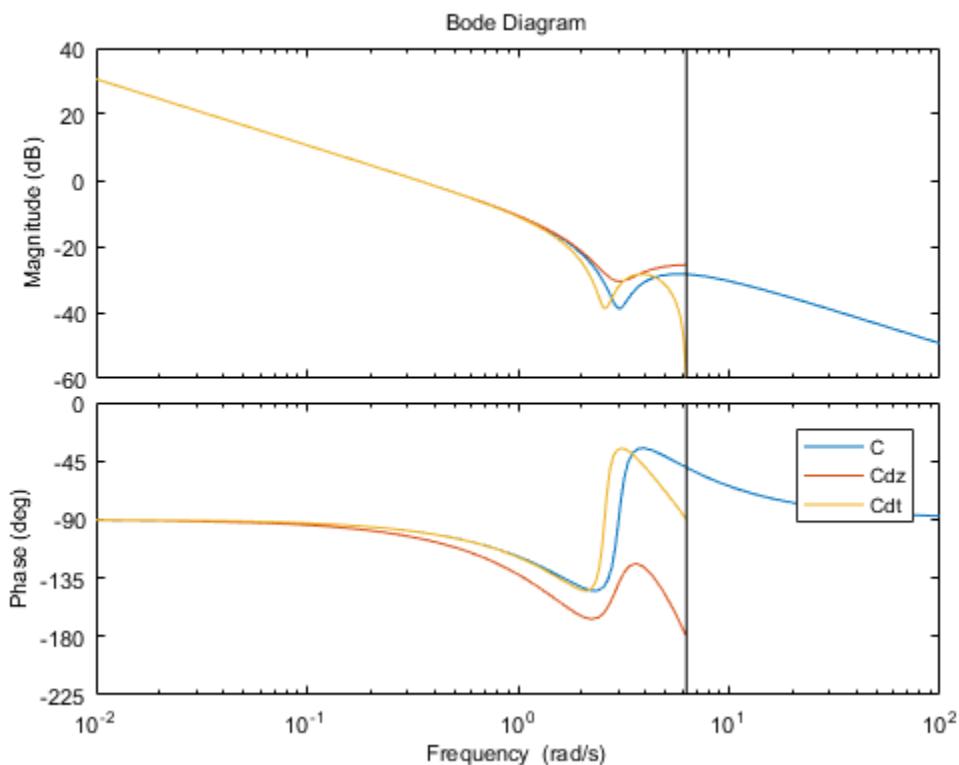
```
bodeplot(C,Cdz)
legend('C','Cdz');
```


The vertical line marks the Nyquist frequency, π/T_s, where T_s is the sample time. Near the Nyquist frequency, the response of the discretized compensator is distorted relative to the continuous-time response. As a result, the discretized notched filter may not properly counteract the plant resonance.

To fix this, try discretizing the compensator using the Tustin method and compare to the ZOH result. The Tustin discretization method often yields a better match in the frequency domain than the ZOH method.
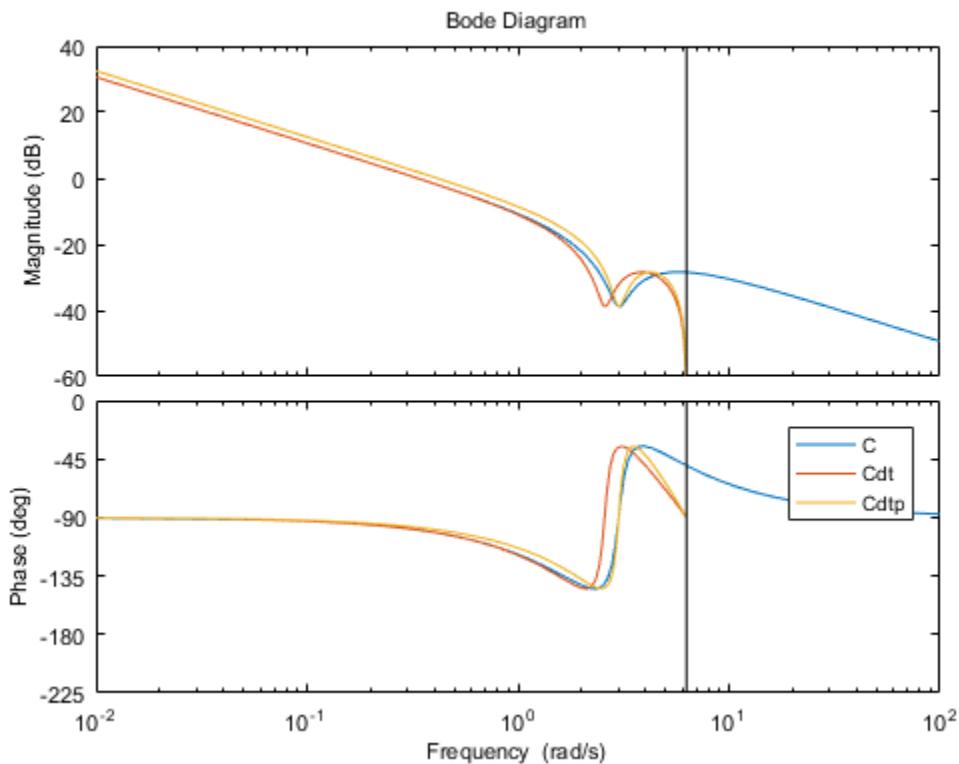
```
Cdt = c2d(C,0.5,'tustin');
plotopts = bodeoptions;
plotopts.Ylim = {[-60,40],[-225,0]};
bodeplot(C,Cdz,Cdt,plotopts)
legend('C','Cdz','Cdt')
```


The Tustin method preserves the depth of the notch. However, the method introduces a frequency shift that is unacceptable for many applications. You can remedy the frequency shift by specifying the notch frequency as the prewarping frequency in the Tustin transform.

Discretize the compensator using the Tustin method with frequency prewarping, and compare the results.

```
discopts = c2dOptions('Method','tustin','PrewarpFrequency',3.0);
Cdtp = c2d(C,0.5,discopts);
bodeplot(C,Cdt,Cdtp,plotopts)
legend('C','Cdt','Cdtp')
```


To specify additional discretization options beyond the discretization method, use c2dOptions. Here, the discretization options set discopts specifies both the Tustin method and the prewarp frequency. The prewarp frequency is 3.0 rad/s, the frequency of the notch in the compensator response.

Using the Tustin method with frequency prewarping yields a better-matching frequency response than Tustin without prewarping.

See Also
c2d | c2dOptions

Related Examples
- "Improve Accuracy of Discretized System with Time Delay" on page 5-18

More About
- "Continuous-Discrete Conversion Methods" on page 5-25

Improve Accuracy of Discretized System with Time Delay

This example shows how to improve the frequency-domain accuracy of a system with a time delay that is a fractional multiple of the sample time.

For systems with time delays that are not integer multiples of the sample time, the `Tustin` and `Matched` methods by default round the time delays to the nearest multiple of the sample time. To improve the accuracy of these methods for such systems, `c2d` can optionally approximate the fractional portion of the time delay by a discrete-time all-pass filter (a Thiran filter). In this example, discretize the system both without and with an approximation of the fractional portion of the delay and compare the results.

Create a continuous-time transfer function with a transfer delay of 2.5 s.

```
G = tf(1,[1,0.2,4],'ioDelay',2.5);
```

Discretize `G` using a sample time of 1 s. `G` has a sharp resonance at 2 rad/s. At a sample time of 1 s, that peak is close to the Nyquist frequency. For a frequency-domain match that preserves dynamics near the peak, use the Tustin method with prewarp frequency 2 rad/s.

```
discopts = c2dOptions('Method','tustin','PrewarpFrequency',2);
Gt = c2d(G,1,discopts)
```

```
Warning: Rounding delays to the nearest multiple of the sampling period. For
more accuracy in the time domain, use the ZOH or FOH methods. For more accuracy
in the frequency domain, use Thiran filters to approximate the fractional delays
(type "help c2dOptions" for more details).

Gt =

            0.1693 z^2 + 0.3386 z + 0.1693
  z^(-3) * ----------------------------
               z^2 + 0.7961 z + 0.913

Sample time: 1 seconds
Discrete-time transfer function.
```
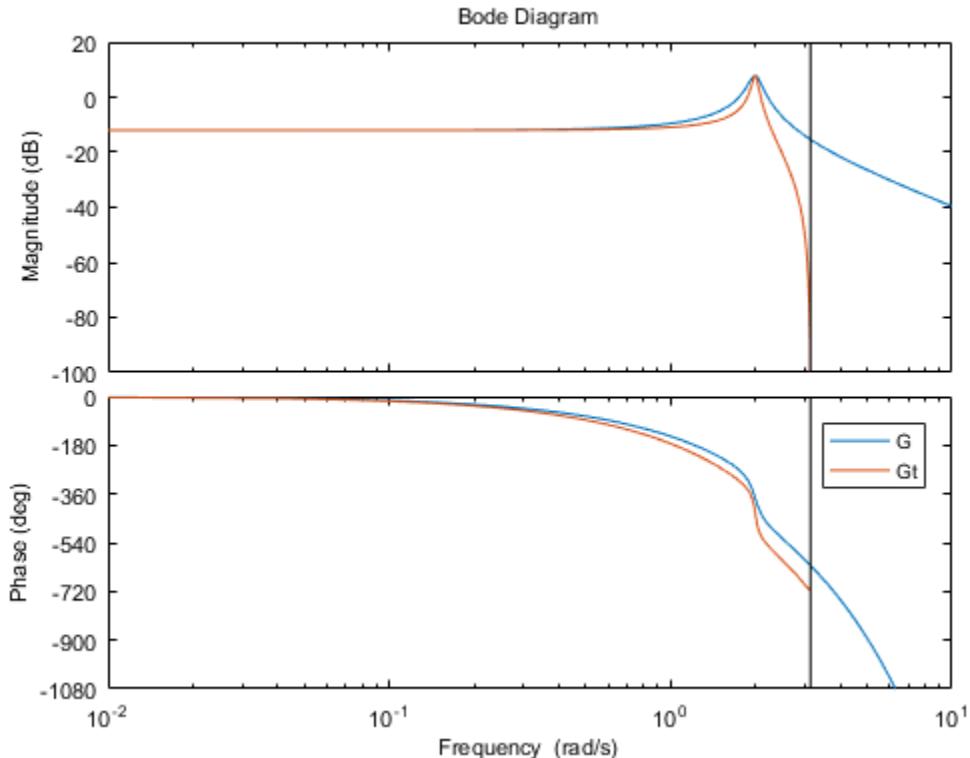
The software warns you that it rounds the fractional time delay to the nearest multiple of the sample time. In this example, the time delay of 2.5 times the sample time (2.5 s) converts to an additional factor of `z^(-3)` in `Gt`.

Compare `Gt` to the continuous-time system `G`.

```
plotopts = bodeoptions;
plotopts.Ylim = {[-100,20],[-1080,0]};
bodeplot(G,Gt,plotopts);
legend('G','Gt')
```


Bode Diagram

There is a phase lag between the discretized system `Gt` and the continuous-time system `G`, which grows as the frequency approaches the Nyquist frequency. This phase lag is largely due to the rounding of the fractional time delay. In this example, the fractional time delay is half the sample time.

Discretize `G` again using a third-order discrete-time all-pass filter (Thiran filter) to approximate the half-period portion of the delay.
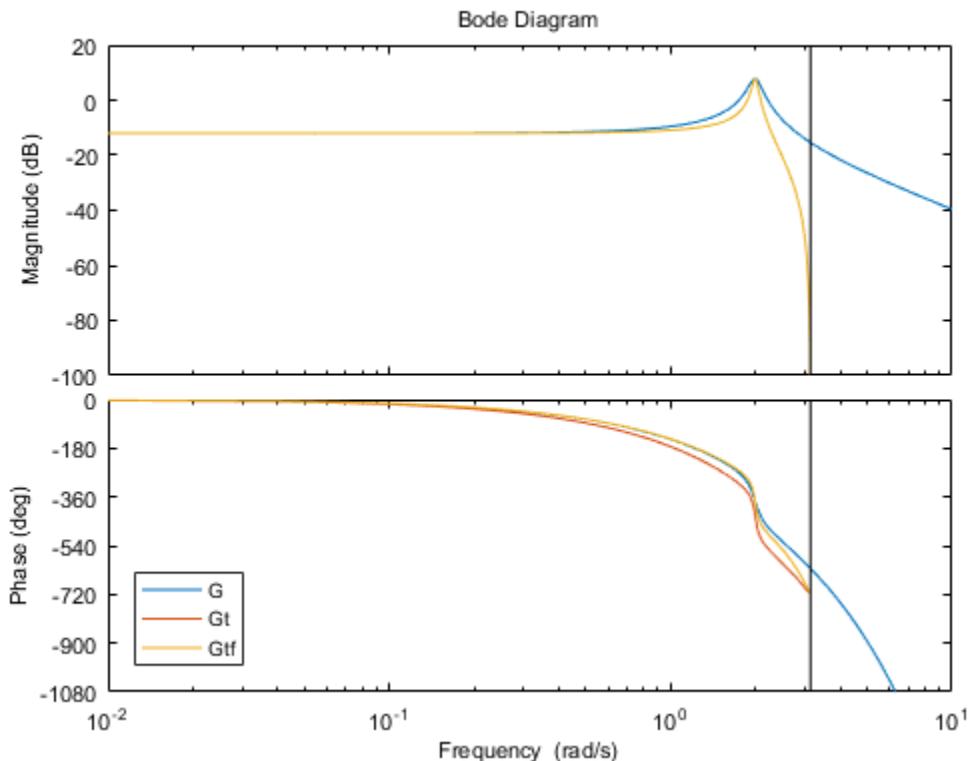
```
discopts.FractDelayApproxOrder = 3;
```

```
Gtf = c2d(G,1,discopts);
```

The `FractDelayApproxOrder` option specifies the order of the Thiran filter that approximates the fractional portion of the delay. The other options in `discopts` are unchanged. Thus `Gtf` is a Tustin discretization of `G` with prewarp at 2 rad/s.

Compare `Gtf` to `G` and `Gt`.

```
plotopts.PhaseMatching = 'on';
bodeplot(G,Gt,Gtf,plotopts);
legend('G','Gt','Gtf','Location','SouthWest')
```


The magnitudes of `Gt` and `Gtf` are identical. However, the phase of `Gtf` provides a better match to the phase of the continuous-time system through the resonance. As the

frequency approaches the Nyquist frequency, this phase match deteriorates. A higher-order approximation of the fractional delay would improve the phase matching closer to the Nyquist frequencies. However, each additional order of approximation adds an additional order (or state) to the discretized system.

If your application requires accurate frequency-matching near the Nyquist frequency, use `c2dOptions` to make `c2d` approximate the fractional portion of the time delay as a Thiran filter.

See Also
c2d | c2dOptions | thiran

Related Examples
- "Discretize a Compensator" on page 5-12

More About
- "Continuous-Discrete Conversion Methods" on page 5-25

Convert Discrete-Time System to Continuous Time

This example shows how to convert a discrete-time system to continuous time using d2c, and compares the results using two different interpolation methods.

Convert the following second-order discrete-time system to continuous time using the zero-order hold (ZOH) method:

$$G(z) = \frac{z + 0.5}{(z + 2)(z - 5)}.$$

```
G = zpk(-0.5,[-2,5],1,0.1);
Gcz = d2c(G)
```

```
Warning: The model order was increased to handle real negative poles.

Gcz =

  2.6663 (s^2 + 14.28s + 780.9)
  -----------------------------
  (s-16.09) (s^2 - 13.86s + 1035)

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

When you call d2c without specifying a method, the function uses ZOH by default. The ZOH interpolation method increases the model order for systems that have real negative poles. This order increase occurs because the interpolation algorithm maps real negative poles in the z domain to pairs of complex conjugate poles in the s domain.

Convert G to continuous time using the Tustin method.

```
Gct = d2c(G,'tustin')
```
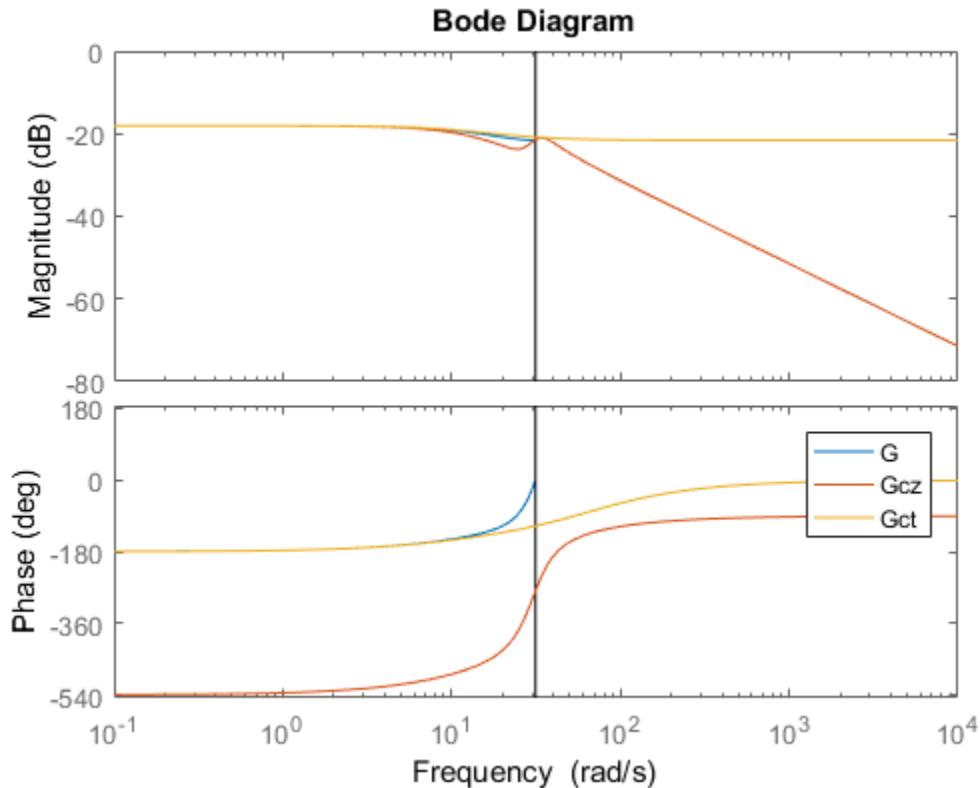
```
Gct =

  0.083333 (s+60) (s-20)
  --------------------
    (s-60) (s-13.33)

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

In this case, there is no order increase.

Compare frequency responses of the interpolated systems with that of G.

```
bode(G,Gcz,Gct)
legend('G','Gcz','Gct')
```

Bode Diagram

In this case, the Tustin method provides a better frequency-domain match between the discrete system and the interpolation. However, the Tustin interpolation method is undefined for systems with poles at $z = $ -1 (integrators), and is ill-conditioned for systems with poles near $z = 1$.

See Also
d2c | d2cOptions

Related Examples

- "Discretize a Compensator" on page 5-12

More About

- "Continuous-Discrete Conversion Methods" on page 5-25

Continuous-Discrete Conversion Methods

In this section...

Choosing a Conversion Method
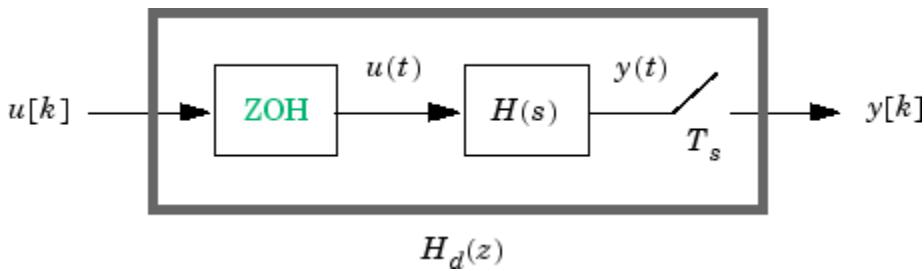
The c2d command discretizes continuous-time models. Conversely, d2c converts discrete-time models to continuous time. Both commands support several discretization and interpolation methods, as shown in the following table.

Discretization Method	Use when:
"Zero-Order Hold" on page 5-26	You want an exact discretization in the time domain for staircase inputs.
"First-Order Hold" on page 5-27	You want an exact discretization in the time domain for piecewise linear inputs.
"Impulse-Invariant Mapping" on page 5-28 (c2d only)	You want an exact discretization in the time domain for impulse train inputs.
"Tustin Approximation" on page 5-29	• You want good matching in the frequency domain between the continuous- and discrete-time models. • Your model has important dynamics at some particular frequency.
"Zero-Pole Matching Equivalents" on page 5-33	You have a SISO model, and you want good matching in the frequency domain between the continuous- and discrete-time models.

Zero-Order Hold

The Zero-Order Hold (ZOH) method provides an exact match between the continuous- and discrete-time systems in the time domain for staircase inputs.

The following block diagram illustrates the zero-order-hold discretization $H_d(z)$ of a continuous-time linear model $H(s)$

The ZOH block generates the continuous-time input signal $u(t)$ by holding each sample value $u(k)$ constant over one sample period:

$$u(t) = u[k], \qquad kT_s \le t \le (k+1)T_s$$

The signal $u(t)$ is the input to the continuous system $H(s)$. The output $y[k]$ results from sampling $y(t)$ every T_s seconds.

Conversely, given a discrete system $H_d(z)$, d2c produces a continuous system $H(s)$. The ZOH discretization of $H(s)$ coincides with $H_d(z)$.
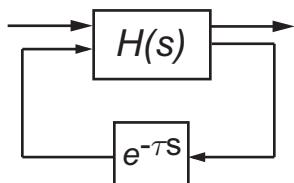
The ZOH discrete-to-continuous conversion has the following limitations:

- d2c cannot convert LTI models with poles at $z = 0$.

- For discrete-time LTI models having negative real poles, ZOH d2c conversion produces a continuous system with higher order. The model order increases because a negative real pole in the z domain maps to a pure imaginary value in the s domain. Such mapping results in a continuous-time model with complex data. To avoid this, the software instead introduces a conjugate pair of complex poles in the s domain. See "Convert Discrete-Time System to Continuous Time" on page 5-22 for an example.

ZOH Method for Systems with Time Delays
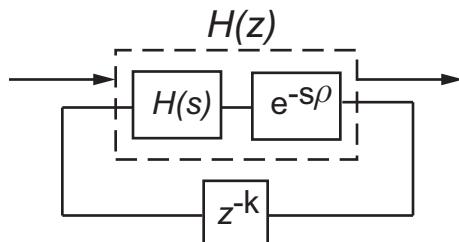
You can use the ZOH method to discretize SISO or MIMO continuous-time models with time delays. The ZOH method yields an exact discretization for systems with input delays, output delays, or transfer delays.

For systems with internal delays (delays in feedback loops), the ZOH method results in approximate discretizations. The following figure illustrates a system with an internal delay.

For such systems, c2d performs the following actions to compute an approximate ZOH discretization:

1 Decomposes the delay τ as $\tau = kT_s + \rho$ with $0 \le \rho < T_s$.

2 Absorbs the fractional delay ρ into $H(s)$.

3 Discretizes $H(s)$ to $H(z)$.

4 Represents the integer portion of the delay kT_s as an internal discrete-time delay z^{-k}. The final discretized model appears in the following figure:

First-Order Hold

The First-Order Hold (FOH) method provides an exact match between the continuous- and discrete-time systems in the time domain for piecewise linear inputs.

FOH differs from ZOH by the underlying hold mechanism. To turn the input samples $u[k]$ into a continuous input $u(t)$, FOH uses linear interpolation between samples:

$$u(t) = u[k] + \frac{t - kT_s}{T_s}\big(u[k+1] - u[k]\big), \quad kT_s \le t \le (k+1)T_s$$

This method is generally more accurate than ZOH for systems driven by smooth inputs.

This FOH method differs from standard causal FOH and is more appropriately called *triangle approximation* (see [2], p. 228). The method is also known as ramp-invariant approximation.
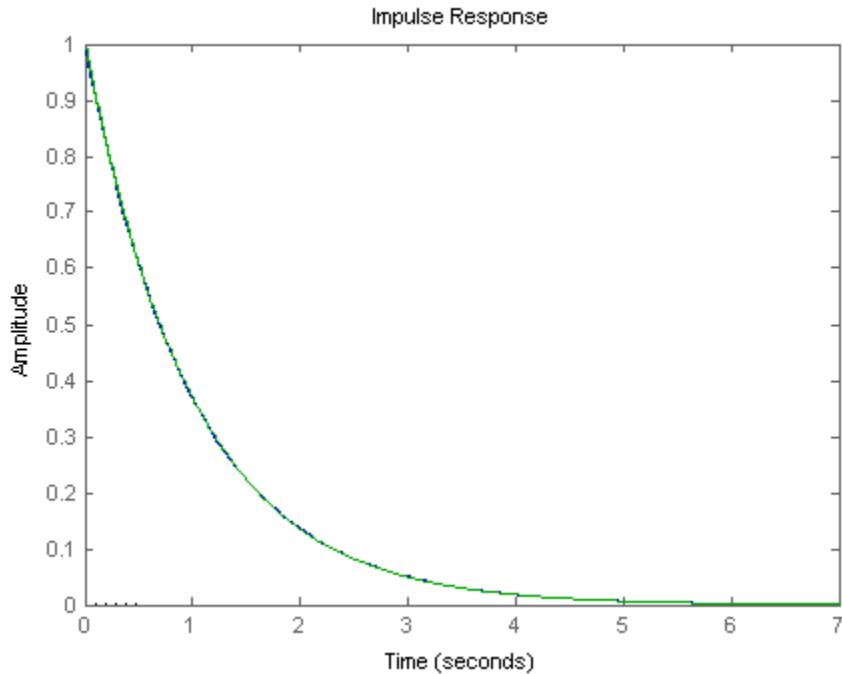
FOH Method for Systems with Time Delays

You can use the FOH method to discretize SISO or MIMO continuous-time models with time delays. The FOH method handles time delays in the same way as the ZOH method. See "ZOH Method for Systems with Time Delays" on page 5-27.

Impulse-Invariant Mapping

The impulse-invariant mapping produces a discrete-time model with the same impulse response as the continuous time system. For example, compare the impulse response of a first-order continuous system with the impulse-invariant discretization:

```
G = tf(1,[1,1]);
Gd1 = c2d(G,0.01,'impulse');
impulse(G,Gd1)
```

The impulse response plot shows that the impulse responses of the continuous and discretized systems match.

Impulse-Invariant Mapping for Systems with Time Delays

You can use impulse-invariant mapping to discretize SISO or MIMO continuous-time models with time delay, except that the method does not support `ss` models with internal delays. For supported models, impulse-invariant mapping yields an exact discretization of the time delay.

Tustin Approximation

The Tustin or bilinear approximation yields the best frequency-domain match between the continuous-time and discretized systems. This method relates the *s*-domain and *z*-domain transfer functions using the approximation:

$$z = e^{sT_s} \approx \frac{1 + sT_s/2}{1 - sT_s/2}.$$

In c2d conversions, the discretization $H_d(z)$ of a continuous transfer function $H(s)$ is:

$$H_d(z) = H(s'), \qquad s' = \frac{2}{T_s} \frac{z-1}{z+1}$$

Similarly, the d2c conversion relies on the inverse correspondence

$$H(s) = H_d(z'), \qquad z' = \frac{1 + sT_s/2}{1 - sT_s/2}$$

When you convert a state-space model using the Tustin method, the states are not preserved. The state transformation depends upon the state-space matrices and whether the system has time delays. For example, for an explicit ($E = I$) continuous-time model with no time delays, the state vector $w[k]$ of the discretized model is related to the continuous-time state vector $x(t)$ by:

$$w[kT_s] = \left(I - A\frac{T_s}{2} \right) x(kT_s) - \frac{T_s}{2} Bu(kT_s) = x(kT_s) - \frac{T_s}{2}\left(Ax(kT_s) + Bu(kT_s) \right).$$

T_s is the sample time of the discrete-time model. A and B are state-space matrices of the continuous-time model.

Tustin Approximation with Frequency Prewarping

If your system has important dynamics at a particular frequency that you want the transformation to preserve, you can use the Tustin method with frequency prewarping. This method ensures a match between the continuous- and discrete-time responses at the prewarp frequency.

The Tustin approximation with frequency prewarping uses the following transformation of variables:

$$H_d(z) = H(s'), \qquad s' = \frac{\omega}{\tan(\omega T_s/2)} \frac{z-1}{z+1}$$

This change of variable ensures the matching of the continuous- and discrete-time frequency responses at the prewarp frequency ω, because of the following correspondence:

$$H(j\omega) = H_d\left(e^{j\omega T_s}\right)$$
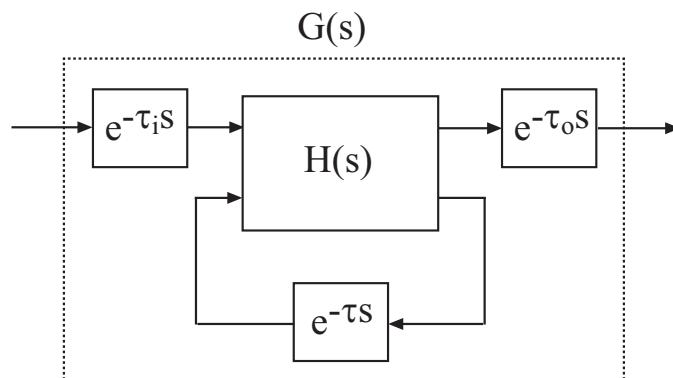
Tustin Approximation for Systems with Time Delays

You can use the Tustin approximation to discretize SISO or MIMO continuous-time models with time delays.
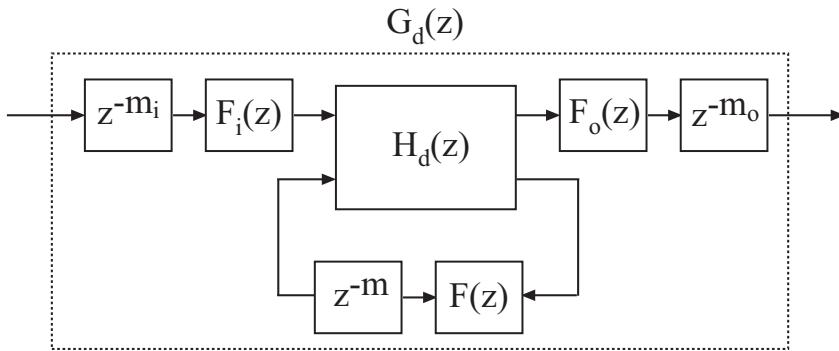
By default, the Tustin method rounds any time delay to the nearest multiple of the sample time. Therefore, for any time delay `tau`, the integer portion of the delay, `k*Ts`, maps to a delay of `k` sampling periods in the discretized model. This approach ignores the residual fractional delay, `tau - k*Ts`.

You can to approximate the fractional portion of the delay by a discrete all-pass filter (Thiran filter) of specified order. To do so, use the `FractDelayApproxOrder` option of `c2dOptions`. See "Improve Accuracy of Discretized System with Time Delay" on page 5-18 for an example.

To understand how the Tustin method handles systems with time delays, consider the following SISO state-space model $G(s)$. The model has input delay τ_i, output delay τ_o, and internal delay τ.

The following figure shows the general result of discretizing $G(s)$ using the Tustin method.

$$G_d(z)$$

By default, `c2d` converts the time delays to pure integer time delays. The `c2d` command computes the integer delays by rounding each time delay to the nearest multiple of the sample time T_s. Thus, in the default case, $m_i = \text{round}(\tau_i / T_s)$, $m_o = \text{round}(\tau_o / T_s)$, and $m = \text{round}(\tau / T_s)$.. Also in this case, $F_i(z) = F_o(z) = F(z) = 1$.

If you set `FractDelayApproxOrder` to a non-zero value, `c2d` approximates the fractional portion of the time delays by Thiran filters $F_i(z)$, $F_o(z)$, and $F(z)$.

The Thiran filters add additional states to the model. The maximum number of additional states for each delay is `FractDelayApproxOrder`.

For example, for the input delay τ_i, the order of the Thiran filter $F_i(z)$ is:
$\text{order}(F_i(z)) = \max(\text{ceil}(\tau_i / T_s), \texttt{FractDelayApproxOrder})$.

If $\text{ceil}(\tau_i / T_s) < $ `FractDelayApproxOrder`, the Thiran filter $F_i(z)$ approximates the entire input delay τ_i. If $\text{ceil}(\tau_i / T_s) > $ `FractDelayApproxOrder`, the Thiran filter only approximates a portion of the input delay. In that case, `c2d` represents the remainder of the input delay as a chain of unit delays z^{-m_i}, where $m_i = \text{ceil}(\tau_i / T_s) - $ `FractDelayApproxOrder`.

`c2d` uses Thiran filters and `FractDelayApproxOrder` in a similar way to approximate the output delay τ_o and the internal delay τ.

When you discretize`tf` and `zpk` models using the Tustin method, `c2d` first aggregates all input, output, and transfer delays into a single transfer delay τ_{TOT} for each channel. `c2d` then approximates τ_{TOT} as a Thiran filter and a chain of unit delays in the same way as described for each of the time delays in `ss` models.

For more information about Thiran filters, see the `thiran` reference page and [4].

Zero-Pole Matching Equivalents

The method of conversion by computing zero-pole matching equivalents applies only to SISO systems. The continuous and discretized systems have matching DC gains. Their poles and zeros are related by the transformation:

$$z_i = e^{s_i T_s}$$

where:

- z_i is the ith pole or zero of the discrete-time system.
- s_i is the ith pole or zero of the continuous-time system.
- T_s is the sample time.

See [2] for more information.

Zero-Pole Matching for Systems with Time Delays

You can use zero-pole matching to discretize SISO continuous-time models with time delay, except that the method does not support `ss` models with internal delays. The zero-pole matching method handles time delays in the same way as the Tustin approximation. See "Tustin Approximation for Systems with Time Delays" on page 5-31.

References

[1] Åström, K.J. and B. Wittenmark, *Computer-Controlled Systems: Theory and Design*, Prentice-Hall, 1990, pp. 48-52.

[2] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

[3] Smith, J.O. III, "Impulse Invariant Method", *Physical Audio Signal Processing*, August 2007. http://www.dsprelated.com/dspbooks/pasp/Impulse_Invariant_Method.html.

[4] T. Laakso, V. Valimaki, "Splitting the Unit Delay", *IEEE Signal Processing Magazine*, Vol. 13, No. 1, p.30-60, 1996.

See Also

c2d | c2dOptions | d2c | d2cOptions | d2d | thiran

Related Examples

- "Discretize a Compensator" on page 5-12
- "Improve Accuracy of Discretized System with Time Delay" on page 5-18
- "Convert Discrete-Time System to Continuous Time" on page 5-22

Upsample a Discrete-Time System

This example shows how to upsample a system using both the `d2d` and `upsample` commands and compares the results of both to the original system.

Upsampling a system can be useful, for example, when you need to implement a digital controller at a faster rate than you originally designed it for.

Create the discrete-time system $G(z) = \dfrac{z + 0.4}{z - 0.7}$ with a sample time of 0.3 s.

```
G = tf([1,0.4],[1,-0.7],0.3);
```

Resample the system at 0.1 s using `d2d`.

```
G_d2d = d2d(h1,0.1)

G_d2d =

  z - 0.4769
  ----------
  z - 0.8879

Sample time: 0.1 seconds
Discrete-time transfer function.
```

By default, `d2d` uses the zero-order-hold (ZOH) method to resample the system. The resampled system has the same order as `G`.

Resample the system again at 0.1 s, using `upsample`.

```
G_up = upsample(h1,3)

G_up =

  z^3 + 0.4
  ---------
  z^3 - 0.7
```
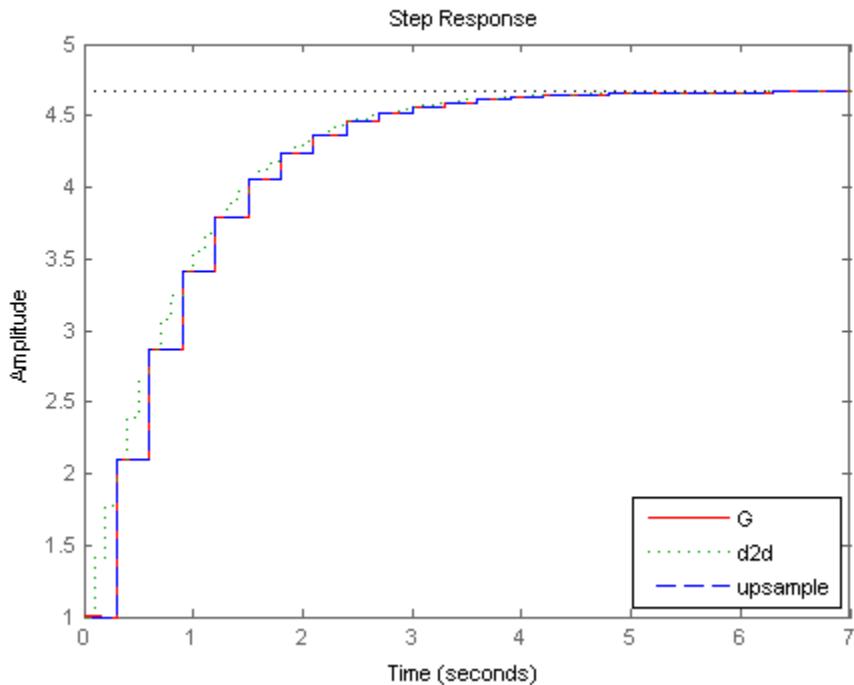
The second input, 3, tells `upsample` to resample `G` at a sample time three times faster than the sample time of `G`. This input to `upsample` must be an integer.

`G_up` has three times as many poles and zeroes as `G`.

5-35

Compare the step responses of the original model G with the resampled models G_d2d and G_up.
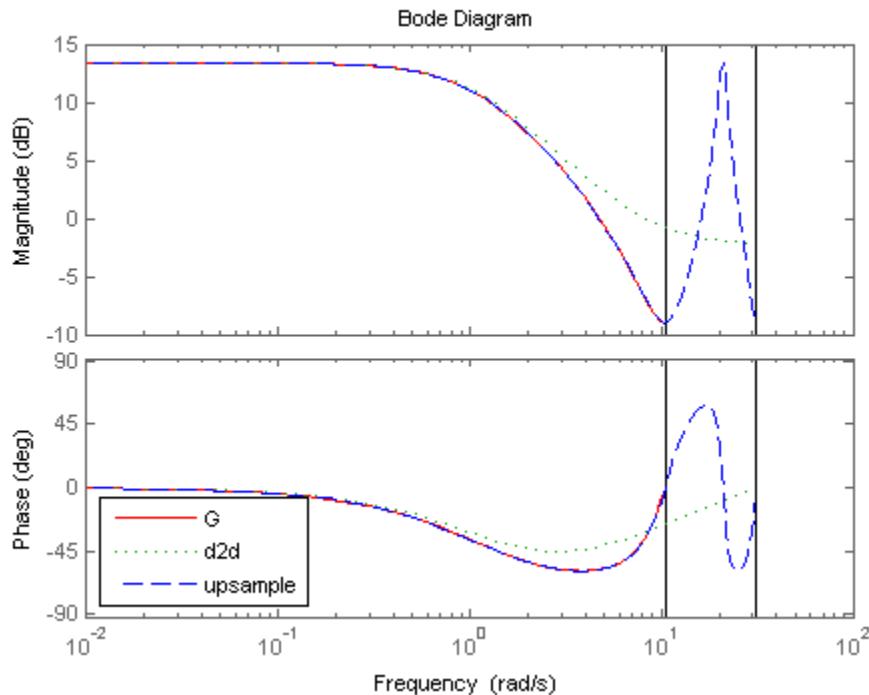
```
step(G,'-r',G_d2d,':g',G_up,'--b')
legend('G','d2d','upsample','Location','SouthEast')
```


The step response of the upsampled model G_up matches exactly the step response of the original model G. The response of the resampled model G_d2d matches only at every third sample.

Compare the frequency response of the original model with the resampled models.

```
bode(G,'-r',G_d2d,':g',G_up,'--b')
legend('G','d2d','upsample','Location','SouthWest')
```

In the frequency domain as well, the model G_up created with the upsample command matches the original model exactly up to the Nyquist frequency of the original model.

Using upsample provides a better match than d2d in both the time and frequency domains. However, upsample increases the model order, which can be undesirable. Additionally, upsample is only available where the original sample time is an integer multiple of the new sample time.

See Also
d2d | d2dOptions | upsample

More About
• "Choosing a Resampling Command" on page 5-38

Choosing a Resampling Command

You can resample a discrete-time model using the commands described in the following table.

To...	Use the command...
• Downsample a system. • Upsample a system without any restriction on the new sample time.	d2d
Upsample a system with the highest accuracy when: • The new sample time is integer-value-times faster than the sample time of the original model. • Your new model can have more states than the original model.	upsample

See Also

d2d | d2dOptions | upsample

Related Examples

• "Upsample a Discrete-Time System" on page 5-35

6

Model Simplification

Model Reduction Basics
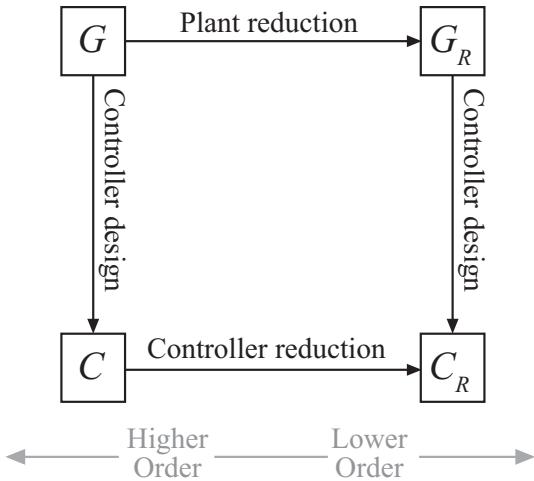
Working with lower-order models can simplify analysis and control design, relative to higher-order models. Simpler models are also easier to understand and manipulate. High-order models obtained by linearizing complex Simulink models or from other sources can contain states that do not contribute much to the dynamics of particular interest to your application. Therefore, it can be useful to reduce model order while preserving model characteristics that are important for your application.

When to Reduce Model Order

Cases where you might want to reduce model order include these situations:

- You are working with a relatively high-order model obtained from linearizing a Simulink model, performing a finite-element calculation, interconnecting model elements, or other source.

- You want to improve the simulation speed of a Simulink model at a certain operating point. In that case, you can linearize a portion of the model at that operating point and compute a reduced-order simplification or approximation of the linearized model. You can then replace the portion of the model with an LTI Block containing the reduced-order model.

- You design a high-order controller that you want to implement as a lower-order controller, such as a PID controller. For example, controller design using Linear-Quadratic-Gaussian methods or H_∞ synthesis techniques can yield a high-order result. In this case, you can try reducing the plant order before synthesis, reducing the controller order after synthesis, or both.

- You want to simplify a model obtained by identification with System Identification Toolbox software.

The following diagram illustrates the relationship between model reduction and control design.

In general, when designing a controller for a system represented by a high-order model, G, it is useful start by simplifying the plant model. Then, design a relatively low-order controller, C_R, for the lower-order plant model G_R. After you design a controller for either the original or the reduced plant model, you can try to reduce the controller further.

Reducing the plant or controller can include:

- Discarding states that do not contribute to the system dynamics, such as structurally disconnected states or canceling pole-zero pairs.

- Discarding low-energy states that contribute relatively little to system dynamics.

- Focusing on a particular frequency region and discarding dynamics outside that region. For example, if your control bandwidth is limited by actuator dynamics, discard higher-frequency dynamics.

In any case, when you reduce model order, you want to preserve model characteristics that are important for your application. Whenever you compute a reduced-order model, verify that the reduced model preserves time-domain or frequency-domain behavior that you care about. For example, for control design, it is useful to verify that the reduced closed-loop system is stable. It is also useful to check that the reduced open-loop transfer function $C_R G_R$ adequately matches the original models where the open-loop gain GC is close to 1 (in the gain crossover region).

Choosing a Model Reduction Method

To reduce the order of a model, you can either simplify your model, or compute a lower-order approximation. The following table summarizes the differences among several model-reduction approaches.

Approach	Command Line	Model Reducer App
Simplification — Reduce model order exactly by canceling pole-zero pairs or eliminating states that have no effect on the overall model response	• `sminreal` — Eliminate states that are structurally disconnected from the inputs or outputs. • `minreal` — Eliminate canceling or near-canceling pole-zero pairs from transfer functions. Eliminate unobservable or uncontrollable states from state-space models.	"Pole-Zero Simplification" on page 6-45 method — Eliminate: • Structurally disconnected states • Unobservable or uncontrollable states from state-space models • Canceling or near-canceling pole-zero pairs from transfer functions
Approximation — Compute a lower-order approximation of your model.	`balred` — Discard states that have relatively low effect on the overall model response.	Balanced Truncation method — Discard states that have relatively low effect on the overall model response.
Mode Selection — Eliminate poles and zeros that fall outside a specific frequency range of interest.	`freqsep` — Separate model into slow and fast dynamics around a specified cutoff frequency.	Mode Selection method — Select frequency range of interest and discard dynamics outside that range.

Sometimes, approximation can yield better results, even if the model looks like a good candidate for simplification. For example, models with near pole-zero cancelations are sometimes better reduced by approximation than simplification. Similarly, using `balred` to reduce state-space models can yield more accurate results than `minreal`.

When you use a reduced-order model, always verify that the simplification or approximation preserves model characteristics that are important for your application. For example, compare the frequency responses of the original and reduced models using `bodeplot` or `sigmaplot`. Or, compare the open-loop responses for the original and reduced plant and controller models.

See Also
balred | freqsep | minreal | Model Reducer | sminreal

Related Examples
- "Balanced Truncation Model Reduction" on page 6-16
- "Mode-Selection Model Reduction" on page 6-55
- "Pole-Zero Simplification" on page 6-45

Reduce Model Order Using the Model Reducer App
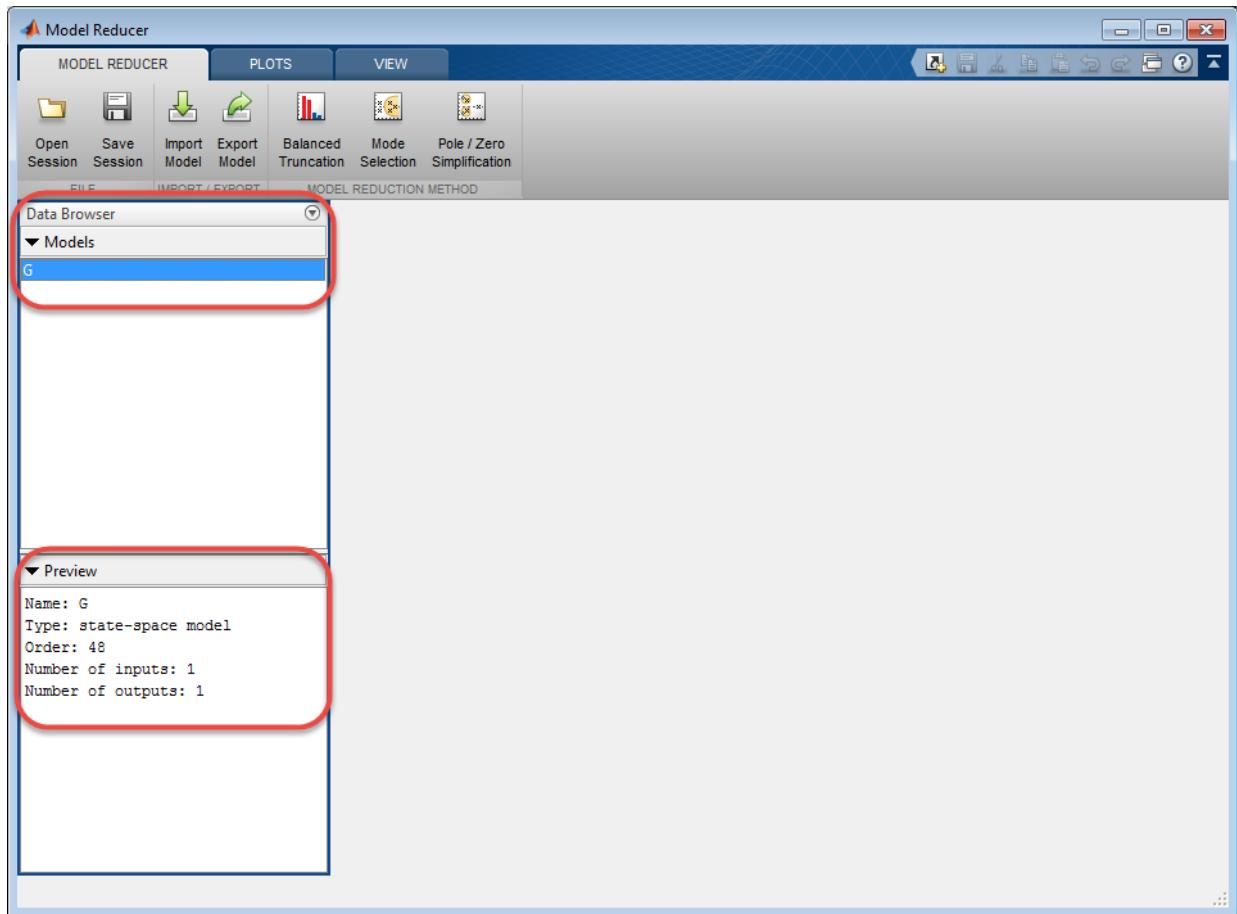
This example shows how to reduce model order while preserving important dynamics using the **Model Reducer** app. This example illustrates the Balanced Truncation method, which eliminates states based on their energy contributions to the system response.

Open Model Reducer With a Building Model

This example uses a model of the Los Angeles University Hospital building. The building has eight floors, each with three degrees of freedom: two displacements and one rotation. The input-output relationship for any one of these displacements is represented as a 48-state model, where each state represents a displacement or its rate of change (velocity). Load the building model and open **Model Reducer** with that model.
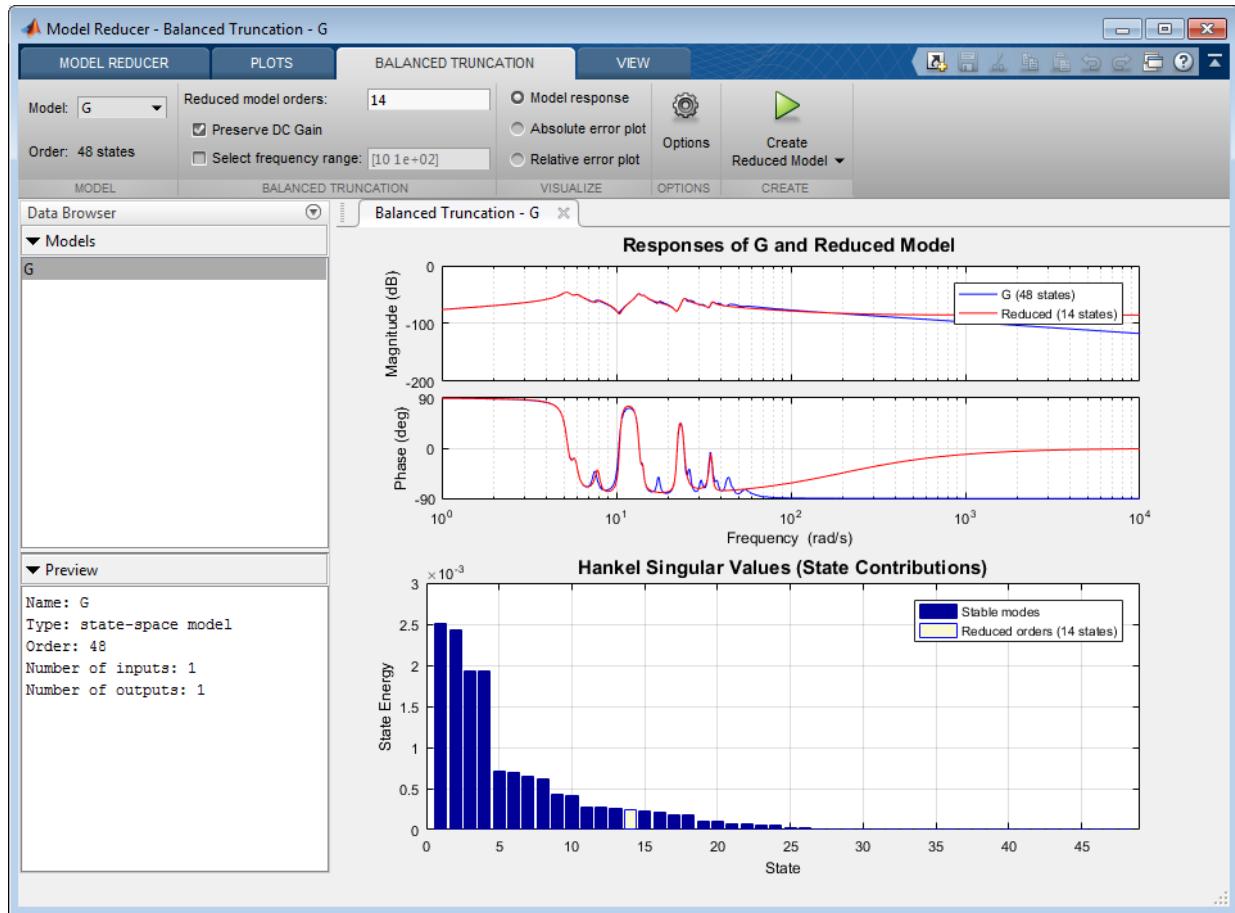
```
load build.mat
modelReducer(G)
```

Select the model in the Data Browser to display some information about the model in the Preview section. Double-click the model to see more detailed information.

Open the Balanced Truncation Tab

Model Reducer has three model reduction methods: Balanced Truncation, Mode Selection, and Pole/Zero Simplification. For this example, click **Balanced Truncation**.

Model Reducer opens the **Balanced Truncation** tab and automatically generates a reduced-order model. The top plot compares the original and reduced model in the frequency domain. The bottom plot shows the energy contribution of each state, where the states are sorted from high energy to low energy. The order of the reduced model, 14, is highlighted in the bar chart. In the reduced model, all states with lower energy contribution than this one are discarded.

Compute Multiple Approximations
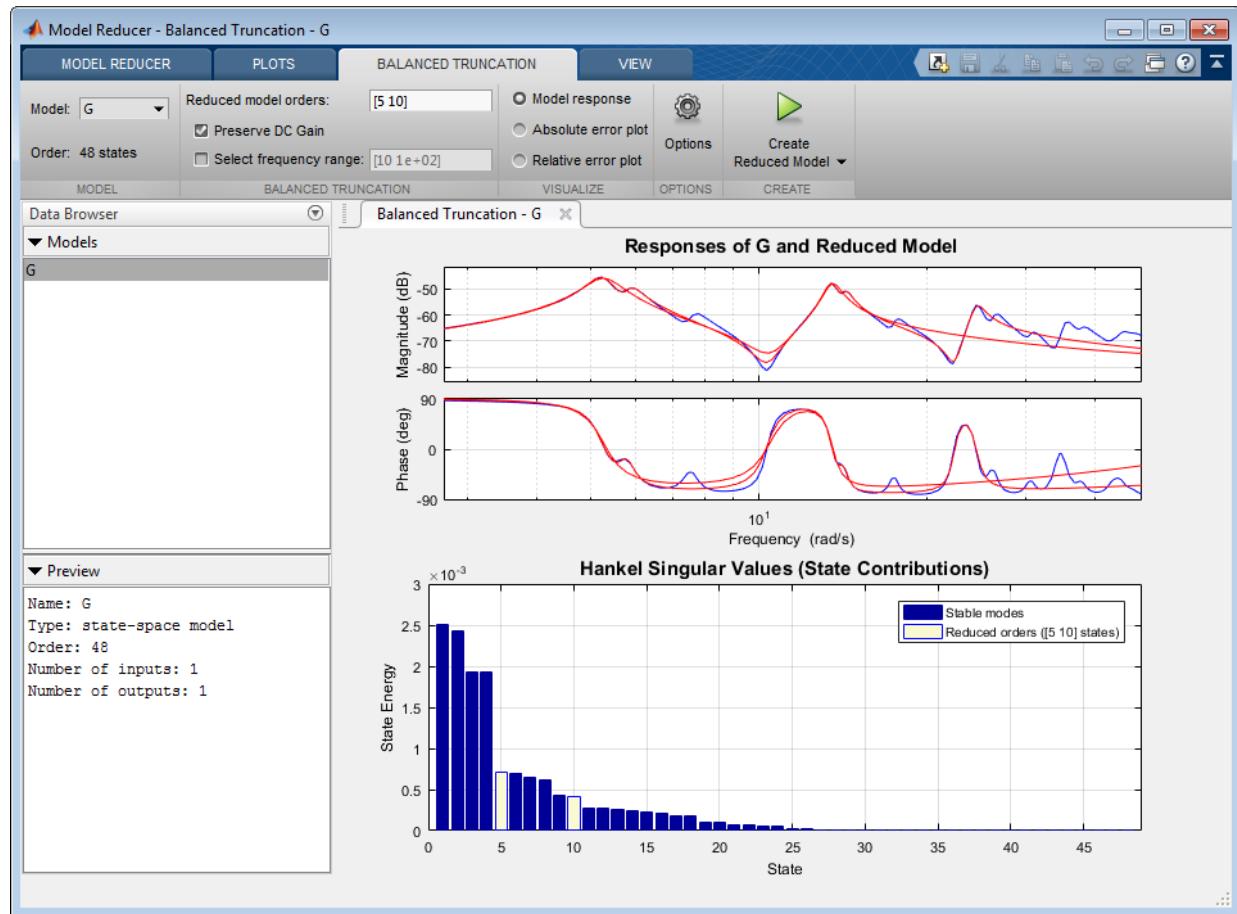
Suppose that you want to preserve the first, second, and third peaks of the model response, around 5.2 rad/s, 13 rad/s, and 25 rad/s. Try other model orders to see whether

you can achieve this goal with a lower model order. Compute a 5th-order and a 10th-order approximation in one of the following ways:

- In the **Reduced model orders** text box, enter `[5 10]`.
- In the state-contribution plot, ctrl-click the bars for state 5 and state 10.
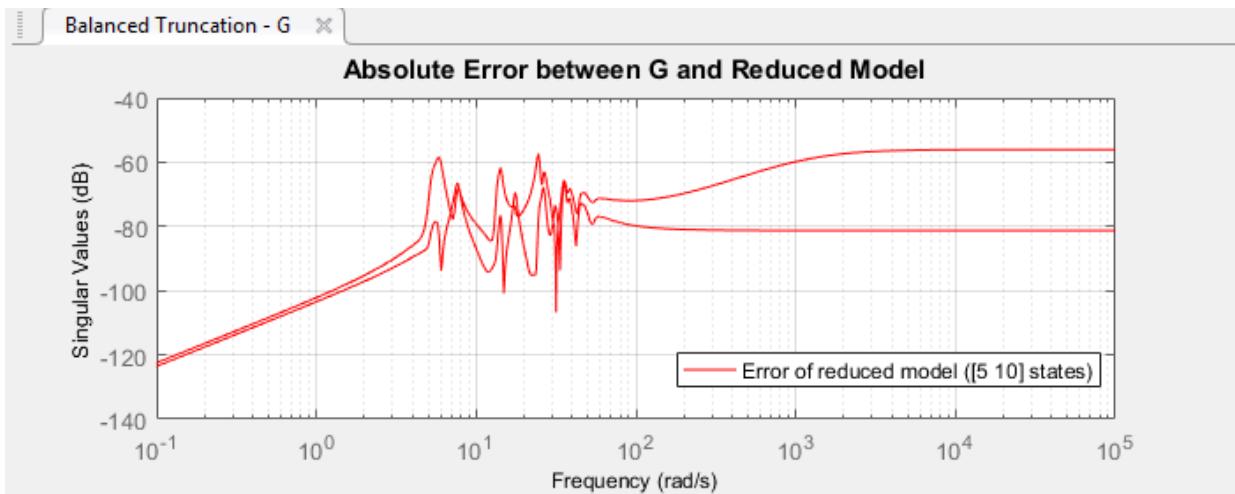
Model Reducer computes two new reduced-order models and displays them on the response plot with the original model `G`. To examine the three peaks more closely, Zoom in on the relevant frequency range. The 10th-order model captures the three peaks successfully, while the 5th-order model only approximates the first two peaks. (For information about zooming and other interactions with the analysis plots, see "Visualize Reduced-Order Models in the Model Reducer App".)

Compare Reduced Models With Different Visualizations
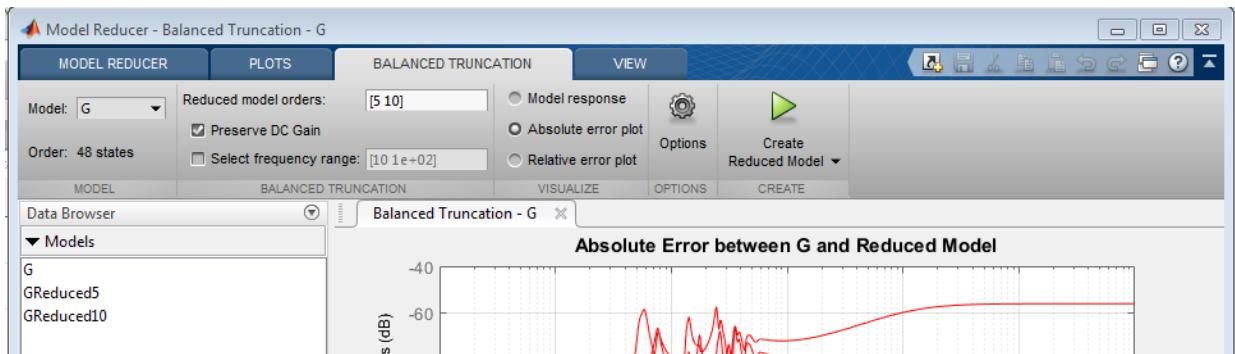
In addition to the frequency response plot of all three models, **Model Reducer** lets you examine the absolute and relative error between the original and reduced models. Select **Absolute error plot** to see the difference between the building and reduced models.

The 5th-order reduced model has at most -60dB error in the frequency region of the first two peaks, below about 30 rad/s. The error increases at higher frequencies. The 10th-order reduced model has smaller error over all frequencies.

Create Reduced Models in Data Browser
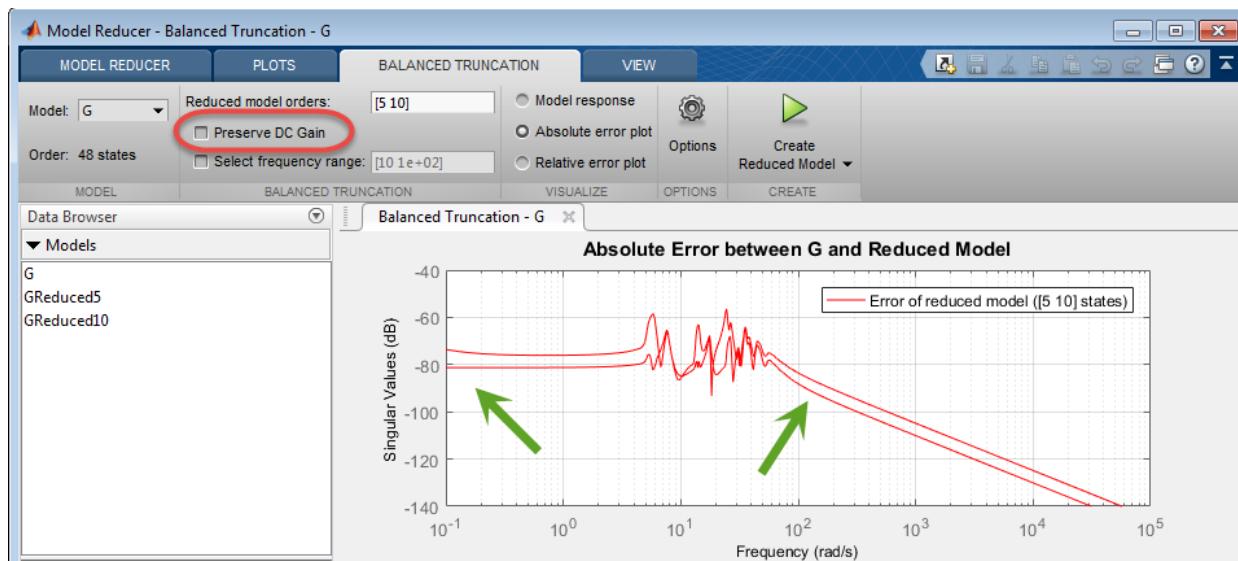
Store the reduced models in the Data Browser by clicking **Create Reduced Model**. The 5th-order and 10th-order reduced models appear in the Data Browser with names `GReduced5` and `Greduced10`.

You can continue to change the model-reduction parameters and generate additional reduced models. As you do so, `GReduced5` and `Greduced10` remain unchanged in the Data Browser.

Focus on Dynamics at Particular Frequencies
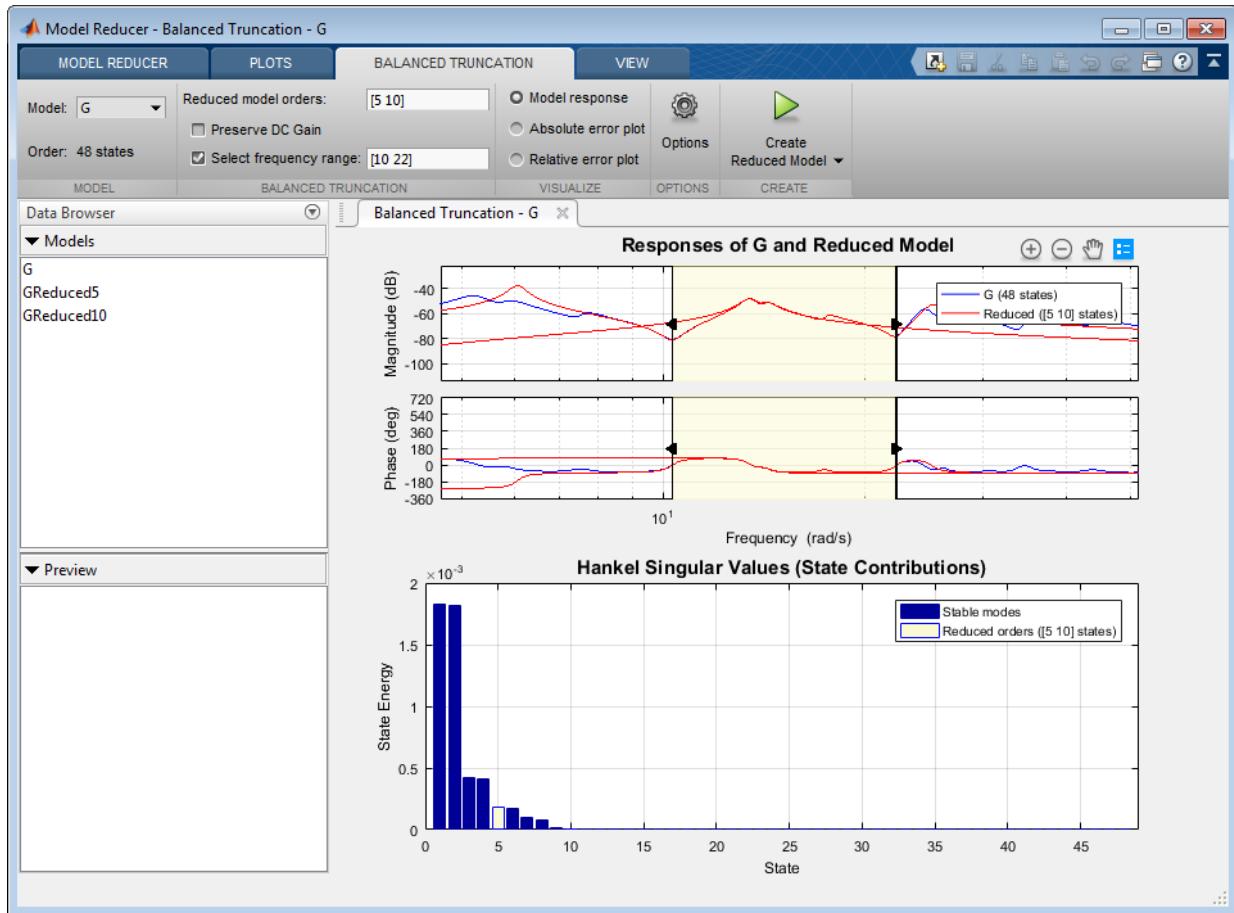
By default, balanced truncation in **Model Reducer** preserves DC gain, matching the steady-state response of the original and reduced models. Clear the **Preserve DC Gain** checkbox to better approximate high-frequency dynamics. **Model Reducer** computes new reduced models. The error in the high-frequency region is decreased at the cost of a slight increase in error at low frequencies.

You can also focus the balanced truncation on the model dynamics in a particular frequency interval. For example, approximate only the second peak of the building model around 13 rad/s. First, select the **Model response** plot to see the Bode plots of models. Then check **Select frequency range** checkbox. **Model Reducer** analyzes state contributions in the highlighted frequency interval only.

You can drag the boundaries to change the frequency range interactively. As you change the frequency interval, the Hankel Singular Value plot reflects the changes in the energy contributions of the states.

Enter the frequency limits [10 22] into the text box next to **Select frequency range**. The 5th-order reduced model captures the essential dynamics. The 10th-order model has almost the same dynamics as the original building model within this frequency range.
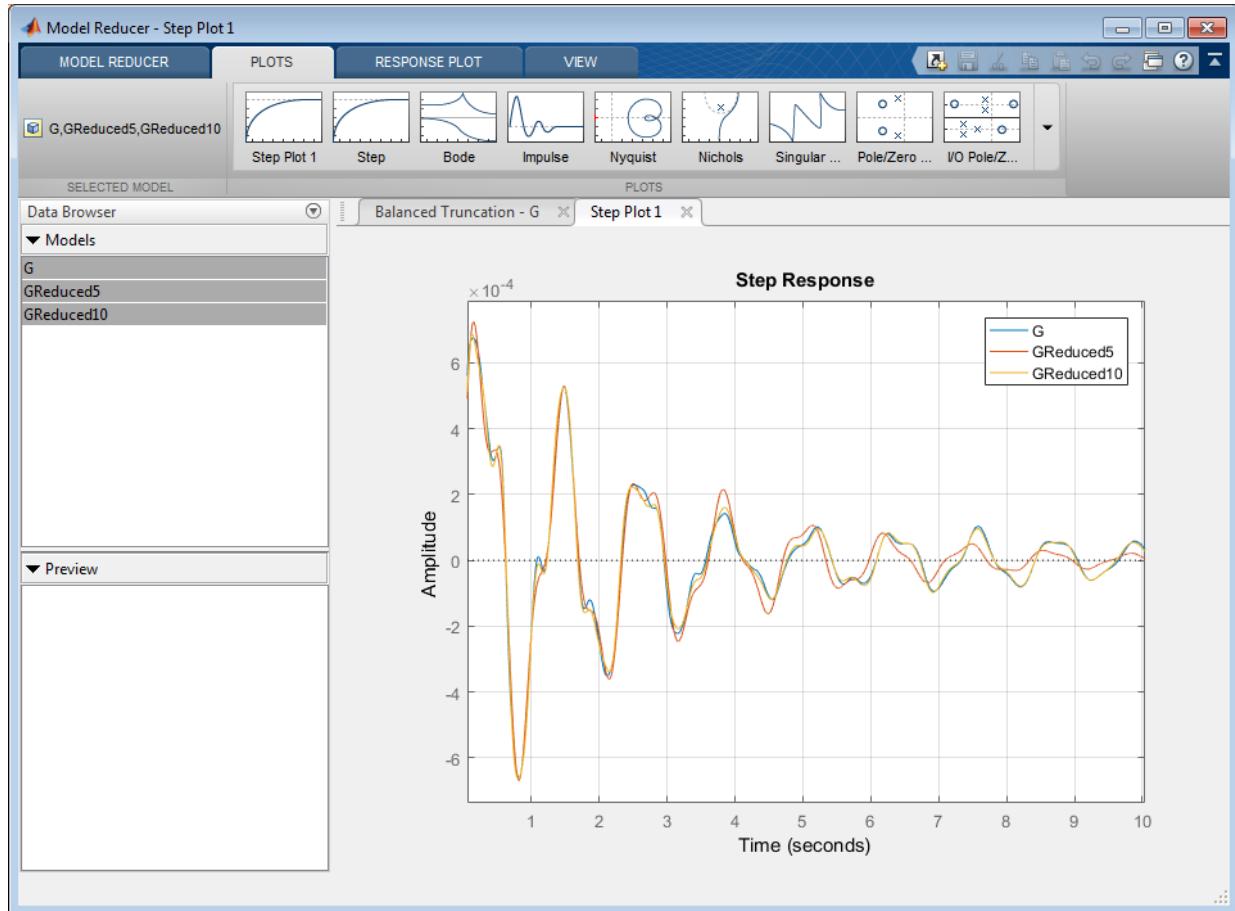
Optionally, store these additional models in the Data Browser by clicking **Create Reduced Model**.

Compare Models In Time Domain

You can compare time-domain responses of the stored reduced models and the original in the **Plots** tab. In the Data Browser, control-click to select the models you want to compare, G, GReduced5, and GReduced10. Then, click Step. **Model Reducer** creates a step plot with all three models.
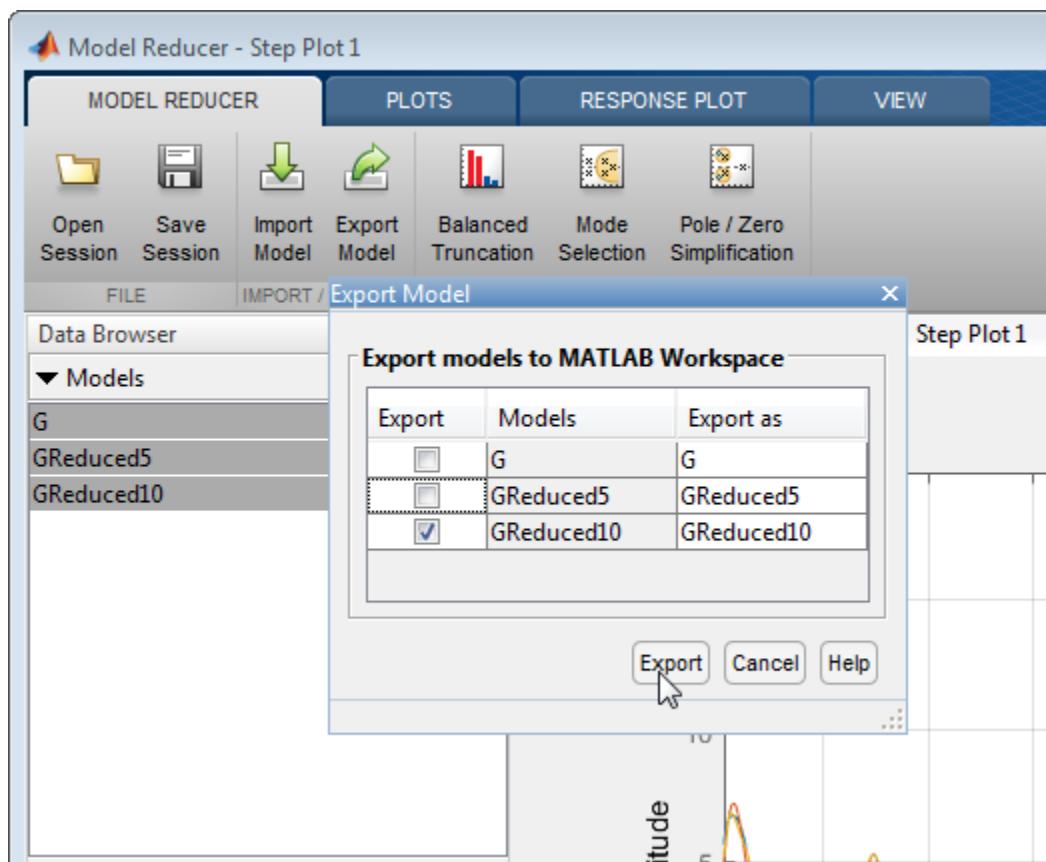
Zooming on the transient behavior of this plot shows that GReduced10 captures the time domain behavior of the original model well. However, the response of GReduced5 deviates from the original model after about 3 seconds.

Export Model for Further Analysis

Comparison of the reduced and original models in the time and frequency domains shows that GReduced10 adequately captures the dynamics of interest. Export that model to the MATLAB® workspace for further analysis and design. In the **Model Reducer** tab, click **Export Model**. Clear the check boxes for G and Greduced5, and click **Export** to export Greduced10.

Greduced10 appears in the MATLAB workspace as a state-space (ss) model.

See Also

Model Reducer

Related Examples

- "Model Reduction Basics" on page 6-2
- "Balanced Truncation Model Reduction" on page 6-16
- "Pole-Zero Simplification" on page 6-45
- "Mode-Selection Model Reduction" on page 6-55

Balanced Truncation Model Reduction

Balanced truncation computes a lower-order approximation of your model by neglecting states that have relatively low effect on the overall model response. Using a lower-order approximation that preserves the dynamics of interest can simplify analysis and control design. In the balanced truncation method of model reduction, the software measures state contributions by Hankel singular values (see `hsvd`) and discards states with smaller values. There are two ways to compute a reduced-order model by balanced truncation:

- At the command line, using the `balred` command.
- In the **Model Reducer**, using the **Balanced Truncation** method.

For more general information about model reduction, see "Model Reduction Basics" on page 6-2.

In this section...
"Balanced Truncation in the Model Reducer App" on page 6-16
"Approximate Model by Balanced Truncation at the Command Line" on page 6-25
"Compare Truncated and DC Matched Low-Order Model Approximations" on page 6-29
"Approximate Model with Unstable or Near-Unstable Pole" on page 6-34
"Frequency-Limited Balanced Truncation" on page 6-39
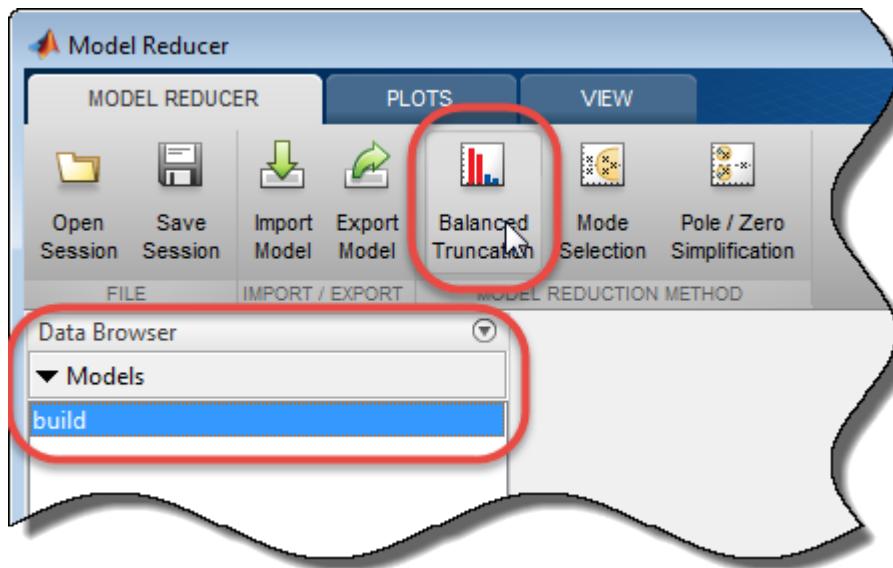
Balanced Truncation in the Model Reducer App

Model Reducer provides an interactive tool for performing model reduction and examining and comparing the responses of the original and reduced-order models. To approximate a model by balanced truncation in **Model Reducer**:
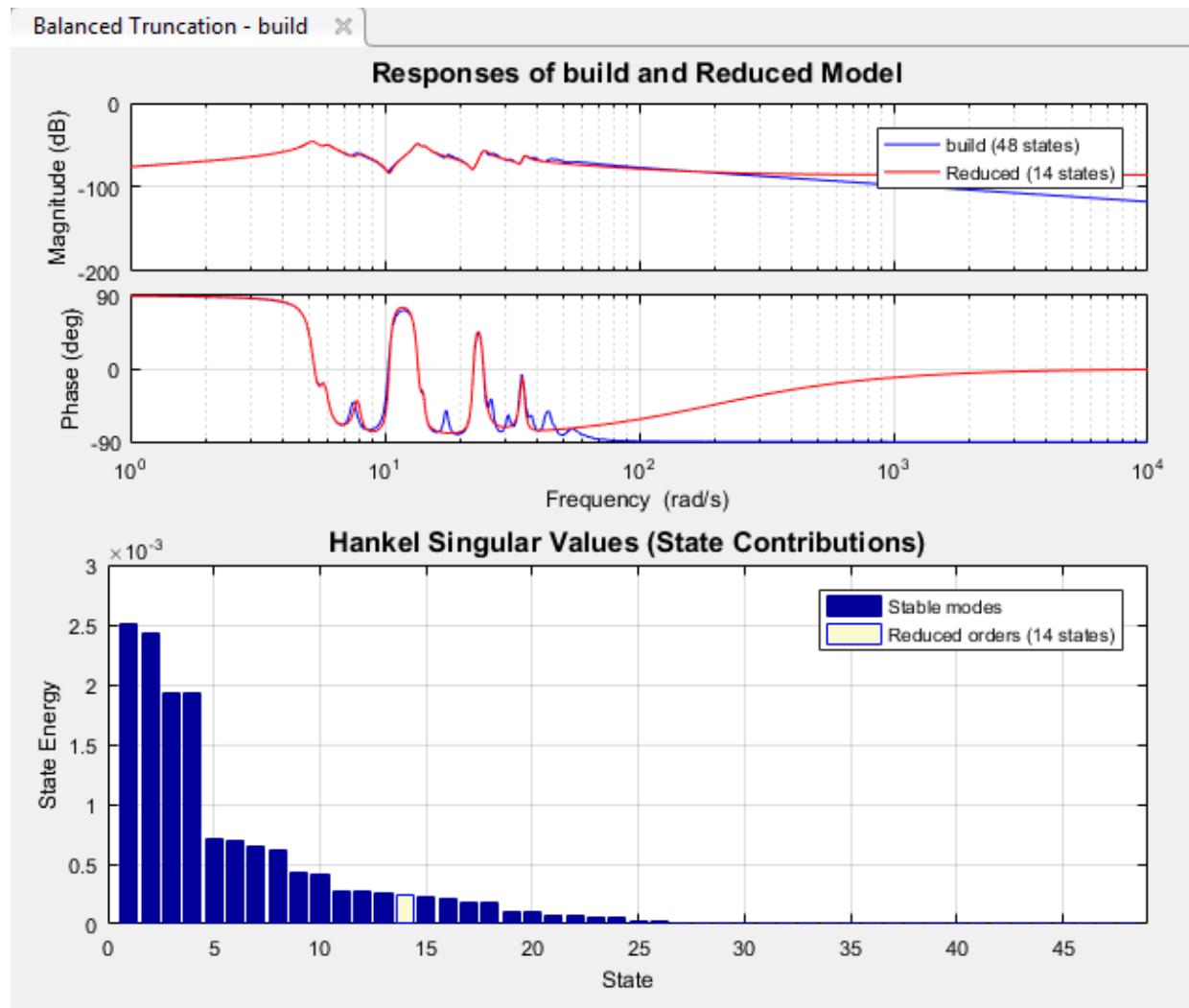
1 Open the app, and import an LTI model to reduce. For instance, suppose that there is a model named `build` in the MATLAB workspace. The following command opens **Model Reducer** and imports the model.

   ```
   modelReducer(build)
   ```

2 In the **Data Browser**, select the model to reduce. Click ▮▮▮ **Balanced Truncation**.

In the **Balanced Truncation** tab, **Model Reducer** displays a plot of the frequency response of the original model and a reduced version of the model. The frequency response is a Bode plot for SISO models, and a singular-value plot for MIMO models. The app also displays a Hankel singular-value plot of the original model.
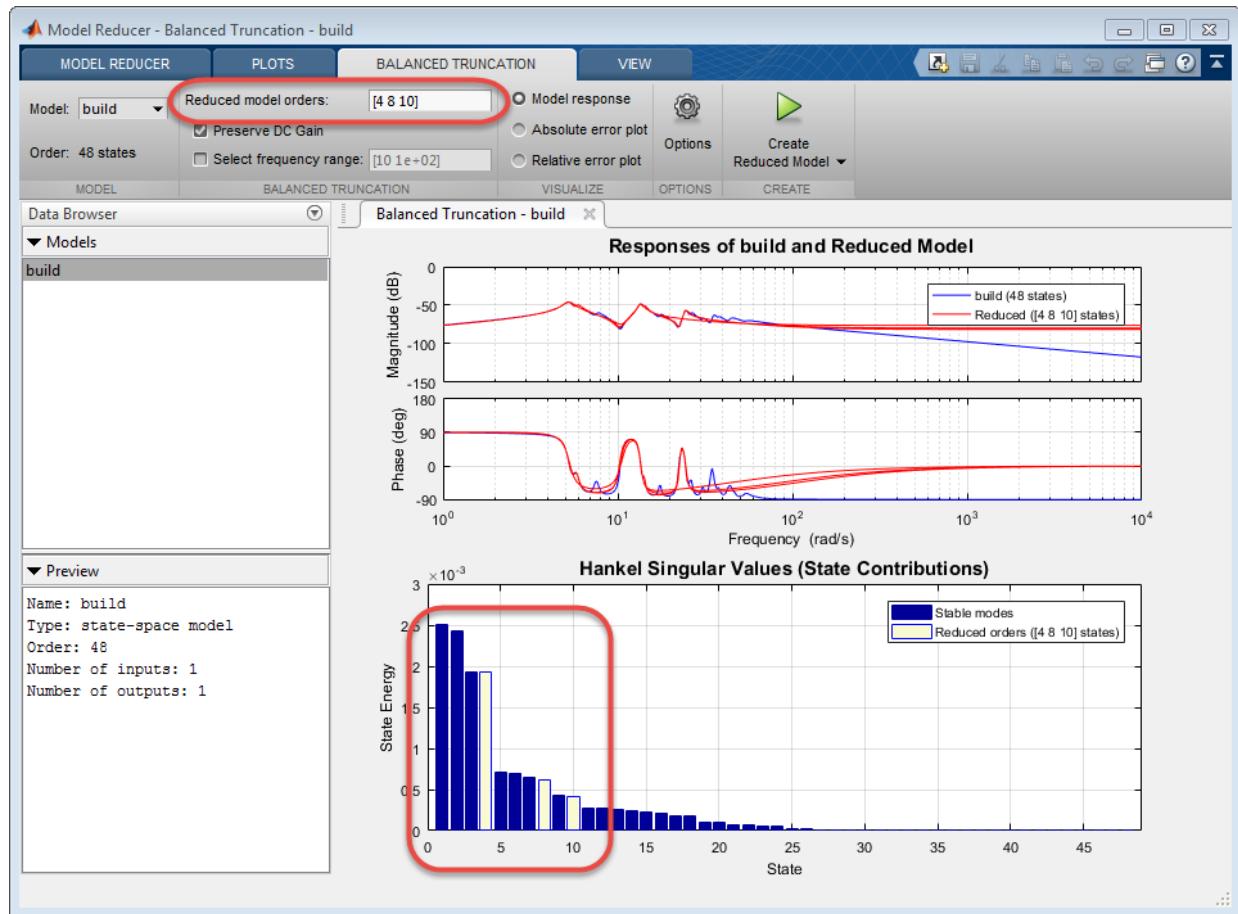
The Hankel singular-value plot shows the relative energy contributions of each state in the system. **Model Reducer** computes an initial reduced-order model based on these values. The highlighted bar is the lowest-energy state in the initial reduced-order model. **Model Reducer** discards states that have lower Hankel singular values than the highlighted bar.
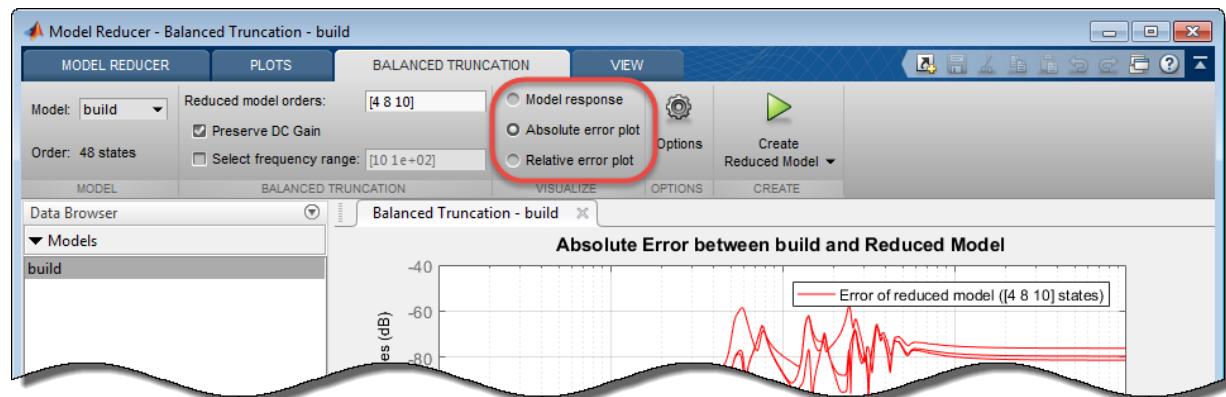
3   Try different reduced-model orders to find the lowest-order model that preserves the dynamics that are important for your application. To specify different orders, either:

   •   Enter model orders in the **Reduced model orders** field. You can enter a single integer or an array of integers, such as `10:14` or `[8,11,12]`.

   •   Click a bar on the Hankel singular-value plot to specify the lowest-energy state of the reduced-order model. Ctrl-click to specify multiple values.

When you change the specified reduced model order, **Model Reducer** automatically computes a new reduced-order model. If you specify multiple model orders, **Model Reducer** computes multiple reduced-order models and displays their responses on the plot.
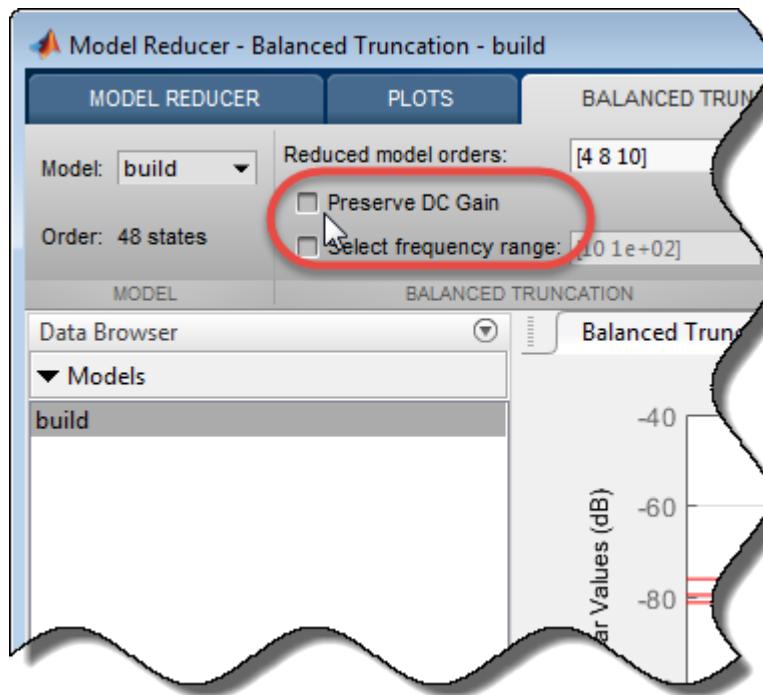
4   Optionally, examine the absolute or relative error between the original and reduced-order model, in addition to the frequency response. Select the error-plot type using the buttons on the **Balanced Truncation** tab.

For more information about using the analysis plots, see "Visualize Reduced-Order Models in the Model Reducer App" on page 6-65.

5   If low-frequency dynamics are not important to your application, you can clear the **Preserve DC Gain** checkbox. Doing so sometimes yields a better match at higher frequencies between the original and reduced-order models.
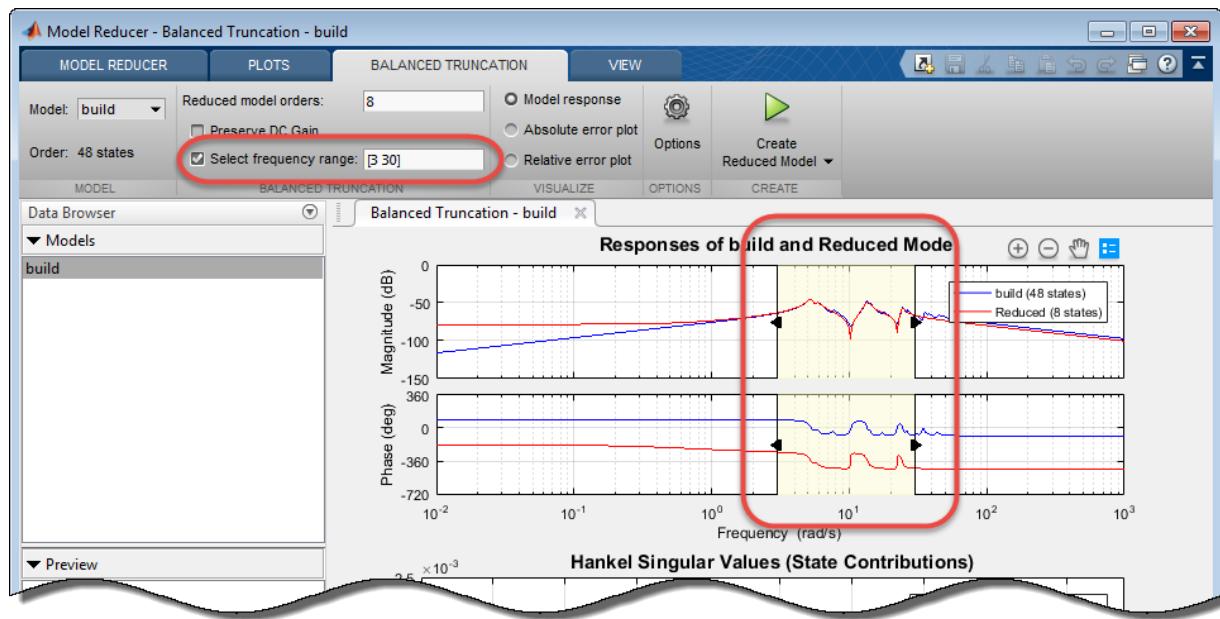
When you check or clear the **Preserve DC Gain** checkbox, **Model Reducer** automatically computes new reduced-order models. For more information about this option, see "Compare Truncated and DC Matched Low-Order Model Approximations" on page 6-29.

6 Optionally, limit the Hankel singular-value computation to a specific frequency range. Such a limit is useful when the model has modes outside the region of interest to your particular application. When you apply a frequency limit, **Model Reducer** determines which states to truncate based on their energy contribution within the specified frequency range only. Neglecting energy contributions outside that range can yield an even lower-order approximation that is still adequate for your application.

To limit the singular-value computation, check **Select frequency range**. Then, specify the frequency range by:

- In the text box, entering a vector of the form [fmin,fmax]. Units are rad/TimeUnit, where TimeUnit is the TimeUnit property of the model you are reducing.

- On the response plot or error plot, dragging the boundaries of the shaded region or the shaded region itself. **Model Reducer** analyzes the state contributions within the shaded region only.
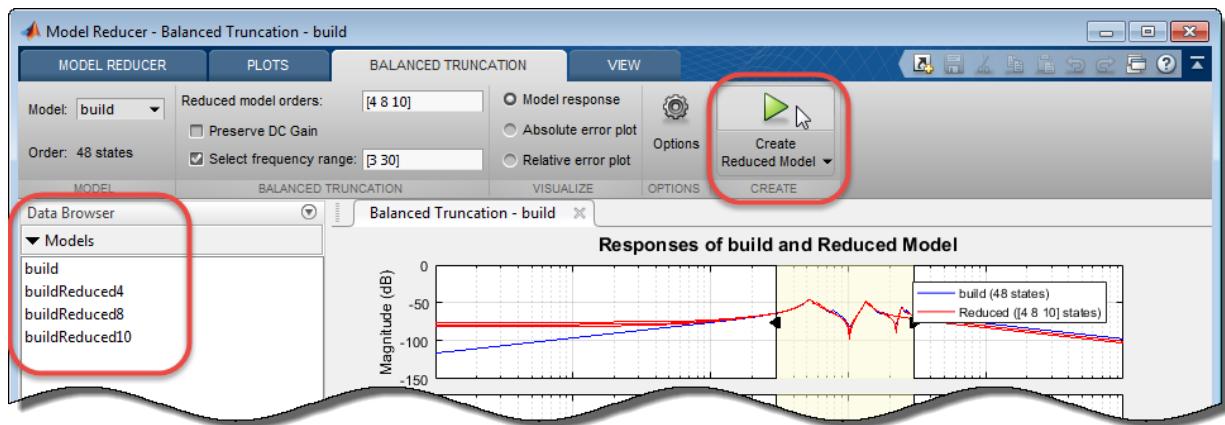
When you check or clear the **Select frequency range** checkbox or change the selected range, **Model Reducer** automatically computes new reduced-order models.

Note: Checking **Select frequency range** automatically clears **Preserve DC Gain**. To enforce a DC match even when using frequency limits, recheck **Preserve DC Gain**.

7 When you have one or more reduced models that you want to store and analyze

further, click ▶. The new models appear in the **Data Browser**. If you have

specified multiple orders, each reduced model appears separately. Model names reflect the reduced model order.
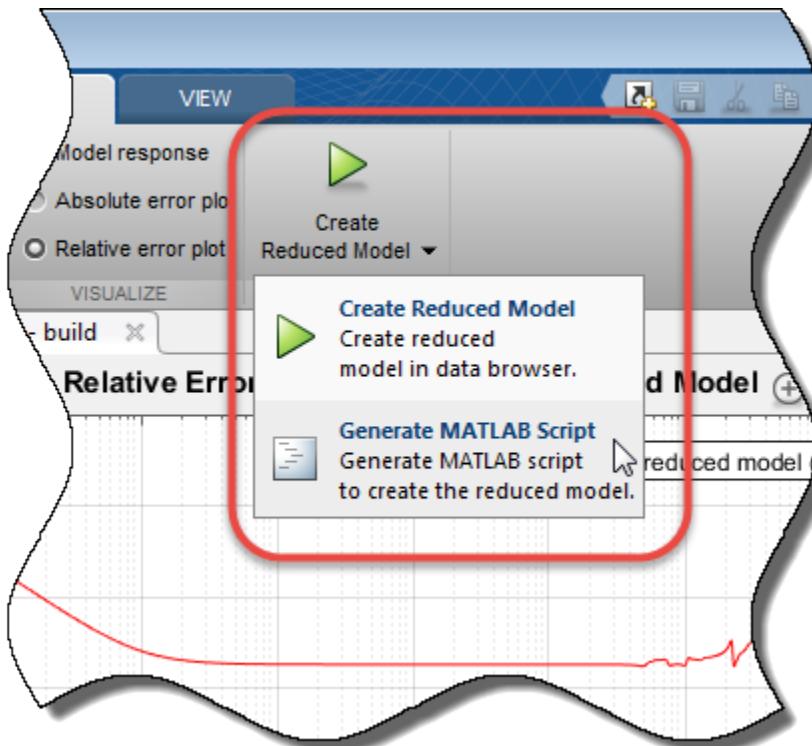
After creating reduced models in the **Data Browser**, you can continue changing the reduction parameters and create reduced models with different orders for analysis and comparison.

You can now perform further analysis with the reduced model. For example:

- Examine other responses of the reduced system, such as the step response or Nichols plot. To do so, use the tools on the **Plots** tab. See "Visualize Reduced-Order Models in the Model Reducer App" on page 6-65 for more information.

- Export reduced models to the MATLAB workspace for further analysis or control design. On the **Model Reducer** tab, click ⬛ **Export**.

Generate MATLAB Code for Balanced Truncation

To create a MATLAB script you can use for further model-reduction tasks at the command line, click **Create Reduced Model**, and select **Generate MATLAB Script**.

Model Reducer creates a script that uses the `balred` command to perform model reduction with the parameters and options you have set on the **Balanced Truncation** tab. The script opens in the MATLAB editor.

Approximate Model by Balanced Truncation at the Command Line

At the command-line, use `balred` to compute a reduced-order approximation of a model.

To do so, first examine the contribution of the various states to the overall model behavior. Choose the approximation order based on the number of states that make a significant contribution to the overall model behavior.

For this example, load a high-order model. `hplant` is a 23rd-order SISO model.
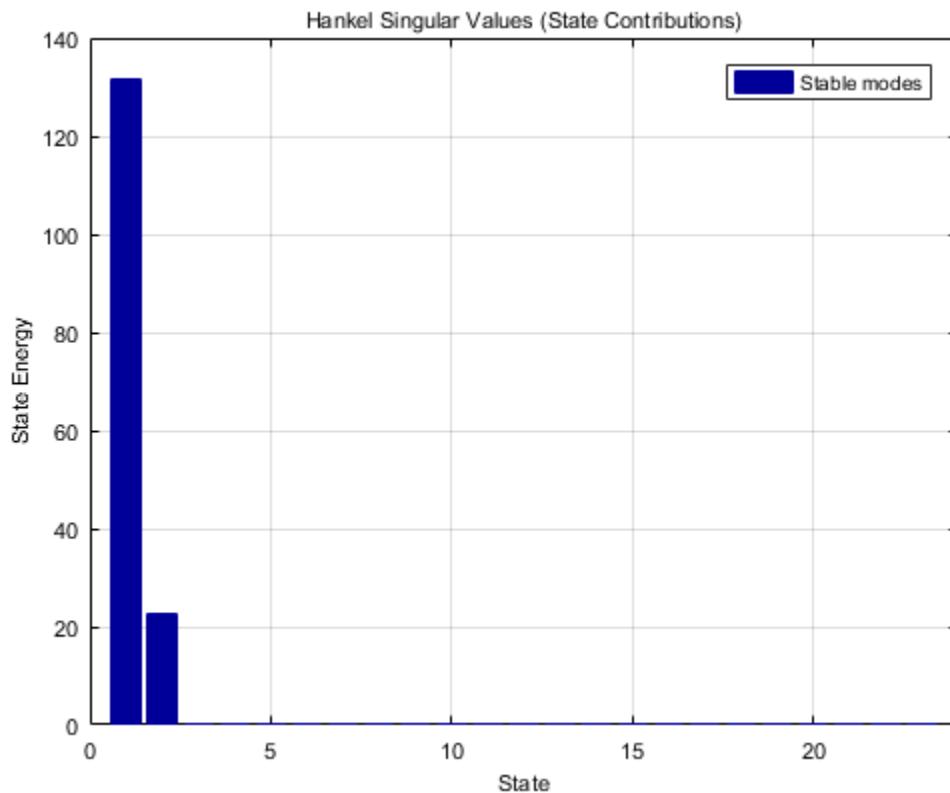
```
load ltiexamples hplant
order(hplant)
```

```
ans =

    23
```

Examine the relative amount of energy per state in `hplant` using a Hankel singular-value (HSV) plot.

```
hsvplot(hplant)
```
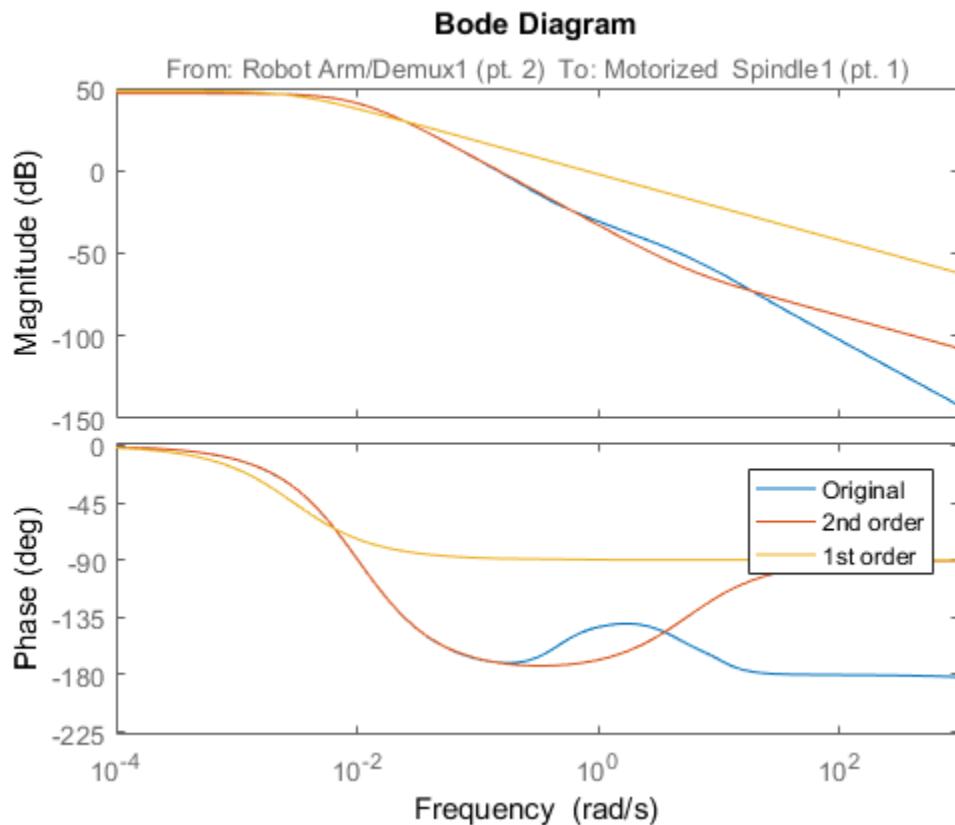

Small Hankel singular values indicate that the associated states contribute little to the behavior of the system. The plot shows that two states account for most of the energy in the system. Therefore, try simplifying the model to just first or second order.

```
opts = balredOptions('StateElimMethod','Truncate');
hplant1 = balred(hplant,1,opts);
hplant2 = balred(hplant,2,opts);
```

The second argument to `balred` specifies the target approximation order, so that `hplant1` is a first-order approximation and `hplant2` is a second-order approximation of `hplant`. By default, `balred` discards the states with the smallest Hankel singular values, and alters the remaining states to preserve the DC gain of the system. Setting the `StateElimMethod` option to `Truncate` causes `balred` to discard low-energy states without altering the remaining states.

When working with reduced-order models, it is important to verify that the approximation does not introduce inaccuracies at frequencies that are important for your application. Therefore, compare the frequency responses of the original and approximated systems. For MIMO systems, use the `sigmaplot` command. For this SISO system, examine a Bode plot.

```
bodeplot(hplant,hplant2,hplant1)
legend('Original','2nd order','1st order')
```

Bode Diagram

From: Robot Arm/Demux1 (pt. 2) To: Motorized Spindle1 (pt. 1)
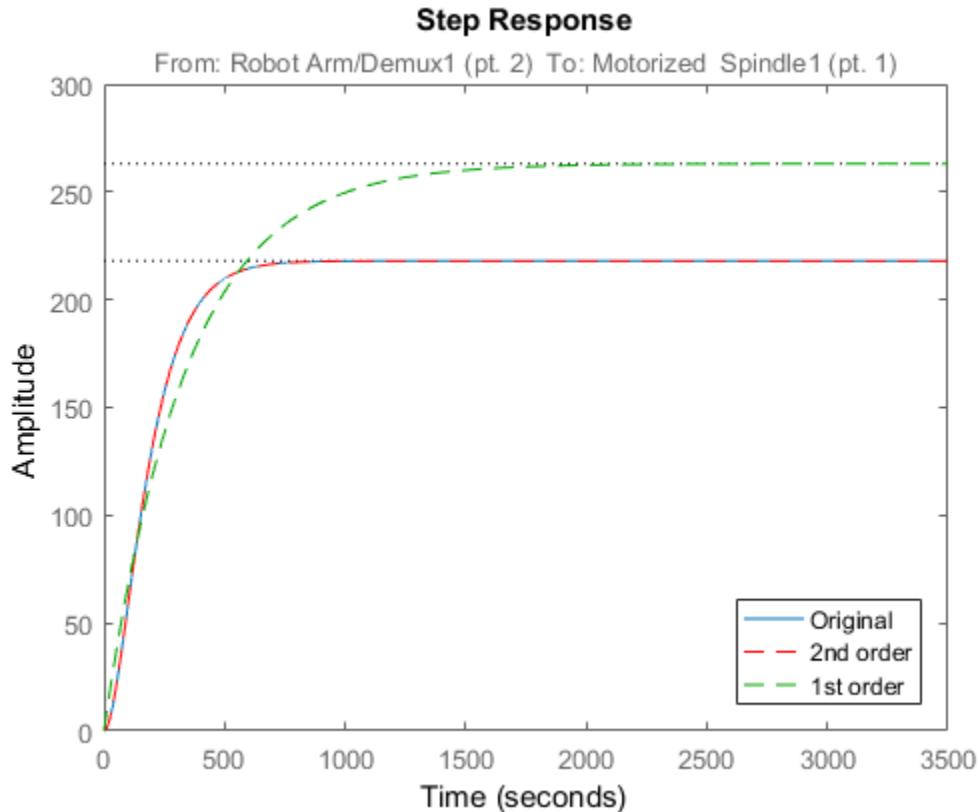
The second-order approximation `hplant2` matches the original 23rd-order system very well, especially at lower frequencies. The first-order system does not match as well.

In general, as you decrease the order of the approximated model, the frequency response of the approximated model begins to differ from the original model. Choose an approximation that is sufficiently accurate in the bands that are important to you. For example, in a control system you might want good accuracy inside the control bandwidth. Accuracy at frequencies far above the control bandwidth, where the gain rapidly rolls off, might be less important.

You can also validate the approximation in the time domain. For instance, examine the step responses of the original and reduced-order systems.

```
stepplot(hplant,hplant2,'r--',hplant1,'g--')
```

```
legend('Original','2nd order','1st order','Location','SouthEast')
```


This result confirms that the second-order approximation is a good match to the original 23rd-order system.

Compare Truncated and DC Matched Low-Order Model Approximations
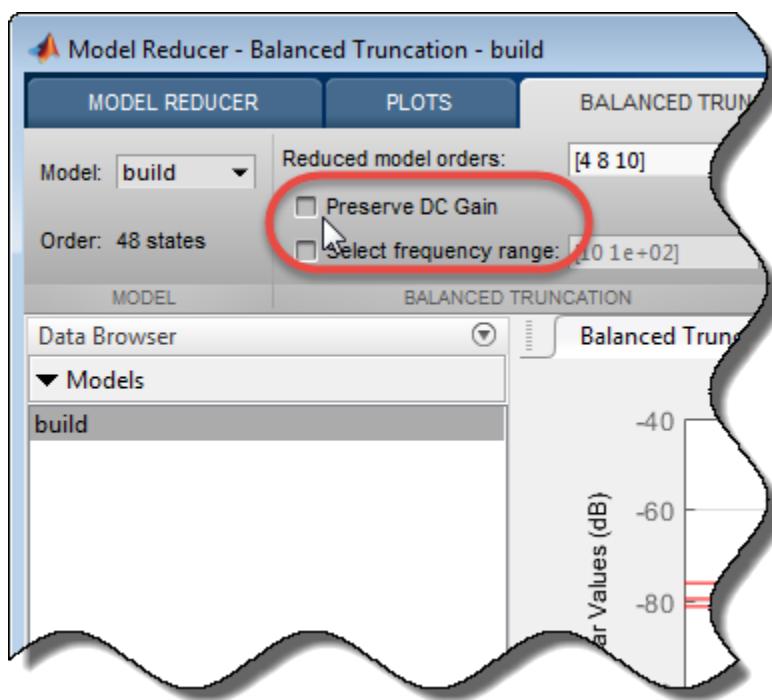
This example shows how to compute a low-order approximation in two ways and compares the results. When you compute a low-order approximation by the balanced truncation method, you can either:

- Discard the states that make the smallest contribution to system behavior, altering the remaining states to preserve the DC gain of the system.
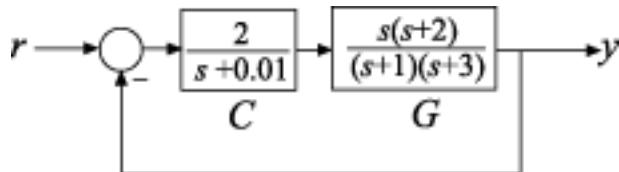
- Discard the low-energy states without altering the remaining states.

Which method you choose depends on what dynamics are most important to your application. In general, preserving DC gain comes at the expense of accuracy in higher-frequency dynamics. Conversely, state truncation can yield more accuracy in fast transients, at the expense of low-frequency accuracy.

This example compares the state-elimination methods of the `balred` command, `Truncate` and `MatchDC`. You can similarly control the state-elimination method in the **Model Reducer** app, on the **Balanced Truncation** tab, using the **Preserve DC Gain** check box, as shown.

Consider the following system.

Create a closed-loop model of this system from *r* to *y*.

```
G = zpk([0 -2],[-1 -3],1);
C = tf(2,[1 1e-2]);
T = feedback(G*C,1)
```

```
T =

            2 s (s+2)
  -------------------------------
  (s+0.004277) (s+1.588) (s+4.418)

Continuous-time zero/pole/gain model.
```
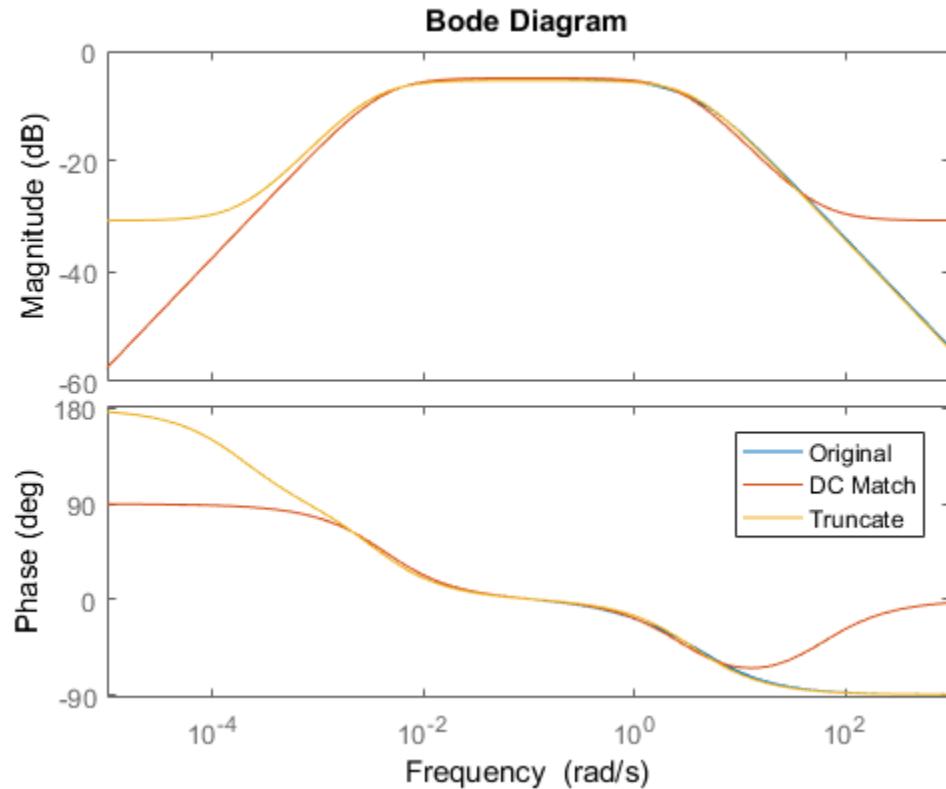
T is a third-order system that has a pole-zero near-cancellation close to *s* = 0. Therefore, it is a good candidate for order reduction by approximation.

Compute two second-order approximations to T, one that preserves the DC gain and one that truncates the lowest-energy state without changing the other states. Use `balredOptions` to specify the approximation methods, `MatchDC` and `Truncate`, respectively.

```
matchopt = balredOptions('StateElimMethod','MatchDC');
truncopt = balredOptions('StateElimMethod','Truncate');
Tmatch = balred(T,2,matchopt);
Ttrunc = balred(T,2,truncopt);
```

Compare the frequency responses of the approximated models.
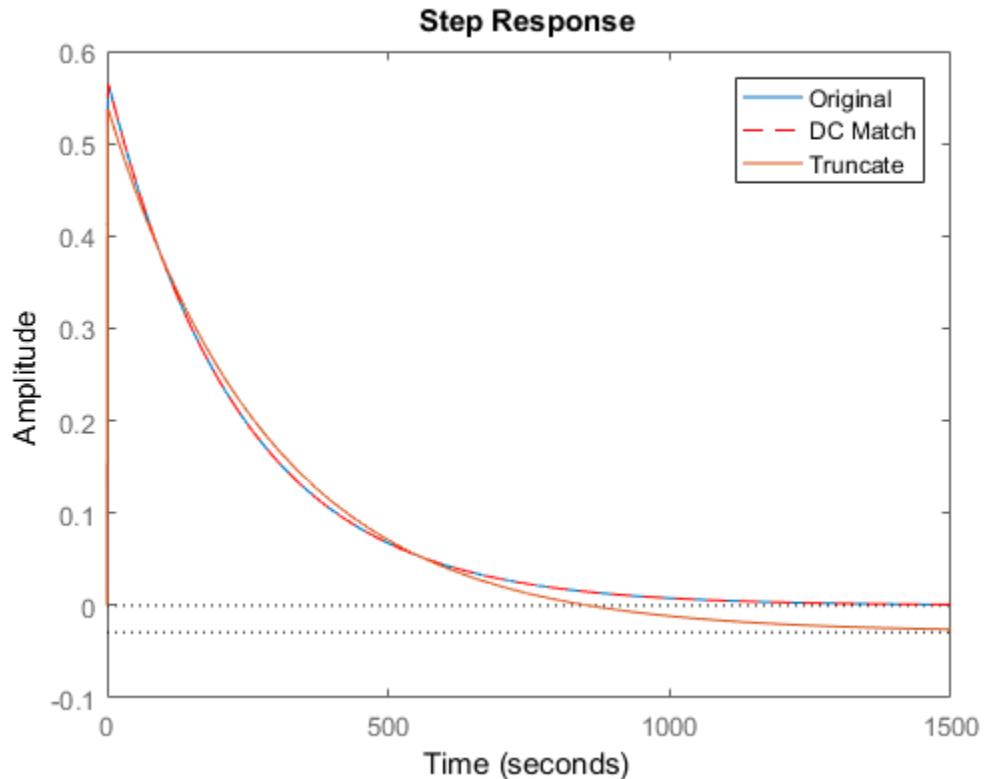
```
bodeplot(T,Tmatch,Ttrunc)
legend('Original','DC Match','Truncate')
```

The truncated model `Ttrunc` matches the original model well at high frequencies, but differs considerably at low frequency. Conversely, `Tmatch` yields a good match at low frequencies as expected, at the expense of high-frequency accuracy.

You can also see the differences between the two methods by examining the time-domain response in different regimes. Compare the slow dynamics by looking at the step response of all three models with a long time horizon.
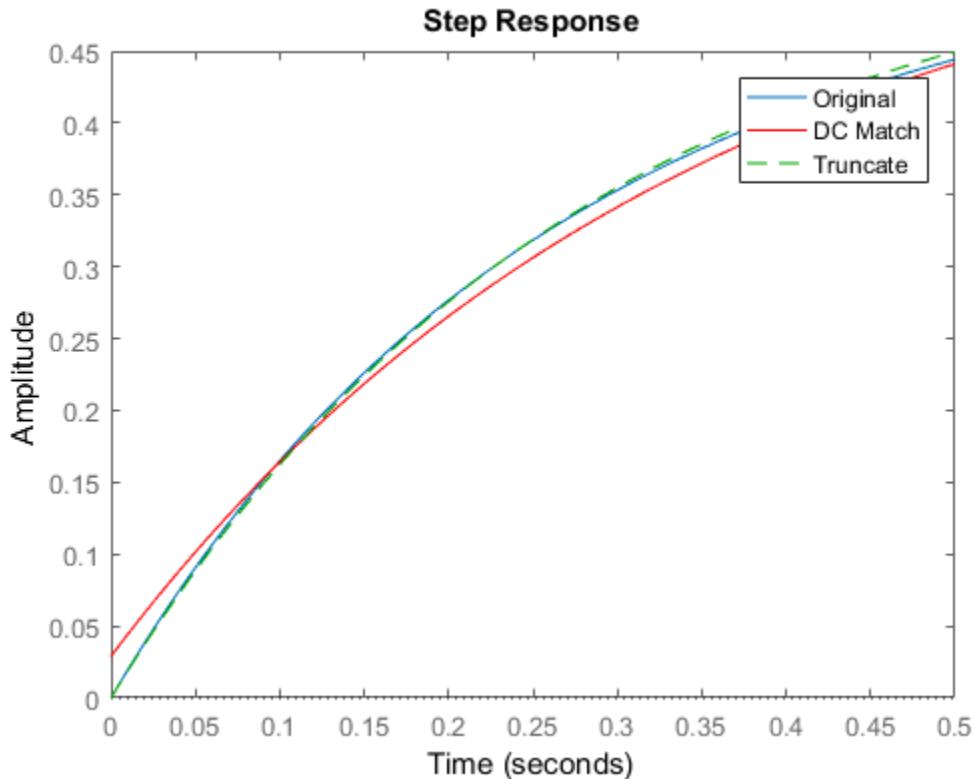
```
stepplot(T,Tmatch,'r--',Ttrunc,1500)
legend('Original','DC Match','Truncate')
```

Step Response

As expected, on long time scales the DC-matched approximation `Tmatch` has a very similar response to the original model.

Compare the fast transients in the step response.

```
stepplot(T,Tmatch,'r',Ttrunc,'g--',0.5)
legend('Original','DC Match','Truncate')
```
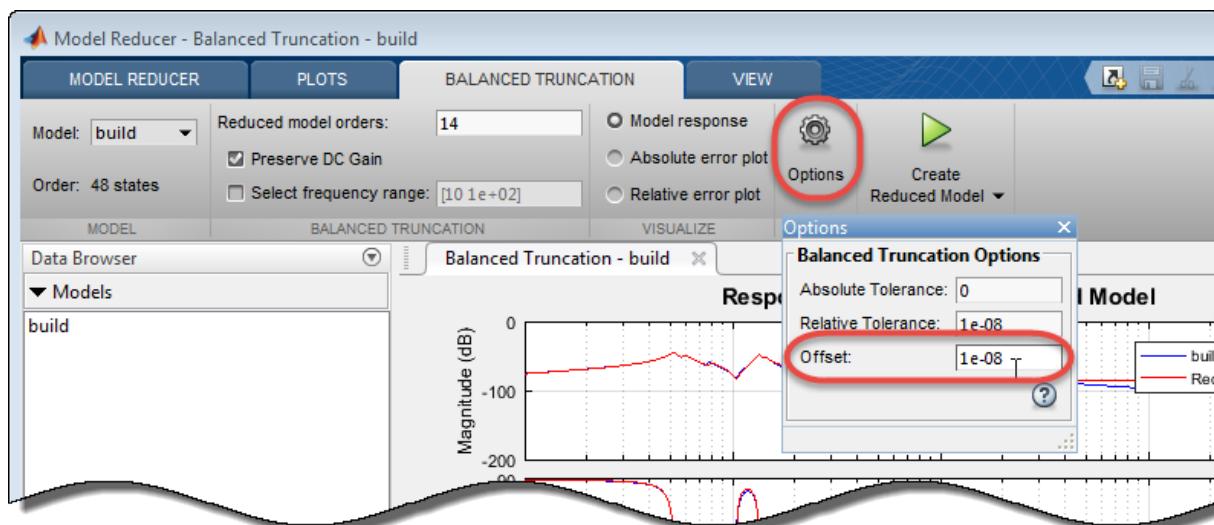
On short time scales, the truncated approximation `Ttrunc` provides a better match to the original model. Which approximation method you should use depends on which regime is most important for your application.

Approximate Model with Unstable or Near-Unstable Pole

This example shows how to compute a reduced-order approximation of a system when the system has unstable or near-unstable poles.

When computing a reduced-order approximation, the `balred` command (or the **Model Reducer** app) does not eliminate unstable poles because doing so would fundamentally change the system dynamics. Instead, the software decomposes the model into stable and unstable parts and reduces the stable part of the model.

If your model has near-unstable poles, you might want to ensure that the reduced-order approximation preserves these dynamics. This example shows how to use the `Offset` option of `balred` to preserve poles that are close to the stable-unstable boundary. You can achieve the same result in the **Model Reducer** app, on the **Balanced Truncation** tab, under **Options**, using the **Offset** field, as shown:
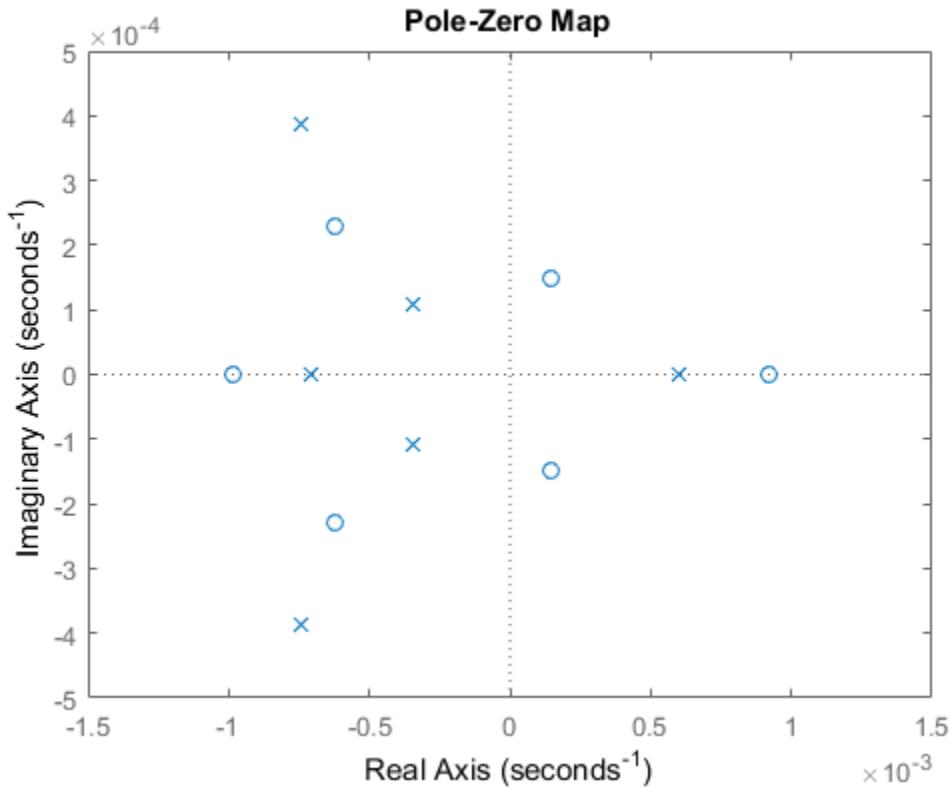
Load a model with unstable and near-unstable poles.

```
load('reduce.mat','gasf35unst')
```

`gasf35unst` is a 25-state SISO model with two unstable poles (Re(s) > 0). Examine the system poles to find the near-unstable poles.
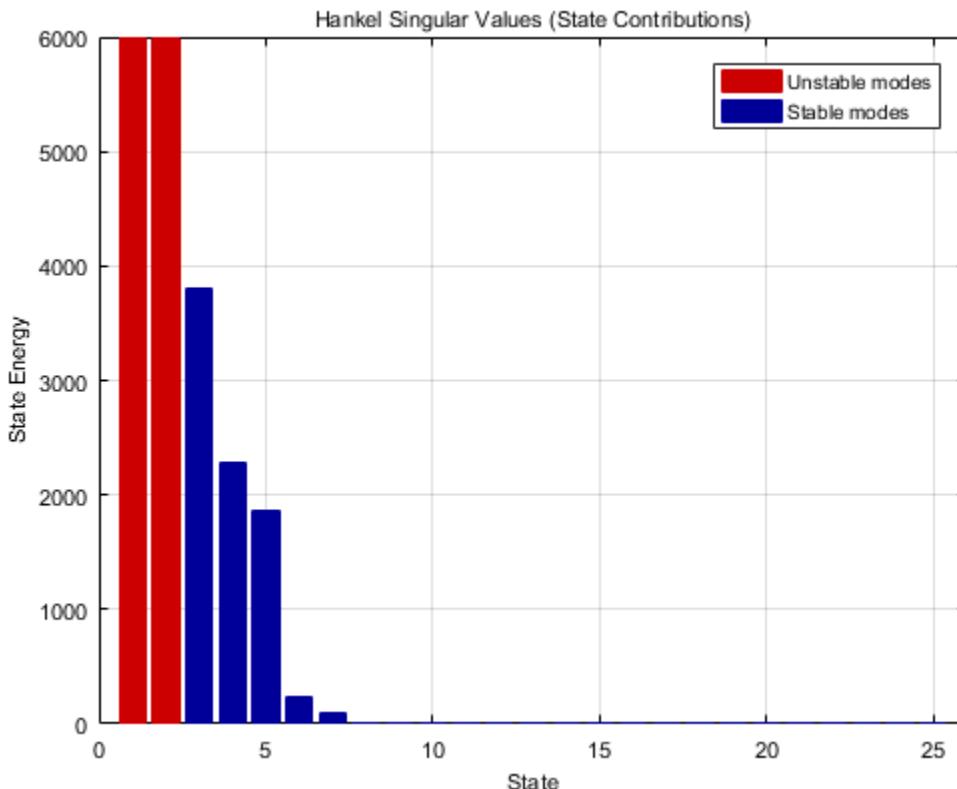
```
pzplot(gasf35unst)
axis([-0.0015 0.0015 -0.0005 0.0005])
```

The pole-zero plot shows several poles (marked by x) that fall in the left half-plane, but relatively close to the imaginary axis. These are the near-unstable poles. Two of these fall within 0.0005 of instability. Three more fall within 0.001 of instability.

Examine a Hankel singular-value plot of the model.

```
hsvplot(gasf35unst)
```

The plot shows the two unstable modes, but you cannot easily determine the energy contribution of the near-unstable poles. In your application, you might want to reduce the model without discarding those poles nearest to instability, even if they are of relatively low energy. Use the `Offset` option of `balred` to calculate a reduced-order system that preserves the two stable poles that are closest to the imaginary axis. The `Offset` option sets the boundary between poles that `balred` can discard, and poles that `balred` must preserve (treat as unstable).
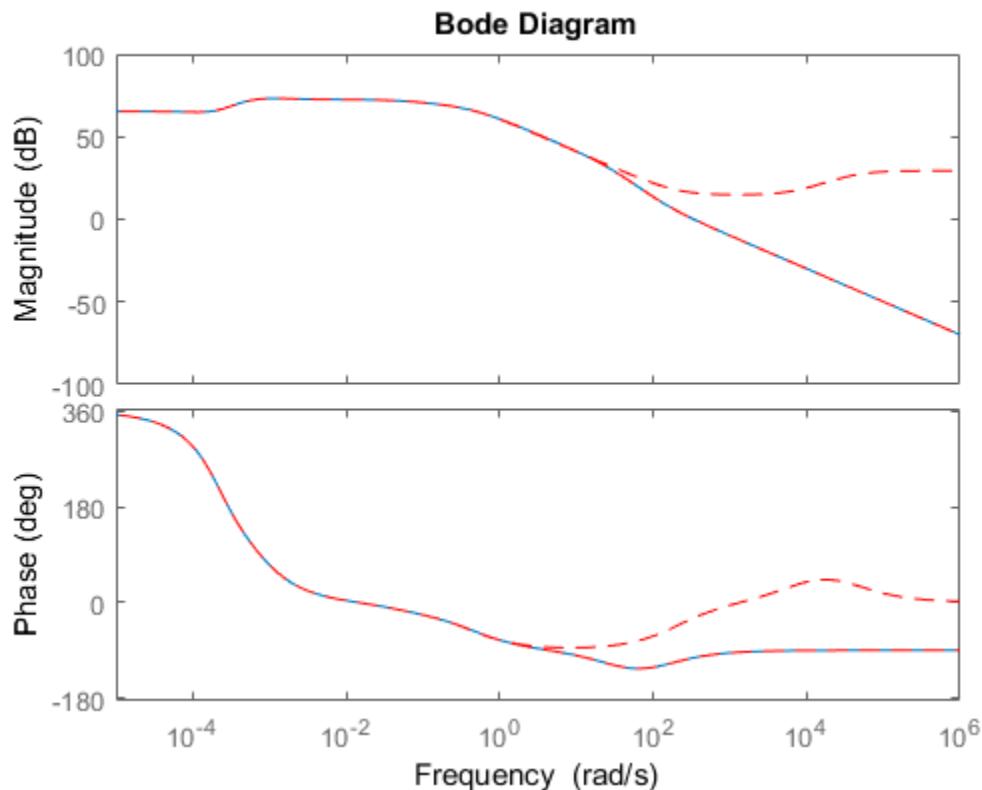
```
opts = balredOptions('Offset',0.0005);
gasf_arr = balred(gasf35unst,[10 15],opts);
```

Providing `balred` an array of target approximation orders `[10 15]` causes `balred` to return an array of approximated models. The array `gasf_arr` contains two models, a

10th-order and a 15th-order approximation of `gasf35unst`. In both approximations, `balred` does not discard the two unstable poles or the two nearly-unstable poles.

Compare the reduced-order approximations to the original model.

```
bodeplot(gasf35unst,gasf_arr,'r--')
```
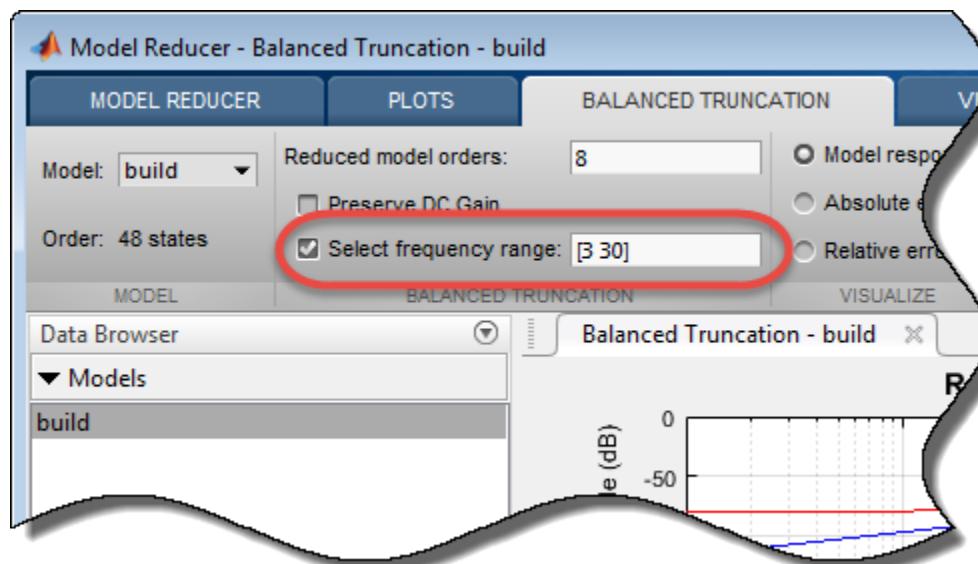

The 15th order approximation is a good frequency-domain match to the original model. However, the 10th-order approximation shows changes in high-frequency dynamics,

which might be too large to be acceptable. The 15th-order approximation is likely a better choice.

Frequency-Limited Balanced Truncation
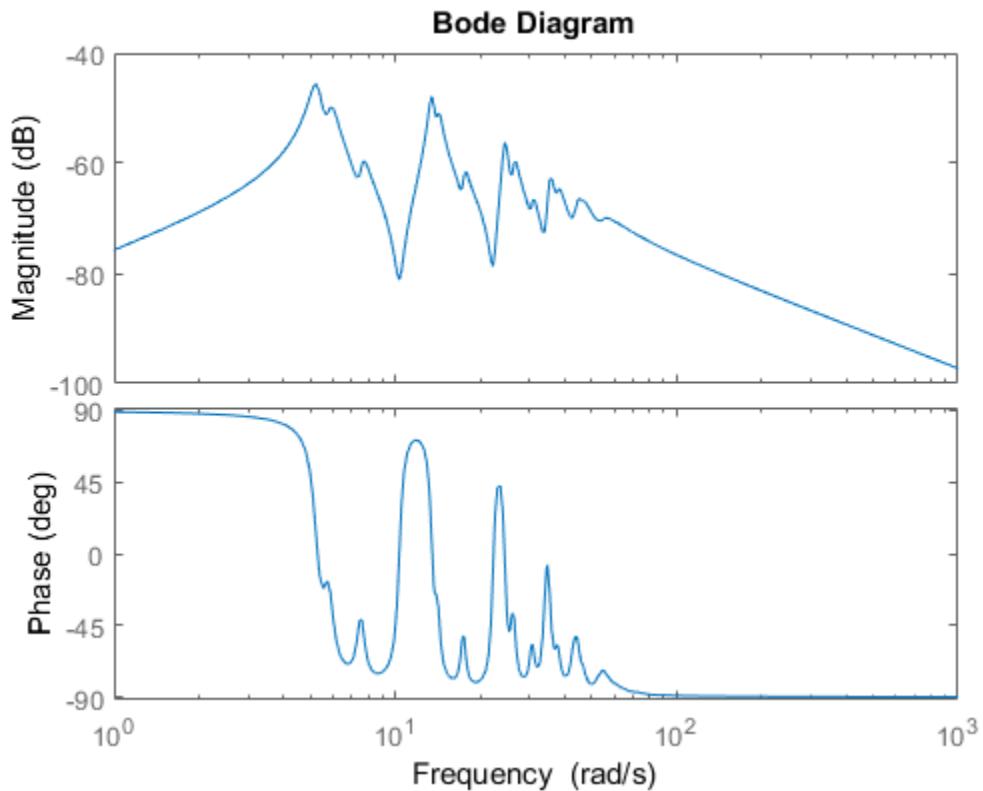
Focusing the energy-contribution calculation on a particular frequency region sometimes yields a good approximation to the dynamics of interest at a lower order than a reduction that takes all frequencies into account. For this example, reduce a high-order model with a focus on the dynamics in a particular interval.

This example demonstrates frequency-limited balanced truncation at the command line, using options for the `balred` command. You can also perform frequency-limited balanced trunacation in the **Model Reducer** app, on the **Balanced Truncation** tab, using the **Select frequency range** check box, as shown.
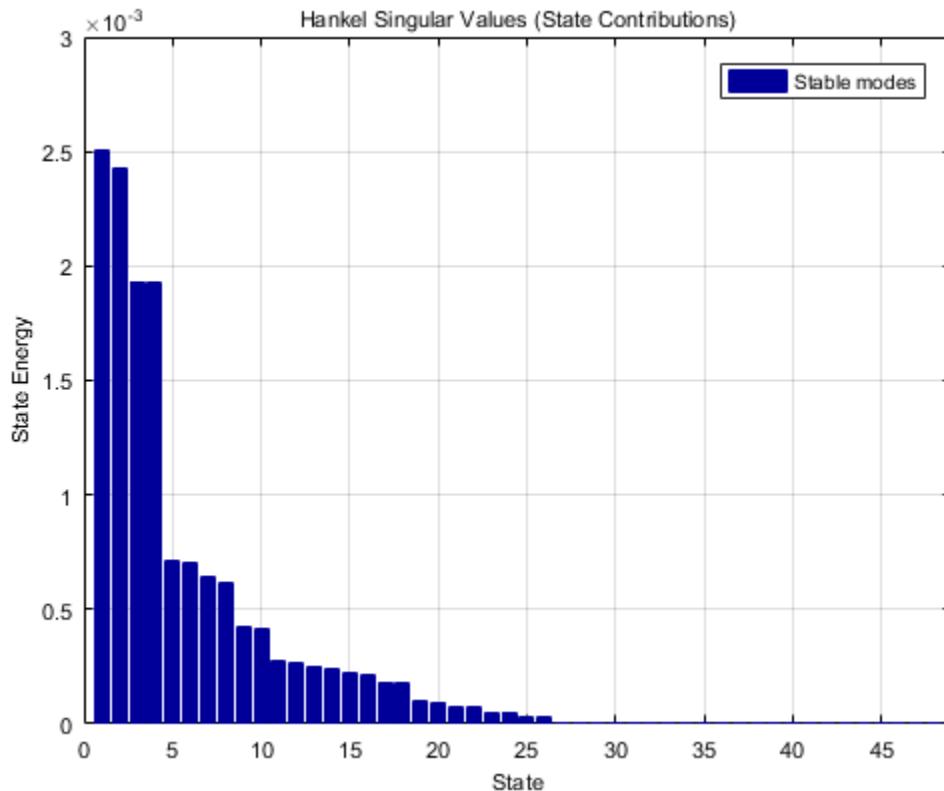
Load a model and examine its frequency response.

```
load build G
bodeplot(G)
```

G is a 48th-order model with several large peak regions around 5.2 rad/s, 13.5 rad/s, and 24.5 rad/s, and smaller peaks scattered across many frequencies. Examine the Hankel singular-value plot to see the energy contributions of the model's 48 states.
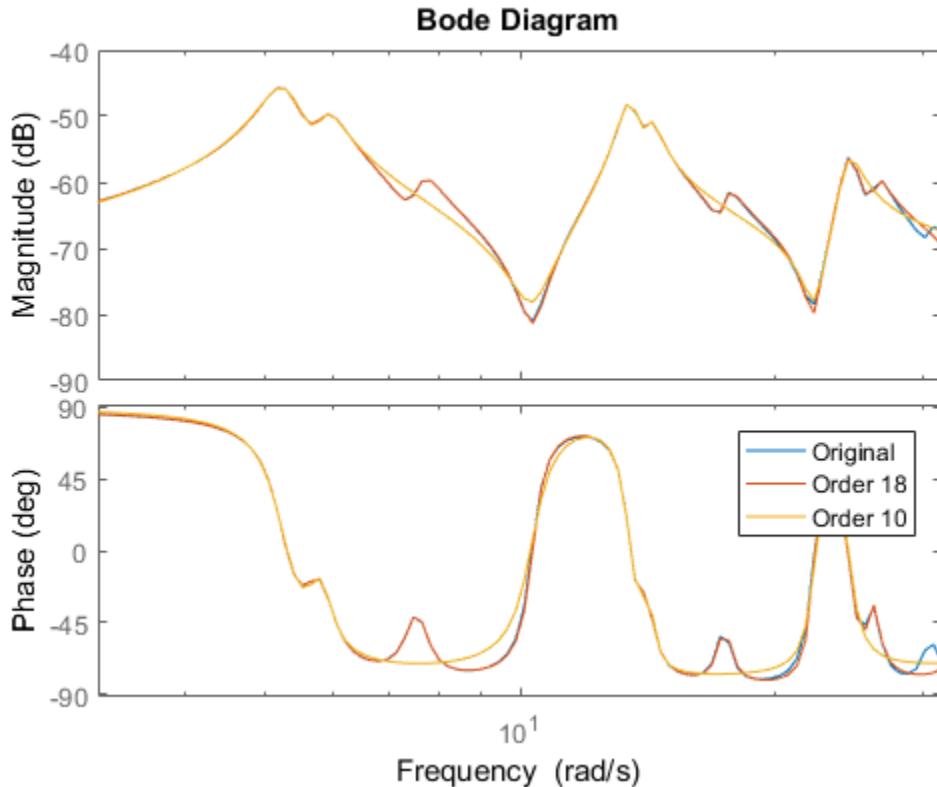
```
hsvd(G)
```

The singular-value plot suggests that you can discard at least 20 states without significant impact on the overall system response. Suppose that for your application you are only interested in the dynamics near the second large peak, between 10 rad/s and 22 rad/s. Try a few reduced-model orders based on the Hankel singular value plot. Compare their frequency responses to the original model, especially in the region of that peak.

```
G18 = balred(G,18);
G10 = balred(G,10);

bodeplot(G,G18,G10,logspace(0.5,1.5,100));
legend('Original','Order 18','Order 10');
```
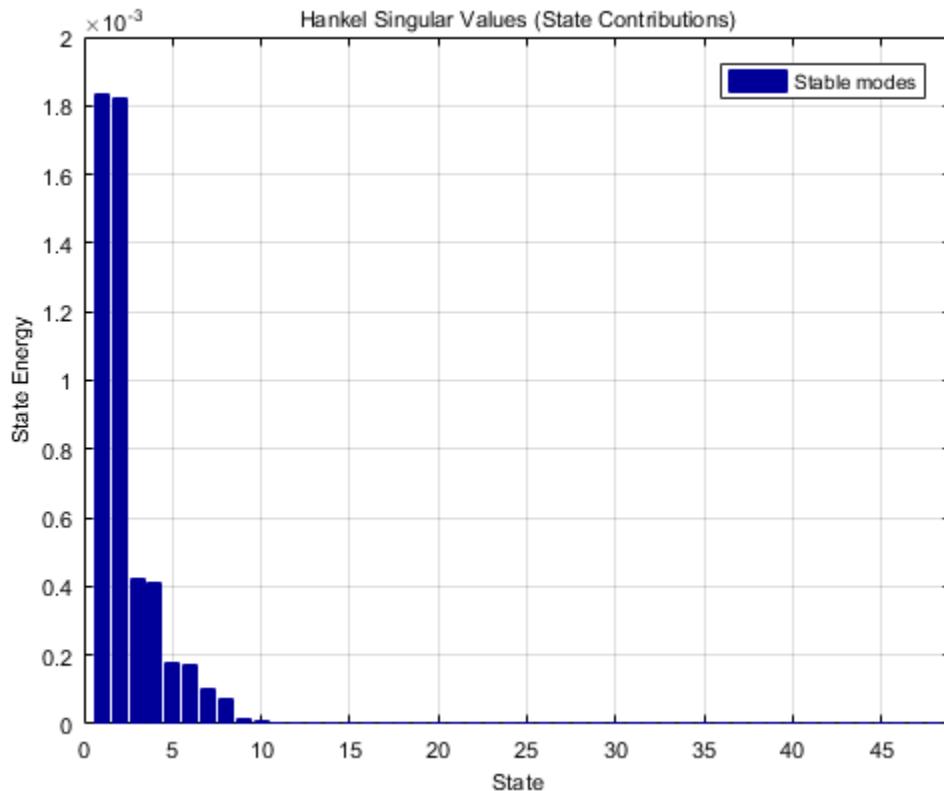
Bode Diagram

The 18th-order model is a good match to the dynamics in the region of interest. At 10th order, however, there is some degradation of the match.

Focus the model reduction on the region of interest to obtain a good match with a lower-order approximation. First, examine the state energy contributions in that frequency region only. Use `hsvdOptions` to specify the frequency interval for `hsvd`.

```
hopt = hsvdOptions('FreqIntervals',[10,22]);
hsvd(G,hopt)
```
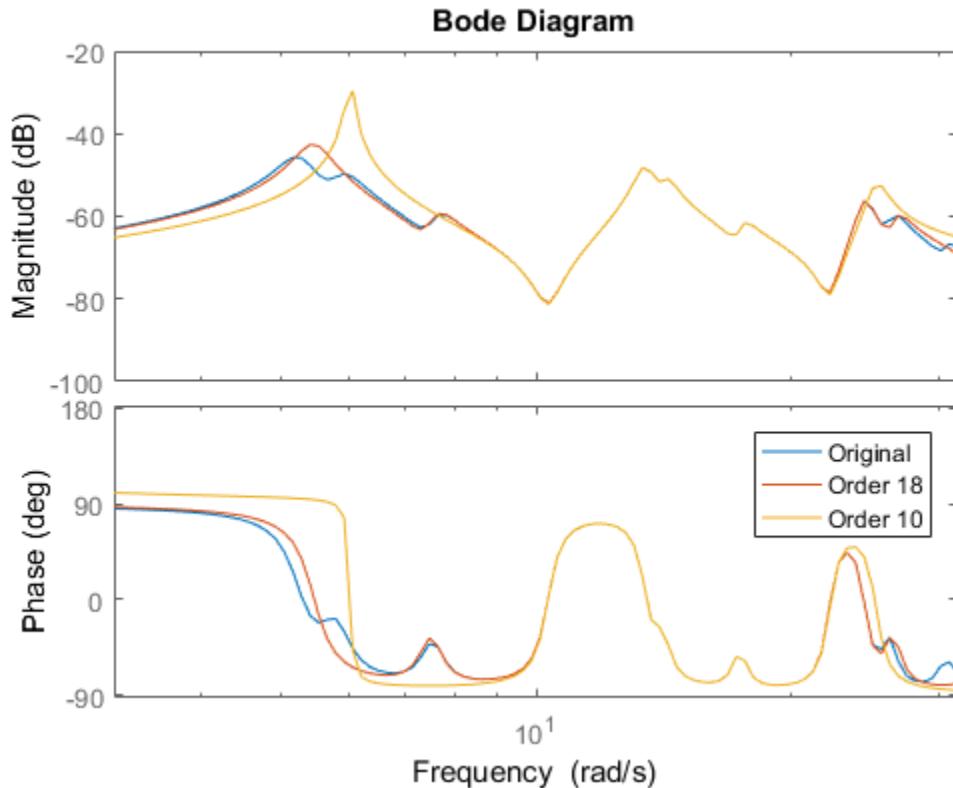
Comparing this plot to the previous Hankel singular-value plot shows that in this frequency region, many fewer states contribute significantly to the dynamics than contribute to the overall dynamics.

Try the same reduced-model orders again, this time choosing states to eliminate based only on their contribution to the frequency interval. Use `balredOptions` to specify the frequency interval for `balred`.

```
bopt = balredOptions('FreqIntervals',[10,22]);
GLim18 = balred(G,18,bopt);
GLim10 = balred(G,10,bopt);

bodeplot(G,GLim18,GLim10,logspace(0.5,1.5,100));
legend('Original','Order 18','Order 10');
```

With the frequency-limited energy computation, a 10th-order approximation is as good in the region of interest as the 18th-order approximation computed without frequency limits.

See Also

balred | hsvplot | Model Reducer

Related Examples

- "Mode-Selection Model Reduction" on page 6-55
- "Pole-Zero Simplification" on page 6-45
- "Model Reduction Basics" on page 6-2

Pole-Zero Simplification

Pole-zero simplification reduces the order of your model exactly by canceling pole-zero pairs or eliminating states that have no effect on the overall model response. Canceling pole-zero pairs can be introduced, for example, when you construct closed-loop architectures. Normal small errors associated with numerical computation can convert such canceling pairs to near-canceling pairs. Removing these states preserves the model response characteristics while simplifying analysis and control design. Types of pole-zero simplification include:

- Structural elimination — Eliminate states that are structurally disconnected from the inputs or outputs. Eliminating structurally disconnected states is a good first step in model reduction because the process does not involve any numerical computation. It also preserves the state structure of the remaining states. At the command line, perform structural elimination with `sminreal`.

- Pole-zero cancelation or minimal realization — Eliminate canceling or near-canceling pole-zero pairs from transfer functions. Eliminate unobservable or uncontrollable states from state-space models. At the command line, perform this kind of simplification with `minreal`

In the **Model Reducer** app, the **Pole-Zero Simplification** method automatically eliminates structurally disconnected states and also performs pole-zero cancelation or minimal realization.

In this section...
"Pole-Zero Simplification in the Model Reducer App" on page 6-45
"Pole-Zero Cancelation at the Command Line" on page 6-51

Pole-Zero Simplification in the Model Reducer App
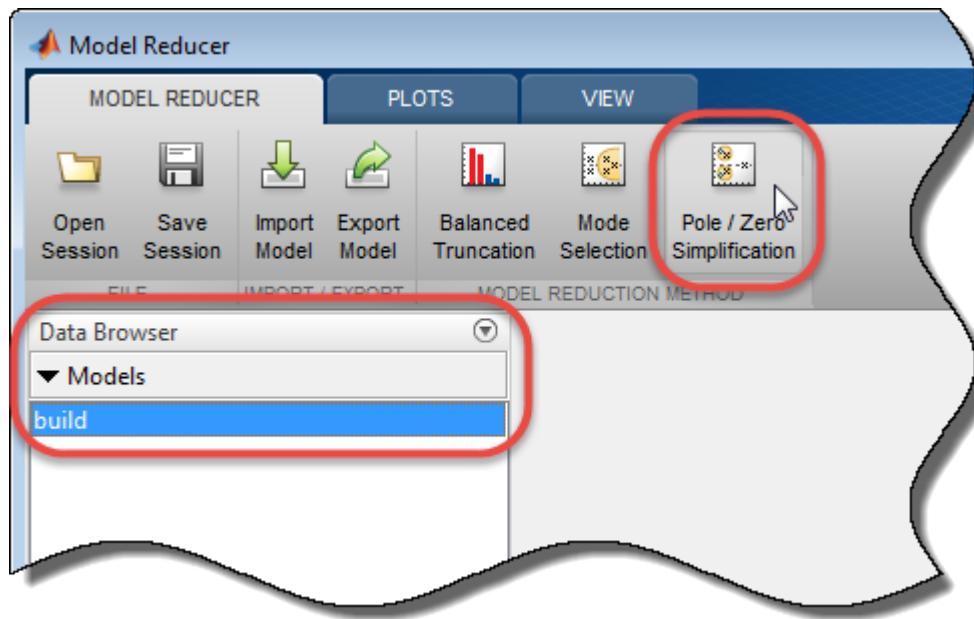
Model Reducer provides an interactive tool for performing model reduction and examining and comparing the responses of the original and reduced-order models. To reduce a model by pole-zero simplification in **Model Reducer**:

1 Open the app and import a model to reduce. For instance, suppose that there is a model named `build` in the MATLAB workspace. The following command opens **Model Reducer** and imports the LTI model `build`.

```
modelReducer(build)
```

2
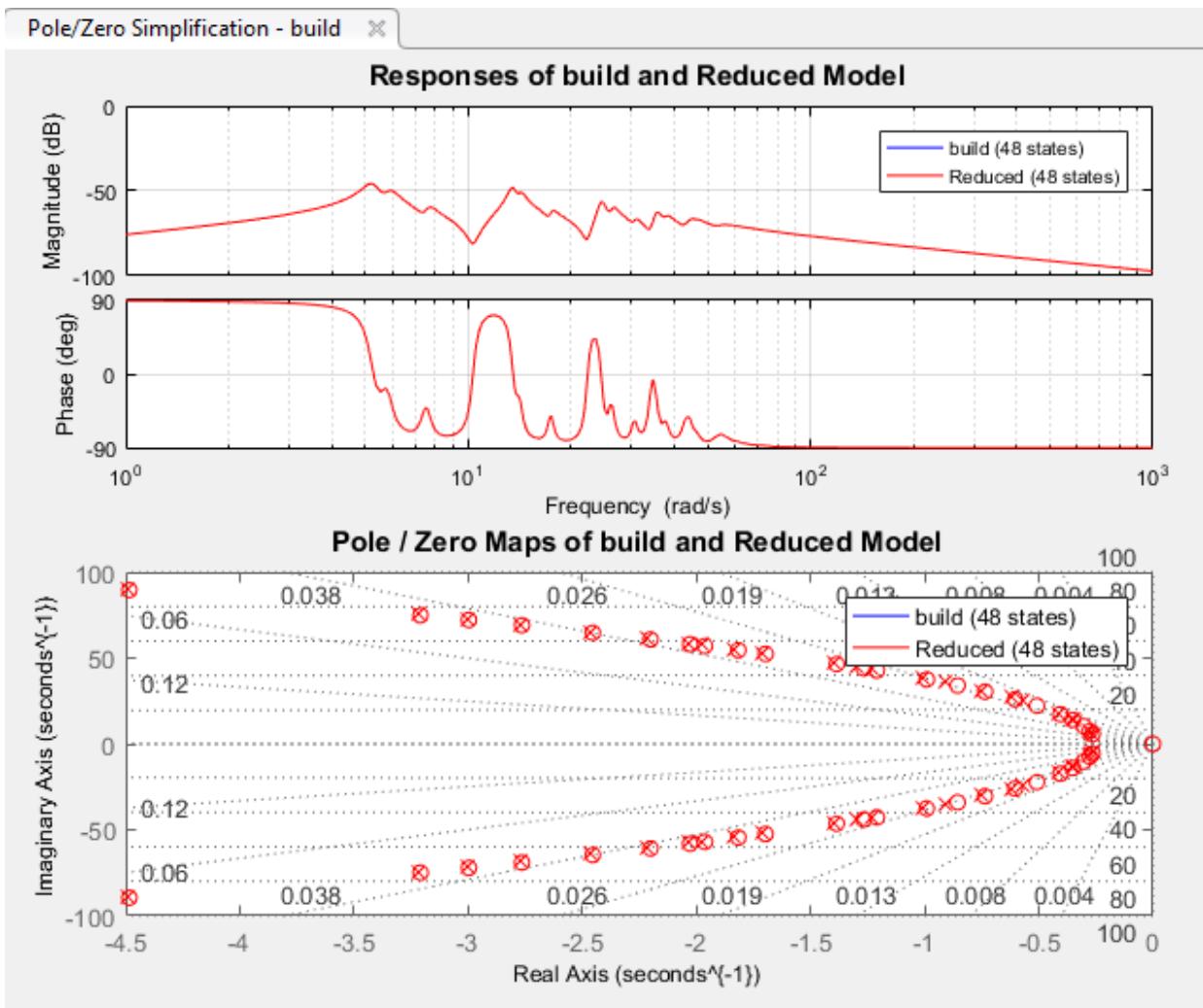
In the **Data Browser**, select the model to reduce. Click **Pole-Zero Simplification**.
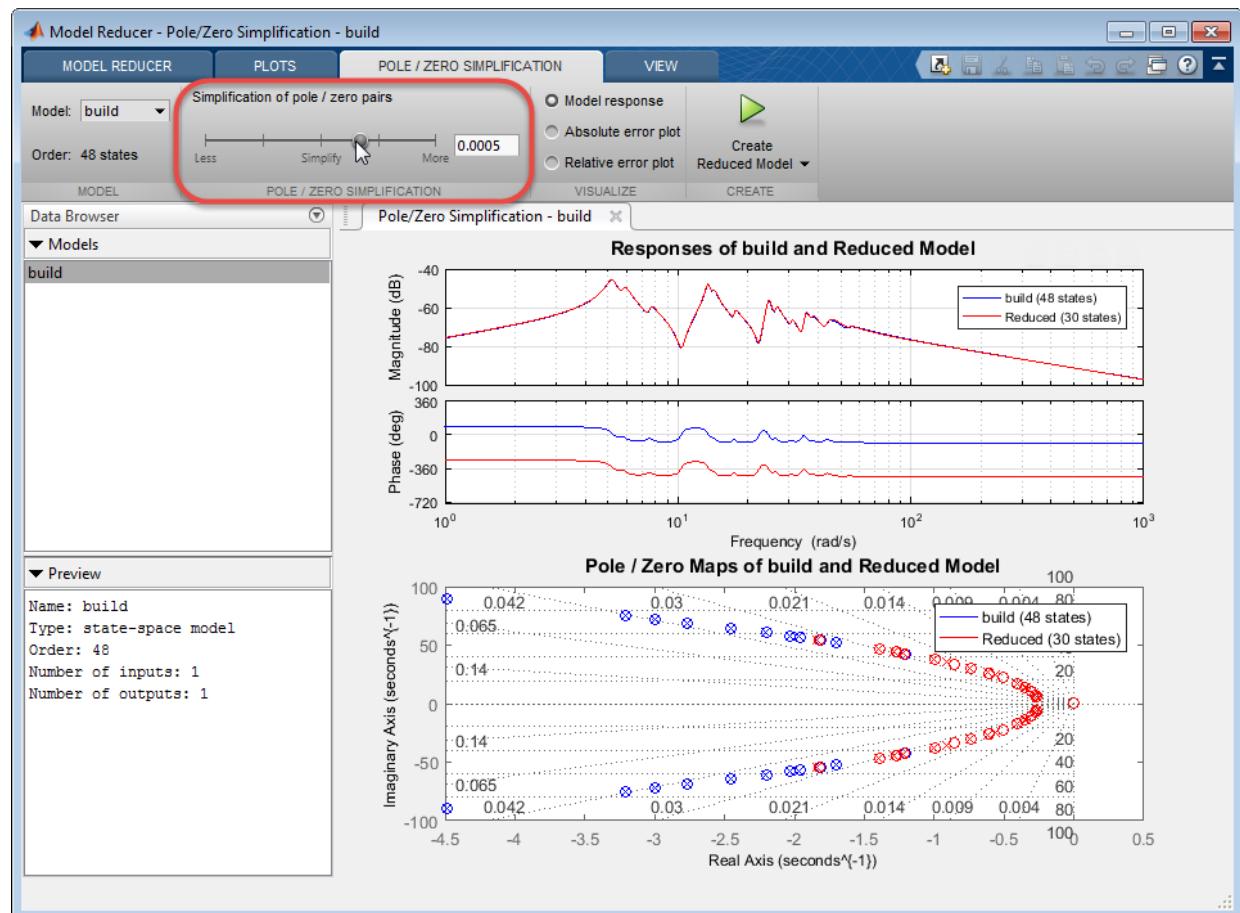
In the **Pole-Zero Simplification** tab, **Model Reducer** displays a plot of the frequency response of the original model and a reduced version of the model. The app also displays a pole-zero map of both models.

The pole-zero map marks pole locations with x and zero locations with o.
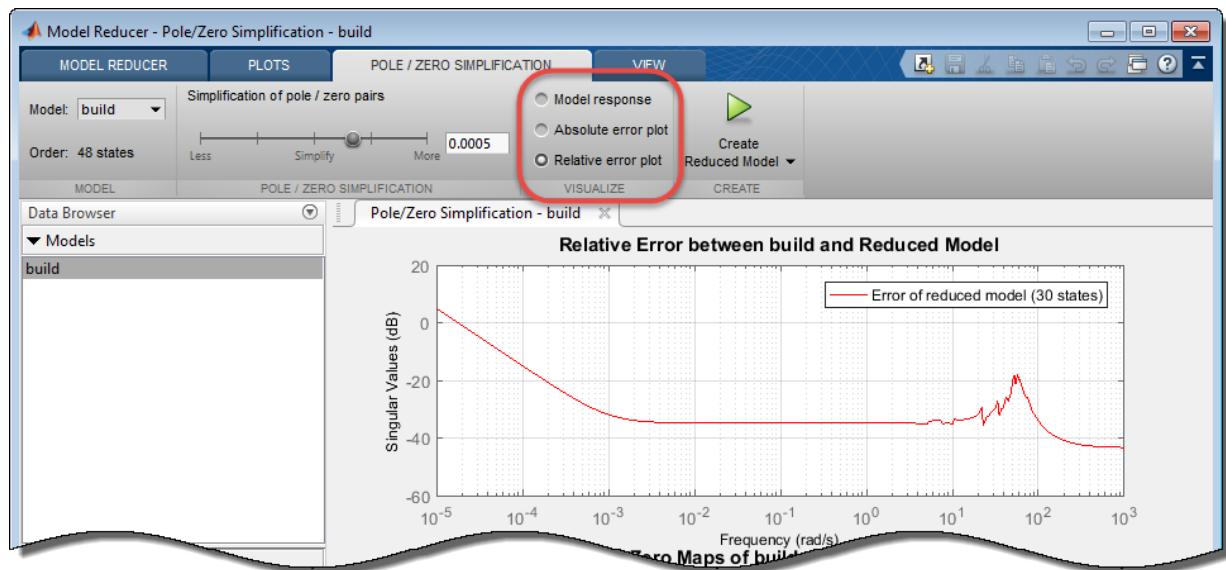
Note: The frequency response is a Bode plot for SISO models, and a singular-value plot for MIMO models.

3 Optionally, change the tolerance with which **Model Reducer** identifies canceling pole-zero pairs. **Model Reducer** cancels pole-zero pairs that fall within the tolerance specified by the **Simplification of pole/zero pairs** value. In this case, no pole-zero pairs are close enough together for **Model Reducer** to cancel them at the default tolerance of 1e-05. To cancel pairs that are a little further apart, move the slider to the right or enter a larger value in the text box.
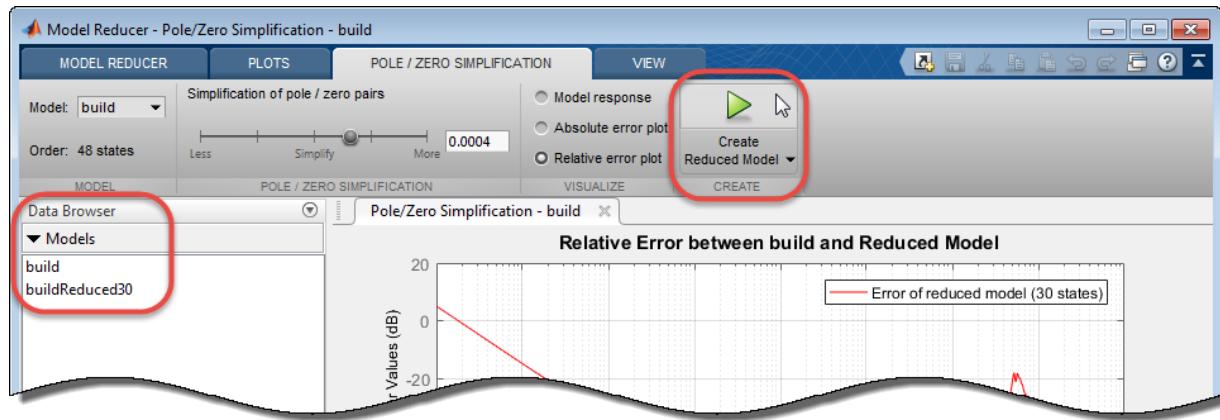
The blue x and o marks on the pole-zero map show the near-canceling pole-zero pairs in the original model that are eliminated from the simplified model. Poles and zeros remaining in the simplified model are marked with red x and o.

4 Try different simplification tolerances while observing the frequency response of the original and simplified model. Remove as many poles and zeros as you can while preserving the system behavior in the frequency region that is important for your application. Optionally, examine absolute or relative error between the original and simplified model. Select the error-plot type using the buttons on the **Pole-Zero Simplification** tab.

For more information about using the analysis plots, see "Visualize Reduced-Order Models in the Model Reducer App" on page 6-65.

5 When you have a simplified model that you want to store and analyze further, click

   . The new model appears in the **Data Browser** with a name that reflects the reduced model order.
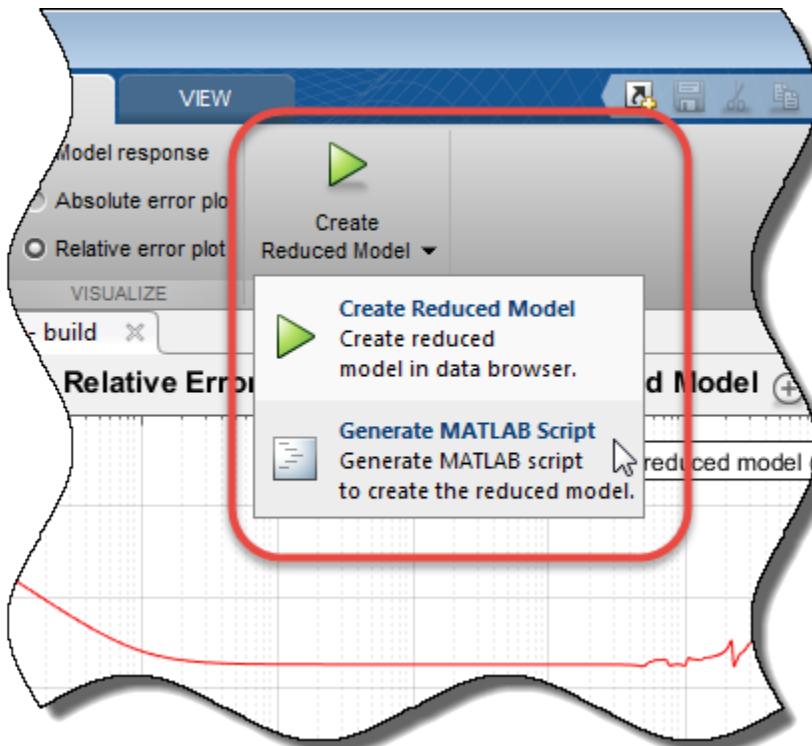
After creating a reduced model in the **Data Browser**, you can continue changing the simplification parameters and create reduced models with different orders for analysis and comparison.

You can now perform further analysis with the reduced model. For example:

- Examine other responses of the reduced system, such as the step response or Nichols plot. To do so, use the tools on the **Plots** tab. See "Visualize Reduced-Order Models in the Model Reducer App" on page 6-65 for more information.

- Export reduced models to the MATLAB workspace for further analysis or control design. On the **Model Reducer** tab, click **Export**.

Generate MATLAB Code for Pole-Zero Simplification

To create a MATLAB script you can use for further model-reduction tasks at the command line, click **Create Reduced Model**, and select **Generate MATLAB Script**.
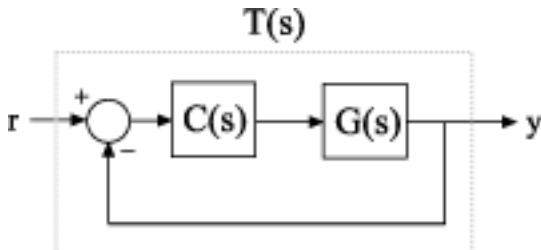
Model Reducer creates a script that uses the `minreal` command to perform model reduction with the parameters you have set on the **Pole-Zero Simplification** tab. The script opens in the MATLAB editor.

Pole-Zero Cancelation at the Command Line

To reduce the order of a model by pole-zero cancelation at the command line, use `minreal`.

Create a model of the following system, where `C` is a PI controller, and `G` has a zero at 3×10^{-8} rad/s. Such a low-frequency zero can arise from derivative action somewhere in the plant dynamics. For example, the plant may include a component that computes speed from position measurements.

```
G = zpk(3e-8,[-1,-3],1);
C = pid(1,0.3);
T = feedback(G*C,1)


T =

    (s+0.3) (s-3e-08)
  ---------------------
  s (s+4.218) (s+0.7824)

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

In the closed-loop model T, the integrator $(1/s)$ from C very nearly cancels the low-frequency zero of G.

Force a cancelation of the integrator with the zero near the origin.

```
Tred = minreal(T,1e-7)


Tred =

        (s+0.3)
  ------------------
  (s+4.218) (s+0.7824)

Continuous-time zero/pole/gain model.
```
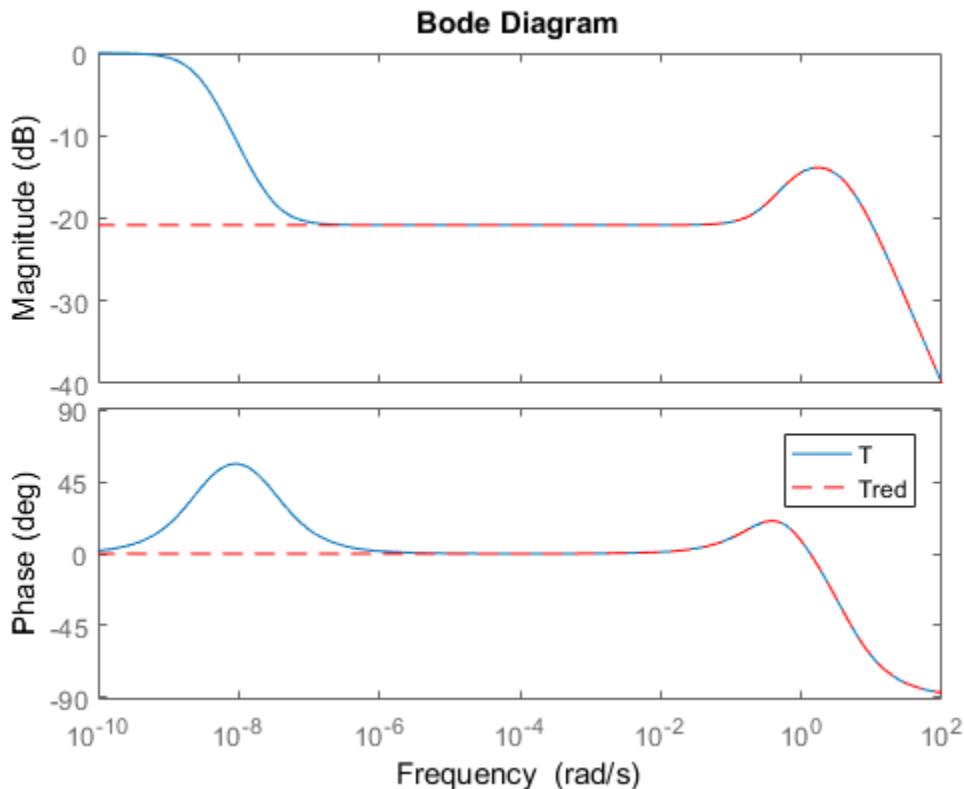
By default, minreal reduces transfer function order by canceling exact pole-zero pairs or near pole-zero pairs within sqrt(eps). Specifying 1e-7 as the second input causes minreal to eliminate pole-zero pairs within 10^{-7} rad/s of each other.

The reduced model `Tred` includes all the dynamics of the original closed-loop model `T`, except for the near-canceling zero-pole pair.

Compare the frequency responses of the original and reduced systems.

```
bode(T,Tred,'r--')
legend('T','Tred')
```


Because the canceled pole and zero do not match exactly, some extreme low-frequency dynamics evident in the original model are missing from `Tred`. In many applications, you can neglect such extreme low-frequency dynamics. When you increase the matching

tolerance of `minreal`, make sure that you do not eliminate dynamic features that are relevant to your application.

See Also
`minreal` | Model Reducer | `sminreal`

Related Examples
- "Balanced Truncation Model Reduction" on page 6-16
- "Mode-Selection Model Reduction" on page 6-55
- "Model Reduction Basics" on page 6-2

Mode-Selection Model Reduction

Model selection eliminates poles that fall outside a specific frequency range of interest. This method is useful when you want to focus your analysis on a particular subset of system dynamics. For instance, if you are working with a control system with bandwidth limited by actuator dynamics, you might discard higher-frequency dynamics in the plant. Eliminating dynamics outside the frequency range of interest reduces the numerical complexity of calculations with the model. There are two ways to compute a reduced-order model by mode selection:

- At the command line, using the `freqsep` command.
- In the **Model Reducer**, using the **Mode Selection** method.

For more general information about model reduction, see "Model Reduction Basics" on page 6-2

In this section...
"Mode Selection in the Model Reducer App" on page 6-55
"Mode Selection at the Command Line" on page 6-61
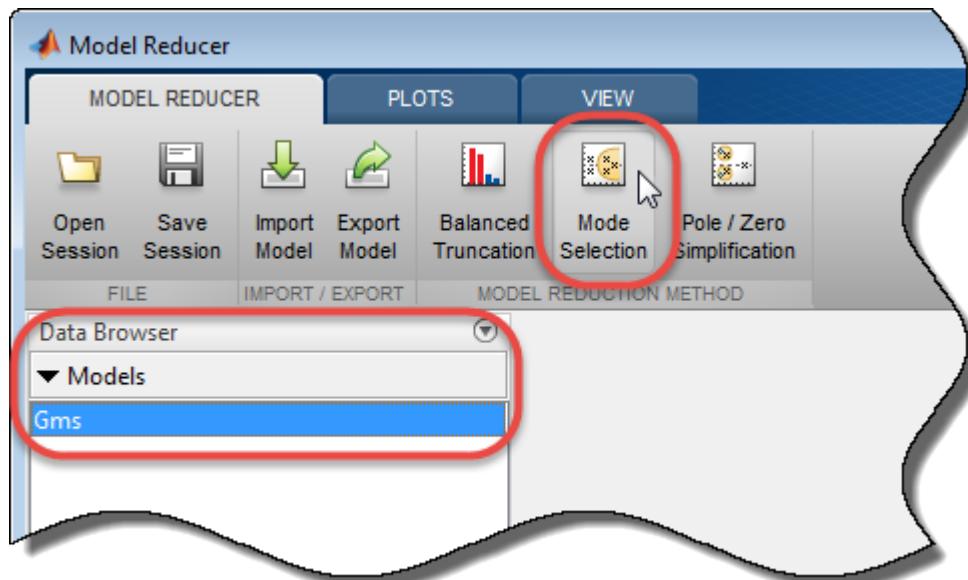
Mode Selection in the Model Reducer App

Model Reducer provides an interactive tool for performing model reduction and examining and comparing the responses of the original and reduced-order models. To approximate a model by mode selection in **Model Reducer**:
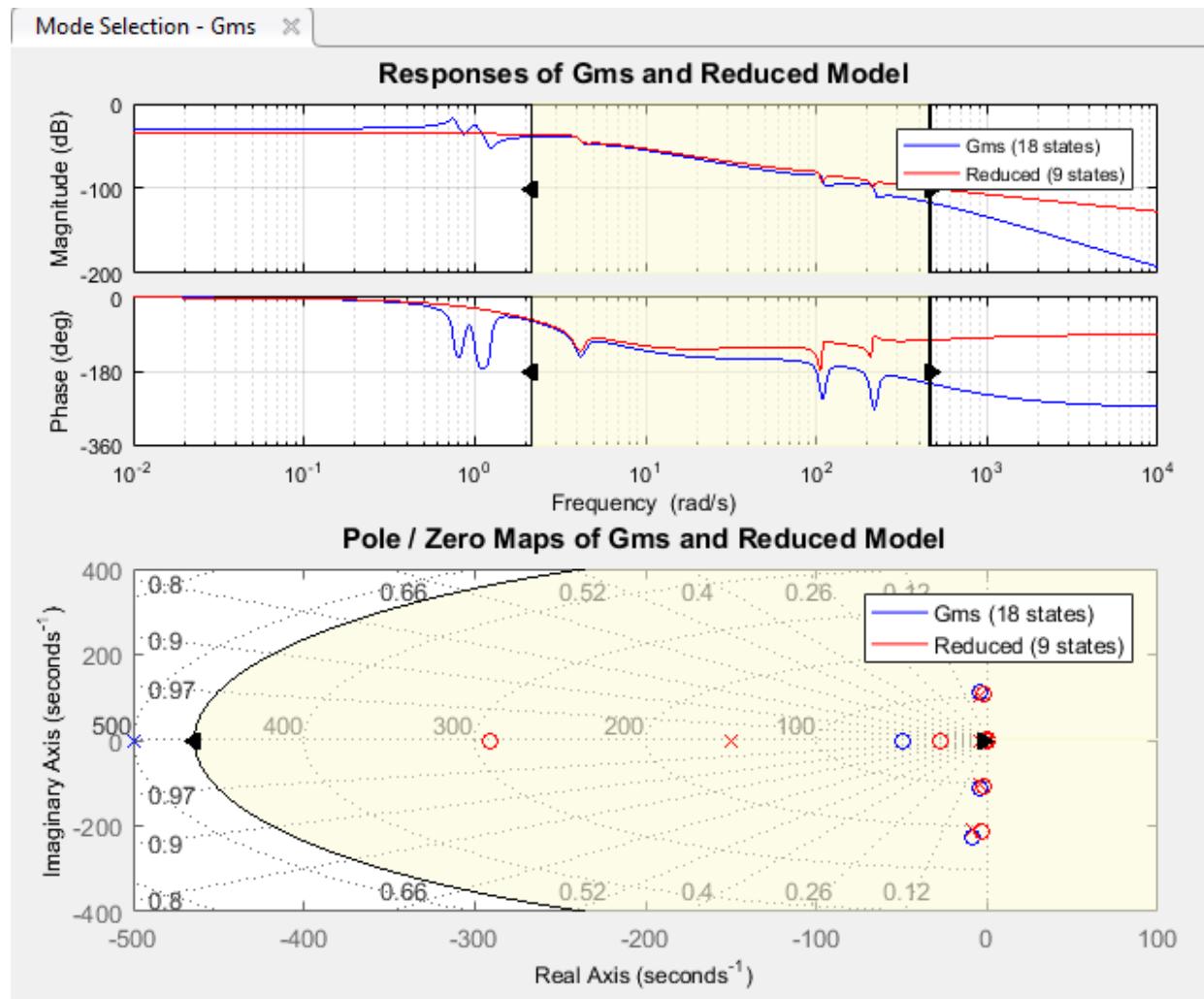
1   Open the app and import an LTI model to reduce. For instance, suppose that there is a model named `Gms` in the MATLAB workspace. The following command opens **Model Reducer** and imports the model.

```
modelReducer(Gms)
```

2   In the **Data Browser**, select the model to reduce. Click [icon] **Mode Selection**.

In the **Mode Selection** tab, **Model Reducer** displays a plot of the frequency response of the original model and a reduced version of the model. The app also displays a pole-zero map of both models.
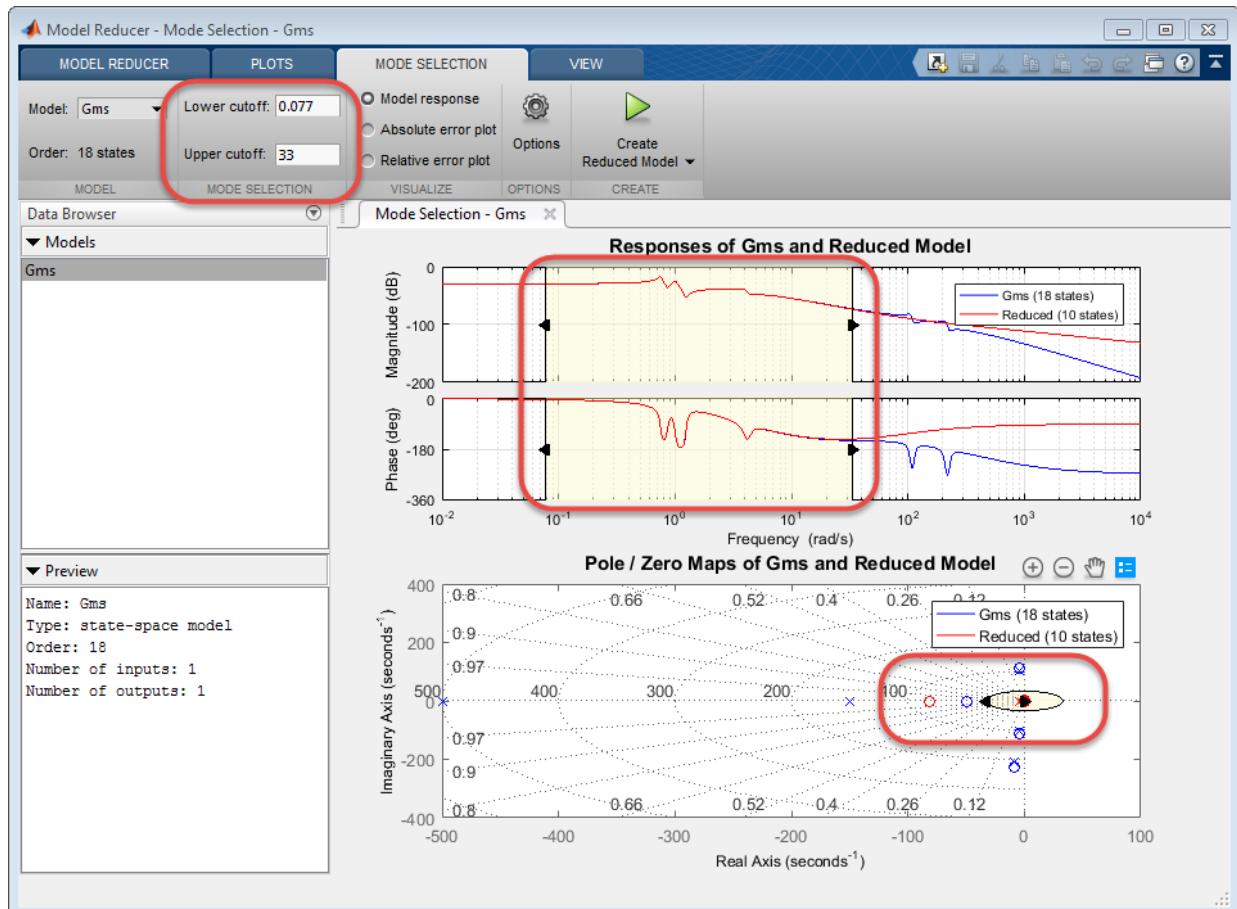
The pole-zero map marks pole locations with x and zero locations with o.

Note: The frequency response is a Bode plot for SISO models, and a singular-value plot for MIMO models.

---

3 **Model Reducer** eliminates poles that lie outside the shaded region. Change the shaded region to capture only the dynamics you want to preserve in the reduced model. There are two ways to do so.
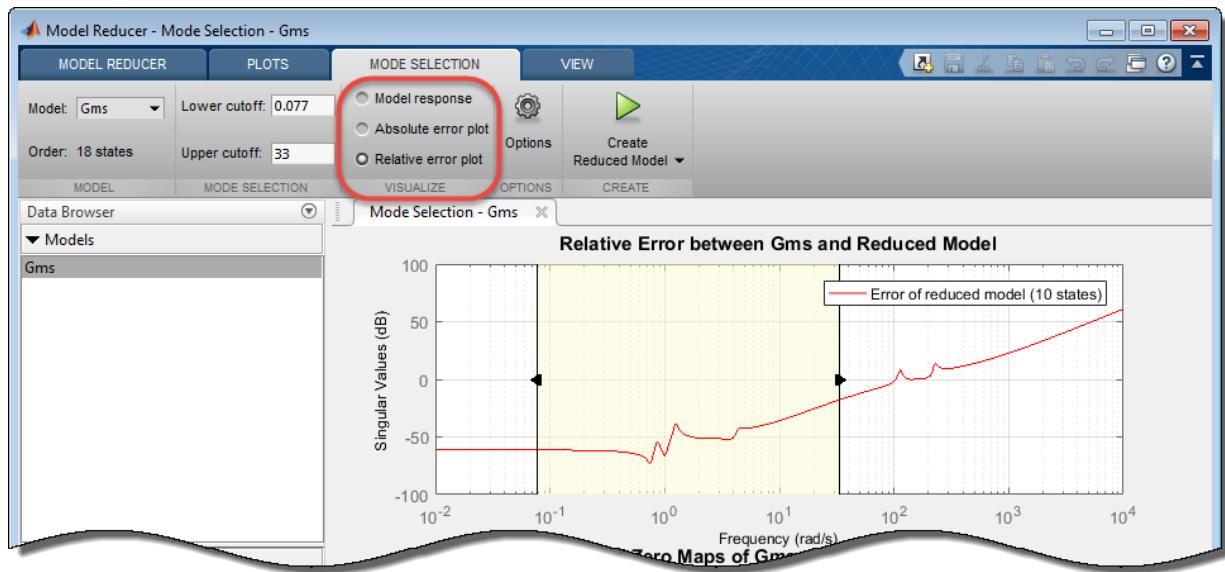
- On either the response plot or the pole-zero map, drag the boundaries of the shaded region or the shaded region itself.

- On the **Mode Selection** tab, enter lower and upper cutoff frequencies.

When you change the shaded regions or cutoff frequencies, **Model Reducer** automatically computes a new reduced-order model. All poles retained in the reduced
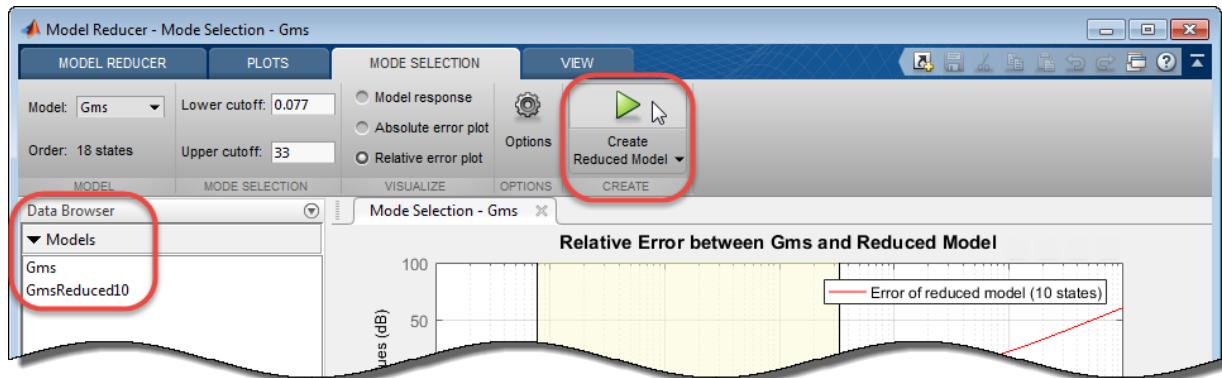
model fall within the shaded region on the pole-zero map. The reduced model might contain zeros that fall outside the shaded region.

4   Optionally, examine absolute or relative error between the original and simplified model. Select the error-plot type using the buttons on the **Mode Selection** tab.

For more information about using the analysis plots, see "Visualize Reduced-Order Models in the Model Reducer App" on page 6-65.

5   When you have one or more reduced models that you want to store and analyze further, click [>]. The new model appears in the **Data Browser**.
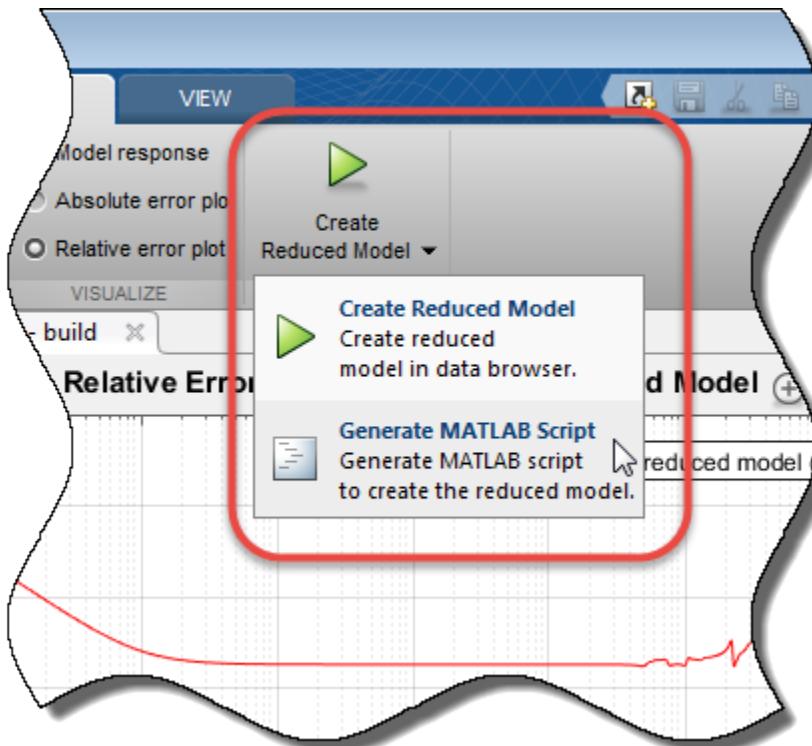
After creating a reduced model in the **Data Browser**, you can continue adjusting the mode-selection region to create reduced models with different orders for analysis and comparison.

You can now perform further analysis with the reduced model. For example:

- Examine other responses of the reduced system, such as the step response or Nichols plot. To do so, use the tools on the **Plots** tab. See "Visualize Reduced-Order Models in the Model Reducer App" on page 6-65 for more information.

- Export reduced models to the MATLAB workspace for further analysis or control design. On the **Model Reducer** tab, click ![export icon] **Export**.

Generate MATLAB Code for Mode Selection

To create a MATLAB script you can use for further model-reduction tasks at the command line, click **Create Reduced Model**, and select **Generate MATLAB Script**.
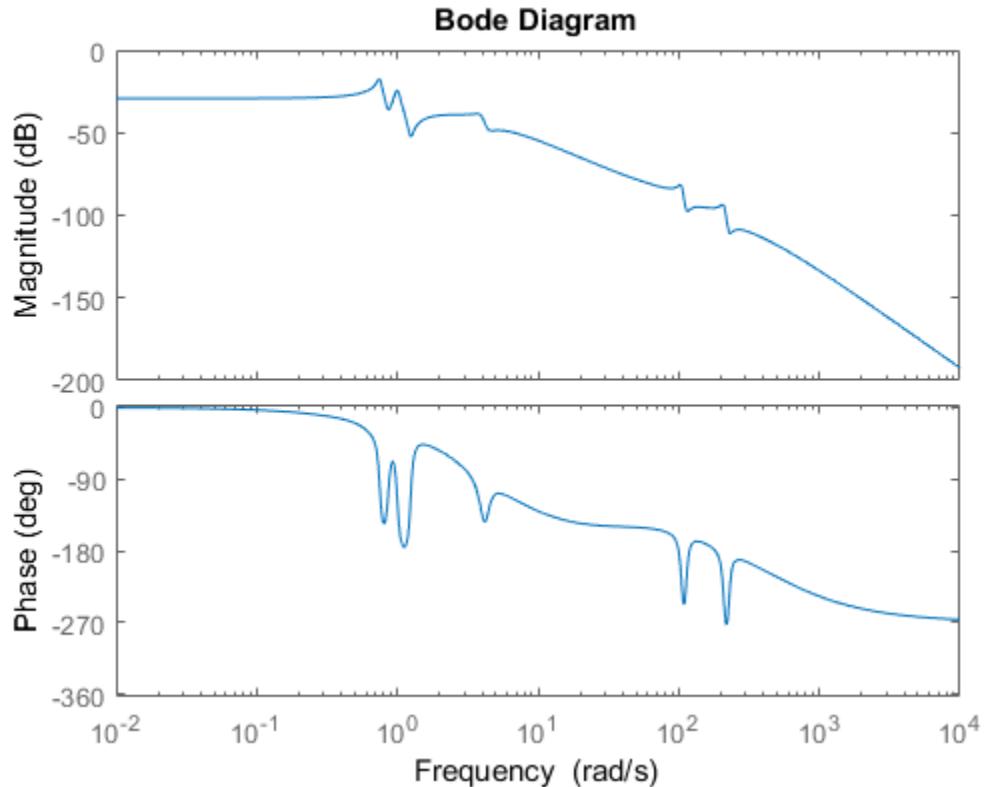
Model Reducer creates a script that uses the `freqsep` command to perform model reduction with the parameters you have set on the **Mode Selection** tab. The script opens in the MATLAB editor.

Mode Selection at the Command Line

To reduce the order of a model by mode selection at the command line, use `freqsep`. This command separates a dynamic system model into slow and fast components around a specified frequency.

For this example, load the model `Gms` and examine its frequency response.

```
load modeselect Gms
bodeplot(Gms)
```
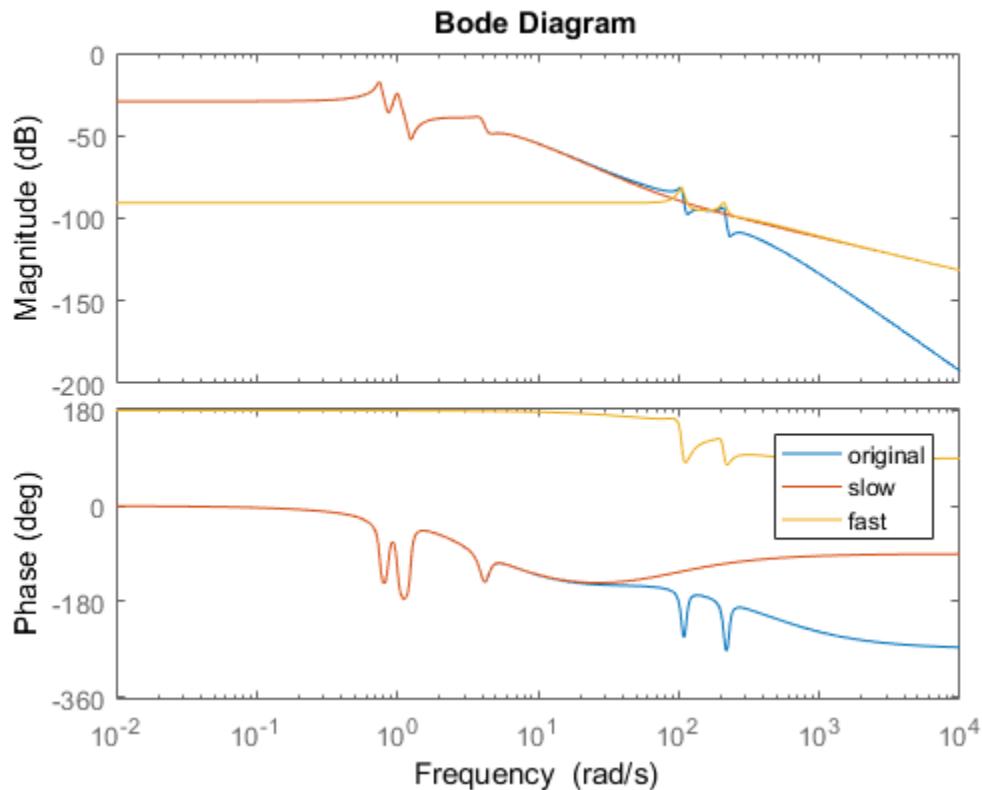
Gms has two sets of resonances, one at relatively low frequency and the other at relatively high frequency. Suppose that you want to tune a controller for Gms, but the actuator in your system is limited to a bandwidth of about 3 rad/s, in between the two groups of resonances. To simplify calculation and tuning using Gms, you can use mode selection to eliminate the high-frequency dynamics.

```
[Gms_s,Gms_f] = freqsep(Gms,30);
```

freqsep decomposes Gms into slow and fast components such that Gms = Gms_s + Gms_f. All modes (poles) with natural frequency less than 30 are in Gms_s, and the higher-frequency poles are in Gms_f.

```
bodeplot(Gms,Gms_s,Gms_f)
legend('original','slow','fast')
```

Bode Diagram

The slow component, `Gms_s`, contains only the lower-frequency resonances and matches the DC gain of the original model. Examine the orders of both models.

```
order(Gms)
order(Gms_s)
```

```
ans =

    18
```

```
ans =
```

10

When the high-frequency dynamics are unimportant for your application, you can use the 10th-order `Gms_s` instead of the original 18th-order model. If neglecting low-frequency dynamics is appropriate for your application, you can use `Gms_f`. To select modes that fall between a low-frequency and a high-frequency cutoff, use additional calls to `freqsep`.

See Also
`freqsep` | Model Reducer

Related Examples
- "Balanced Truncation Model Reduction" on page 6-16
- "Pole-Zero Simplification" on page 6-45
- "Model Reduction Basics" on page 6-2

Visualize Reduced-Order Models in the Model Reducer App

The plotting tools in the **Model Reducer** app let you examine and compare time-domain and frequency-domain responses of the original model and the reduced models you create in the app. Use these tools to help verify that the reduced-order model you choose to work with preserves the system behaviors that are important for your application.

For more general information about model reduction, see "Model Reduction Basics" on page 6-2
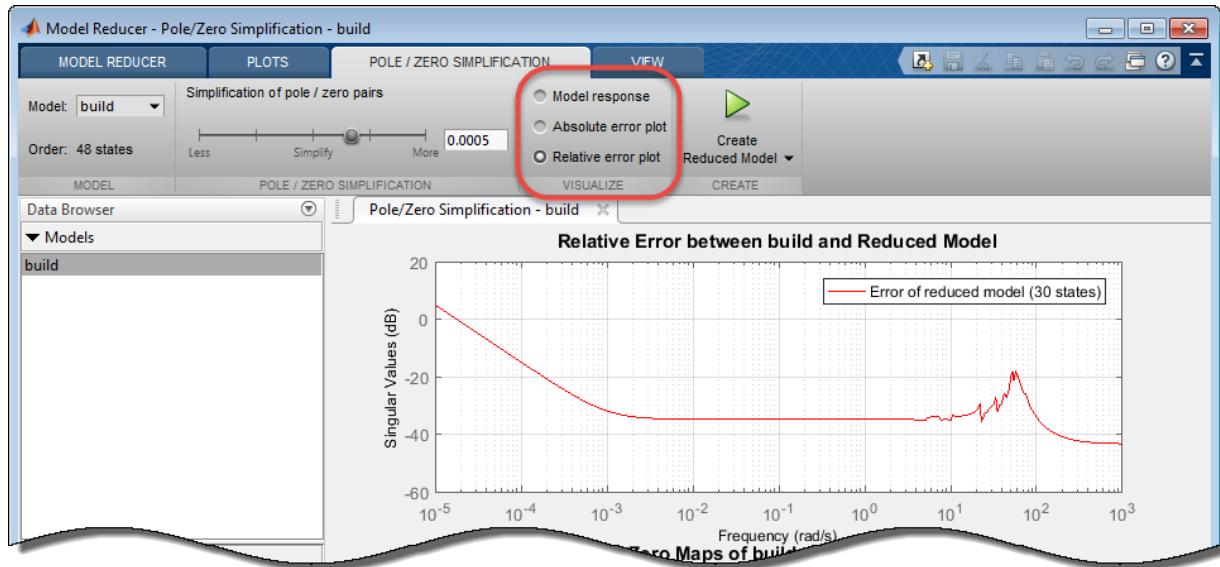
In this section...
"Error Plots" on page 6-65
"Response Plots" on page 6-66
"Plot Characteristics" on page 6-69
"Plot Tools" on page 6-71

Error Plots

By default, for any model reduction method, **Model Reducer** shows a frequency-response plot of both the original and reduced models. This plot is a Bode plot for SISO models, and a singular-value plot for MIMO models.

To more closely examine the differences between an original model and a reduced model, you can use absolute error or relative error plots. On any model reduction tab, click **Absolute error plot** or **Relative error plot** to view these plots.

- **Absolute error plot** — Shows the singular values of `G-Gr`, where `G` is the original model and `Gr` is the current reduced model.
- **Relative error plot** — Shows the singular values of `(G-Gr)/G`. This plot is useful when the model has very high or very low gain in the region that is important to your application. In such regions, absolute error can be misleading.
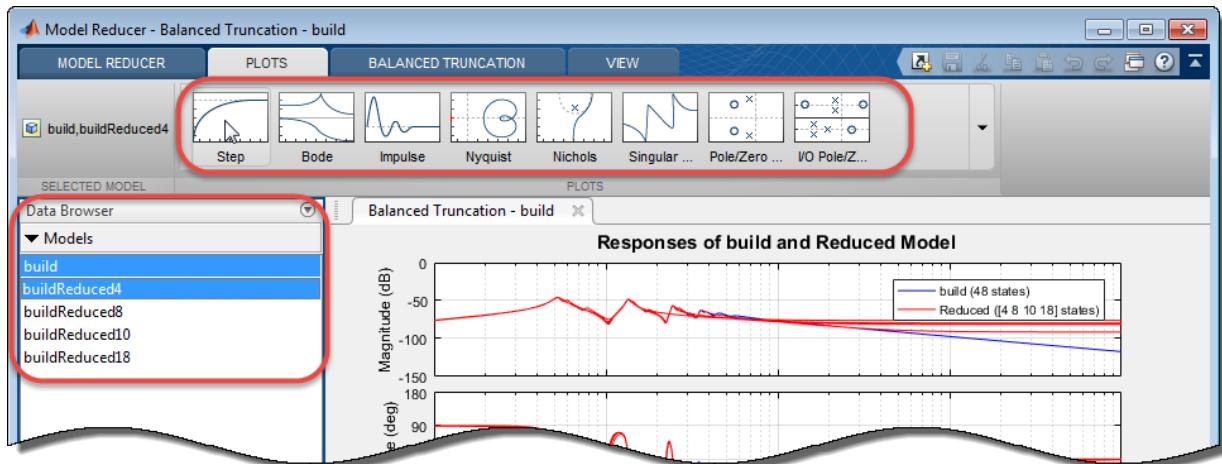
For SISO models, the singular-value plot is the magnitude of the frequency response.

Response Plots
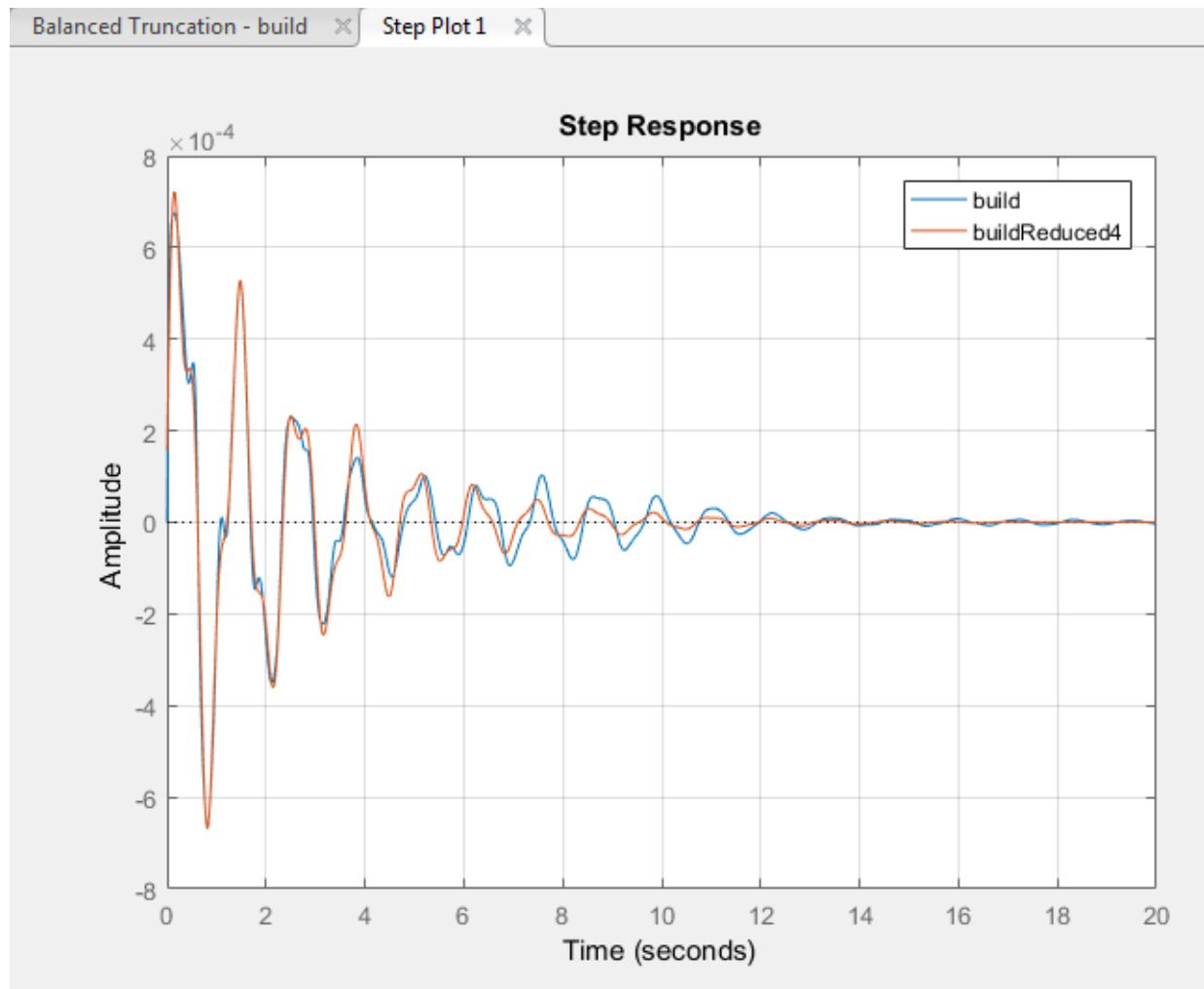
After you click to add one or more reduced models to the **Data Browser**, compare additional responses of the original and reduced models using the **Plots** tab.

Create New Response Plot

In the **Data Browser**, select one or more models to plot. (Ctrl-click to select multiple models.) Then, on the **Plots** tab, click the type of plot you want to create.
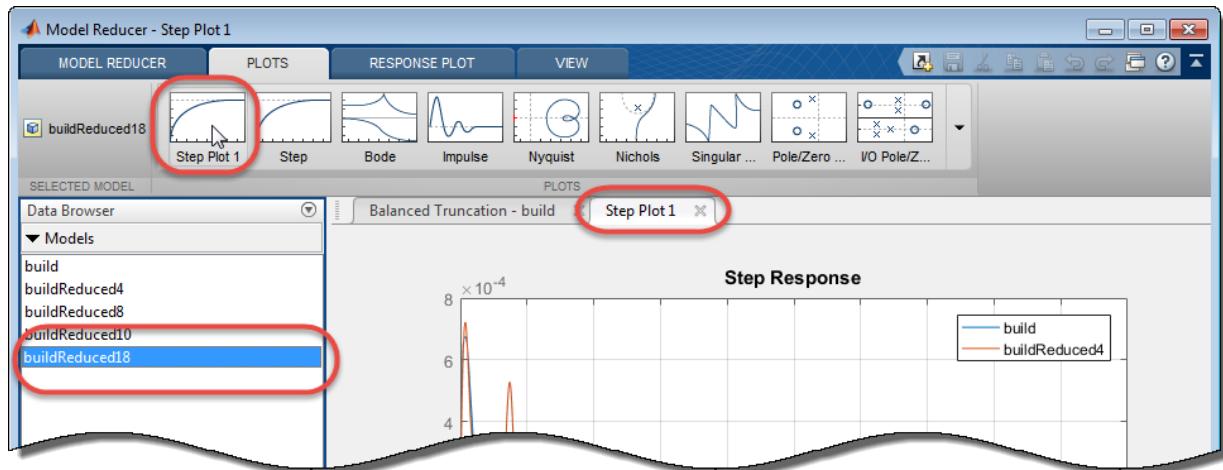
Model Reducer creates the plot.

Add Model to Existing Plot

In the **Data Browser**, select the model to add. Then, on the **Plots** tab, click the icon corresponding to the plot you want to update. Plots you have created appear on the left side of the plot gallery.
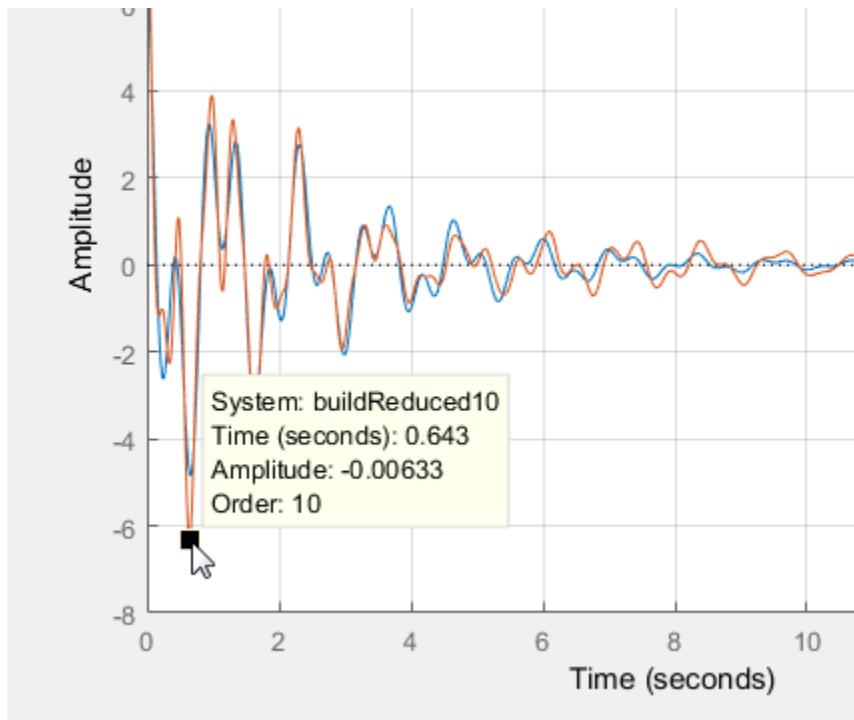
Model Reducer updates the plot with the new model.

Tip To expand the gallery view, click ▼.
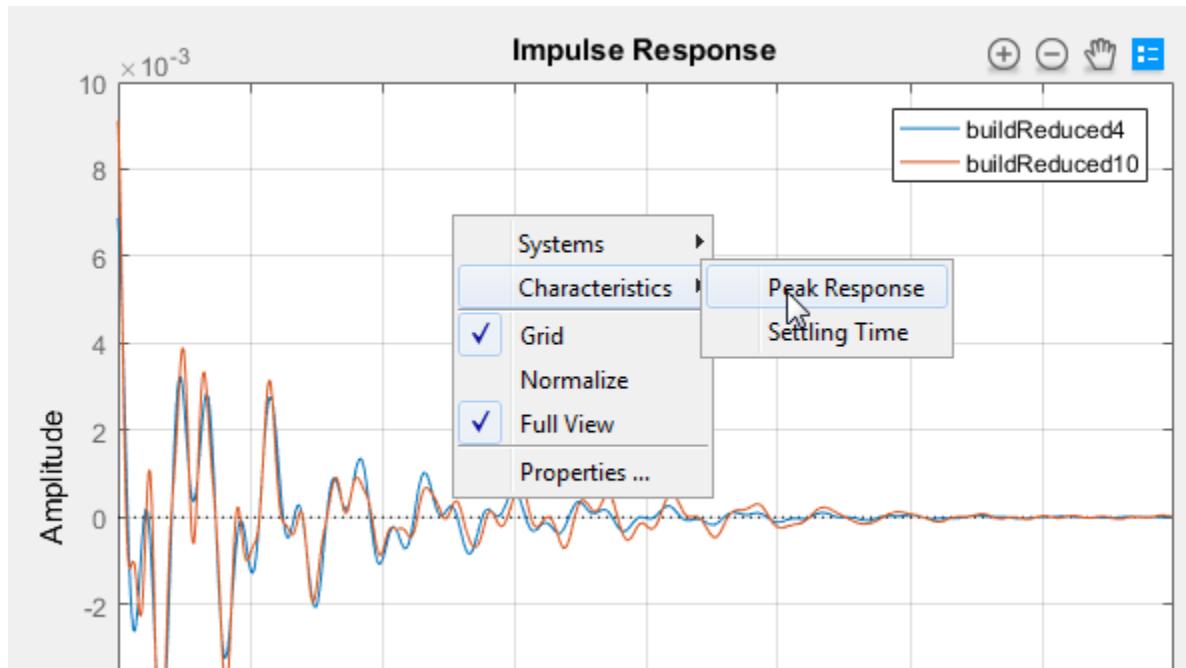
Plot Characteristics

On any plot in **Model Reducer**:

- To see response information and data values, click a line on the plot.
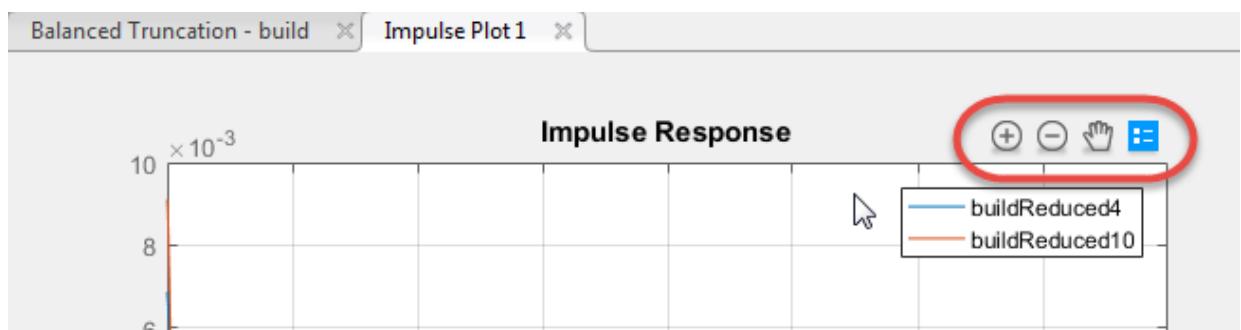
- To view system characteristics, right-click anywhere on the plot, as described in "System Characteristics on Response Plots" on page 8-10.
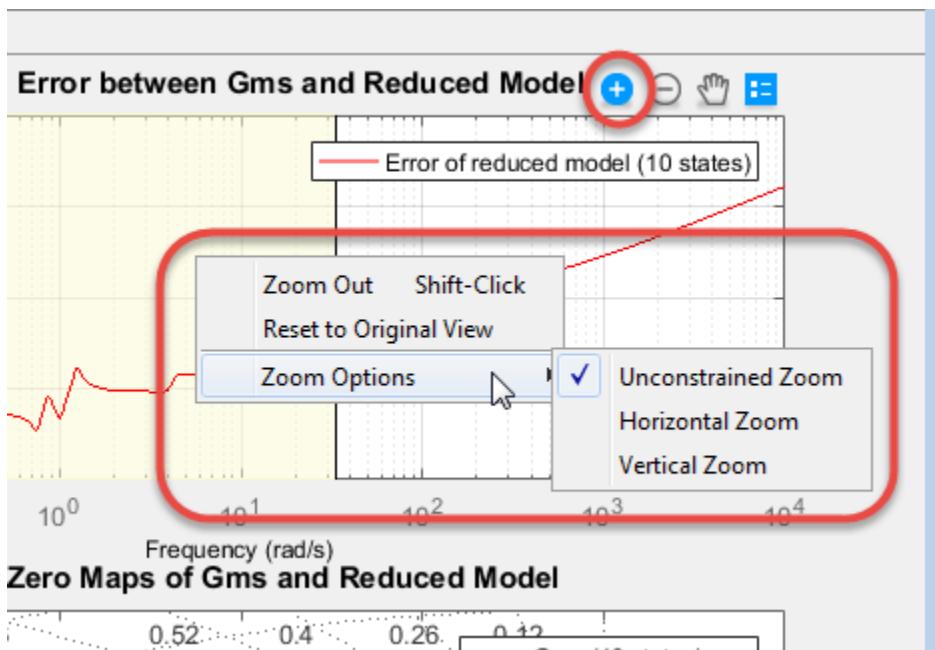
Plot Tools

Mouse over any plot to access plot tools at the upper right corner of the plot.
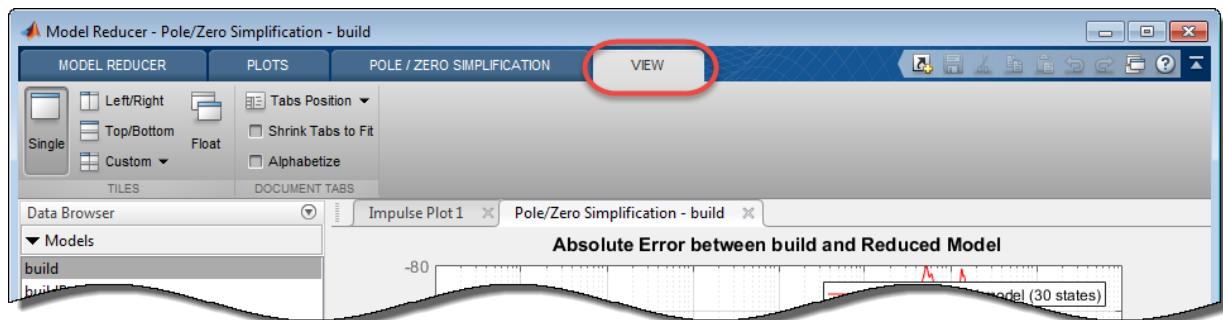
- ⊕ and ⊖ — Zoom in and zoom out. Click to activate, and drag the cursor over the region to zoom. The zoom icon turns dark when zoom is active. Right-click while zoom is active to access additional zoom options.Click the icon again to deactivate.

- 🖐 — Pan. Click to activate, and drag the cursor across the plot area to pan. The pan icon turns dark when pan is active. Right-click while pan is active to access additional pan options. Click the icon again to deactivate.

- ▦ — Legend. By default, the plot legend is active. To toggle the legend off and on, click this icon. To move the legend, drag it to a new location on the plot.

To change the way plots are tiled or sorted, use the options on the **View** tab.

See Also

Model Reducer

Related Examples

- "Balanced Truncation Model Reduction" on page 6-16
- "Mode-Selection Model Reduction" on page 6-55
- "Pole-Zero Simplification" on page 6-45

Linear Analysis

7

Time Domain Analysis

Choosing a Time-Domain Analysis Command

When you perform time-domain analysis of a dynamic system model, you may want one or more of the following:

- A plot of the system response as a function of time.
- Numerical values of the system response in a data array.
- Numerical values of characteristics of the system response such as peak response or settling time.

Control System Toolbox time-domain analysis commands can obtain these results for any kind of dynamic system model (for example, continuous or discrete, SISO or MIMO, or arrays of models) except for frequency response data models.

To obtain numerical data, use:

- `step`,`impulse`,`initial`,`lsim` — System response data at a vector of time points.
- `stepinfo`,`lsiminfo` — Numerical values of system response characteristics such as settling time and overshoot.

To obtain response plots, use:

- `stepplot`,`impulseplot`,`initialplot`,`lsimplot` — Plot system response data, visualize response characteristics on plots, compare responses of multiple systems on a single plot.
- `linearSystemAnalyzer` — Linear analysis tool for plotting many types of system responses simultaneously, including both time-domain and frequency-domain responses

Related Examples
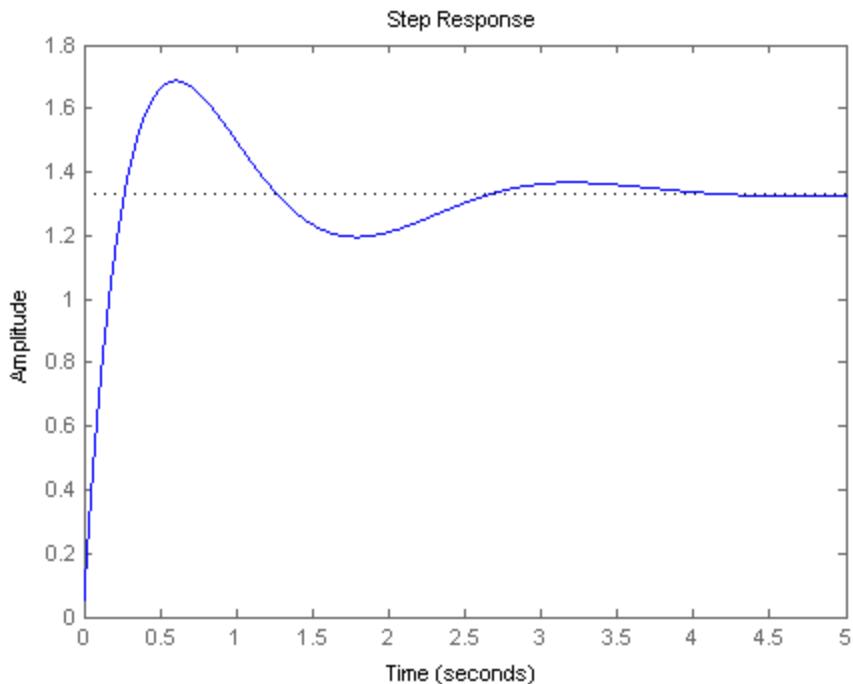
Time-Domain Response Data and Plots

This example shows how to obtain step and impulse response data, as well as step and impulse response plots, from a dynamic system model.

Create a transfer function model and plot its response to a step input at $t = 0$.

```
H = tf([8 18 32],[1 6 14 24]);
stepplot(H);
```


`stepplot` plots the step response on the screen. Unless you specify a time range to plot, `stepplot` automatically chooses a time range that illustrates the system dynamics.
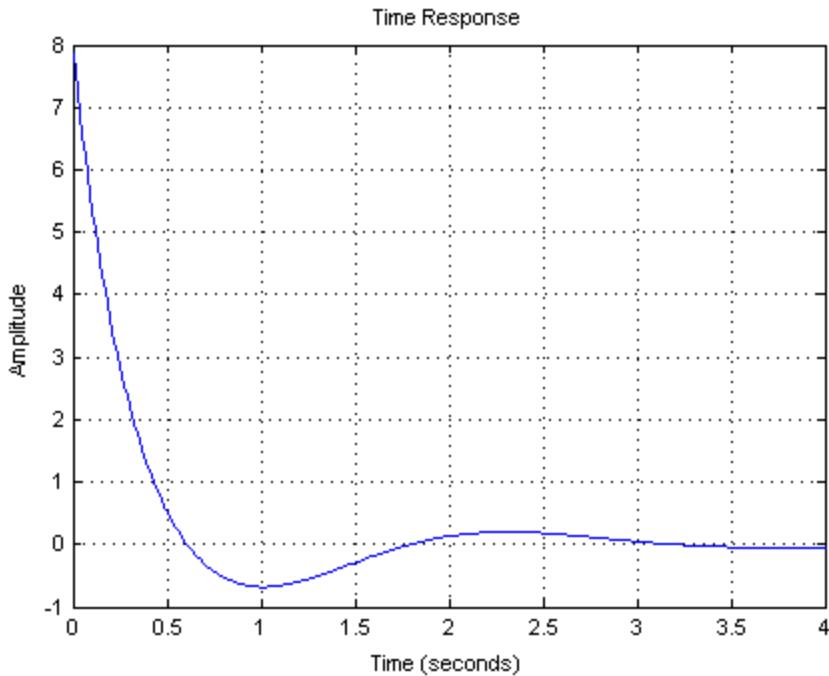
Calculate the step response data from $t = 0$ (application of the step input) to $t = 8$ s.

```
[y,t] = step(H,8);
```

When you call `step` with output arguments, the command returns the step response data `y`. The vector `t` contains corresponding time values.

Plot the response of `H` to an impulse input applied at $t = 0$. Plot the response with a grid.

```
opts = timeoptions;
opts.Grid = 'on';
impulseplot(H,opts)
```


Use the `timeoptions` command to define options sets for time-domain plotting commands like `impulseplot` and `stepplot`.

Calculate 200 points of impulse response data from $t = 1$ (one second after application of the impulse input) to $t = 3$s.

```
[y,t] = impulse(H,linspace(1,3,200));
```

As for `step`, you can omit the time vector to allow `impulse` to automatically select a time range.

See Also
`impulse` | `impulseplot` | `step` | `stepplot` | `timeoptions`

Related Examples
- "System Characteristics on Response Plots" on page 7-6
- "Time-Domain Responses of Multiple Models" on page 7-16
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About
- "Choosing a Time-Domain Analysis Command" on page 7-2
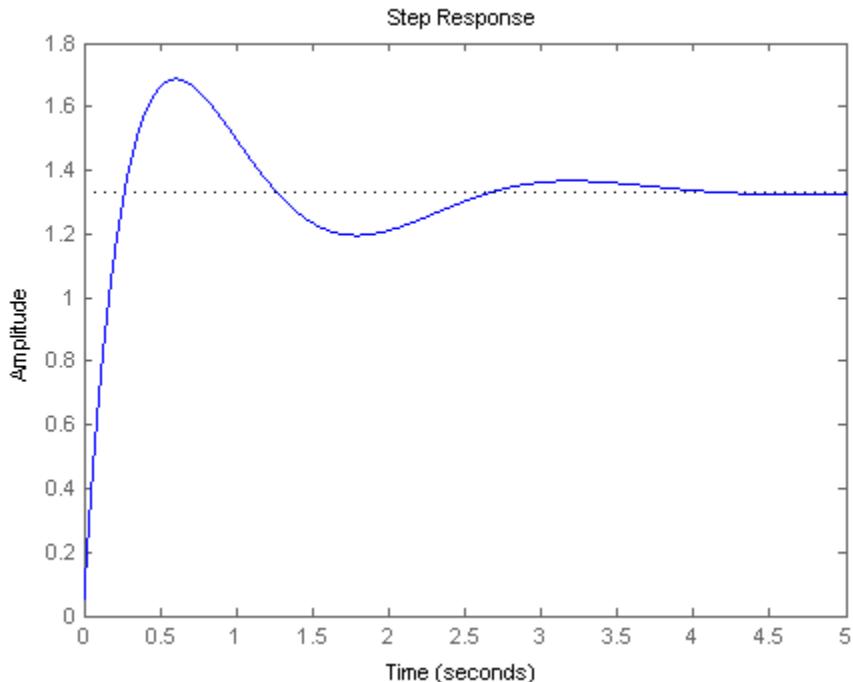
System Characteristics on Response Plots

This example shows how to display system characteristics such as settling time and overshoot on step response plots.

You can use similar procedures to display system characteristics on impulse response plots or initial value response plots, such as peak response or settling time.
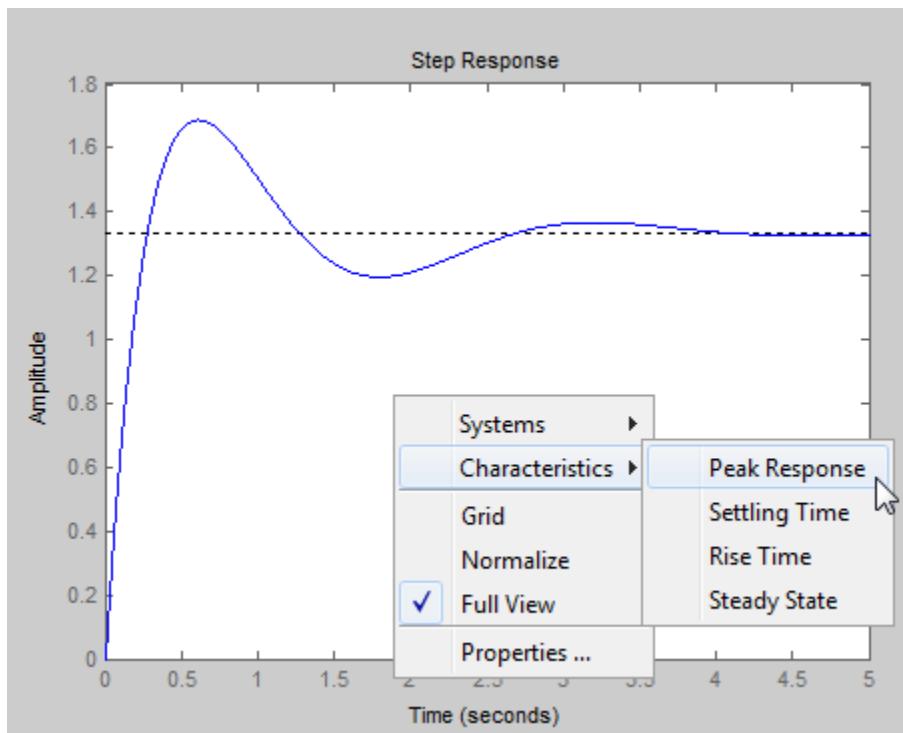
Create a transfer function model and plot its response to a step input at $t = 0$.

```
H = tf([8 18 32],[1 6 14 24]);
stepplot(H)
```
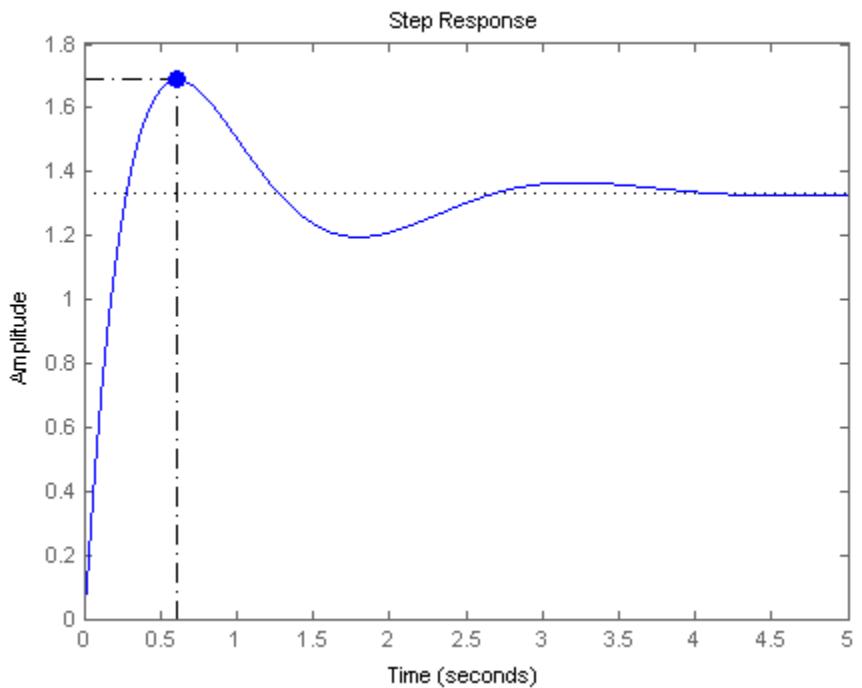

Display the peak response on the plot.
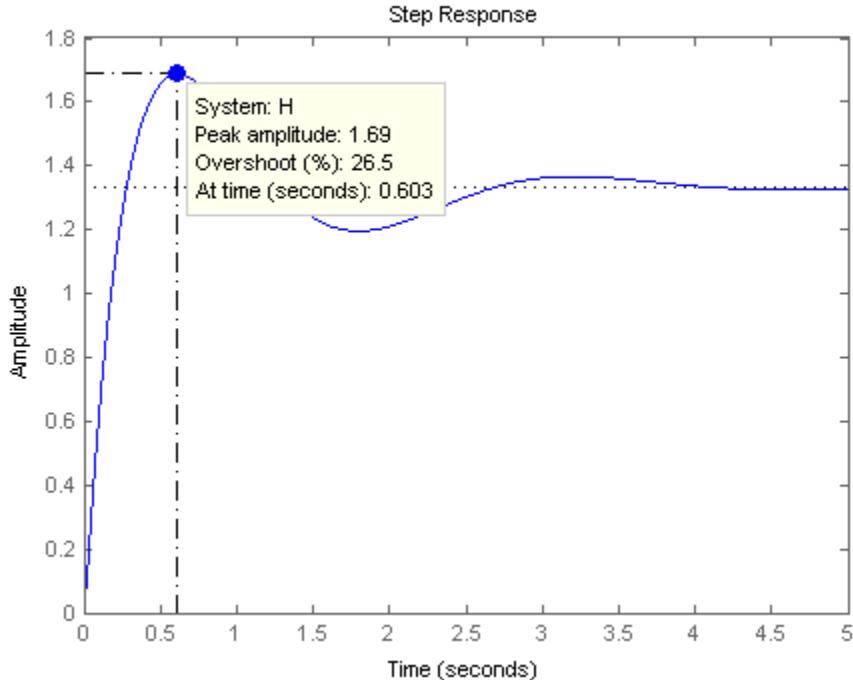
Right-click anywhere in the figure and select **Characteristics** > **Peak Response** from the menu.

A marker appears on the plot indicating the peak response. Horizontal and vertical dotted lines indicate the time and amplitude of that response.

Click the marker to view the value of the peak response and the overshoot in a datatip.

You can use a similar procedure to select other characteristics such as settling time and rise time from the **Characteristics** menu and view the values.

See Also
impulse | lsiminfo | step | stepinfo

Related Examples
- "Numeric Values of Time-Domain System Characteristics" on page 7-10
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About
- "Choosing a Time-Domain Analysis Command" on page 7-2

Numeric Values of Time-Domain System Characteristics

This example shows how to obtain numeric values of step response characteristics such as rise time, settling time, and overshoot using `stepinfo`.

You can use similar techniques with `lsiminfo` to obtain characteristics of the system response to an arbitrary input or initial conditions.

Create a dynamic system model and get numeric values of the system's step response characteristics.

```
H = tf([8 18 32],[1 6 14 24]);
data = stepinfo(H)

data =

        RiseTime: 0.2087
     SettlingTime: 3.4972
      SettlingMin: 1.1956
      SettlingMax: 1.6872
        Overshoot: 26.5401
       Undershoot: 0
             Peak: 1.6872
         PeakTime: 0.6033
```

The output is a structure that contains values for several step response characteristics. To access these values or refer to them in other calculations, use dot notation. For example, `data.Overshoot` is the overshoot value.

Calculate the time it takes the step response of `H` to settle within 0.5% of its final value.

```
data = stepinfo(H,'SettlingTimeThreshold',0.005);
t05 = data.SettlingTime

t05 =

    4.8897
```

By default, `stepinfo` defines the settling time as the time it takes for the output to settle within 0.02 (2%) of its final value. Specifying a more stringent `'SettlingTimeThreshold'` of 0.005 results in a longer settling time.

For more information about the options and the characteristics, see the `stepinfo` reference page.

See Also
`lsiminfo` | `stepinfo`

Related Examples
- "System Characteristics on Response Plots" on page 7-6
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About
- "Choosing a Time-Domain Analysis Command" on page 7-2
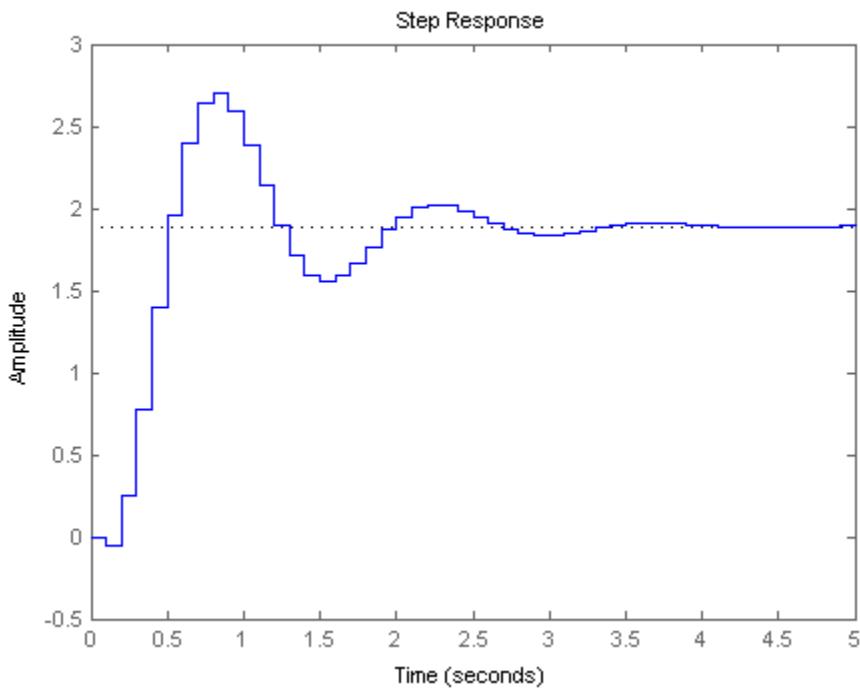
Time-Domain Responses of Discrete-Time Model

This example shows how to obtain a step response and step response data for a discrete-time dynamic system model.

You can use the techniques of this example with commands such as `impulse`, `initial`, `impulseplot`, and `initialpot` to obtain time-domain responses of discrete-time models. Obtaining time-domain responses of discrete-time models is the same as for continuous-time models, except that the time sample points are limited by the sample time `Ts` of the model.

Create a discrete-time transfer function model and plot its response to a step input at $t = 0$.

```
H = tf([-0.06,0.4],[1,-1.6,0.78],0.1);
stepplot(H)
```

For discrete-time models, `stepplot` plots the response at multiples of the sample time, assuming a hold between samples.

Compute the step response of H between 0.5 and 2.5 seconds.

```
[y,t] = step(H,0.5:0.1:2.5);
```

When you specify a time vector for the response of a discrete-time model, the time step must match the sample time Ts of the discrete-time model. The vector t contains the time points between 0.5 and 2.5 seconds, at multiples of the sample time of H, 0.1 s. The vector y contains the corresponding step response values.

See Also

`impulse` | `impulseplot` | `initial` | `initialplot` | `step` | `stepplot`

Related Examples

- "Time-Domain Responses of MIMO Model" on page 7-14
- "Time-Domain Responses of Multiple Models" on page 7-16
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About
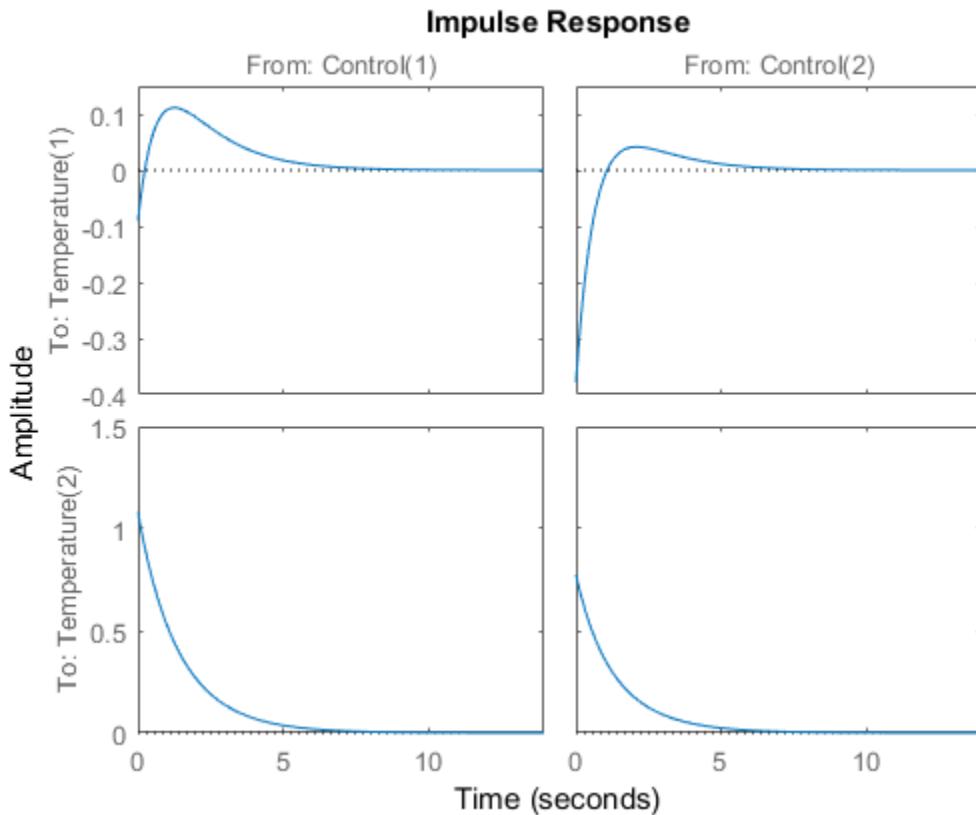
- "Choosing a Time-Domain Analysis Command" on page 7-2

Time-Domain Responses of MIMO Model

This example shows how to obtain impulse response data and plots for a multi-input, multi-output (MIMO) model using `impulse`. You can use the same techniques to obtain other types of time-domain responses of MIMO models.

Create a MIMO model and plot its response to a `t = 0` impulse at all inputs.

```
H = rss(2,2,2);
H.InputName = 'Control';
H.OutputName = 'Temperature';
impulseplot(H)
```

impulseplot plots the response of each output to an impulse applied at each input. (Because rss generates a random state-space model, you might see different responses from those pictured.) The first column of plots shows the response of each output to an impulse applied at the first input, Control(1). The second column shows the response of each output to an impulse applied at the second input, Control(2).

Calculate the impulse responses of all channels of H, and examine the size of the output.

```
[y,t] = impulse(H);
size(y)

ans =

   207    2    2
```

The first dimension of the data array y is the number of samples in the time vector t. The impulse command determines this number automatically if you do not supply a time vector. The remaining dimensions of y are the numbers of outputs and inputs in H. Thus, y(:,i,j) is the response at the i th output of H to an impulse applied at the j th input.

See Also
impulse | impulseplot | initial | initialplot | step | stepplot

Related Examples
- "Time-Domain Responses of Multiple Models" on page 7-16
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About
- "Choosing a Time-Domain Analysis Command" on page 7-2

Time-Domain Responses of Multiple Models
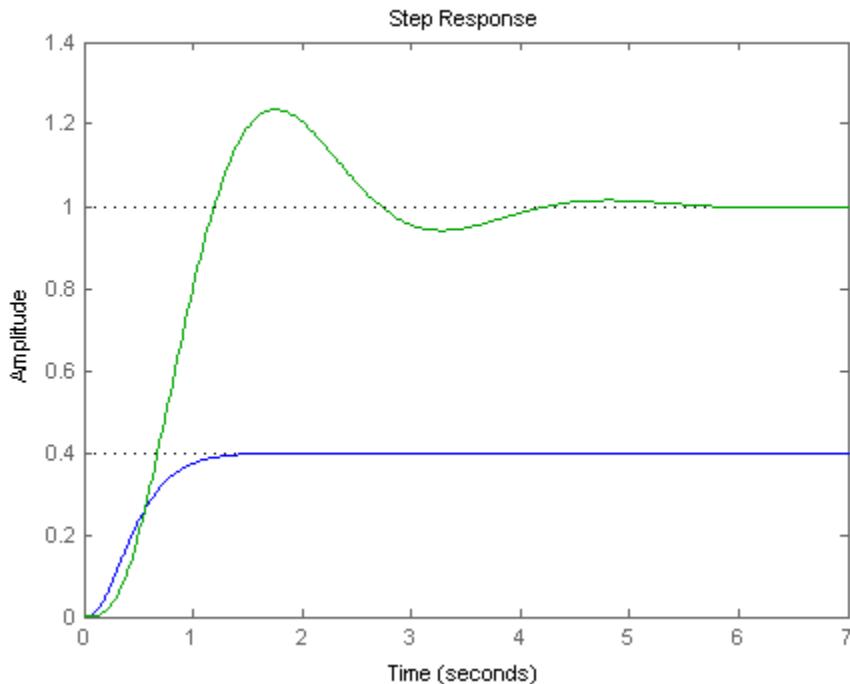
This example shows how to compare the step responses of multiple models on a single plot using `step`. This example compares the step response of an uncontrolled plant to the closed-loop step response of the plant with two different PI controllers.

You can use similar techniques with other response commands, such as `impulse` or `initial`, to obtain plots of responses of multiple models.

Obtain two models whose time responses you want to compare, and plot them on a single step plot.

For example, you can compare a third-order plant `G`, and the closed-loop response of `G` with a controller `C1` having integral action.

```
G = zpk([],[-5 -5 -10],100);

C1 = pid(0,4.4);
CL1 = feedback(G*C1,1);

step(G,CL1);
```
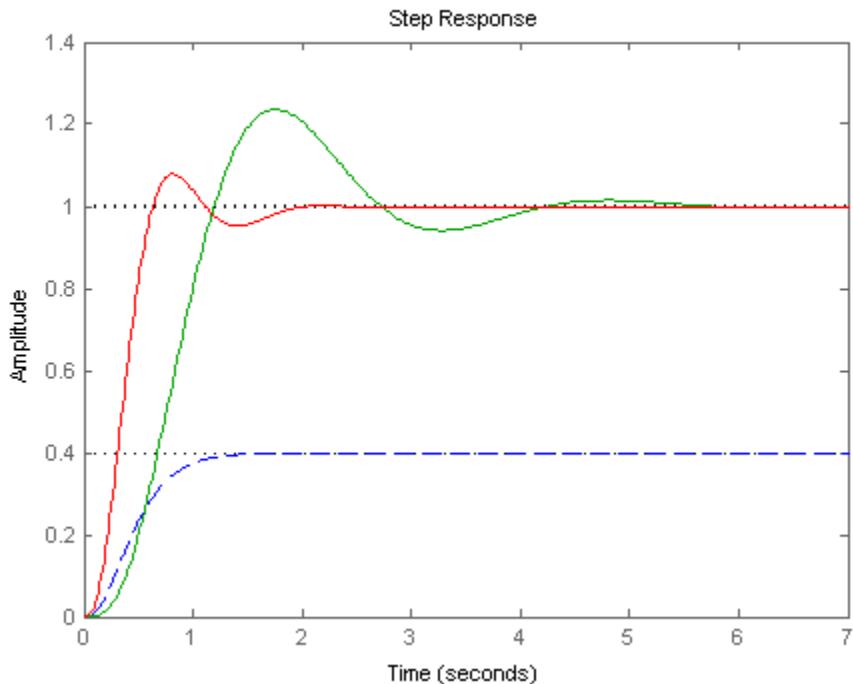
When you provide multiple models to `step` as input arguments, the command displays the responses of both models on the same plot. If you do not specify a time range to plot, `step` attempts to choose a time range that illustrates the dynamics of all the models.

Compare the step response of the closed-loop model with another controller. Specify plot colors and styles for each response.

```
C2 = pid(2.9,7.1);
CL2 = feedback(G*C2,1);

step(G,'b--',CL1,'g-',CL2,'r-')
```

You can use formatting strings to specify plot color and style for each response in the plot. For example, `'g-'` specifies a solid green line for response CL2. For additional plot customization options, use `stepplot`.

See Also

`impulse` | `impulseplot` | `initial` | `initialplot` | `linearSystemAnalyzer` | `step` | `stepplot`

Related Examples

- "Time-Domain Responses of MIMO Model" on page 7-14
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About

- "Choosing a Time-Domain Analysis Command" on page 7-2

Joint Time- and Frequency-Domain Analysis

This example shows how to compare multiple types of responses side by side, including both time-domain and frequency-domain responses, using the interactive Linear System Analyzer tool.
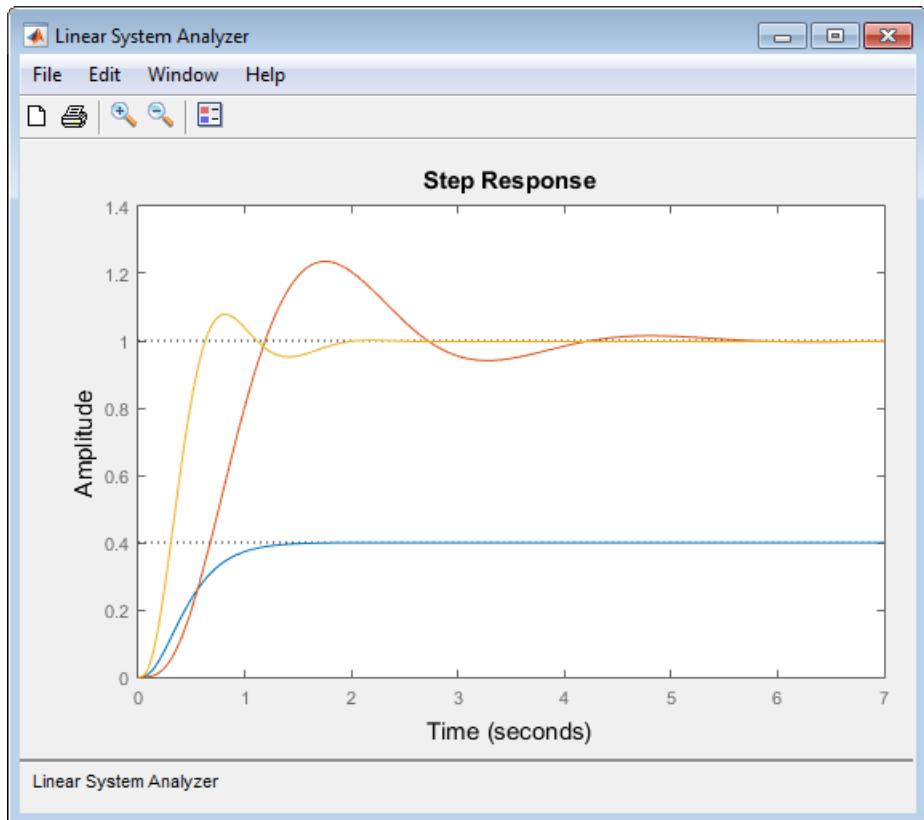
Obtain models whose responses you want to compare.

For example, compare a third-order plant G, and the closed-loop responses of G with two different controllers, C1 and C2.

```
G = zpk([],[-5 -5 -10],100);
C1 = pid(0,4.4);
T1 = feedback(G*C1,1);
C2 = pid(2.9,7.1);
T2 = feedback(G*C2,1);
```

Open the Linear System Analyzer tool to examine the responses of the plant and the closed-loop systems.
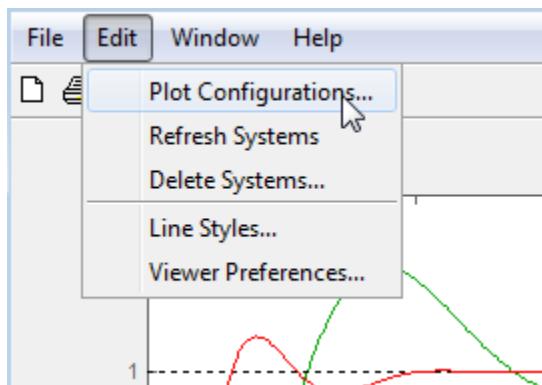
```
linearSystemAnalyzer(G,T1,T2)
```
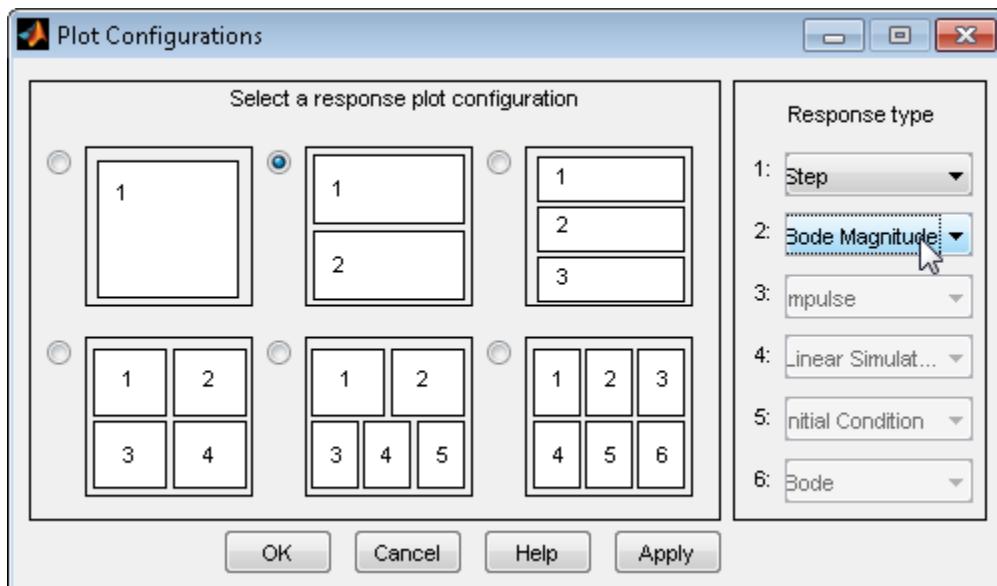
By default, the Linear System Analyzer launches with a plot of the step response of the three systems. Click ![icon] to add a legend to the plot.

Add plots of the impulse responses to the Linear System Analyzer display.

In the Linear System Analyzer, select **Edit** > **Plot Configurations** to open the Plot Configurations dialog box.
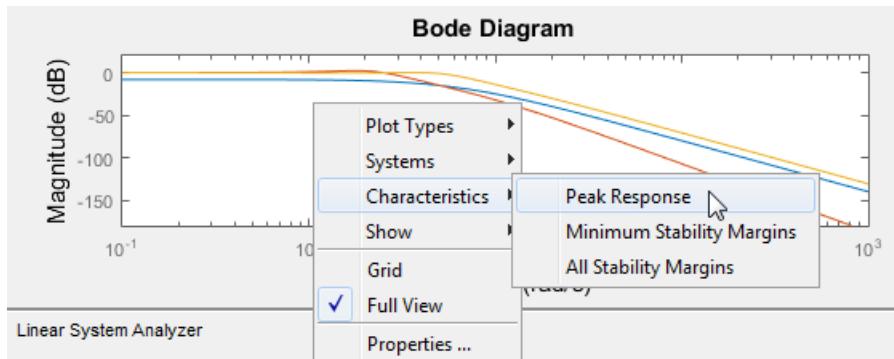
Select the two-plot configuration. In the Response Type area, select `Bode Magnitude` for the second plot type.
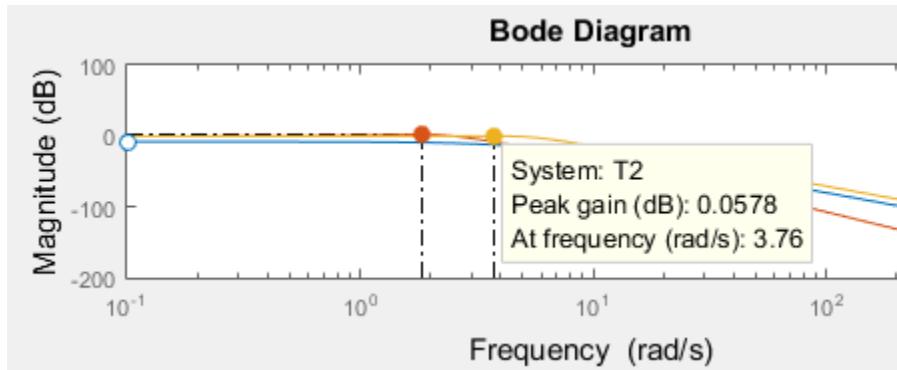
Click **OK** to add the Bode plots to the Linear System Analyzer display.

Display the peak values of the Bode responses on the plot.

Right-click anywhere in the Bode Magnitude plot and select **Characteristics** > **Peak Response** from the menu.
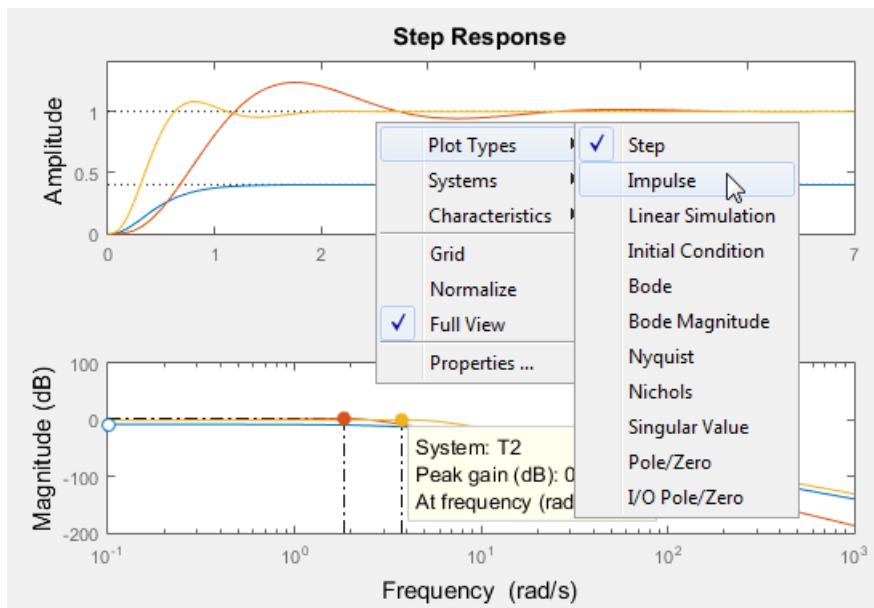
Markers appear on the plot indicating the peak response values. Horizontal and vertical dotted lines indicate the frequency and amplitude of those responses. Click on a marker to view the value of the peak response in a datatip.

You can use a similar procedure to select other characteristics such as settling time and rise time from the **Characteristics** menu and view the values.

You can also change the type of plot displayed in the Linear System Analyzer. For example, to change the first plot type to a plot of the impulse response, right-click anywhere in the plot. Select **Plot Types** > **Impulse**

The displayed plot changes to show the impulse of the three systems.

See Also

`impulse` | `impulseplot` | `initial` | `initialplot` | `linearSystemAnalyzer` |
`step` | `stepplot`

Related Examples

- "Time-Domain Responses of Multiple Models" on page 7-16

More About

- "Choosing a Time-Domain Analysis Command" on page 7-2

Response from Initial Conditions

This example shows how to compute and plot the response of a state-space (`ss`) model to specified initial state values using `initialplot` and `initial`.

Load a state-space model.

```
load ltiexamples sys_dc
sys_dc.InputName = 'Volts';
sys_dc.OutputName = 'w';
sys_dc.StateName = {'Current','w'};
sys_dc
```

```
sys_dc =

  A =
            Current          w
   Current       -4      -0.03
   w            0.75        -10

  B =
            Volts
   Current       2
   w             0

  C =
      Current          w
   w        0          1

  D =
      Volts
   w      0

Continuous-time state-space model.
```
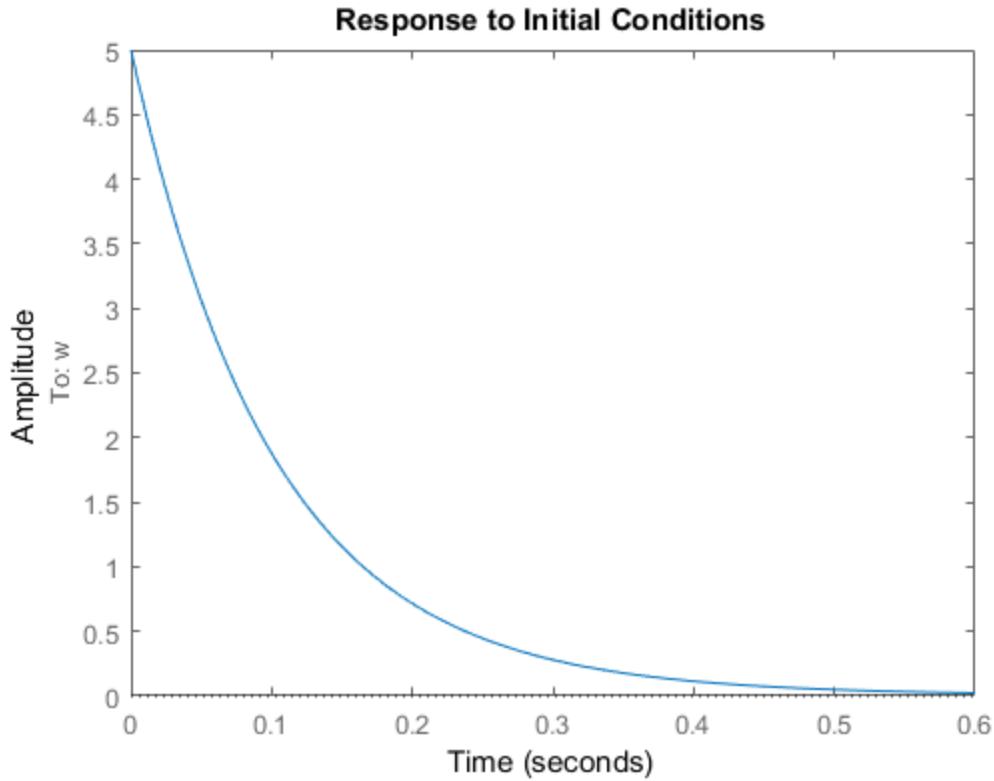
This example uses the SISO, 2-state model `sys_dc`. This model represents a DC motor. The input is an applied voltage, and the output is the angular rate of the motor ω. The states of the model are the induced current i (`x1`), and ω (`x2`). The model display in the command window shows the labeled input, output, and states.

Plot the undriven evolution of the motor's angular rate from an initial state in which the induced current is 1.0 amp and the initial rotation rate is 5.0 rad/s.

```
x0 = [1.0 5.0];
initialplot(sys_dc,x0)
```

Response to Initial Conditions

`initialplot` plots the time evolution from the specified initial condition on the screen. Unless you specify a time range to plot, `impulse` automatically chooses a time range that illustrates the system dynamics.

Calculate the time evolution of the output and the states of sys_dc from $t = 0$ (application of the step input) to $t = 1$ s.

```
t = 0:0.01:1;
[y,t,x] = initial(sys_dc,x0,t);
```

The vector y contains the output at each time step in t. The array x contains the state values at each time step. Therefore, in this example x is a 2-by-101 array. Each row of x contains the values of the two states of sys_dc at the corresponding time step.

See Also

impulse | initial | initialplot | step

Related Examples

- "Time-Domain Response Data and Plots" on page 7-3
- "Numeric Values of Time-Domain System Characteristics" on page 7-10
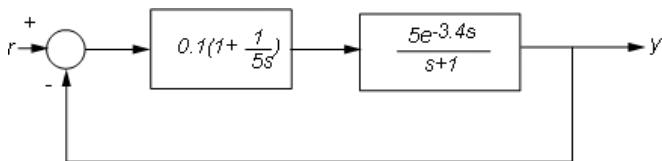
More About

- "Choosing a Time-Domain Analysis Command" on page 7-2

Analysis of Systems with Time Delays

You can use analysis commands such as `step`, `bode`, or `margin` to analyze systems with time delays. The software makes no approximations when performing such analysis.

For example, consider the following control loop, where the plant is modeled as first-order plus dead time:
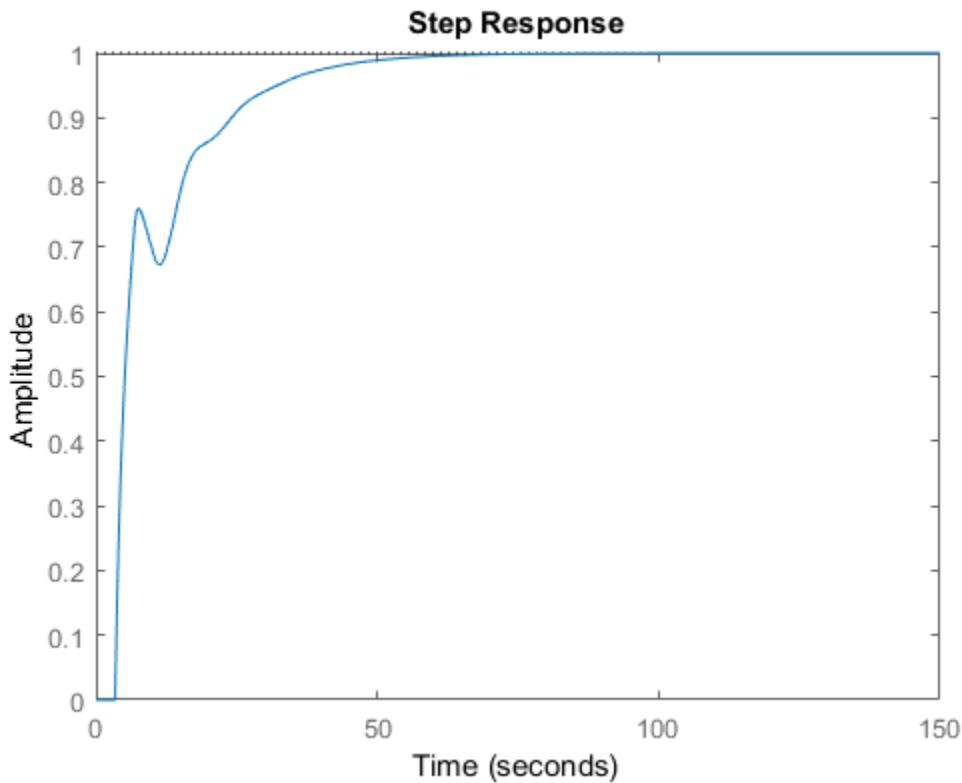
You can model the closed-loop system from `r` to `y` with the following commands:

```
s = tf('s');
P = 5*exp(-3.4*s)/(s+1);
C = 0.1 * (1 + 1/(5*s));
T = feedback(P*C,1);
```
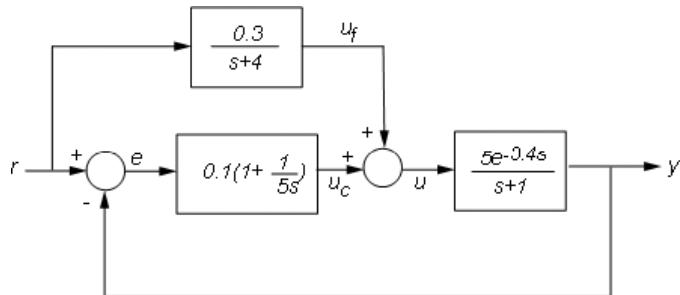
`T` is a state-space model with an internal delay. For more information about models with internal delays, see "Closing Feedback Loops with Time Delays" on page 2-46.

Plot the step response of `T`:

```
stepplot(T)
```

Step Response

For more complicated interconnections, you can name the input and output signals of each block and use `connect` to automatically take care of the wiring. Suppose, for example, that you want to add feedforward to the control loop of the previous model.
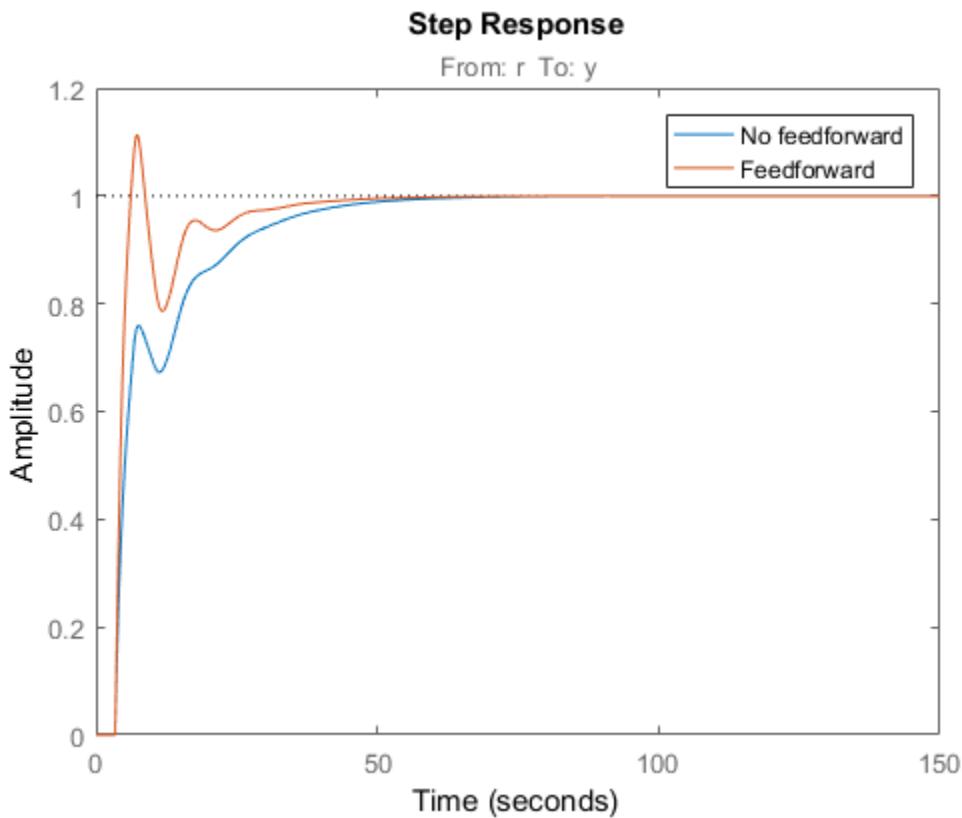
You can derive the corresponding closed-loop model `Tff` by

```
F = 0.3/(s+4);
P.InputName = 'u';
P.OutputName = 'y';
C.InputName = 'e';
C.OutputName = 'uc';
F.InputName = 'r';
F.OutputName = 'uf';
Sum1 = sumblk('e','r','y','+-');     % e = r-y
Sum2 = sumblk('u','uf','uc','++');   % u = uf+uc
Tff = connect(P,C,F,Sum1,Sum2,'r','y');
```

and compare its response with the feedback only design.

```
stepplot(T,Tff)
legend('No feedforward','Feedforward')
```

The state-space representation keeps track of the internal delays in both models.
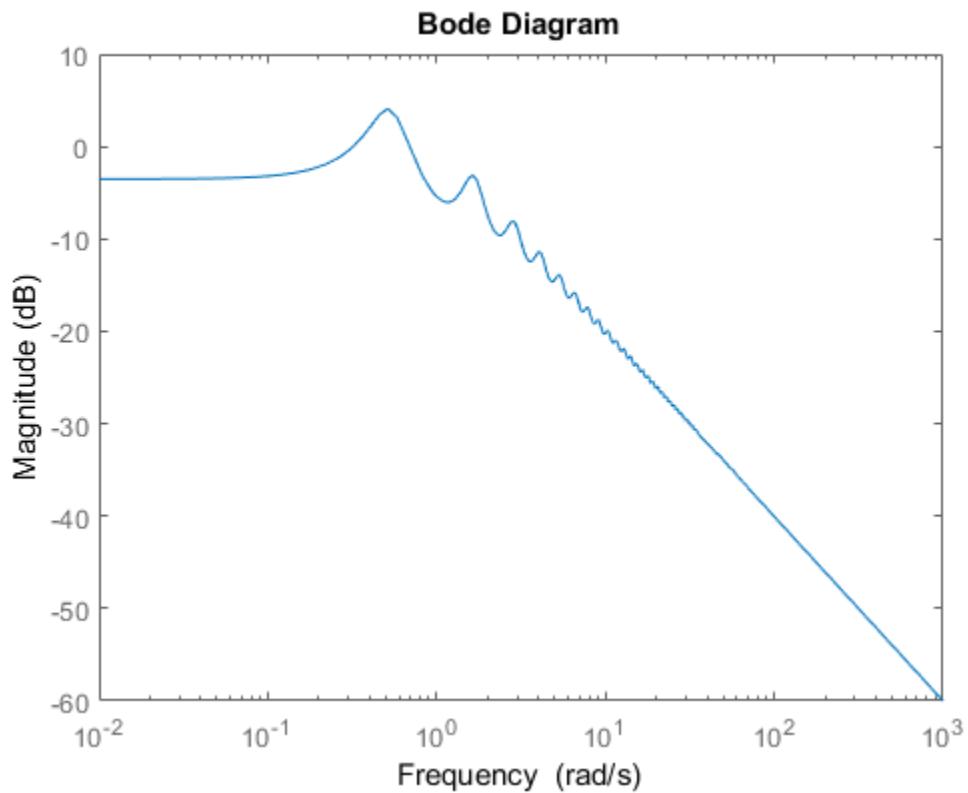
Considerations to Keep in Mind when Analyzing Systems with Internal Time Delays

The time and frequency responses of delay systems can look odd and suspicious to those only familiar with delay-free LTI analysis. Time responses can behave chaotically, Bode plots can exhibit gain oscillations, etc. These are not software or numerical quirks but real features of such systems. Below are a few illustrations of these phenomena.

Gain ripple:
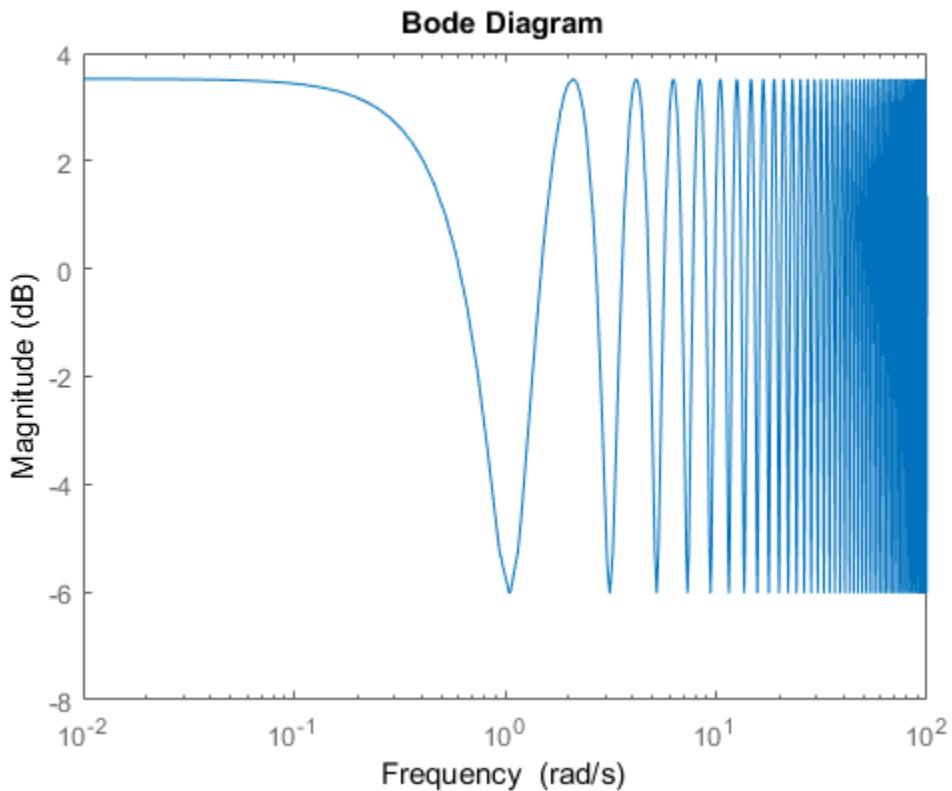
```
s = tf('s');
```

```
G = exp(-5*s)/(s+1);
T = feedback(G,.5);
bodemag(T)
```
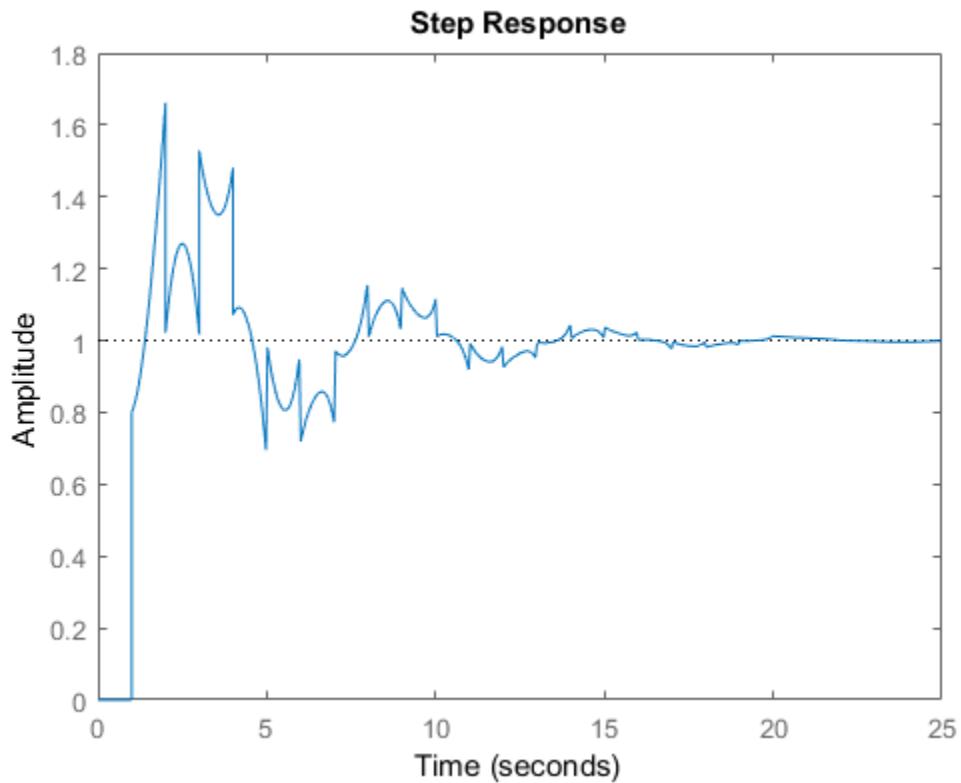

Gain oscillations:

```
G = 1 + 0.5 * exp(-3*s);
bodemag(G)
```

Jagged step response:
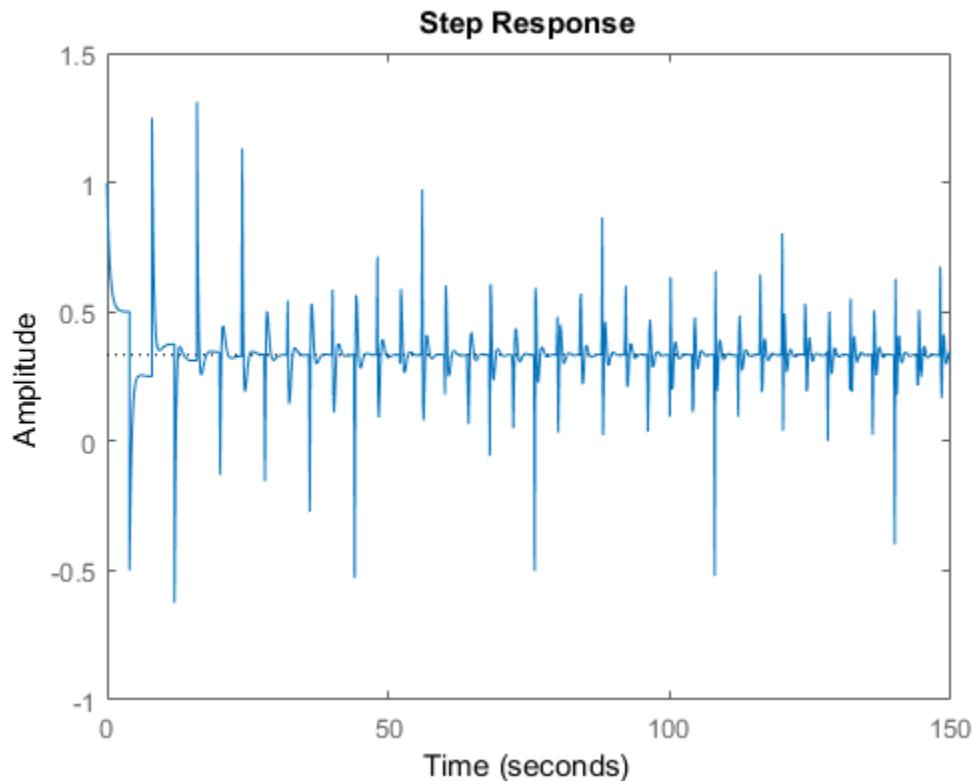
```
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(G,1);
stepplot(T)
```

Step Response

Note the rearrivals (echoes) of the initial step function.

Chaotic response:

```
G = 1/(s+1) + exp(-4*s);
T = feedback(1,G);
stepplot(T,150)
```

You can use Control System Toolbox tools to model and analyze these and other strange-appearing artifacts of internal delays.

Related Examples

- "Closing Feedback Loops with Time Delays" on page 2-46

More About

- "Time Delays in Linear Systems" on page 2-41
- "Internal Delays" on page 2-72

8

Frequency Domain Analysis

Choosing a Frequency-Domain Analysis Command

When you perform time-domain analysis of a dynamic system model, you may want one or more of the following:

- A plot of the system response as a function of frequency, or plots of pole and zero locations.
- Numerical values of the system response in a data array.
- Numerical values of characteristics of the system response such as stability margins, peak gains, or singular values.

Control System Toolbox frequency-domain analysis commands can obtain these results for any kind of dynamic system model (for example, continuous or discrete, SISO or MIMO, or arrays of models).

To obtain numerical data, use:

- `bode`,`bodemag`,`freqresp`,`nichols`,`nyquist` — System response data at a vector of frequency points.
- `margin`,`getPeakGain`,`getGainCrossover`,`sigma` — Numerical values of system response characteristics such as gain margins, phase margins, and singular values.

To obtain response plots, use:

- `bodeplot`,`nicholsplot`,`nyquistplot`,`sigmaplot` — Plot system response data, visualize response characteristics on plots, compare responses of multiple systems on a single plot.
- `linearSystemAnalyzer` — Linear analysis tool for plotting many types of system responses simultaneously, including both time-domain and frequency-domain responses

To obtain pole-zero maps, use:

- `pzplot`, `iopzplot` — Plot pole and zero locations in the complex plane.

If you have a generalized state-space (`genss`) model of a control system, you can extract various transfer functions from it for analysis using frequency-domain and time-domain analysis commands. Extract responses from such models using `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`.

Related Examples
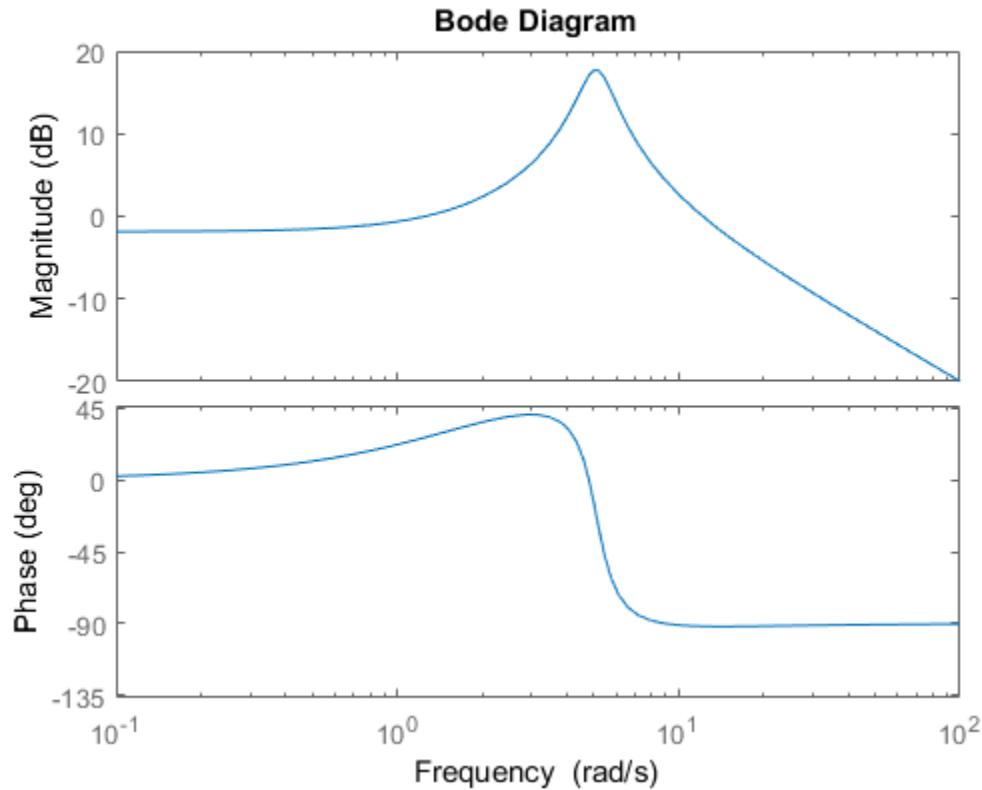
Frequency Response of a SISO System

This example shows how to plot the frequency response and obtain frequency response data for a single-input, single-output (SISO) dynamic system model.

Create a transfer function model and plot its frequency response.

```
H = tf([10,21],[1,1.4,26]);
bodeplot(H)
```


bodeplot plots the frequency response on the screen. Unless you specify a time range to plot, stepplot automatically chooses a time range that illustrates the system dynamics.

Calculate the frequency response between 1 and 13 rad/s.

```
[mag,phase,w] = bode(H,{1,13});
```

When you call `bode` with output arguments, the command returns vectors `mag` and `phase` containing the magnitude and phase of the frequency response. The cell array input `{1,13}` tells `bode` to calculate the response at a grid of frequencies between 1 and 13 rad/s. `bode` returns the frequency points in the vector `w`.

See Also
bode | bodeoptions | bodeplot

Related Examples
- "Frequency Response of a MIMO System" on page 8-6
- "Numeric Values of Frequency-Domain Characteristics of SISO Model" on page 8-13

More About
- "Choosing a Frequency-Domain Analysis Command" on page 8-2

Frequency Response of a MIMO System

This example shows how to examine the frequency response of a multi-input, multi-output (MIMO) system in two ways: by computing the frequency response, and by computing the singular values.

Calculate the frequency response of a MIMO model and examine the size of the output.

```
H = rss(2,2,2);
H.InputName = 'Control';
H.OutputName = 'Temperature';
[mag,phase,w] = bode(H);
size(mag)
```

```
ans =

    2    2    70
```

The first and second dimension of the data array `mag` are the number of outputs and inputs of `H`. The third dimension is the number of points in the frequency vector `w`. (The `bode` command determines this number automatically if you do not supply a frequency vector.) Thus, `mag(i,j,:)` is the frequency response from the `j` th input of `H` to the `i` th output, in absolute units. The phase data array `phase` takes the same form as `mag`.

Plot the frequency response of each input/output pair in `H`.

```
bodeplot(H)
```

Bode Diagram

bodeplot plots the magnitude and the phase of the frequency response of each input/output pair in H. (Because rss generates a random state-space model, you might see different responses from those pictured.) The first column of plots shows the response from the first input, Control(1), to each output. The second column shows the response from the second input, Control(2), to each output.

Plot the singular values of H as a function of frequency.

```
sigmaplot(H)
```

sigmaplot plots the singular values of the MIMO system H as a function of frequency. The maximum singular value at a particular frequency is the maximum gain of the system over all linear combinations of inputs at that frequency. Singular values can provide a better indication of the overall response, stability, and conditioning of a MIMO system than a channel-by-channel Bode plot.

Calculate the singular values of H between 0.1 and 10 rad/s.

```
[sv,w] = sigma(H,{0.1,10});
```

When you call sigma with output arguments, the command returns the singular values in the data array sv. The cell array input {0.1,10} tells sigma to calculate the singular

values at a grid of frequencies between 0.1 and 10 rad/s. `sigma` returns these frequencies in the vector `w`. Each row of `sv` contains the singular values of `H` at the frequencies of `w`.

See Also
bode | bodeplot | sigma | sigmaplot

Related Examples

- "Numeric Values of Frequency-Domain Characteristics of SISO Model" on page 8-13
- "Joint Time- and Frequency-Domain Analysis" on page 7-20
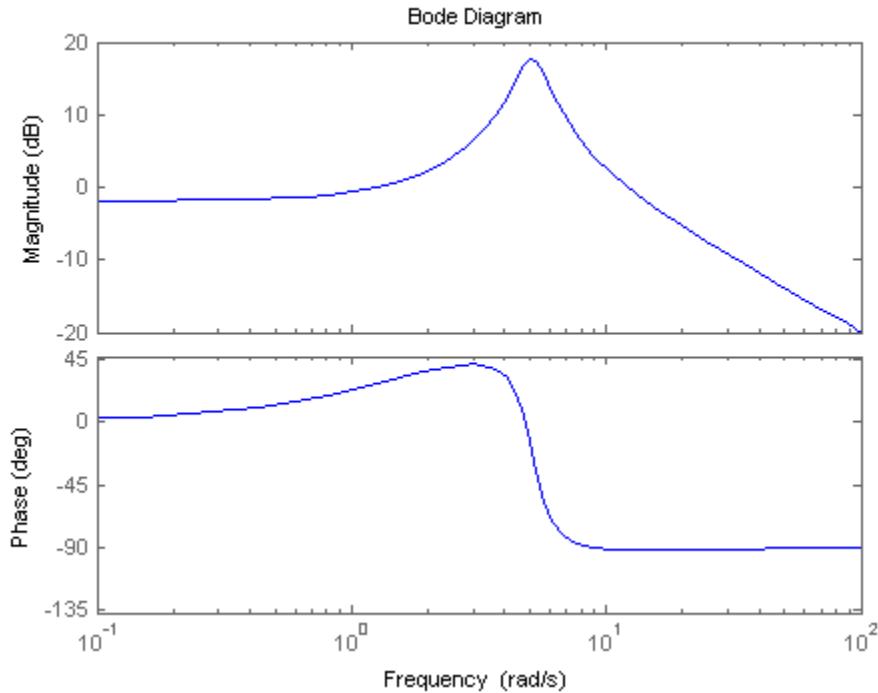
System Characteristics on Response Plots

This example shows how to display system characteristics such as peak response on Bode response plots.

You can use similar procedures to display system characteristics on other types of response plots.
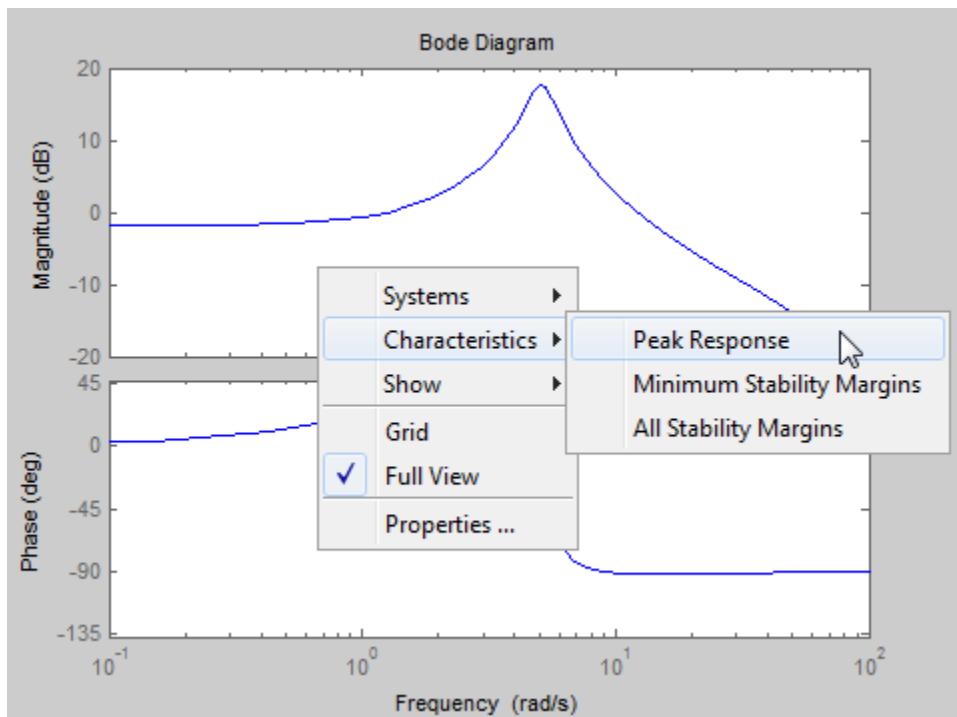
Create a transfer function model and plot its frequency response.

```
H = tf([10,21],[1,1.4,26]);
bodeplot(H)
```
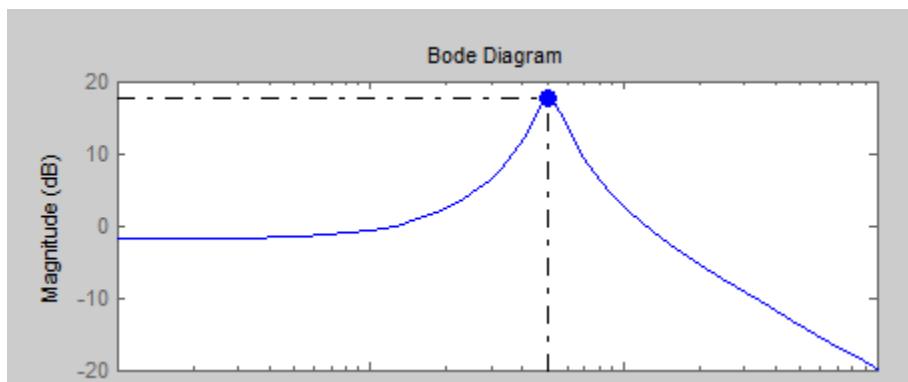

Display the peak response on the plot.
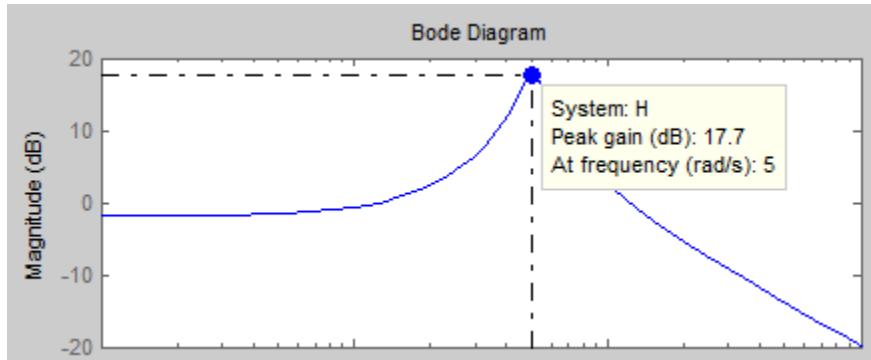
Right-click anywhere in the figure and select **Characteristics** > **Peak Response** from the menu.

A marker appears on the plot indicating the peak response. Horizontal and vertical dotted lines indicate the frequency and magnitude of that response. The other menu options add other system characteristics to the plot.

Click the marker to view the magnitude and frequency of the peak response in a datatip.

Related Examples
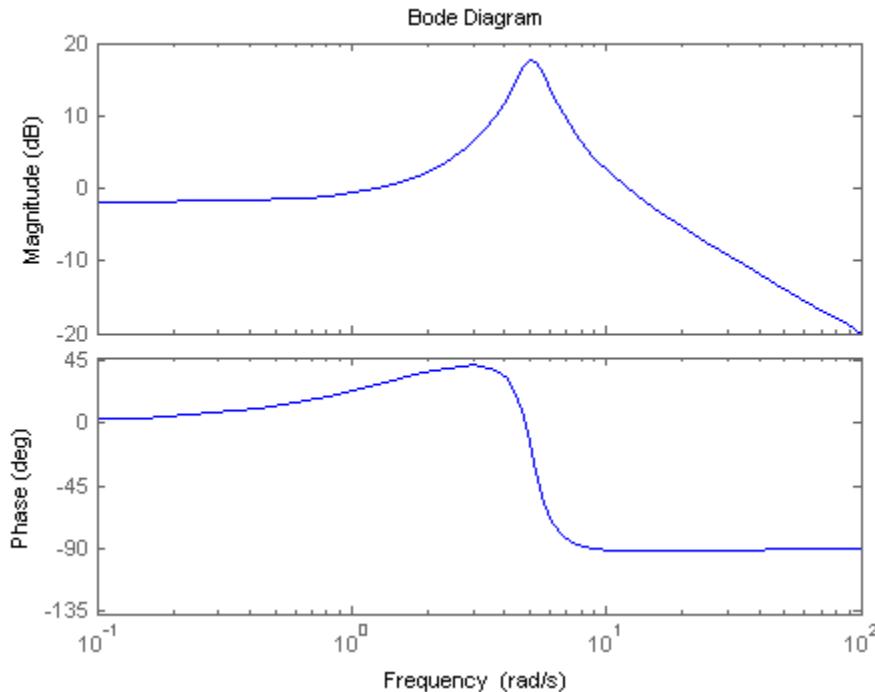
- "Numeric Values of Frequency-Domain Characteristics of SISO Model" on page 8-13
- "Joint Time- and Frequency-Domain Analysis" on page 7-20
- "Pole and Zero Locations" on page 8-16

Numeric Values of Frequency-Domain Characteristics of SISO Model

This example shows how to obtain numeric values of several frequency-domain characteristics of a SISO dynamic system model, including the peak gain, dc gain, system bandwidth, and the frequencies at which the system gain crosses a specified frequency.

Create a transfer function model and plot its frequency response.

```
H = tf([10,21],[1,1.4,26]);
bodeplot(H)
```


Plotting the frequency response gives a rough idea of the frequency-domain characteristics of the system. H includes a pronounced resonant peak, and rolls off at 20 dB/decade at high frequency. It is often desirable to obtain specific numeric values for such characteristics.

Calculate the peak gain and the frequency of the resonance.

```
[gpeak,fpeak] = getPeakGain(H);
gpeak_dB = mag2db(gpeak)

gpeak_dB =

   17.7579
```

getPeakGain returns both the peak location fpeak and the peak gain gpeak in absolute units. Using mag2db to convert gpeak to decibels shows that the gain peaks at almost 18 dB.

Find the band within which the system gain exceeds 0 dB, or 1 in absolute units.

```
wc = getGainCrossover(H,1)

wc =

    1.2582
   12.1843
```

getGainCrossover returns a vector of frequencies at which the system response crosses the specified gain. The resulting wc vector shows that the system gain exceeds 0 dB between about 1.3 and 12.2 rad/s.

Find the dc gain of H.

The Bode response plot shows that the gain of H tends toward a finite value as the frequency approaches zero. The dcgain command finds this value in absolute units.

```
k = dcgain(H);
```

Find the frequency at which the response of H rolls off to –10 dB relative to its dc value.

```
fb = bandwidth(H,-10);
```

bandwidth returns the first frequency at which the system response drops below the dc gain by the specified value in dB.

See Also
bandwidth | getGainCrossover | getPeakGain

Related Examples

- "Pole and Zero Locations" on page 8-16

More About

- "Choosing a Frequency-Domain Analysis Command" on page 8-2

Pole and Zero Locations

This example shows how to examine the pole and zero locations of dynamic systems both graphically using `pzplot` and numerically using `pole` and `zero`.

Examining the pole and zero locations can be useful for tasks such as stability analysis or identifying near-canceling pole-zero pairs for model simplification. This example compares two closed-loop systems that have the same plant and different controllers.
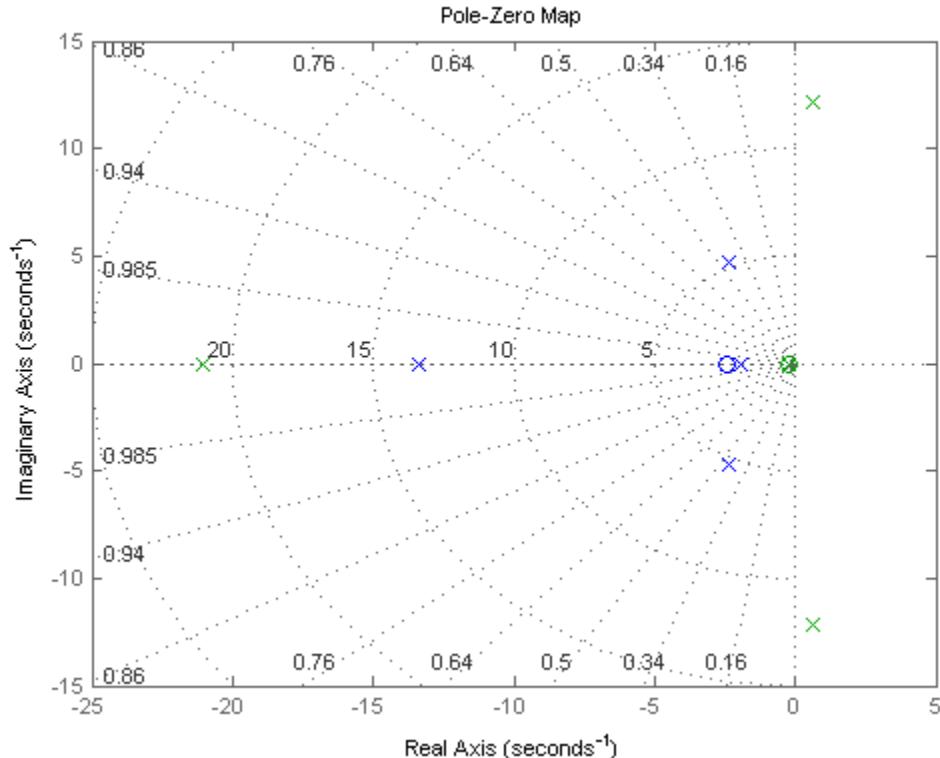
Create dynamic system models representing the two closed-loop systems.

```
G = zpk([],[-5 -5 -10],100);
C1 = pid(2.9,7.1);
CL1 = feedback(G*C1,1);
C2 = pid(29,7.1);
CL2 = feedback(G*C2,1);
```

The controller `C2` has a much higher proportional gain. Otherwise, the two closed-loop systems `CL1` and `CL2` are the same.

Graphically examine the pole and zero locations of `CL1` and `CL2`.

```
pzplot(CL1,CL2)
grid
```

Pole-Zero Map

`pzplot` plots pole and zero locations on the complex plane as `x` and `o` marks, respectively. When you provide multiple models, `pzplot` plots the poles and zeros of each model in a different color. Here, there poles and zeros of `CL1` are blue, and those of `CL2` are green.

The plot shows that all poles of `CL1` are in the left half-plane, and therefore `CL1` is stable. From the radial grid markings on the plot, you can read that the damping of the oscillating (complex) poles is approximately 0.45. The plot also shows that `CL2` contains poles in the right half-plane and is therefore unstable.

Compute numerical values of the pole and zero locations of `CL2`.

```
z = zero(CL2);
```

8-17

```
p = pole(CL2);
```

`zero` and `pole` return column vectors containing the zero and pole locations of the system.

See Also
`pole` | `pzplot` | `zero`

Related Examples
*   "Numeric Values of Frequency-Domain Characteristics of SISO Model" on page 8-13

More About
*   "Choosing a Frequency-Domain Analysis Command" on page 8-2

Assessing Gain and Phase Margins

This example shows how to examine the effect of stability margins on closed-loop response characteristics of a control system.

Stability of a Feedback Loop

Stability generally means that all internal signals remain bounded. This is a standard requirement for control systems to avoid loss of control and damage to equipment. For linear feedback systems, stability can be assessed by looking at the poles of the closed-loop transfer function. Consider for example the SISO feedback loop:
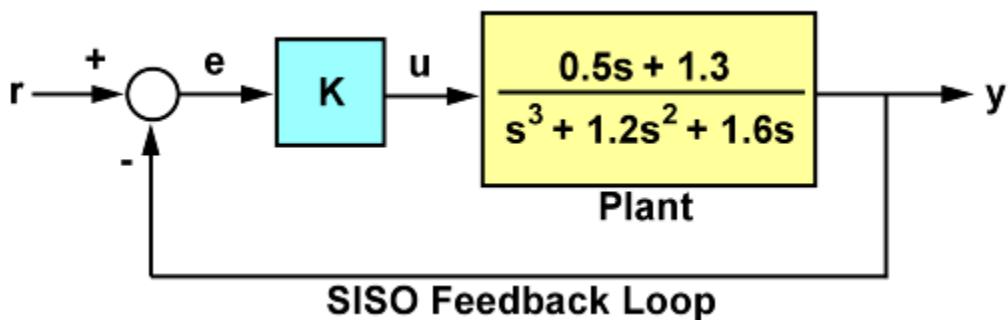
Figure 1: SISO Feedback Loop.

For a unit loop gain k, you can compute the closed-loop transfer function T using:

```
G = tf([.5 1.3],[1 1.2  1.6 0]);
T = feedback(G,1);
```

To obtain the poles of T, type

```
pole(T)
```
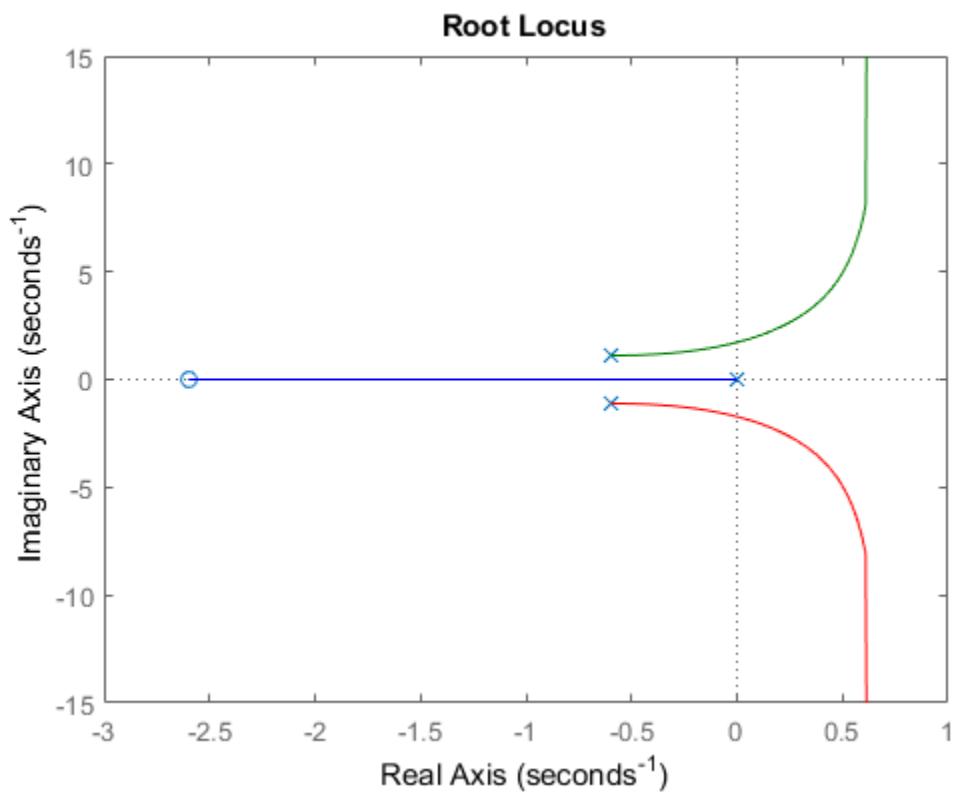
```
ans =

  -0.2305 + 1.3062i
  -0.2305 - 1.3062i
  -0.7389 + 0.0000i
```

The feedback loop for k=1 is stable since all poles have negative real parts.

How Stable is Stable?

Checking the closed-loop poles gives us a binary assessment of stability. In practice, it is more useful to know how robust (or fragile) stability is. One indication of robustness is how much the loop gain can change before stability is lost. You can use the root locus plot to estimate the range of k values for which the loop is stable:

rlocus(G)

Clicking on the point where the locus intersects the y axis reveals that this feedback loop is stable for

$$0 < k < 2.7$$

This range shows that with `k=1`, the loop gain can increase 270% before you lose stability.
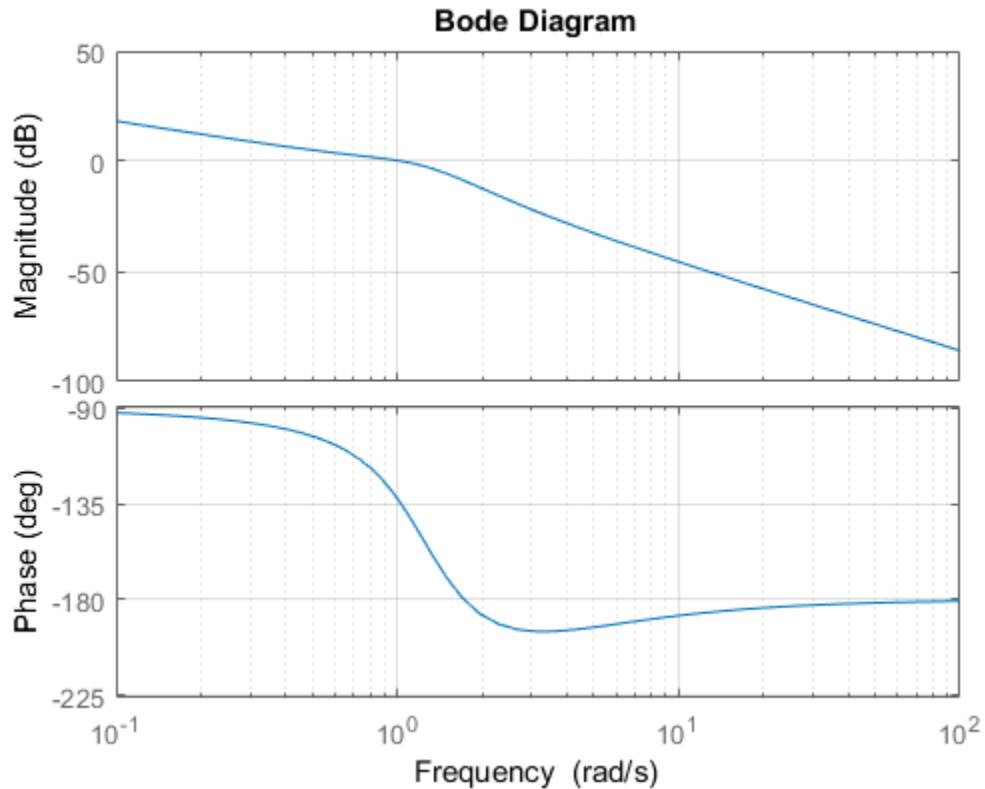
Gain and Phase Margins

Changes in the loop gain are only one aspect of robust stability. In general, imperfect plant modeling means that both gain and phase are not known exactly. Because modeling errors are most damaging near the gain crossover frequency (frequency where open-loop gain is 0dB), it also matters how much phase variation can be tolerated at this frequency.
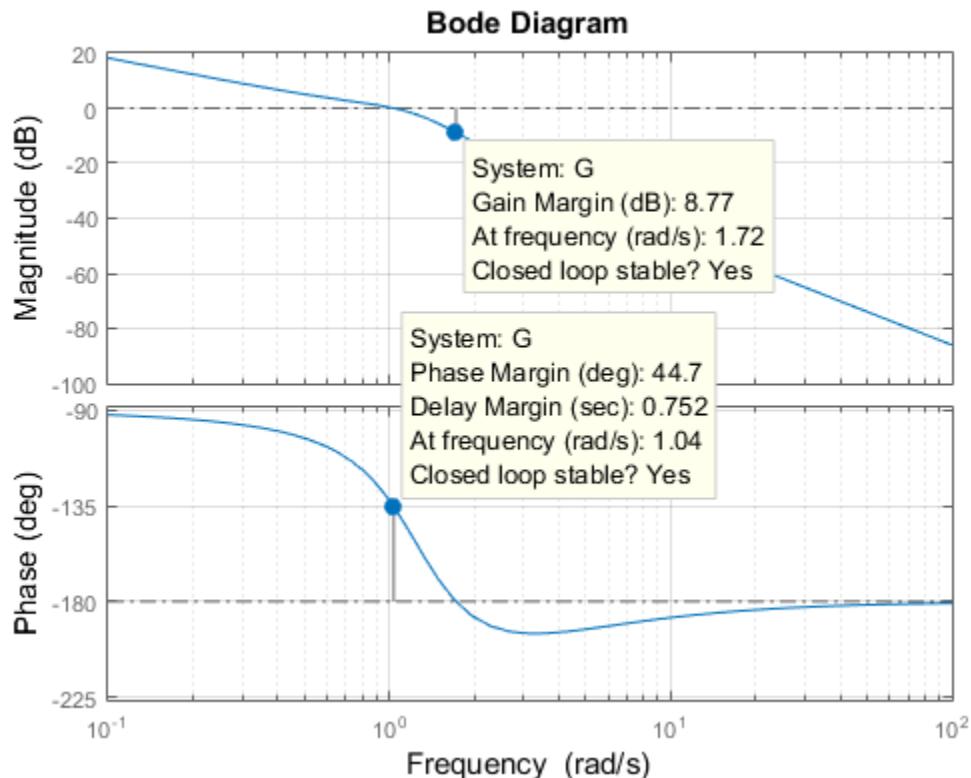
The phase margin measures how much phase variation is needed at the gain crossover frequency to lose stability. Similarly, the gain margin measures what relative gain variation is needed at the gain crossover frequency to lose stability. Together, these two numbers give an estimate of the "safety margin" for closed-loop stability. The smaller the stability margins, the more fragile stability is.

You can display the gain and phase margins on a Bode plot as follows. First create the plot:

```
bode(G), grid
```
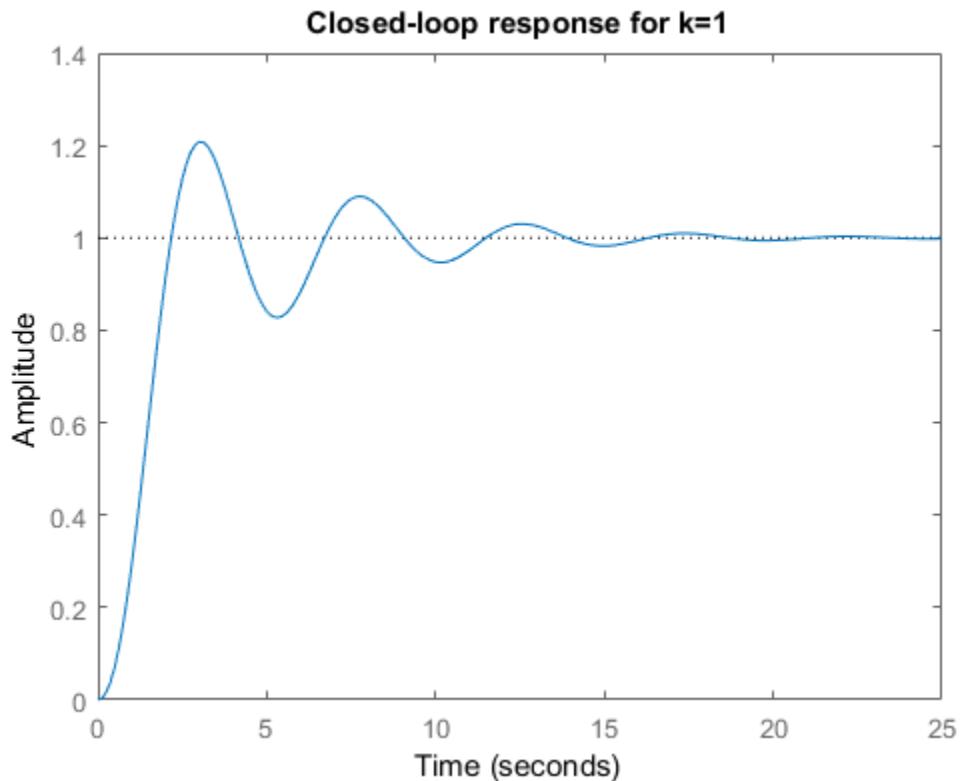
Then, right-click on the plot and select the **Characteristics -> Minimum Stability Margins** submenu. Finally, click on the blue dot markers. The resulting plot is shown below:

This indicates a gain margin of about 9 dB and a phase margin of about 45 degrees. The corresponding closed-loop step response exhibits about 20% overshoot and some oscillations.

```
step(T), title('Closed-loop response for k=1')
```

Closed-loop response for k=1

If we increase the gain to k=2, the stability margins are reduced to

```
[Gm,Pm] = margin(2*G);
GmdB = 20*log10(Gm)    % gain margin in dB
Pm   % phase margin in degrees


GmdB =

    2.7471


Pm =
```
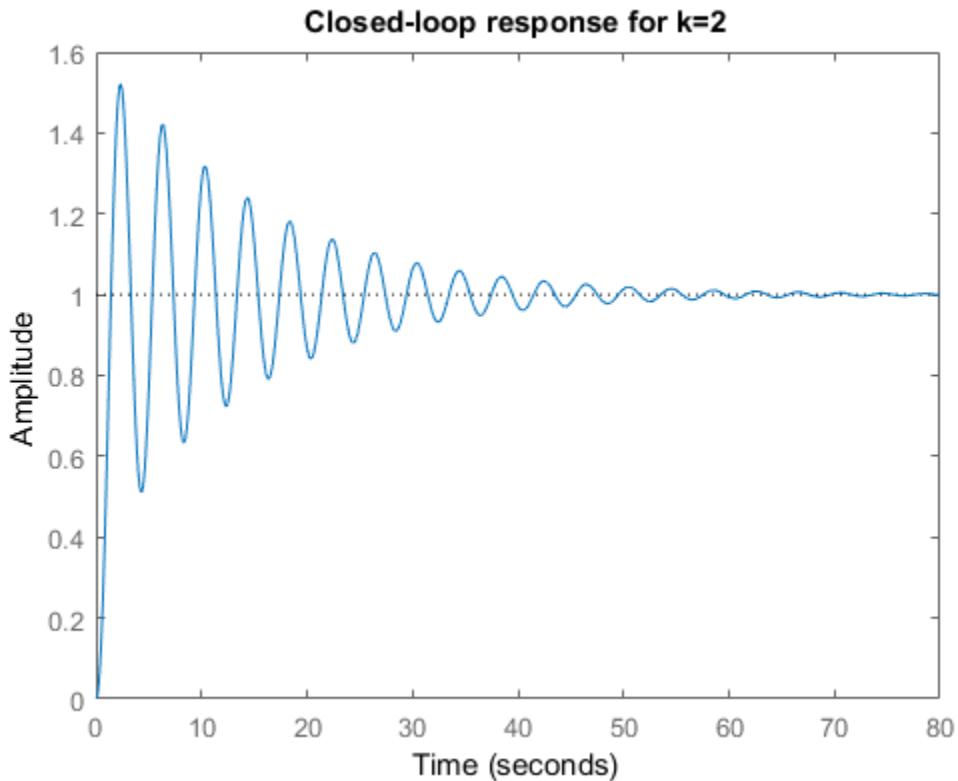
```
8.6328
```

and the closed-loop response has poorly damped oscillations, a sign of near instability.

```
step(feedback(2*G,1)), title('Closed-loop response for k=2')
```


Systems with Multiple Gain or Phase Crossings

Some systems have multiple gain crossover or phase crossover frequencies, which leads to multiple gain or phase margin values. For example, consider the feedback loop
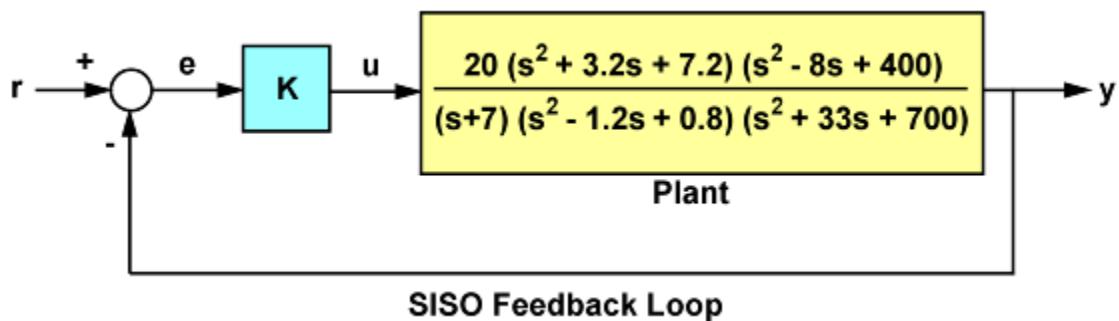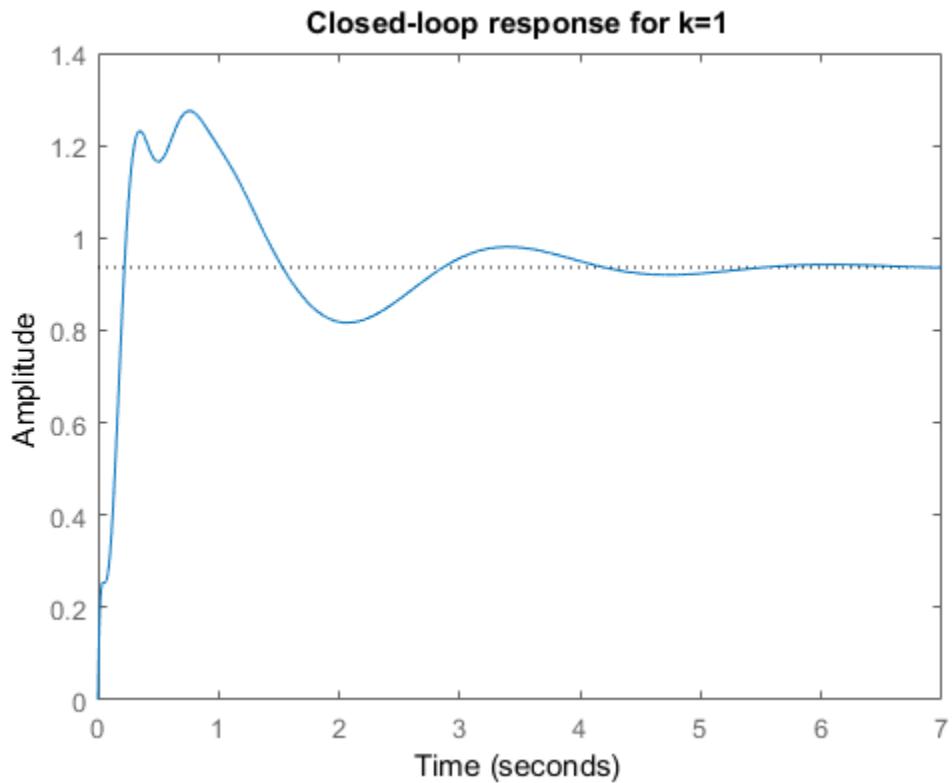
Figure 2: Feedback Loop with Multiple Phase Crossovers
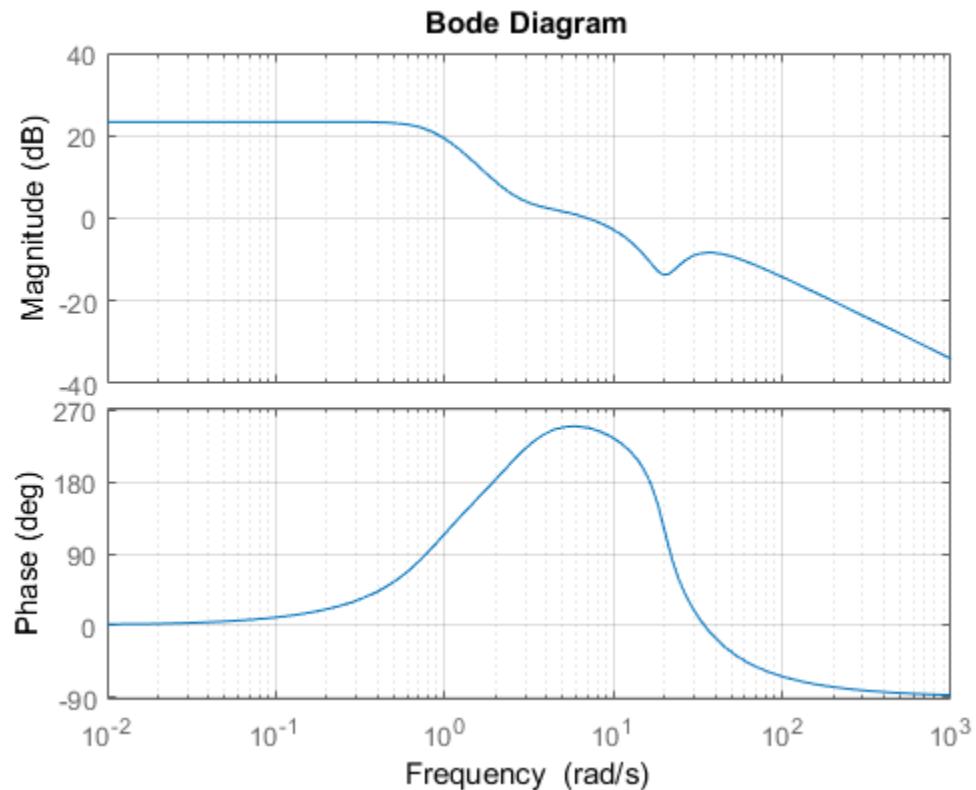
The closed-loop response for k=1 is stable:

```
G = tf(20,[1 7]) * tf([1 3.2 7.2],[1 -1.2 0.8]) * tf([1 -8 400],[1 33 700]);
T = feedback(G,1);
step(T), title('Closed-loop response for k=1')
```
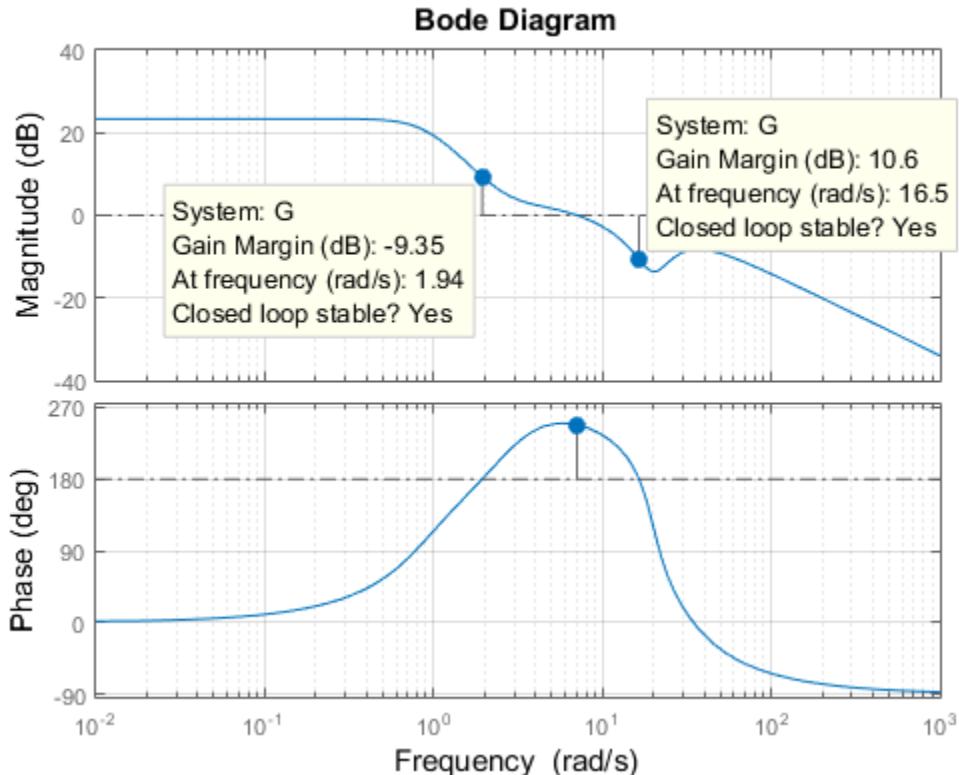
To assess how robustly stable this loop is, plot its Bode response:

```
bode(G), grid
```

Bode Diagram
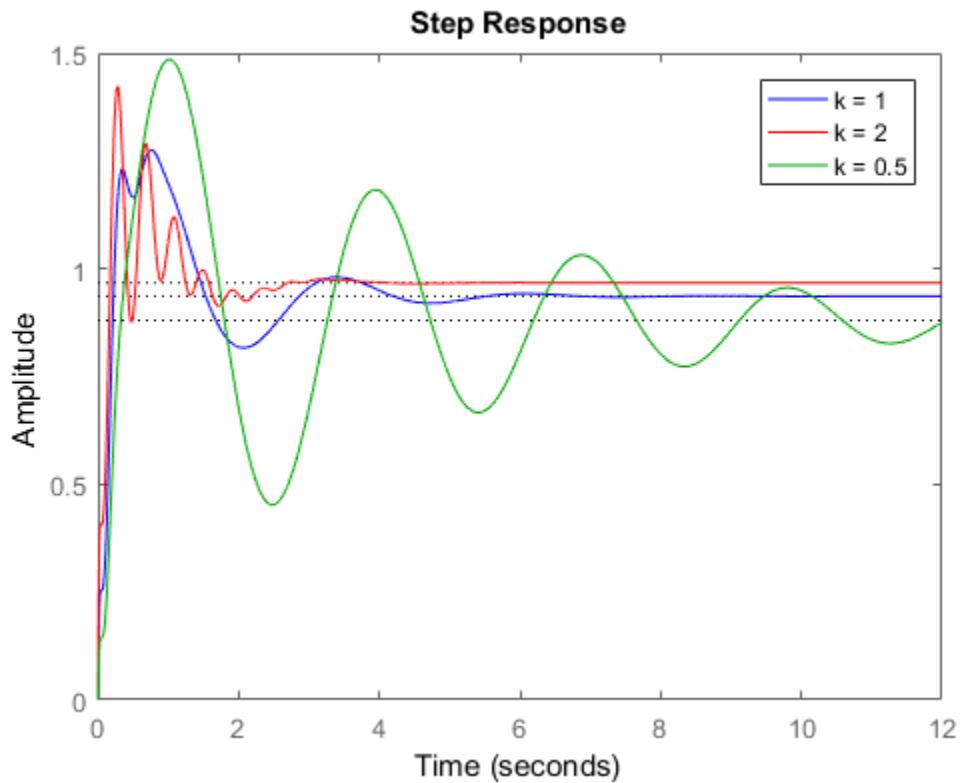
Then, right-click on the plot and select the **Characteristics -> All Stability Margins** submenu to show all the crossover frequencies and associated stability margins. The resulting plot is shown below.

Bode Diagram

Note that there are two 180 deg phase crossings with corresponding gain margins of -9.35dB and +10.6dB. Negative gain margins indicate that stability is lost by decreasing the gain, while positive gain margins indicate that stability is lost by increasing the gain. This is confirmed by plotting the closed-loop step response for a plus/minus 6dB gain variation about k=1:

```
k1 = 2;     T1 = feedback(G*k1,1);
k2 = 1/2;   T2 = feedback(G*k2,1);
step(T,'b',T1,'r',T2,'g',12),
legend('k = 1','k = 2','k = 0.5')
```

The plot shows increased oscillations for both smaller and larger gain values.

You can use the command `allmargin` to compute all stability margins. Note that gain margins are expressed as gain ratios, not dB. Use `mag2db` to convert the values to dB.

```
m = allmargin(G)

GainMargins_dB = mag2db(m.GainMargin)


m =

    GainMargin: [0.3408 3.3920]
   GMFrequency: [1.9421 16.4807]
```

```
    PhaseMargin: 68.1178
    PMFrequency: 7.0762
    DelayMargin: 0.1680
    DMFrequency: 7.0762
         Stable: 1


GainMargins_dB =

  -9.3510   10.6091
```
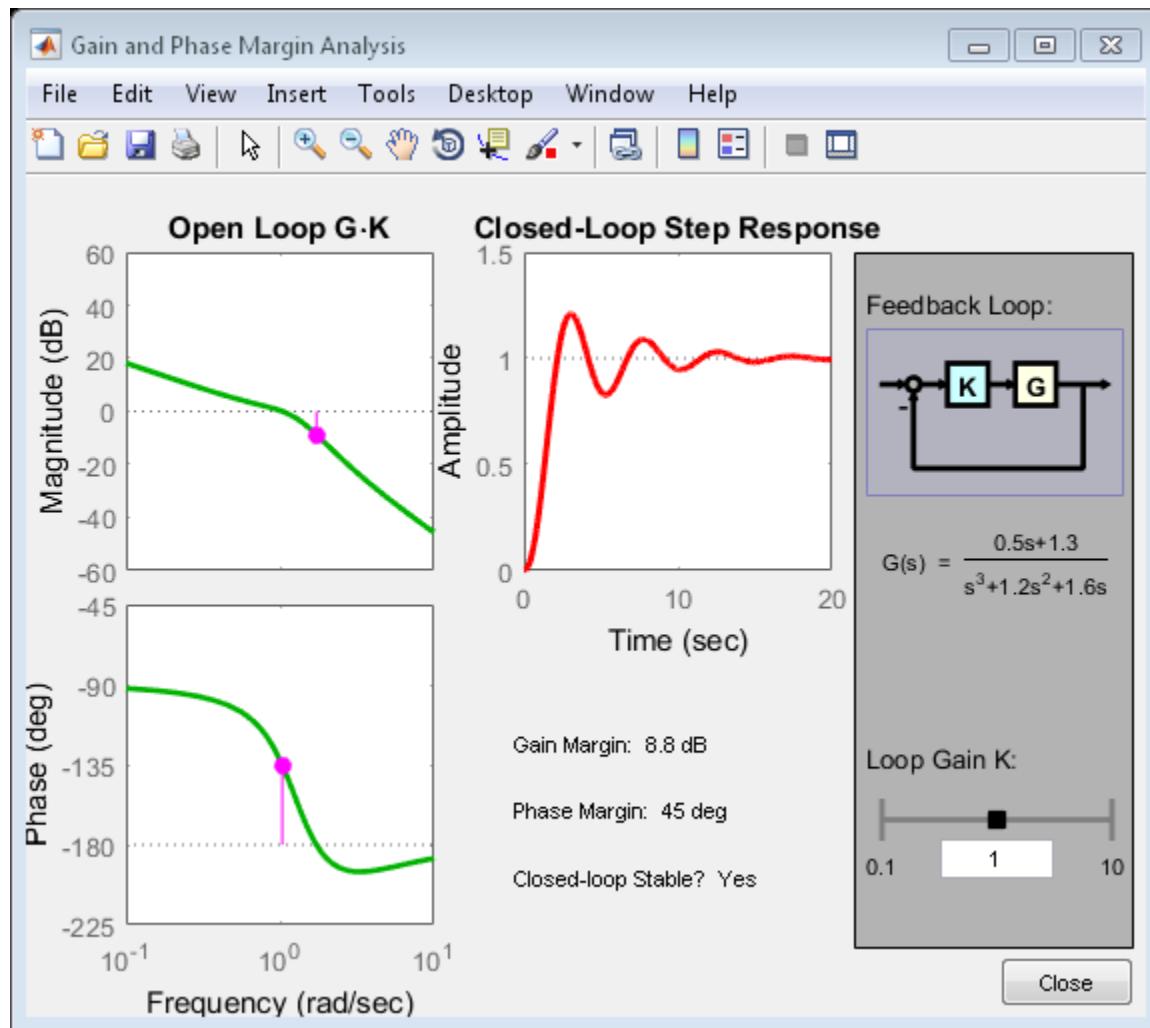
Interactive GUI

To gain additional insight into the connection between stability margins and closed-loop responses, click on the link below to launch an interactive GUI for tuning the loop gain k and seeing the effect on margins and closed-loop responses.

Open the Gain and Phase Margin GUI

```
margin_gui
```

See Also

margin | pole

Related Examples

- "Pole and Zero Locations" on page 8-16

Analyzing Control Systems with Delays

This example shows how to use Control System Toolbox™ to analyze and design control systems with delays.
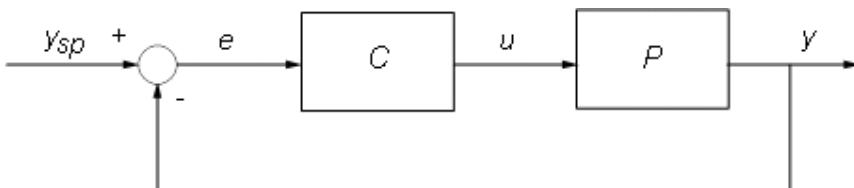
Control of Processes with Delays

Many processes involve dead times, also referred to as transport delays or time lags. Controlling such processes is challenging because delays cause linear phase shifts that limit the control bandwidth and affect closed-loop stability.

Using the state-space representation, you can create accurate open- or closed-loop models of control systems with delays and analyze their stability and performance without approximation. The state-space (SS) object automatically keeps track of "internal" delays when combining models, see the "Specifying Time Delays" tutorial for more details.

Example: PI Control Loop with Dead Time

Consider the standard setpoint tracking loop:

where the process model P has a 2.6 second dead time and the compensator C is a PI controller:

$$P(s) = \frac{e^{-2.6s}(s+3)}{s^2 + 0.3s + 1}, \quad C(s) = 0.06(1 + \frac{1}{s})$$

You can specify these two transfer functions as

```
s = tf('s');
P = exp(-2.6*s)*(s+3)/(s^2+0.3*s+1);
C = 0.06 * (1 + 1/s);
```

To analyze the closed-loop response, construct a model T of the closed-loop transfer from ysp to y. Because there is a delay in this feedback loop, you must convert P and C to state space and use the state-space representation for analysis:

```
T = feedback(P*C,1)


T =

  A =
         x1      x2      x3
    x1  -0.36   -1.24   -0.18
    x2     1       0       0
    x3     0       1       0

  B =
         u1
    x1   0.5
    x2    0
    x3    0

  C =
         x1     x2     x3
    y1   0.12   0.48   0.36

  D =
         u1
    y1    0

  (values computed with all internal delays set to zero)

  Internal delays (seconds): 2.6

Continuous-time state-space model.
```
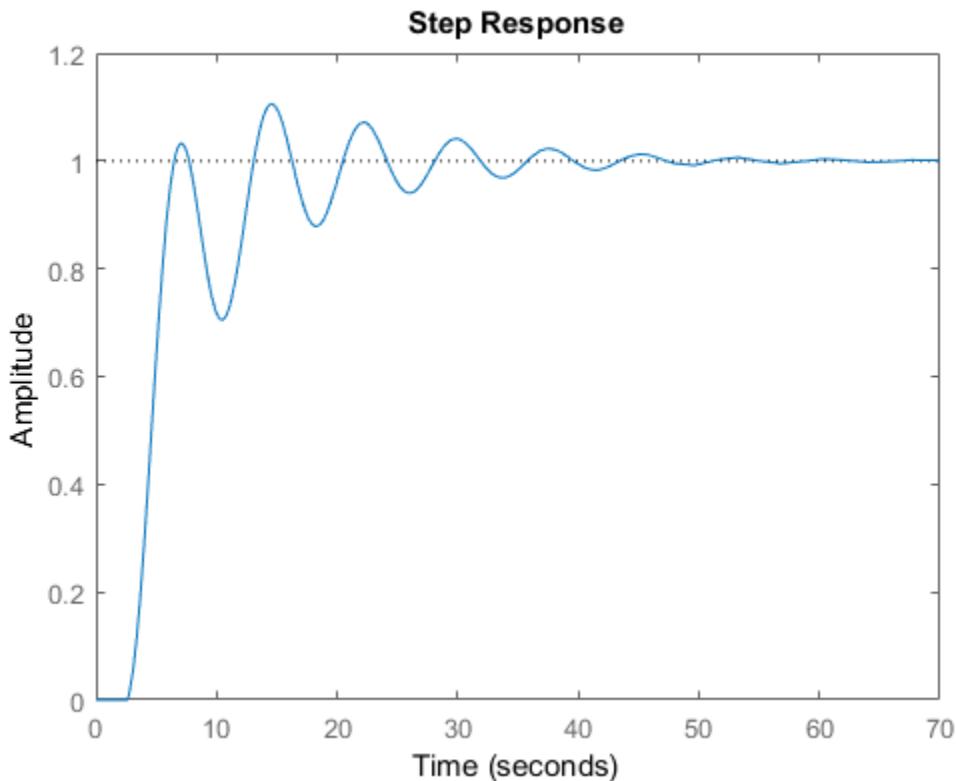
The result is a third-order model with an internal delay of 2.6 seconds. Internally, the state-space object T tracks how the delay is coupled with the remaining dynamics. This structural information is not visible to users, and the display above only gives the A,B,C,D values when the delay is set to zero.
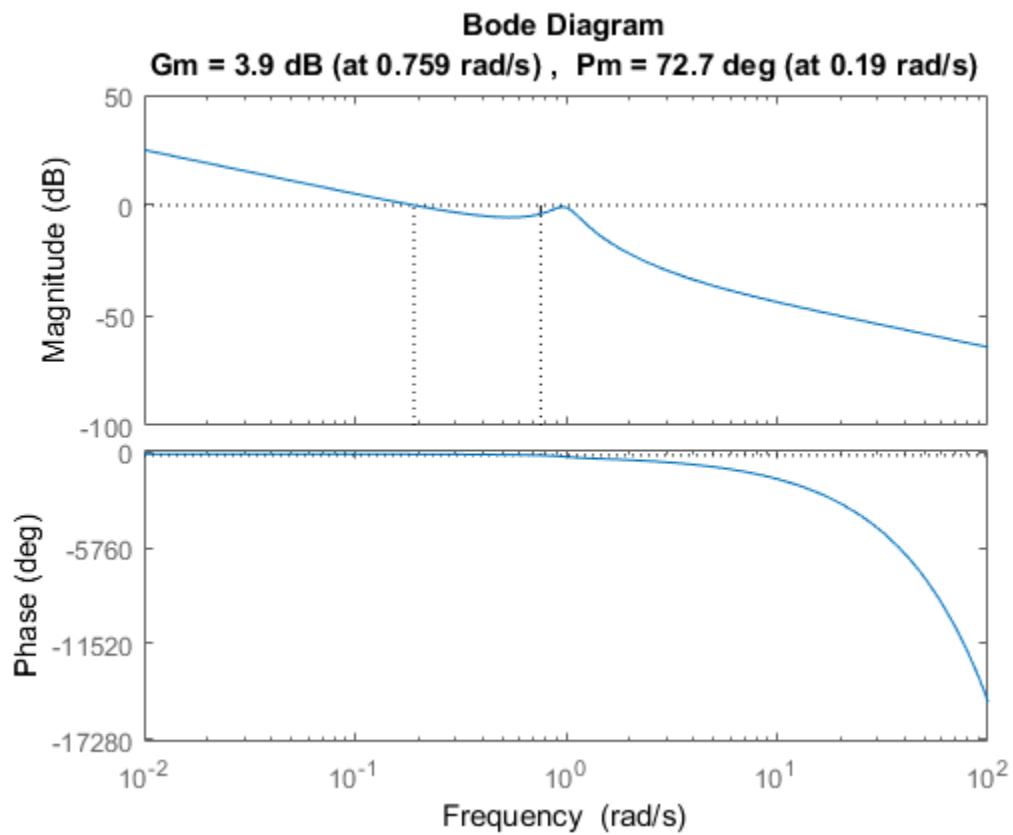
Use the STEP command to plot the closed-loop step response from ysp to y:

```
step(T)
```
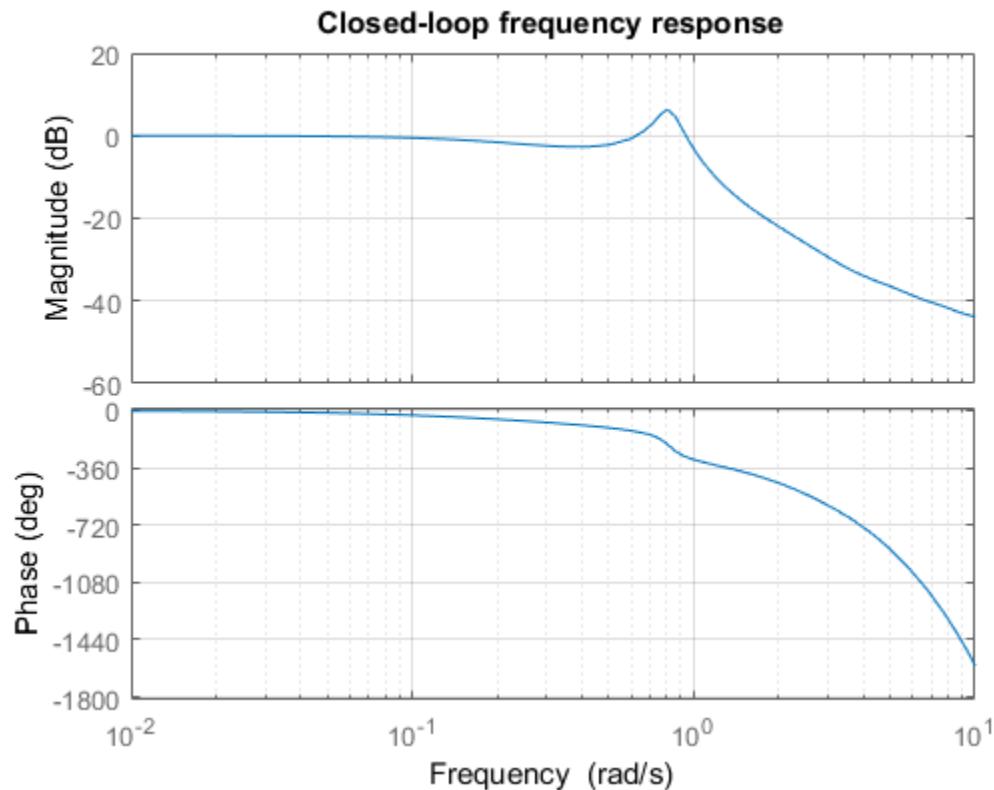
The closed-loop oscillations are due to a weak gain margin as seen from the open-loop response `P*C`:

```
margin(P*C)
```

Bode Diagram
Gm = 3.9 dB (at 0.759 rad/s) , Pm = 72.7 deg (at 0.19 rad/s)

There is also a resonance in the closed-loop frequency response:
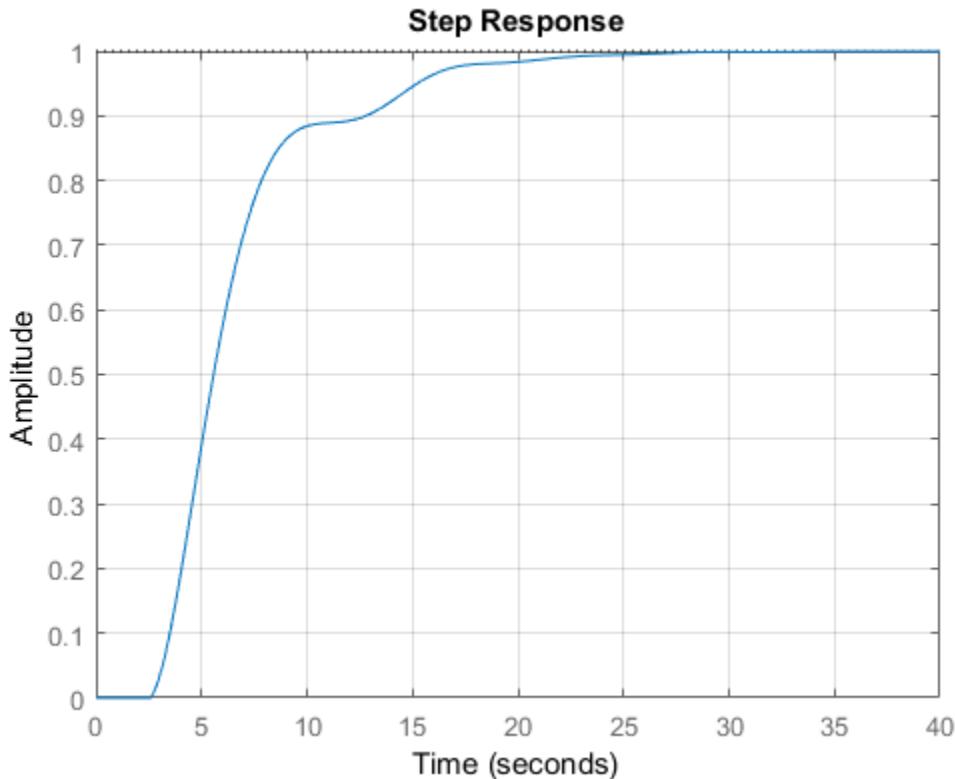
```
bode(T)
grid, title('Closed-loop frequency response')
```

Closed-loop frequency response

To improve the design, you can try to notch out the resonance near 1 rad/s:

```
notch = tf([1 0.2 1],[1 .8 1]);
C = 0.05 * (1 + 1/s);
Tnotch = feedback(P*C*notch,1);

step(Tnotch), grid
```

8-37

Step Response
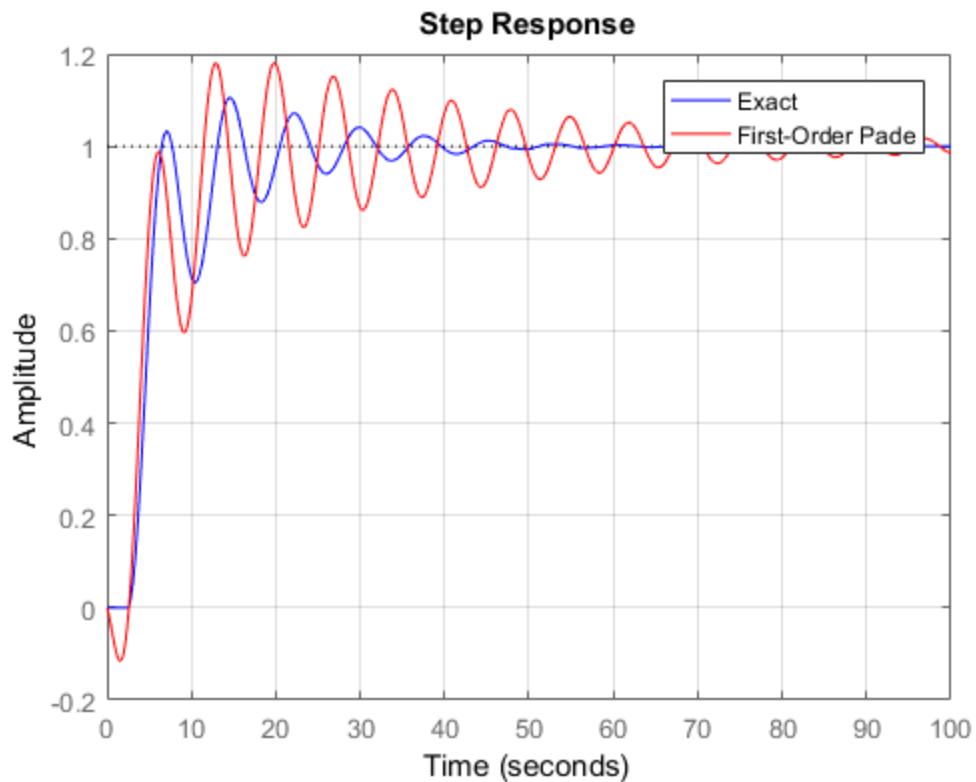
Pade Approximation of Time Delays

Many control design algorithms cannot handle time delays directly. A common workaround consists of replacing delays by their Pade approximations (all-pass filters). Because this approximation is only valid at low frequencies, it is important to compare the true and approximate responses to choose the right approximation order and check the approximation validity.

Use the PADE command to compute Pade approximations of LTI models with delays. For the PI control example above, you can compare the exact closed-loop response T with the response obtained for a first-order Pade approximation of the delay:
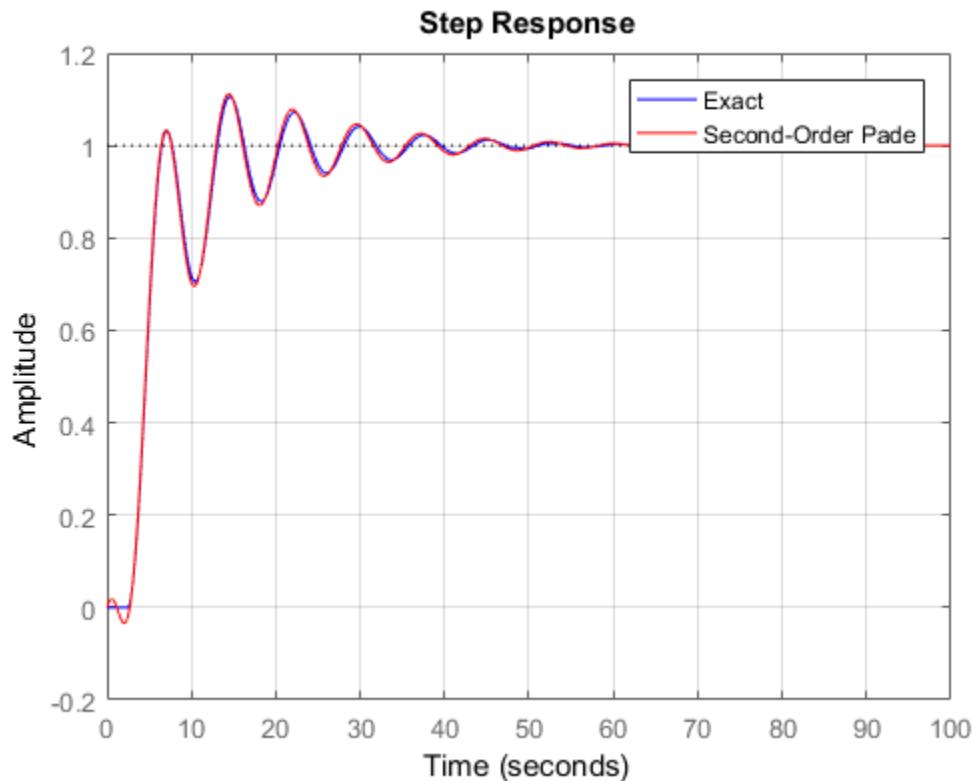
```
T1 = pade(T,1);
```

```
step(T,'b',T1,'r',100)
grid, legend('Exact','First-Order Pade')
```


The approximation error is fairly large. To get a better approximation, try a second-order Pade approximation of the delay:

```
T2 = pade(T,2);
step(T,'b',T2,'r',100)
grid, legend('Exact','Second-Order Pade')
```

The responses now match closely except for the non-minimum phase artifact introduced by the Pade approximation.

Sensitivity Analysis

Delays are rarely known accurately, so it is often important to understand how sensitive a control system is to the delay value. Such sensitivity analysis is easily performed using LTI arrays and the InternalDelay property.
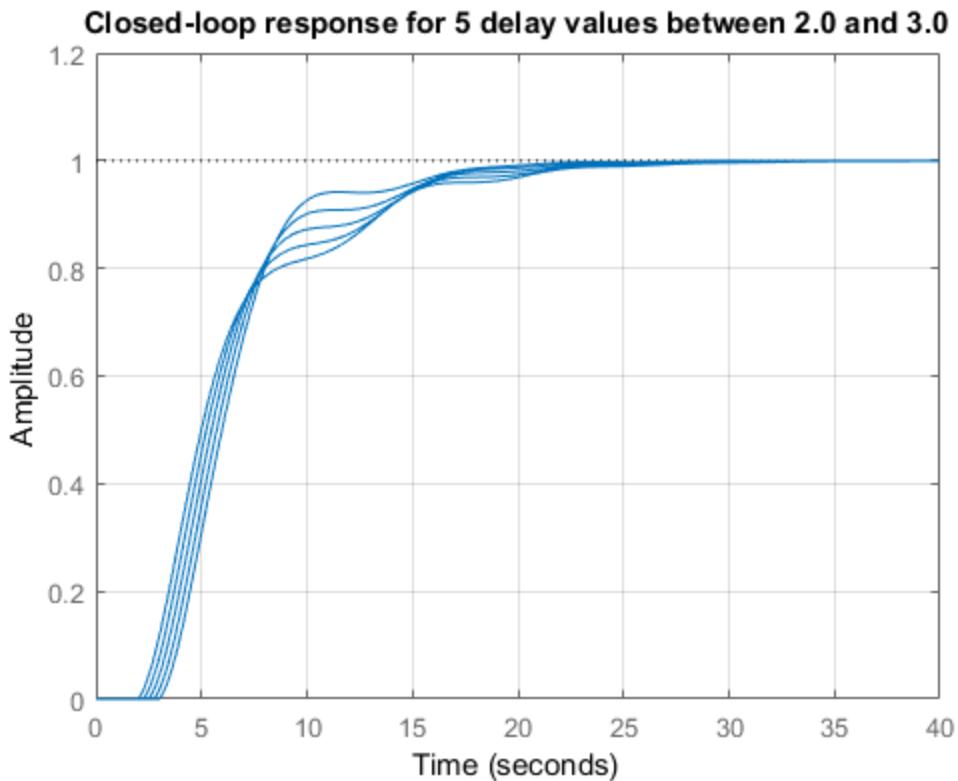
For example, to analyze the sensitivity of the notched PI control above, create 5 models with delay values ranging from 2.0 to 3.0:

```
tau = linspace(2,3,5);                      % 5 delay values
Tsens = repsys(Tnotch,[1 1 5]);             % 5 copies of Tnotch
```

```
for j=1:5
  Tsens(:,:,j).InternalDelay = tau(j);     % jth delay value -> jth model
end
```
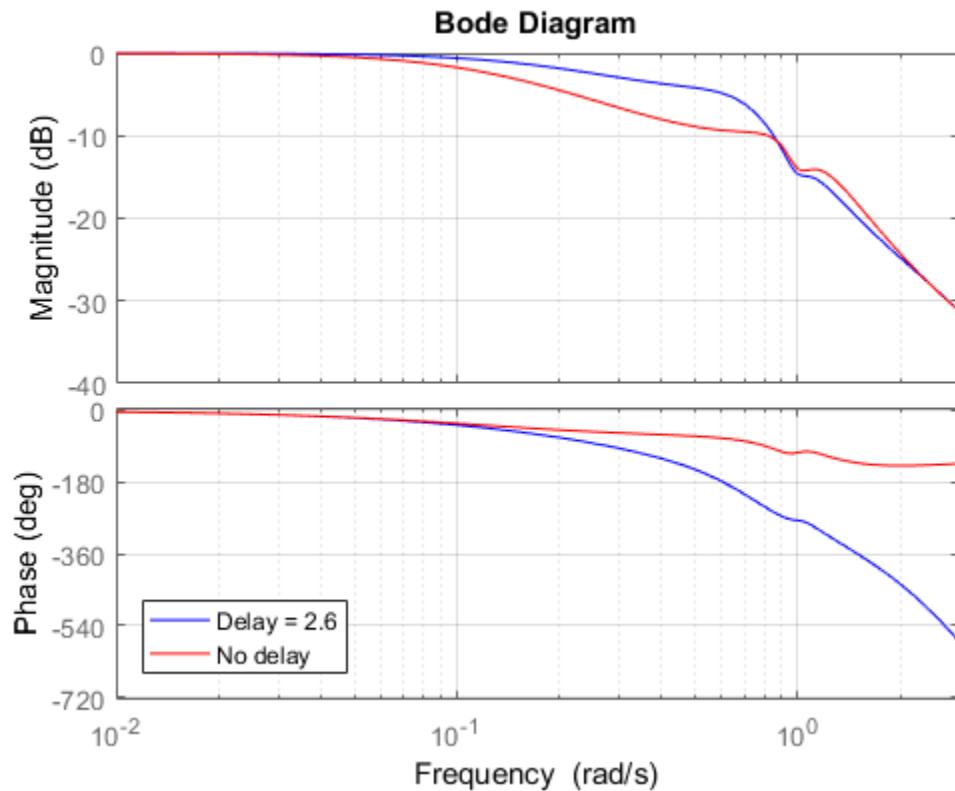
Then use STEP to create an envelope plot:

```
step(Tsens)
grid, title('Closed-loop response for 5 delay values between 2.0 and 3.0')
```


This plot shows that uncertainty on the delay value has little effect on closed-loop characteristics. Note that while you can change the values of internal delays, you cannot change how many there are because this is part of the model structure. To eliminate some internal delays, set their value to zero or use PADE with order zero:

```
Tnotch0 = Tnotch;
Tnotch0.InternalDelay = 0;
bode(Tnotch,'b',Tnotch0,'r',{1e-2,3})
grid, legend('Delay = 2.6','No delay','Location','SouthWest')
```
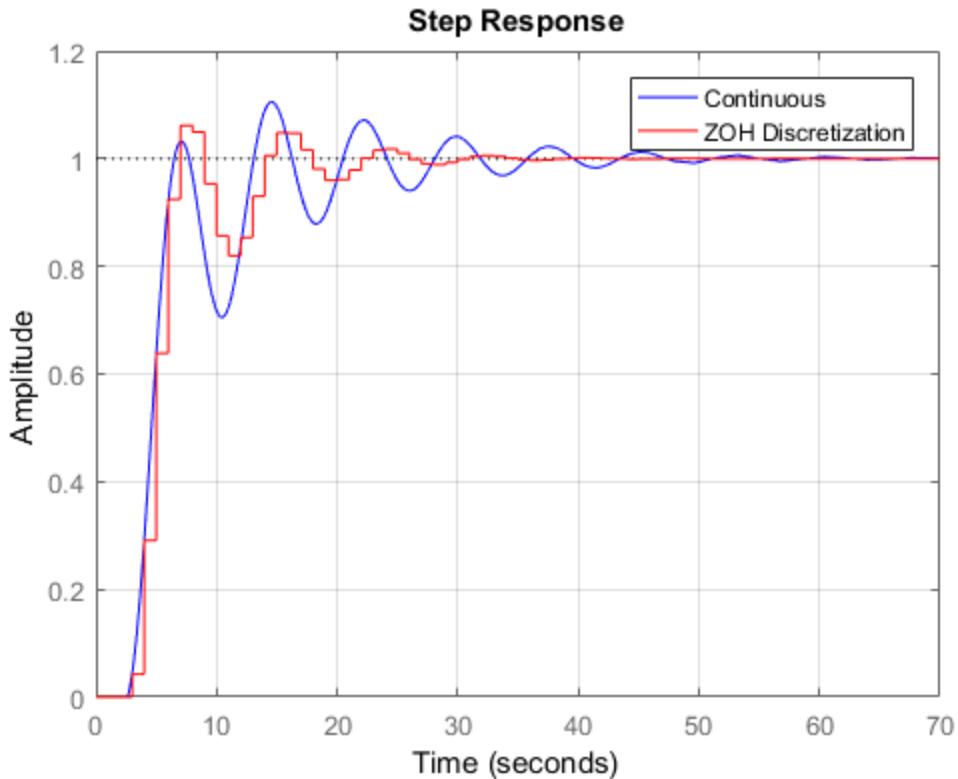

Discretization

You can use C2D to discretize continuous-time delay systems. Available methods include zero-order hold (ZOH), first-order hold (FOH), and Tustin. For models with internal delays, the ZOH discretization is not always "exact," i.e., the continuous and discretized step responses may not match:

```
Td = c2d(T,1);
step(T,'b',Td,'r')
```

```
grid, legend('Continuous','ZOH Discretization')
```
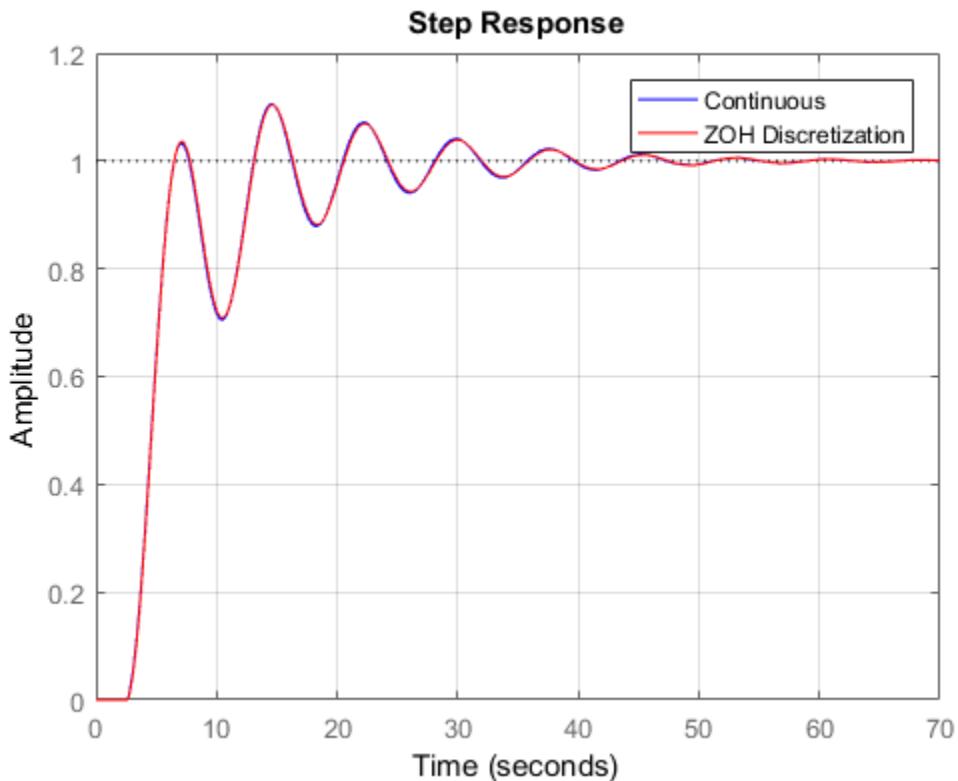
Warning: Discretization is only approximate due to internal delays. Use faster
sampling rate if discretization error is large.

To correct such discretization gaps, reduce the sampling period until the continuous and discrete responses match closely:

```
Td = c2d(T,0.05);
step(T,'b',Td,'r')
grid, legend('Continuous','ZOH Discretization')
```

Warning: Discretization is only approximate due to internal delays. Use faster
sampling rate if discretization error is large.

Step Response

Note that internal delays remain internal in the discretized model and do not inflate the model order:

```
order(Td)
Td.InternalDelay
```

```
ans =

    3
```

```
ans =
```

52
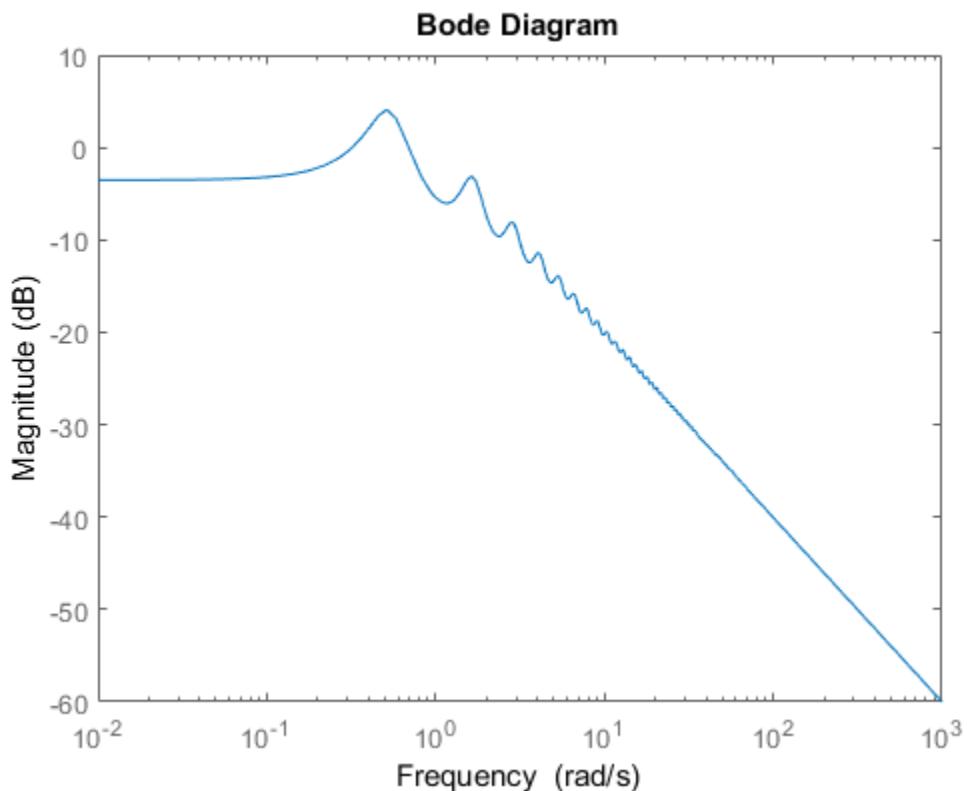
Some Unique Features of Delay Systems

The time and frequency responses of delay systems can look bizarre and suspicious to those only familiar with delay-free LTI analysis. Time responses can behave chaotically, Bode plots can exhibit gain oscillations, etc. These are not software quirks but real features of such systems. Below are a few illustrations of these phenomena
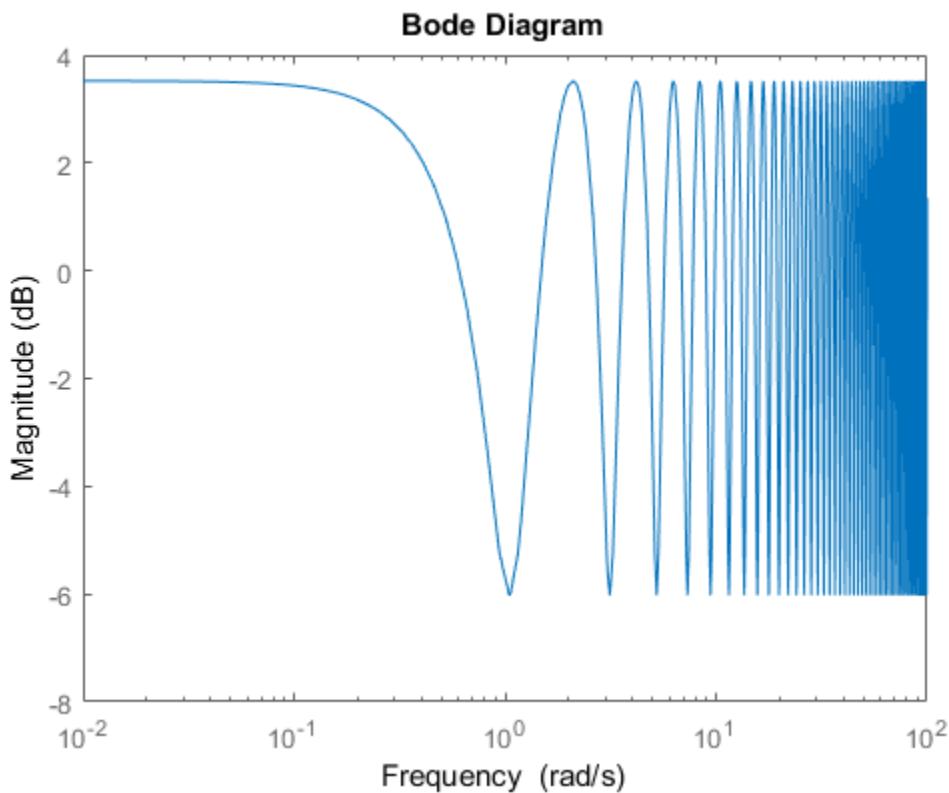
Gain ripples:

```
G = exp(-5*s)/(s+1);
T = feedback(G,.5);
bodemag(T)
```

Gain oscillations:

```
G = 1 + 0.5 * exp(-3*s);
bodemag(G)
```

Bode Diagram
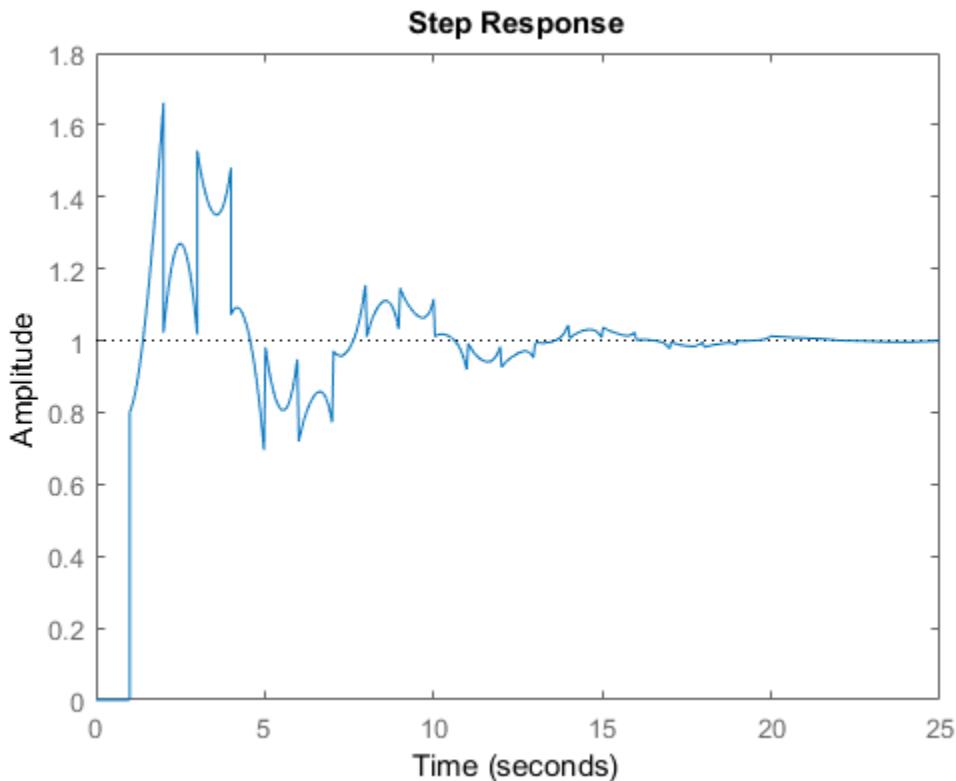
Jagged step response (note the "echoes" of the initial step):

```
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(G,1);
step(T)
```
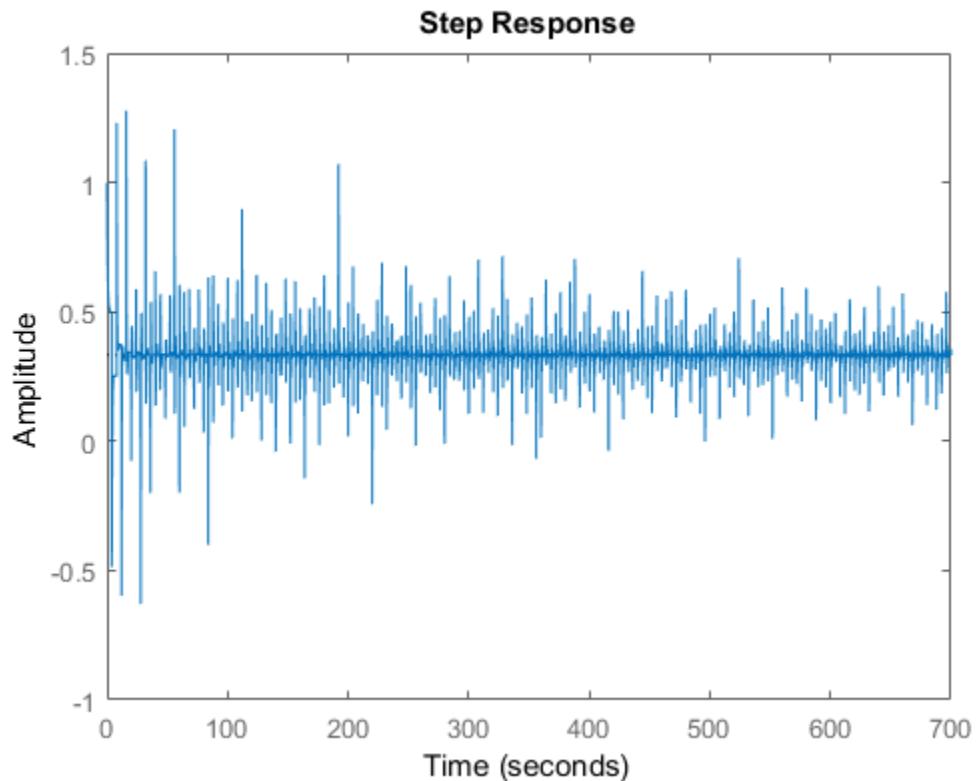
Step Response

Chaotic response:

```
G = 1/(s+1) + exp(-4*s);
T = feedback(1,G);

step(T)
```

See Also

`margin | pade`

Related Examples

*   "Analyzing the Response of an RLC Circuit" on page 8-49

More About

*   "Time Delays in Linear Systems" on page 2-41
*   "Time-Delay Approximation" on page 2-49

Analyzing the Response of an RLC Circuit

This example shows how to analyze the time and frequency responses of common RLC circuits as a function of their physical parameters using Control System Toolbox™ functions.

Bandpass RLC Network

The following figure shows the parallel form of a bandpass RLC circuit:
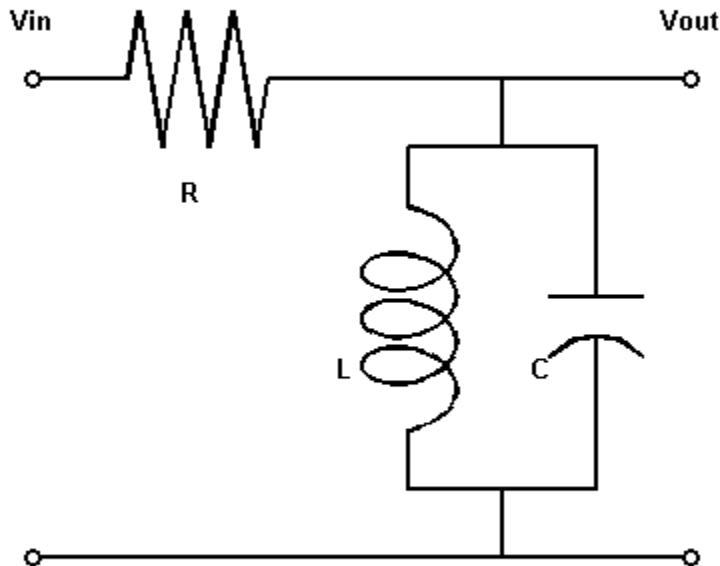
Figure 1: Bandpass RLC Network.

The transfer function from input to output voltage is:

$$G(s) = \frac{s/(RC)}{s^2 + s/(RC) + 1/(LC)}$$

The product LC controls the bandpass frequency while RC controls how narrow the passing band is. To build a bandpass filter tuned to the frequency 1 rad/s, set L=C=1 and use R to tune the filter band.

Analyzing the Frequency Response of the Circuit

The Bode plot is a convenient tool for investigating the bandpass characteristics of the RLC network. Use tf to specify the circuit's transfer function for the values

```
%|R=L=C=1|:
R = 1; L = 1; C = 1;
G = tf([1/(R*C) 0],[1 1/(R*C) 1/(L*C)])


G =

        s
   -----------
   s^2 + s + 1

Continuous-time transfer function.
```
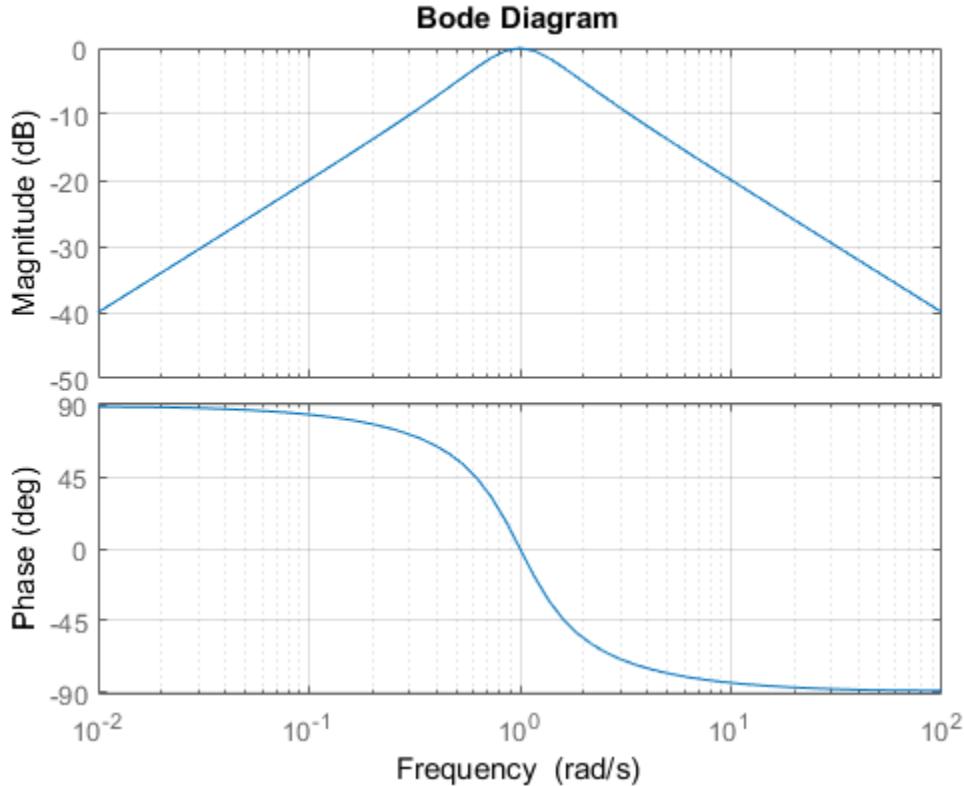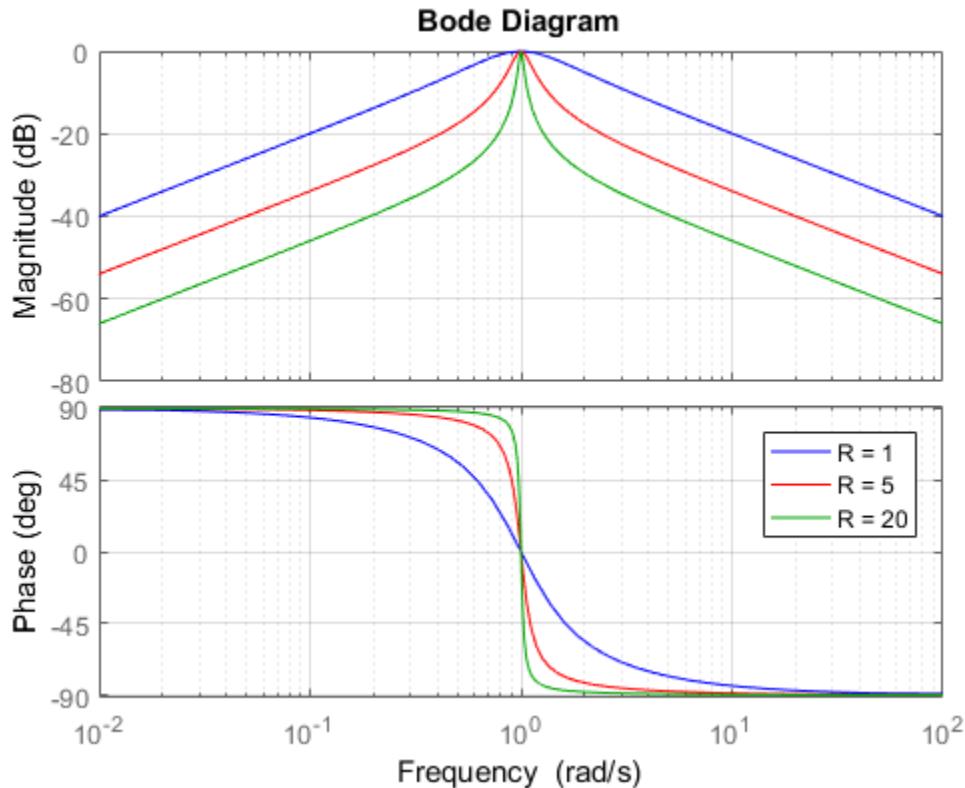
Next, use bode to plot the frequency response of the circuit:

```
bode(G), grid
```

Bode Diagram

As expected, the RLC filter has maximum gain at the frequency 1 rad/s. However, the attenuation is only -10dB half a decade away from this frequency. To get a narrower passing band, try increasing values of R as follows:

```
R1 = 5;    G1 = tf([1/(R1*C) 0],[1 1/(R1*C) 1/(L*C)]);
R2 = 20;   G2 = tf([1/(R2*C) 0],[1 1/(R2*C) 1/(L*C)]);
bode(G,'b',G1,'r',G2,'g'), grid
legend('R = 1','R = 5','R = 20')
```
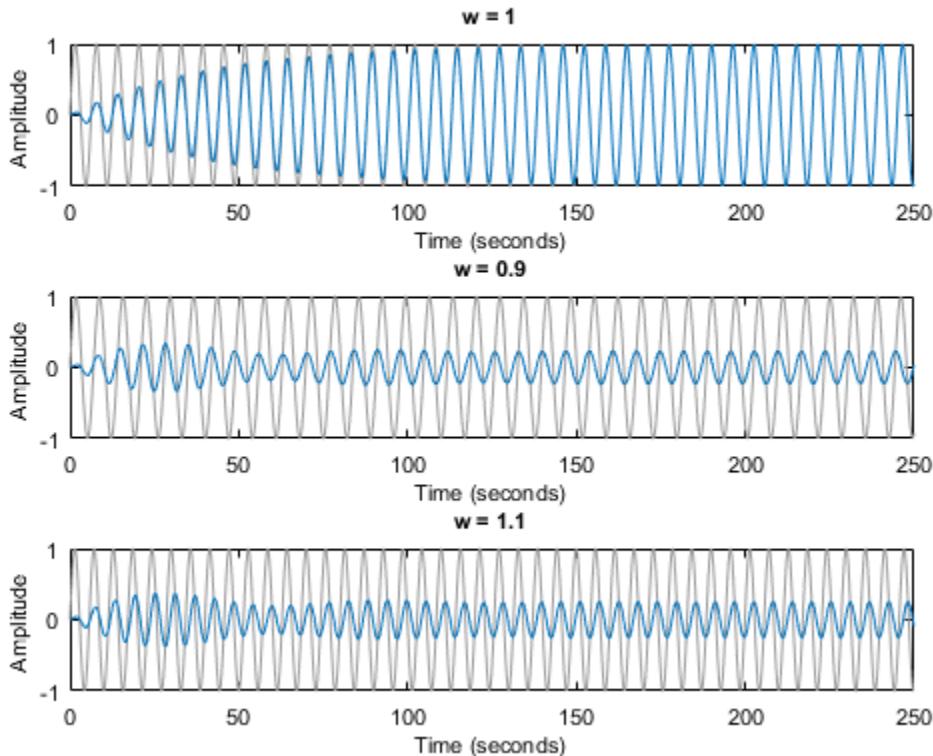
The resistor value `R=20` gives a filter narrowly tuned around the target frequency of 1 rad/s.

Analyzing the Time Response of the Circuit

We can confirm the attenuation properties of the circuit `G2` (`R=20`) by simulating how this filter transforms sine waves with frequency 0.9, 1, and 1.1 rad/s:

```
t = 0:0.05:250;
opt = timeoptions;
opt.Title.FontWeight = 'Bold';
subplot(311), lsim(G2,sin(t),t,opt), title('w = 1')
subplot(312), lsim(G2,sin(0.9*t),t,opt), title('w = 0.9')
subplot(313), lsim(G2,sin(1.1*t),t,opt), title('w = 1.1')
```

The waves at 0.9 and 1.1 rad/s are considerably attenuated. The wave at 1 rad/s comes out unchanged once the transients have died off. The long transient results from the poorly damped poles of the filters, which unfortunately are required for a narrow passing band:

```
damp(pole(G2))
```
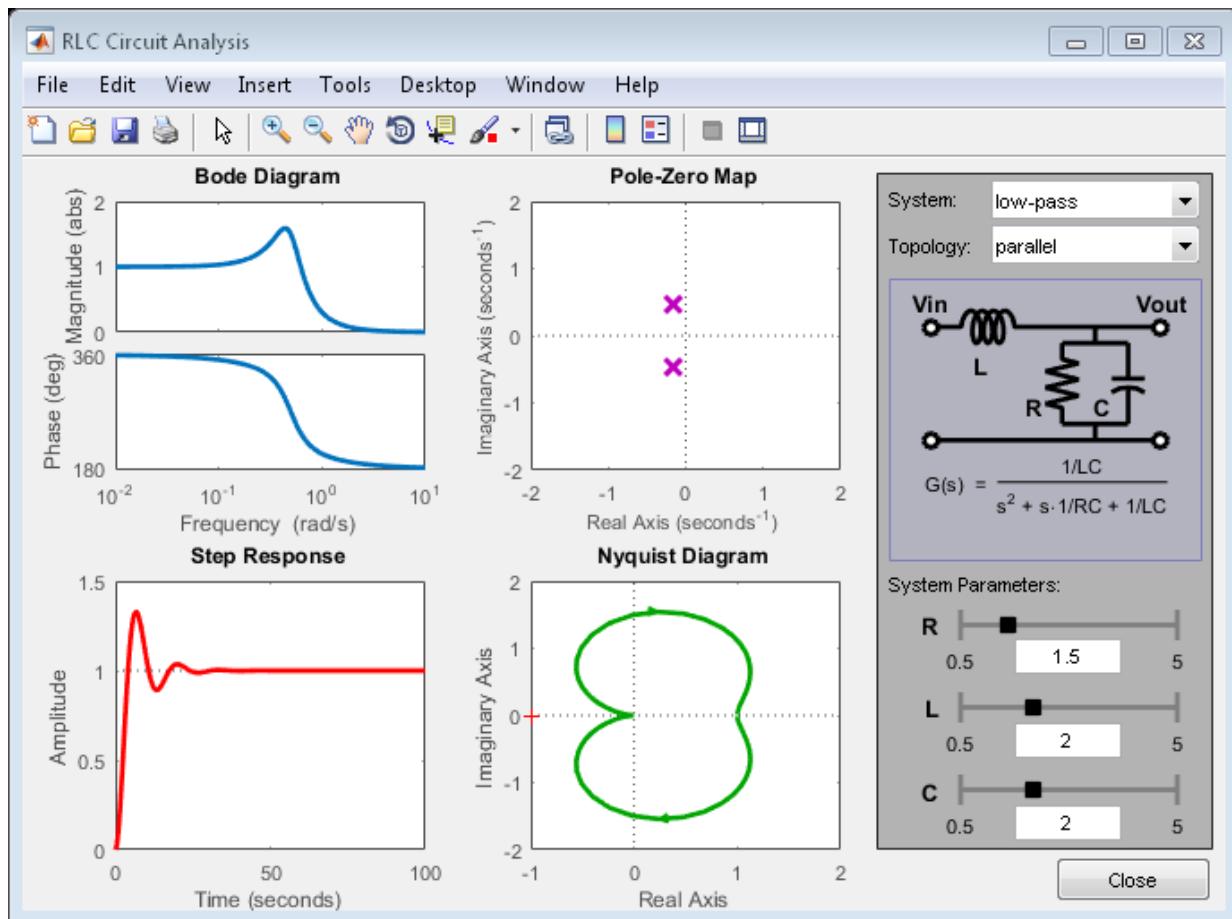
Pole	Damping	Frequency (rad/TimeUnit)	Time Constant (TimeUnit)
-2.50e-02 + 1.00e+00i	2.50e-02	1.00e+00	4.00e+01
-2.50e-02 - 1.00e+00i	2.50e-02	1.00e+00	4.00e+01

Interactive GUI

To analyze other standard circuit configurations such as low-pass and high-pass RLC networks, click on the link below to launch an interactive GUI. In this GUI, you can change the R,L,C parameters and see the effect on the time and frequency responses in real time.

Open the RLC Circuit GUI

```
rlc_gui
```

See Also
bodeplot | lsim | stepplot

Related Examples
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

9

Sensitivity Analysis

Model Array with Single Parameter Variation

This example shows how to create a one-dimensional array of transfer functions using the `stack` command. One parameter of the transfer function varies from model to model in the array. You can use such an array to investigate the effect of parameter variation on your model, such as for sensitivity analysis.

Create an array of transfer functions representing the following low-pass filter at three values of the roll-off frequency, a.

$$F(s) = \frac{a}{s + a}.$$

Create transfer function models representing the filter with roll-off frequency at $a = 3$, 5, and 7.

```
F1 = tf(3,[1 3]);
F2 = tf(5,[1 5]);
F3 = tf(7,[1 7]);
```

Use the `stack` command to build an array.

```
Farray = stack(1,F1,F2,F3);
```

The first argument to `stack` specifies the array dimension along which `stack` builds an array. The remaining arguments specify the models to arrange along that dimension. Thus, `Farray` is a 3-by-1 array of transfer functions.

Concatenating models with MATLAB® array concatenation commands, instead of with `stack`, creates multi-input, multi-output (MIMO) models rather than model arrays. For example:

```
G = [F1;F2;F3];
```

creates a one-input, three-output transfer function model, not a 3-by-1 array.

When working with a model array that represents parameter variations, You can associate the corresponding parameter value with each entry in the array. Set the `SamplingGrid` property to a data structure that contains the name of the parameter and the sampled parameter values corresponding with each model in the array. This assignment helps you keep track of which model corresponds to which parameter value.

```
Farray.SamplingGrid = struct('alpha',[3 5 7]);
Farray
```

```
Farray(:,:,1,1) [alpha=3] =

    3
  -----
  s + 3


Farray(:,:,2,1) [alpha=5] =

    5
  -----
  s + 5


Farray(:,:,3,1) [alpha=7] =

    7
  -----
  s + 7

3x1 array of continuous-time transfer functions.
```

The parameter values in Farray.SamplingGrid are displayed along with the each transfer function in the array.

Plot the frequency response of the array to examine the effect of parameter variation on the filter behavior.

```
bodeplot(Farray)
```

When you use analysis commands such as `bodeplot` on a model array, the resulting plot shows the response of each model in the array. Therefore, you can see the range of responses that results from the parameter variation.

Model Array with Variations in Two Parameters

This example shows how to create a two-dimensional (2-D) array of transfer functions using `for` loops. One parameter of the transfer function varies in each dimension of the array.

You can use the technique of this example to create higher-dimensional arrays with variations of more parameters. Such arrays are useful for studying the effects of multiple-parameter variations on system response.

The second-order single-input, single-output (SISO) transfer function

$$H\left(s\right) = \frac{\omega^2}{s^2 + 2\zeta\omega s + \omega^2}.$$

depends on two parameters: the damping ratio, ζ, and the natural frequency, ω. If both ζ and ω vary, you obtain multiple transfer functions of the form:

$$H_{ij}\left(s\right) = \frac{\omega_j^2}{s^2 + 2\zeta_i\omega_j s + \omega_j^2},$$

where ζ_i and ω_j represent different measurements or sampled values of the variable parameters. You can collect all of these transfer functions in a single variable to create a two-dimensional model array.

Preallocate memory for the model array. Preallocating memory is an optional step that can enhance computation efficiency. To preallocate, create a model array of the required size and initialize its entries to zero.

```
H = tf(zeros(1,1,3,3));
```

In this example, there are three values for each parameter in the transfer function *H*. Therefore, this command creates a 3-by-3 array of single-input, single-output (SISO) zero transfer functions.

Create arrays containing the parameter values.

```
zeta = [0.66,0.71,0.75];
w = [1.0,1.2,1.5];
```

Build the array by looping through all combinations of parameter values.

```
for i = 1:length(zeta)
  for j = 1:length(w)
    H(:,:,i,j) = tf(w(j)^2,[1 2*zeta(i)*w(j) w(j)^2]);
  end
end
```

H is a 3-by-3 array of transfer functions. ζ varies as you move from model to model along a single column of H. The parameter ω varies as you move along a single row.

Plot the step response of H to see how the parameter variation affects the step response.

```
stepplot(H)
```

You can set the `SamplingGrid` property of the model array to help keep track of which set of parameter values corresponds to which entry in the array. To do so, create a grid of parameter values that matches the dimensions of the array. Then, assign these values to `H.SamplingGrid` with the parameter names.

```
[zetagrid,wgrid] = ndgrid(zeta,w);
H.SamplingGrid = struct('zeta',zetagrid,'w',wgrid);
```

When you display `H`, the parameter values in `H.SamplingGrid` are displayed along with the each transfer function in the array.

Study Parameter Variation by Sampling Tunable Model

This example shows how to sample a parametric model of a second-order filter across a grid of parameter values using `sampleBlock`.

Consider the second-order filter represented by:

$$F\left(s\right) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}.$$

Sample this filter at varying values of the damping constant ζ and the natural frequency ω_n. Create a parametric model of the filter by using tunable elements for ζ and ω_n.

```
wn = realp('wn',3);
zeta = realp('zeta',0.8);
F = tf(wn^2,[1 2*zeta*wn wn^2])
```

```
F =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 2 states, and
    wn: Scalar parameter, 5 occurrences.
    zeta: Scalar parameter, 1 occurrences.

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

F is a `genss` model with two tunable Control Design Blocks, the `realp` blocks `wn` and `zeta`. The blocks `wn` and `zeta` have initial values of 3 and 0.8, respectively.

Sample F over a 2-by-3 grid of (`wn`, `zeta`) values.

```
wnvals = [3;5];
zetavals = [0.6 0.8 1.0];
Fsample = sampleBlock(F,'wn',wnvals,'zeta',zetavals);
```

Here, `sampleBlock` samples the model independently over the two ω_n values and three ζ values. Thus, `Fsample` is a 2-by-3 array of state-space models. Each entry in the array is a state-space model that represents F evaluated at the corresponding (`wn`, `zeta`) pair. For example, `Fsample(:,:,2,3)` has `wn` = 5 and `zeta` = 1.0.

Set the `SamplingGrid` property of the model array to help keep track of which set of parameter values corresponds to which entry in the array. To do so, create a grid of

parameter values that matches the dimensions of the array. Then, assign these values to `Fsample.SamplingGrid` in a structure with the parameter names.

```
[wngrid,zetagrid] = ndgrid(wnvals,zetavals);
Fsample.SamplingGrid = struct('wn',wngrid,'zeta',zetagrid);
```

The `ndgrid` command produces the full 2-by-3 grid of (`wn`, `zeta`) combinations. When you display `Fsample` in the command window, the parameter values in `Fsample.SamplingGrid` are displayed along with the each transfer function in the array. The parameter information is also available in response plots. For instance, examine the step response of `Fsample`.

```
stepplot(Fsample)
```

The step response plots show the variation in the natural frequency and damping constant across the six models in the array. When you click on one of the responses in the plot, the datatip includes the corresponding wn and zeta values as specified in Fsample.SamplingGrid.

More About

•   "Models with Tunable Coefficients" on page 1-19

Sensitivity of Control System to Time Delays

This example shows how to examine the sensitivity of a closed-loop control system to time delays within the system.

Time delays are rarely known accurately, so it is often important to understand how sensitive a control system is to the delay value. Such sensitivity analysis is easily performed using LTI arrays and the `InternalDelay` property. For example, consider the notched PI control system developed in "PI Control Loop with Dead Time" from the example "Analyzing Control Systems with Delays." The following commands create an LTI model of that closed-loop system, a third-order plant with an input delay, a PI controller and a notch filter.

```
s = tf('s');
G = exp(-2.6*s)*(s+3)/(s^2+0.3*s+1);
C = 0.06 * (1 + 1/s);
T = feedback(ss(G*C),1);
notch = tf([1 0.2 1],[1 .8 1]);
C = 0.05 * (1 + 1/s);
Tnotch = feedback(ss(G*C*notch),1);
```

Examine the internal delay of the closed-loop system Tnotch.

```
Tnotch.InternalDelay
```

```
ans =

    2.6000
```
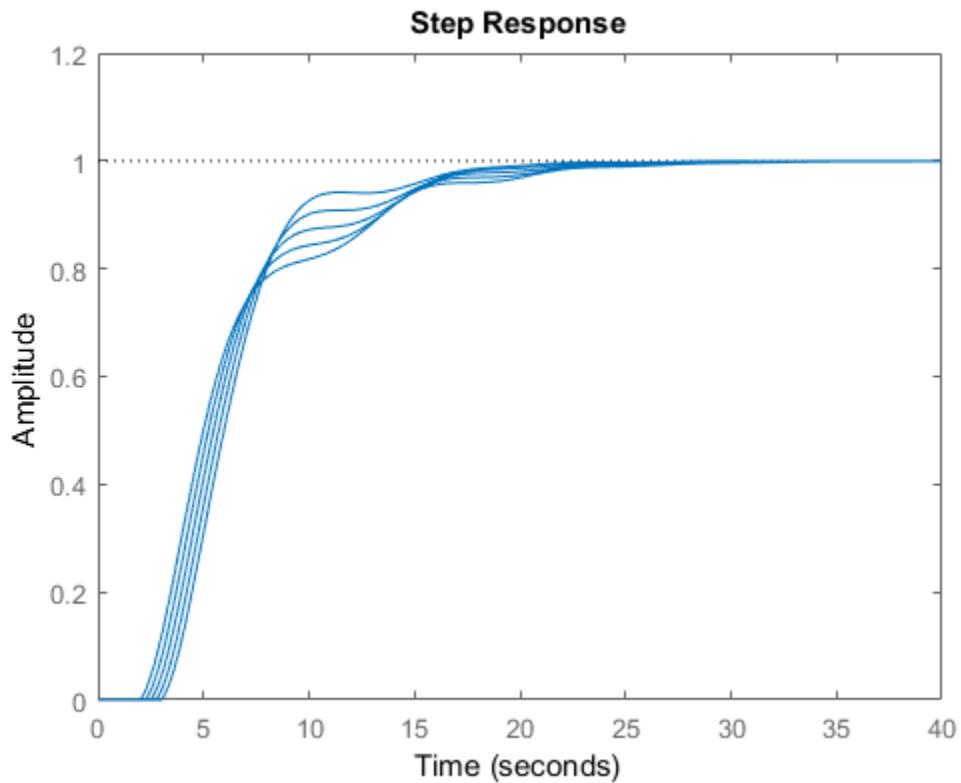
The 2.6-second input delay of the plant `G` becomes an internal delay of 2.6 s in the closed-loop system. To examine the sensitivity of the responses of Tnotch to variations in this delay, create an array of copies of Tnotch. Then, vary the internal delay across the array.

```
Tsens = repsys(Tnotch,[1 1 5]);
tau = linspace(2,3,5);
for j = 1:5;
    Tsens(:,:,j).InternalDelay = tau(j);
end
```

The array `Tsens` contains five models with internal delays that range from 2.0 to 3.0.

Examine the step responses of these models.

```
stepplot(Tsens)
```


The plot shows that uncertainty on the delay value has a small effect on closed-loop characteristics.

Passivity and Conic Sectors

About Passivity and Passivity Indices

Passive control is often part of the safety requirements in applications such as process control, tele-operation, human-machine interfaces, and system networks. A system is *passive* if it cannot produce energy on its own, and can only dissipate the energy that is stored in it initially. More generally, an I/O map is passive if, on average, increasing the output y requires increasing the input u.

For example, a PID controller is passive because the control signal (the output) moves in the same direction as the error signal (the input). But a PID controller with delay is not passive, because the control signal can move in the opposite direction from the error, a potential cause of instability.

Most physical systems are passive. The Passivity Theorem holds that the negative-feedback interconnection of two strictly passive systems is passive and stable. As a result, it can be desirable to enforce passivity of the controller for a passive system, or to *passivate* the operator of a passive system, such as the driver of a car.

In practice, passivity can easily be destroyed by the phase lags introduced by sensors, actuators, and communication delays. These problems have led to extension of the Passivity Theorem that consider excesses or shortages of passivity, frequency-dependent measures of passivity, and a mix of passivity and small-gain properties.

Passive Systems

A linear system $G(s)$ is passive if all input/output trajectories $y(t) = Gu(t)$ satisfy:

$$\int_0^T y^T(t)u(t)dt > 0, \quad \forall T > 0,$$
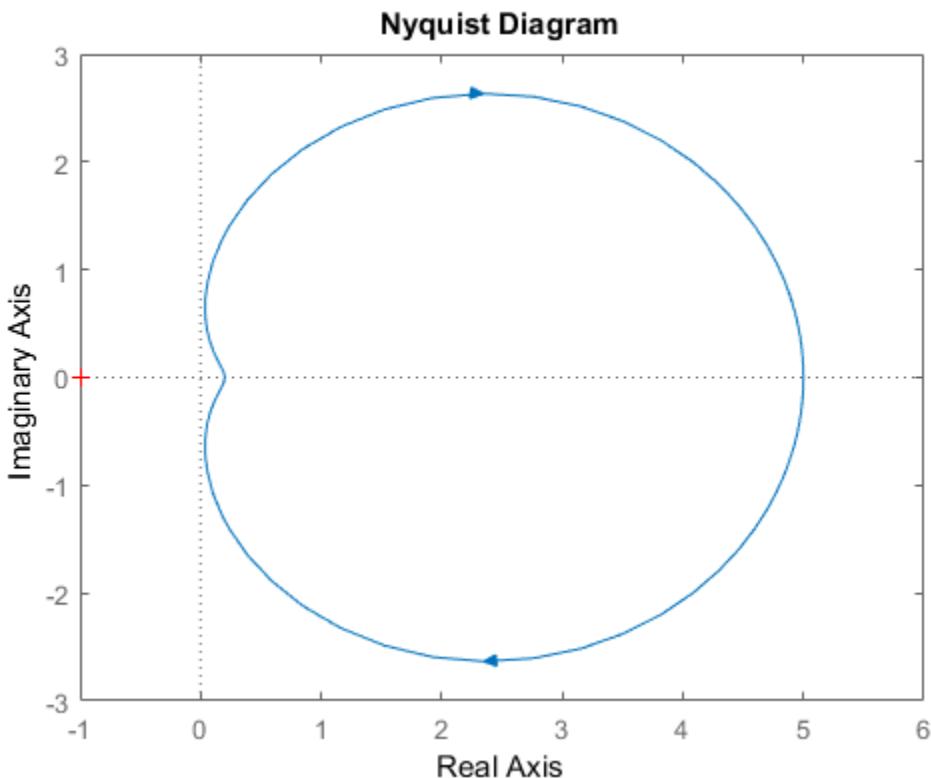
where $y^T(t)$ denotes the transpose of $y(t)$. For physical systems, the integral typically represents the energy going into the system,. Thus passive systems are systems that only consume or dissipate energy. As a result, passive systems are intrinsically stable.

In the frequency domain, passivity is equivalent to the "positive real" condition:

$$G(j\omega) + G^H(j\omega) > 0, \quad \forall \omega \in \mathbf{R}.$$

For SISO systems, this is saying that $\mathrm{Re}(G(j\omega)) > 0$ at all frequencies, so the entire Nyquist plot lies in the right-half plane.

```
nyquist(tf([1 3 5],[5 6 1]))
```

Nyquist Diagram

Nyquist plot of passive system

Passive systems have the following important properties for control purposes:

- The inverse of a passive system is passive.
- The parallel interconnection of passive systems is passive (see "Parallel Interconnection of Passive Systems").

- The feedback interconnection of passive systems is passive (see "Feedback Interconnection of Passive Systems").

When controlling a passive system with unknown or variable characteristics, it is therefore desirable to use a passive feedback law to guarantee closed-loop stability. This task can be rendered difficult given that delays and significant phase lag destroy passivity.
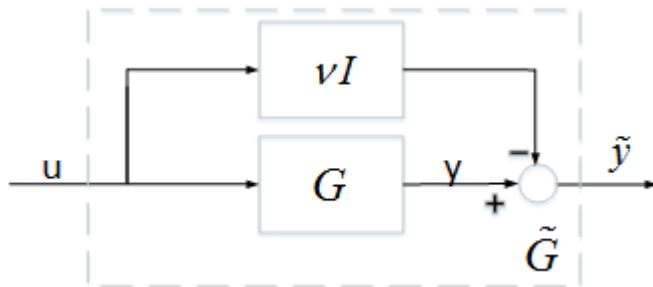
Directional Passivity Indices

For stability, knowing whether a system is passive or not does not tell the full story. It is often desirable to know by how much it is passive or fails to be passive. In addition, a shortage of passivity in the plant can be compensated by an excess of passivity in the controller, and vice versa. It is therefore important to measure the excess or shortage of passivity, and this is where passivity indices come into play.

There are different types of indices with different applications. One class of indices measure the excess or shortage of passivity in a particular direction of the input/output space. For example, the input passivity index is defined as the largest ν such that:

$$\int_0^T y^T(t)u(t)dt > \nu \int_0^T u^T(t)u(t))dt,$$

for all trajectories $y(t) = Gu(t)$ and $T > 0$. The system G is *input strictly passive* (ISP) when $\nu > 0$, and has a shortage of passivity when $\nu < 0$. The input passivity index is also called the input feedforward passivity (IFP) index because it corresponds to the minimum static feedforward action needed to make the system passive.

In the frequency domain, the input passivity index is characterized by:
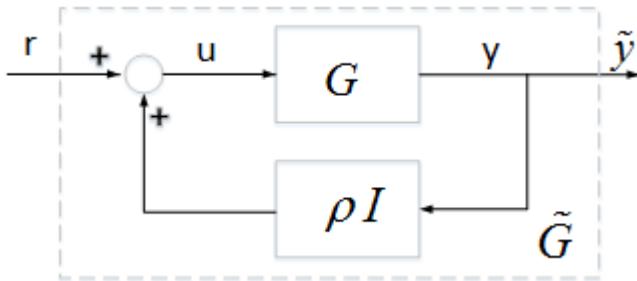
$$\nu = \frac{1}{2}\min_\omega \lambda_{\min}\left(G(j\omega) + G^H(j\omega)\right),$$

where $\lambda_{\min}$ denotes the smallest eigenvalue. In the SISO case, ν is the abscissa of the leftmost point on the Nyquist curve.

Similarly, the output passivity index is defined as the largest ρ such that:

$$\int_0^T (y^T(t)u(t)dt > \rho \int_0^T y^T(t)y(t))dt,$$

for all trajectories $y(t) = Gu(t)$ and $T > 0$. The system G is *output strictly passive* (OSP) when $\rho > 0$, and has a shortage of passivity when $\rho < 0$. The output passivity index is also called the output feedback passivity (OFP) index because it corresponds to the minimum static feedback action needed to make the system passive.

In the frequency domain, the output passivity index of a *minimum-phase* system $G(s)$ is given by:

$$\rho = \frac{1}{2}\min_\omega \lambda_{\min}\left(G^{-1}(j\omega) + G^{-H}(j\omega)\right).$$

In the SISO case, ρ is the abscissa of the leftmost point on the Nyquist curve of $G^{-1}(s)$.

Combining these two notions leads to the I/O passivity index, which is the largest τ such that:

$$\int_0^T y^T(t)u(t)dt > \tau \int_0^T \left(u^T(t)u(t) + y^T(t)y(t)\right)dt.$$

A system with $\tau > 0$ is *very strictly passive*. More generally, we can define the index in the direction δQ as the largest τ such that:

$$\int_0^T y^T(t)u(t)dt > \tau \int_0^T \left(\begin{array}{c} y(t) \\ u(t) \end{array} \right)^T \delta Q \left(\begin{array}{c} y(t) \\ u(t) \end{array} \right) dt.$$

The input, output, and I/O passivity indices all correspond to special choices of δQ and are collectively referred to as *directional passivity indices*. You can use getPassiveIndex to compute any of these indices for linear systems in either parametric or FRD form. You can also use passiveplot to plot the input, output, or I/O passivity indices as a function of frequency. This plot provides insight into which frequency bands have weaker or stronger passivity.

There are many results quantifying how the input and output passivity indices propagate through parallel, series, or feedback interconnections. There are also results quantifying the excess of input or output passivity needed to compensate a given shortage of passivity in a feedback loop. For details, see:

- "Parallel Interconnection of Passive Systems"
- "Series Interconnection of Passive Systems"
- "Feedback Interconnection of Passive Systems"

Relative Passivity Index

The *positive real* condition for passivity:

$$G(j\omega) + G^H(j\omega) > 0 \quad \forall \omega \in \mathbf{R},$$

is equivalent to the small gain condition:

$$\|(I - G(j\omega))(I + G(j\omega))^{-1}\| < 1 \quad \forall \omega \in \mathbf{R}.$$

We can therefore use the peak gain of $(I - G)(I + G)^{-1}$ as a measure of passivity. Specifically, let

$$R := \left\| (I - G)(I + G)^{-1} \right\|_\infty.$$

Then G is passive if and only if $R < 1$, and $R > 1$ indicates a shortage of passivity. Note that R is finite if and only if $I + G$ is minimum phase. We refer to R as the *relative*
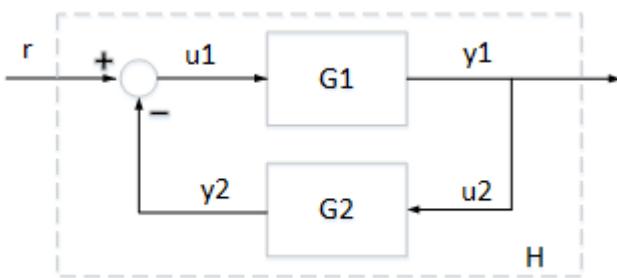
passivity index, or R-index. In the time domain, the R-index is the smallest $r > 0$ such that:

$$\int_0^T ||y - u||^2 dt < r^2 \int_0^T ||y + u||^2 dt,$$

for all trajectories $y(t) = Gu(t)$ and $T > 0$. When $I + G$ is minimum phase, you can use passiveplot to plot the principal gains of $(I - G(j\omega))(I + G(j\omega))^{-1}$. This plot is entirely analogous to the singular value plot (see sigma), and shows how the degree of passivity changes with frequency and direction.

The following result is analogous to the Small Gain Theorem for feedback loops. It gives a simple condition on R-indices for compensating a shortage of passivity in one system by an excess of passivity in the other.

Small-R Theorem: Let $G_1(s)$ and $G_2(s)$ be two linear systems with passivity R-indices R_1 and R_2, respectively. If $R_1 R_2 < 1$, then the negative feedback interconnection of G_1 and G_2 is stable.

See Also
getPassiveIndex | isPassive | passiveplot
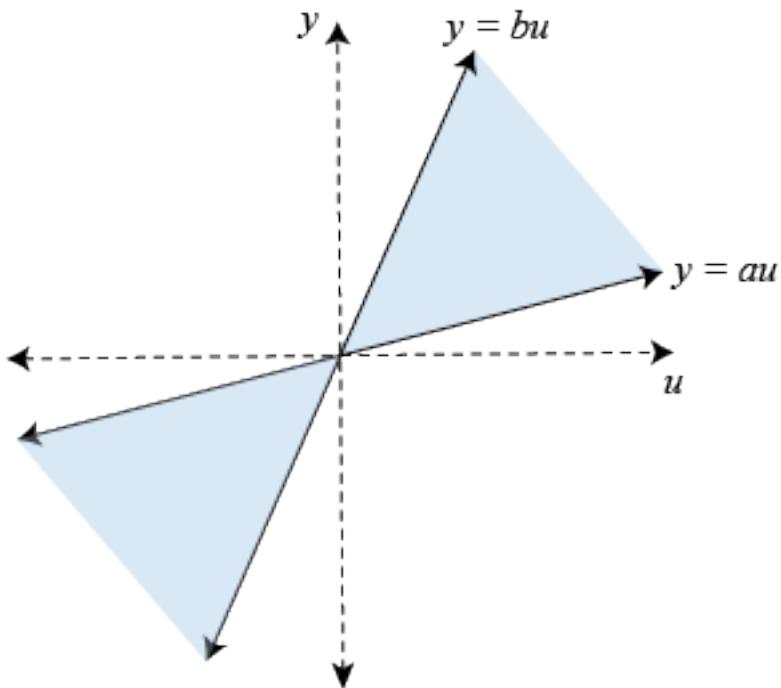
Related Examples
- "Passivity Indices" on page 10-19
- "Parallel Interconnection of Passive Systems" on page 10-25
- "Series Interconnection of Passive Systems" on page 10-28

About Sector Bounds and Sector Indices

Conic Sectors

In its simple form, a conic sector is the 2D region delimited by two lines $y = au$ and $y = bu$.

The shaded region is characterized by the inequality $(y - au)(y - bu) < 0$. More generally, any such sector can be parameterized as:

$$\begin{pmatrix} y \\ u \end{pmatrix}^T Q \begin{pmatrix} y \\ u \end{pmatrix} < 0,$$

where Q is a 2x2 symmetric indefinite matrix (Q has one positive and one negative eigenvalue). We call Q the *sector matrix* . This concept generalizes to higher dimensions. In an N-dimensional space, a conic sector is a set:

$$\mathbf{S} = \left\{ z \in \mathbf{R}^N : \quad z^T Q z < 0 \right\},$$

where Q is again a symmetric indefinite matrix.

Sector Bounds

Sector bounds are constraints on the behavior of a system. Gain constraints and passivity constraints are special cases of sector bounds. If for all nonzero input $u(t)$, the output trajectory $z(t) = (Hu)(t)$ of a linear system $H(s)$ satisfies:

$$\int_0^T z^T(t) Q z(t) dt < 0, \quad \forall T > 0,$$

then the output trajectories of H lie in the conic sector with matrix Q. Selecting different Q matrices imposes different conditions on the system's response. For example, consider trajectories $y(t) = (Gu)(t)$ and the following values:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix}.$$

These values correspond to the sector bound:

$$\int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix} \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt < 0, \quad \forall T > 0.$$

This sector bound is equivalent to the passivity condition for $G(s)$:

$$\int_0^T y^T(t) u(t) dt > 0, \quad \forall T > 0.$$

In other words, passivity is a particular sector bound on the systemdefined by:

$$H = \left(\begin{array}{c} G \\ I \end{array} \right).$$

Frequency-Domain Condition

Because the time-domain condition must hold for all $T > 0$, deriving an equivalent frequency-domain bound takes a little care and is not always possible. Let the following:

$$Q = W_1^T W_1 - W_2^T W_2$$

be (any) decomposition of the indefinite matrix Q into its positive and negative parts. When $W_2^T H(s)$ is square and minimum phase (has no unstable zeros), the time-domain condition:

$$\int_0^T (Hu(t))^T Q (Hu(t))dt < 0, \quad \forall T > 0$$

is equivalent to the frequency-domain condition:

$$H(j\omega)^H Q H(j\omega) < 0 \quad \forall \omega \in \mathbf{R}.$$

It is therefore enough to check the sector inequality for real frequencies. Using the decomposition of Q, this is also equivalent to:

$$\left\| (W_1^T H)(W_2^T H)^{-1} \right\|_\infty < 1.$$
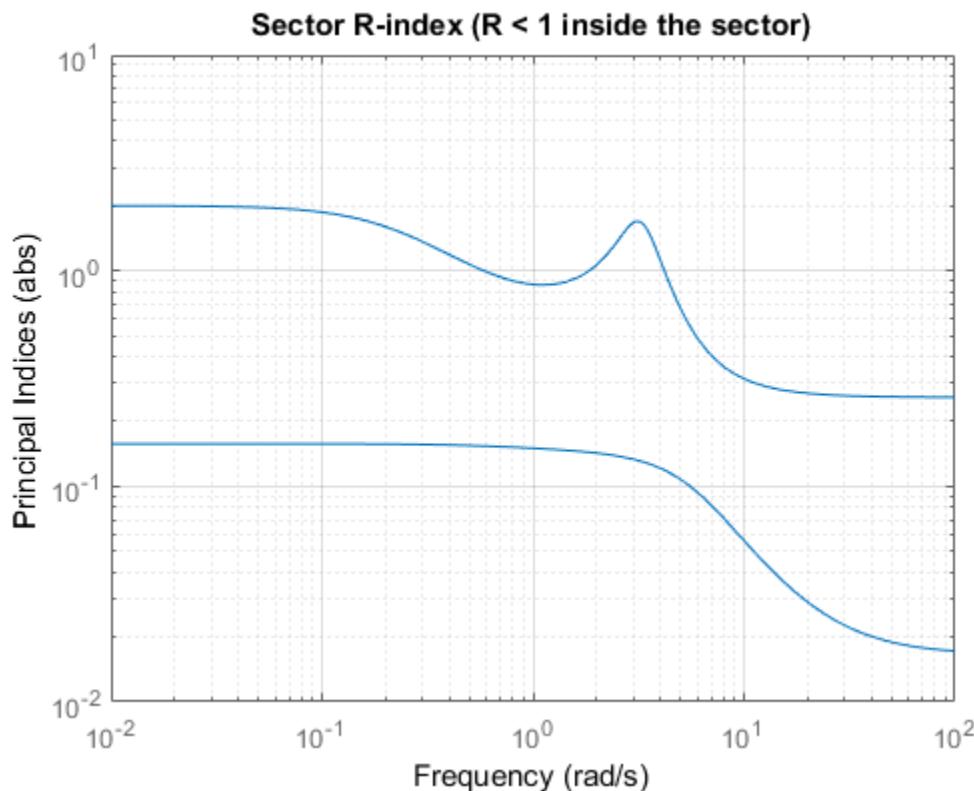
Note that $W_2^T H$ is square when Q has as many negative eigenvalues as input channels in $H(s)$. If this condition is not met, it is no longer enough (in general) to just look at real frequencies. Note also that if $W_2^T H(s)$ is square, then it must be minimum phase for the sector bound to hold.

This frequency-domain characterization is the basis for sectorplot. Specifically, `sectorplot` plots the singular values of $(W_1^T H(j\omega))(W_2^T H(j\omega))^{-1}$ as a function of

frequency. The sector bound is satisfied if and only if the largest singular value stays below 1. Moreover, the plot contains useful information about the frequency bands where the sector bound is satisfied or violated, and the degree to which it is satisfied or violated.

For instance, examine the sector plot of a 2-output, 2-input system for a particular sector.

```
rng(4);
H = rss(3,4,2);
Q = [-5.12    2.16   -2.04    2.17
      2.16   -1.22   -0.28   -1.11
     -2.04   -0.28   -3.35    0.00
      2.17   -1.11    0.00    0.18];
sectorplot(H,Q)
```

The plot shows that the largest singular value of $(W_1^T H(j\omega))(W_2^T H(j\omega))^{-1}$ exceeds 1 below about 0.5 rad/s and in a narrow band around 3 rad/s. Therefore, H does not satisfy the sector bound represented by Q.

Relative Sector Index

We can extend the notion of relative passivity index to arbitrary sectors. Let $H(s)$ be an LTI system, and let:

$$Q = W_1^T W_1 - W_2^T W_2, \quad W_1^T W_2 = 0$$

be an orthogonal decomposition of Q into its positive and negative parts, as is readily obtained from the Schur decomposition of Q. The *relative sector index* R, or R-index, is defined as the smallest $r > 0$ such that for all output trajectories $z(t) = Hu(t)$:

$$\int_0^T z^T(t) \, (W_1^T W_1 - r^2 W_2^T W_2) \, z(t)dt < 0, \quad \forall T > 0.$$
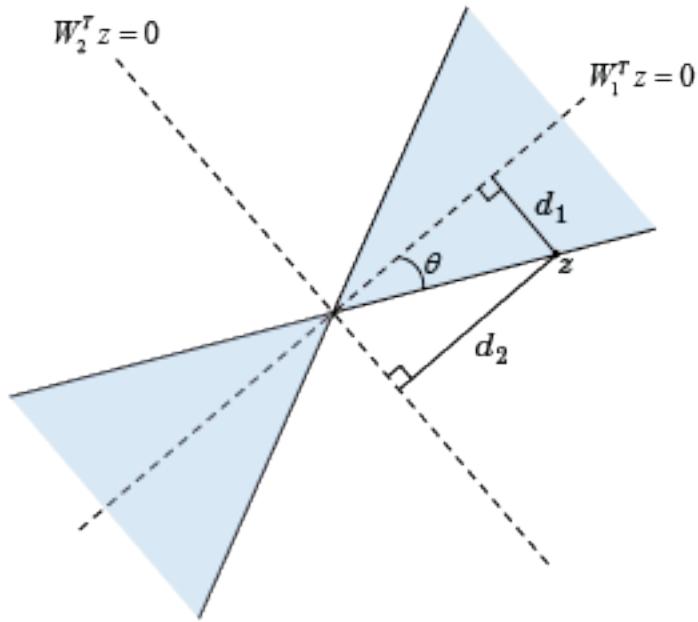
Because increasing r makes $W_1^T W_1 - r^2 W_2^T W_2$ more negative, the inequality is usually satisfied for r large enough. However, there are cases when it can never be satisfied, in which case the R-index is $R = +\infty$. Clearly, the original sector bound is satisfied if and only of $R \leq 1$.

To understand the geometrical interpretation of the R-index, consider the family of cones with matrix $Q(r) = W_1^T W_1 - r^2 W_2^T W_2$. In 2D, the cone slant angle θ is related to r by

$$\tan(\theta) = r\frac{\|W_2\|}{\|W_1\|}$$

(see diagram below). More generally, $\tan(\theta)$ is proportional to R. Thus, given a conic sector with matrix Q, an R-index value $R < 1$ means that we can reduce $\tan(\theta)$ (narrow the cone) by a factor R before some output trajectory of H leaves the conic sector. Similarly, a value $R > 1$ means that we must increase $\tan(\theta)$ (widen the cone) by a factor R to include all output trajectories of H. This clearly makes the R-index a relative measure of how well the response of H fits in a particular conic sector.

10-13

In the diagram,

$$d_1 \frac{\left|W_1^T z\right|}{\|W_1\|}, \quad d_2 \frac{\left|W_2^T z\right|}{\|W_2\|}, \quad R = \frac{\left|W_1^T z\right|}{\left|W_2^T z\right|},$$

and

$$\tan(\theta) = \frac{d_1}{d_2} = R \frac{\|W_2\|}{\|W_1\|}.$$

When $W_2^T H(s)$ is square and minimum phase, the R-index can also be characterized in the frequency domain as the smallest $r > 0$ such that:

$$H(j\omega)^H \left(W_1^T W_1 - r^2 W_2^T W_2\right) H(j\omega) < 0 \quad \forall \omega \in \mathbf{R}.$$

Using elementary algebra, this leads to:

$$R = \max_\omega \left\| (W_1^T H(j\omega))(W_2^T H(j\omega))^{-1} \right\|.$$

In other words, the R-index is the peak gain of the (stable) transfer function $\Phi(s) := (W_1^T H(s))(W_2^T H(s))^{-1}$, and the singular values of $\Phi(jw)$ can be seen as the "principal" R-indices at each frequency. This also explains why plotting the R-index vs. frequency looks like a singular value plot (see sectorplot). There is a complete analogy between relative sector index and system gain. Note, however, that this analogy only holds when $W_2^T H(s)$ is square and minimum phase.

Directional Sector Index

Similarly, we can extend the notion of directional passivity index to arbitrary sectors. Given a conic sector with matrix Q, and a direction δQ, the directional sector index is the largest τ such that for all output trajectories $z(t) = Hu(t)$:

$$\int_0^T z^T(t)\,(Q + \tau\delta Q)\,z(t)dt < 0, \quad \forall T > 0.$$

The directional passivity index for a system $G(s)$ corresponds to:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix}.$$

The directional sector index measures by how much we need to deform the sector in the direction δQ to make it fit tightly around the output trajectories of H. The sector bound is satisfied if and only if the directional index is positive.

Common Sectors

There are many ways to specify sector bounds. Next we review commonly encountered expressions and give the corresponding system H and sector matrix Q for the standard form used by `getSectorIndex` and `sectorplot`:

$$\int_0^T (Hu(t))^T Q(Hu(t))dt < 0, \quad \forall T > 0.$$

For simplicity, these descriptions use the notation:

$$\|x\|_T = \int_0^T \|x(t)\|^2 dt,$$

and omit the $\forall T > 0$ requirement.

Passivity

Passivity is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix}.$$

Gain constraint

The gain constraint $ \| G \|_\infty < \gamma $ % is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} I & 0 \\ 0 & -\gamma^2 I \end{pmatrix}.$$

Ratio of distances

Consider the "interior" constraint,

$$\|y - cu\|_T < r\|u\|_T$$

where c, r are scalars and $y(t) = (Gu)(t)$. This is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} I & -cI \\ -cI & (c^2 - r^2)I \end{pmatrix}.$$

The underlying conic sector is symmetric with respect to $y = cu$. Similarly, the "exterior" constraint,

$$\|y - cu\|_T > r\|u\|_T$$

is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} -I & cI \\ cI & (r^2 - c^2)I \end{pmatrix}.$$

Double inequality

When dealing with static nonlinearities, it is common to consider conic sectors of the form

$$au^2 < yu < bu^2,$$

where $y = \phi(u)$ is the nonlinearity output. While this relationship is not a sector bound per se, it clearly implies:

$$a \int_0^T u(t)^2 dt < \int_0^T y(t)u(t)dt < b \int_0^T u(t)^2 dt$$

along all I/O trajectories and for all $T > 0$. This condition in turn is equivalent to a sector bound with:

$$H(s) = \begin{pmatrix} \phi(.) \\ 1 \end{pmatrix}, \quad Q = \begin{pmatrix} 1 & -(a+b)/2 \\ (a+b)/2 & ab \end{pmatrix}.$$

Product form

Generalized sector bounds of the form:

$$\int_0^T (y(t) - K_1 u(t))^T (y(t) - K_2 u(t))dt < 0$$

correspond to:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 2I & -(K_2 + K_1^T) \\ -(K_1 + K_2^T) & K_1^T K_2 + K_2^T K_1 \end{pmatrix}.$$

As before, the static sector bound:

$$(y - K_1 u)^T (y - K_2 u) < 0$$

implies the integral sector bound above.

QSR dissipative

A system $y = Gu$ is QSR-dissipative if it satisfies:

$$\int_0^T \left(\begin{array}{c} y(t) \\ u(t) \end{array} \right)^T \left(\begin{array}{cc} \mathbf{Q} & \mathbf{S} \\ \mathbf{S^T} & \mathbf{R} \end{array} \right) \left(\begin{array}{c} y(t) \\ u(t) \end{array} \right) dt > 0, \quad \forall T > 0.$$

This is a sector bound with:

$$H(s) = \left(\begin{array}{c} G(s) \\ I \end{array} \right), \quad Q = - \left(\begin{array}{cc} \mathbf{Q} & \mathbf{S} \\ \mathbf{S^T} & \mathbf{R} \end{array} \right).$$

See Also

getSectorCrossover | getSectorIndex | sectorplot

Related Examples

- "About Passivity and Passivity Indices" on page 10-2

Passivity Indices
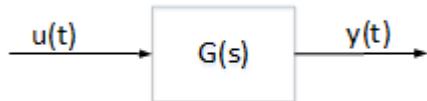
This example shows how to compute various measures of passivity for linear time-invariant systems.

Passive Systems

A linear system G(s) is passive when all I/O trajectories $(u(t), y(t))$ satisfy

$$\int_0^T y^T(t)u(t)dt > 0, \quad \forall T > 0$$

where $y^T(t)$ denotes the transpose of $y(t)$.

To measure "how passive" a system is, we use passivity indices.

- The input passivity index is defined as the largest ν such that

$$\int_0^T y^T(t)u(t)dt > \nu \int_0^T u^T(t)u(t))dt$$

The system G is "input strictly passive" (ISP) when $\nu > 0$. ν is also called the "input feedforward passivity" (IFP) index and corresponds to the minimum feedforward action needed to make the system passive.

- The output passivity index is defined as the largest ρ such that

$$\int_0^T (y^T(t)u(t)dt > \rho \int_0^T y^T(t)y(t))dt$$

The system G is "output strictly passive" (OSP) when $\rho > 0$. ρ is also called the "output feedback passivity" (OFP) index and corresponds to the minimum feedback action needed to make the system passive.

- The I/O passivity index is defined as the largest τ such that
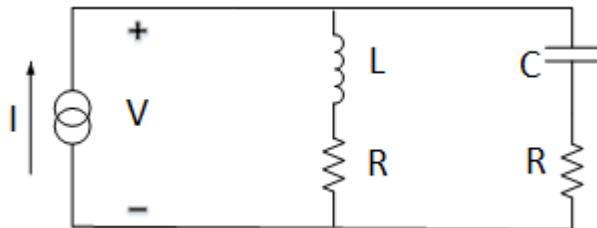
$$\int_0^T y^T(t)u(t)dt > \tau \int_0^T \left(u^T(t)u(t) + y^T(t)y(t)\right) dt$$

The system is "very strictly passive" (VSP) if $\tau > 0$.

Circuit Example

Consider the following example. We take the current I as the input and the voltage V as the output. Based on Kirchhoff's current and voltage law, we obtain the transfer function for $G(s)$,

$$G(s) = \frac{V(s)}{I(s)} = \frac{(Ls + R)(Rs + \frac{1}{C})}{Ls^2 + 2Rs + \frac{1}{C}}.$$

Let $R = 2$, $L = 1$ and $C = 0.1$.

```
R = 2; L = 1; C = 0.1;
s = tf('s');
G = (L*s+R)*(R*s+1/C)/(L*s^2 + 2*R*s+1/C);
```

Use isPassive to check whether $G(s)$ is passive.

```
PF = isPassive(G)
```

```
PF =
```

```
1
```

Since PF = true, $G(s)$ is passive. Use `getPassiveIndex` to compute the passivity indices of $G(s)$.

```
% Input passivity index
nu = getPassiveIndex(G,'in')
% Output passivity index
rho = getPassiveIndex(G,'out')
% I/O passivity index
tau = getPassiveIndex(G,'io')
```

```
nu =

    2.0000


rho =

    0.2857


tau =

    0.2642
```

Since $\tau > 0$, the system $G(s)$ is very strictly passive.

Frequency-Domain Characterization

A linear system is passive if and only if it is "positive real":

$$G(j\omega) + G^H(j\omega) > 0 \quad \forall \omega \in \mathbf{R}.$$

The smallest eigenvalue of the left-hand-side is related to the input passivity index ν:
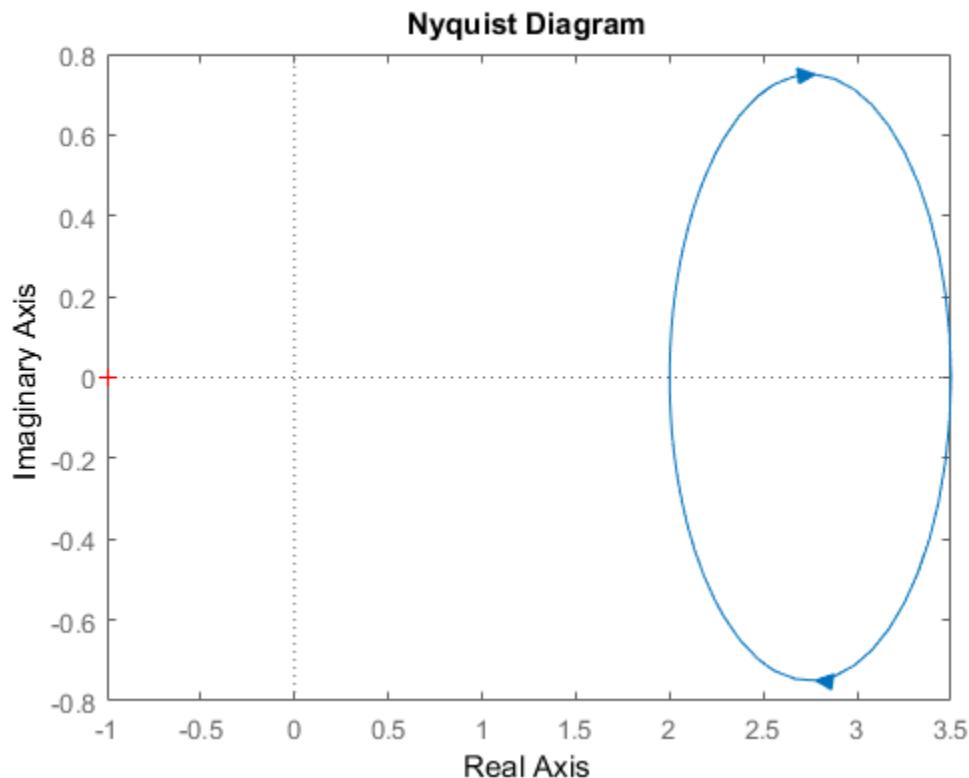
$$\nu = \frac{1}{2} \min_{\omega} \lambda_{\min} \left(G(j\omega) + G^H(j\omega) \right)$$

where $\lambda_{\min}$ denotes the smallest eigenvalue. Similarly, when $G(s)$ is minimum-phase, the output passivity index is given by:

$$\rho = \frac{1}{2} \min_{\omega} \lambda_{\min} \left(G^{-1}(j\omega) + G^{-H}(j\omega) \right).$$

Verify this for the circuit example. Plot the Nyquist plot of the circuit transfer function.

```
nyquist(G)
```

Nyquist Diagram

The entire Nyquist plot lies in the right-half plane so $G(s)$ is positive real. The leftmost point on the Nyquist curve is $(x, y) = (2, 0)$ so the input passivity index is $\nu = 2$, the

same value we obtained earlier. Similarly, the leftmost point on the Nyquist curve for $G^{-1}(s)$ gives the output passivity index value $\rho = 0.286$.

Relative Passivity Index

It can be shown that the "positive real" condition

$$G(j\omega) + G^H(j\omega) > 0 \quad \forall \omega \in \mathbf{R}$$

is equivalent to the small gain condition

$$\|(I - G(j\omega))(I + G(j\omega))^{-1}\| < 1 \quad \forall \omega \in \mathbf{R}.$$

The relative passivity index (R-index) is the peak gain over frequency of $(I - G)(I + G)^{-1}$ when $I + G$ is minimum phase, and $+\infty$ otherwise:

$$R = \left\| (I - G)(I + G)^{-1} \right\|_\infty .$$

In the time domain, the R-index is the smallest $r > 0$ such that

$$\int_0^T \|y - u\|^2 dt < r^2 \int_0^T \|y + u\|^2 dt$$

The system $G(s)$ is passive if and only if $R < 1$, and the smaller R is, the more passive the system is. Use `getPassiveIndex` to compute the R-index for the circuit example.

```
R = getPassiveIndex(G)
```

```
R =

    0.5556
```

The resulting R value indicates that the circuit is a very passive system.

See Also
getPassiveIndex | isPassive
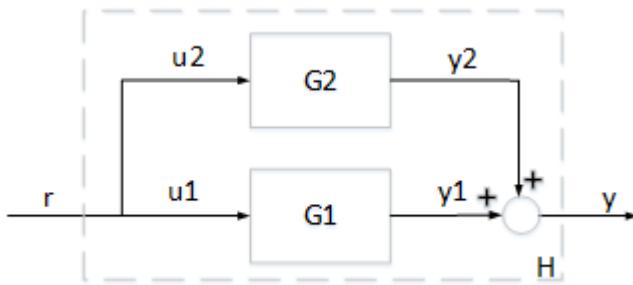
Related Examples

Parallel Interconnection of Passive Systems

This example illustrates the properties of a parallel interconnection of passive systems.

Parallel Interconnection of Passive Systems

Consider an interconnection of two subsystems G_1 and G_2 in parallel. The interconnected system H maps the input r to the output y.

If both systems G_1 and G_2 are passive, then the interconnected system H is guaranteed to be passive. Take for example

$$G_1(s) = \frac{0.1s + 1}{s + 2}; \quad G_2(s) = \frac{s^2 + 2s + 1}{s^2 + 3s + 10}$$

Both systems are passive.

```
G1 = tf([0.1,1],[1,2]);
isPassive(G1)
G2 = tf([1,2,1],[1,3,10]);
isPassive(G2)
```

```
ans =

     1


ans =

     1
```

We can therefore expect their parallel interconnection H to be passive, as confirmed by

```
H = parallel(G1,G2);
isPassive(H)
```

```
ans =

     1
```

Passivity Indices for Parallel Interconnection

There is a relationship between the passivity indices of G_1 and G_2 and the passivity indices of the interconnected system H. Let ν_1 and ν_2 denote the input passivity indices for G_1 and G_2, and let ρ_1 and ρ_2 denote the output passivity indices. If all these indices are nonnegative, then the input passivity index ν and the output passivity index ρ for the parallel interconnection H satisfy

$$\nu \geq \nu_1 + \nu_2, \quad \rho \geq \frac{\rho_1 \rho_2}{\rho_1 + \rho_2}.$$

In other words, we can infer some minimum level of input and output passivity for the parallel connection H from the input and output passivity indices of G_1 and G_2. For details, see the paper by Yu, H., "Passivity and dissipativity as design and analysis tools for networked control systems," *Chapter 2*, PhD Thesis, University of Notre Dame, 2012. Verify the lower bound for the input passivity index ν.

```
% Input passivity index for G1
nu1 = getPassiveIndex(G1,'input');
% Input passivity index for G2
nu2 = getPassiveIndex(G2,'input');
% Input passivity index for H
nu = getPassiveIndex(H,'input')
```

```
nu =

    0.3833
```

```
% Lower bound
```

```
nu1+nu2


ans =

    0.1497
```

Similarly, verify the lower bound for the output passivity index of H.

```
% Output passivity index for G1
rho1 = getPassiveIndex(G1,'output');
% Output passivity index for G2
rho2 = getPassiveIndex(G2,'output');
% Output passivity index for H
rho = getPassiveIndex(H,'output')


rho =

    0.6450


% Lower bound
rho1*rho2/(rho1+rho2)


ans =

    0.2098
```

See Also
getPassiveIndex | isPassive
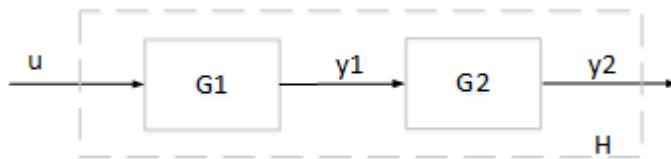
Related Examples
- "About Passivity and Passivity Indices" on page 10-2
- "Series Interconnection of Passive Systems" on page 10-28
- "Feedback Interconnection of Passive Systems" on page 10-32

Series Interconnection of Passive Systems

This example illustrates the properties of a series interconnection of passive systems.

Series Interconnection of Passive Systems

Consider an interconnection of two subsystems G_1 and G_2 in series. The interconnected system H is given by the mapping from input u to output y_2.

In contrast with parallel and feedback interconnections, passivity of the subsystems G_1 and G_2 does not guarantee passivity for the interconnected system H. Take for example

$$G_1(s) = \frac{5s^2 + 3s + 1}{s^2 + 2s + 1}, \quad G_2(s) = \frac{s^2 + s + 5s + 0.1}{s^3 + 2s^2 + 3s + 4}.$$

Both systems are passive as confirmed by

```
G1 = tf([5 3 1],[1,2,1]);
isPassive(G1)
```

```
ans =

     1
```

```
G2 = tf([1,1,5,.1],[1,2,3,4]);
isPassive(G2)
```

```
ans =

     1
```
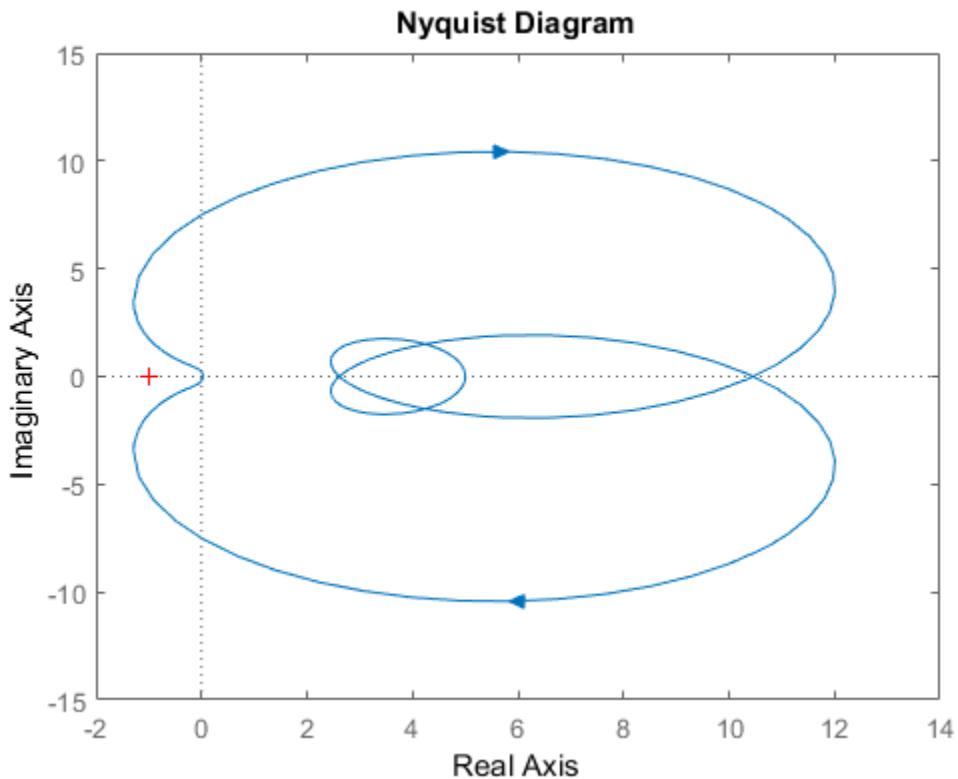
However the series interconnection of G_1 and G_2 is not passive:

```
H = G2*G1;
isPassive(H)
```

ans =

     0

This is confirmed by verifying that the Nyquist plot of G_2G_1 is not positive real.

```
nyquist(H)
```

Passivity Indices for Series Interconnection

While the series interconnection of passive systems is not passive in general, there is a relationship between the passivity indices of G_1 and G_2 and the passivity indices of $H = G_2 G_1$. Let ν_1 and ν_2 denote the input passivity indices for G_1 and G_2, and let ρ_1 and ρ_2 denote the output passivity indices. If all these indices are positive, then the input passivity index ν and the output passivity index ρ for the series interconnection H satisfy

$$\nu \geq -\frac{0.125}{\rho_1 \rho_2}, \quad \rho \geq -\frac{0.125}{\nu_1 \nu_2}.$$

In other words, the shortage of passivity at the inputs or outputs of H is no worse than the right-hand-side expressions. For details, see the paper by Arcak, M. and Sontag, E.D., "Diagonal stability of a class of cyclic systems and its connection with the secant criterion," *Automatica*, Vol 42, No. 9, 2006, pp. 1531-1537. Verify these lower bounds for the example above.

```
% Output passivity index for G1
rho1 = getPassiveIndex(G1,'output');
% Output passivity index for G2
rho2 = getPassiveIndex(G2,'output');
% Input passivity index for H=G2*G1
nu = getPassiveIndex(H,'input')


nu =

   -1.2875


% Lower bound
-0.125/(rho1*rho2)


ans =

   -2.4119
```

Similarly, verify the lower bound for the output passivity index of H.

```
% Input passivity index for G1
```

```
nu1 = getPassiveIndex(G1,'input');
% Input passivity index for G2
nu2 = getPassiveIndex(G2,'input');
% Output passivity index for H=G2*G1
rho = getPassiveIndex(H,'output')


rho =

   -0.6868


% Lower bound
-0.125/(nu1*nu2)


ans =

   -5.9700
```

See Also

getPassiveIndex | isPassive
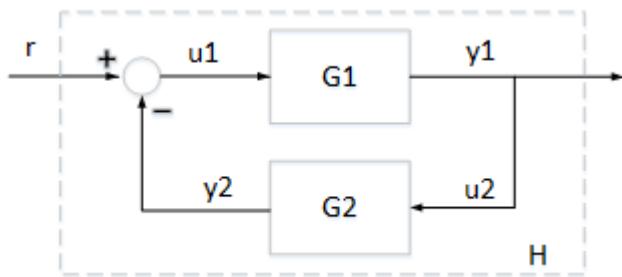
Related Examples

- "About Passivity and Passivity Indices" on page 10-2
- "Parallel Interconnection of Passive Systems" on page 10-25
- "Feedback Interconnection of Passive Systems" on page 10-32

Feedback Interconnection of Passive Systems

This example illustrates the properties of a feedback interconnection of passive systems.

Feedback Interconnection of Passive Systems

Consider an interconnection of two subsystems G_1 and G_2 in feedback. The interconnected system H maps the input r to the output y_1.

If both systems G_1 and G_2 are passive, then the interconnected system H is guaranteed to be passive. Take for example

$$G_1(s) = \frac{s^2 + s + 1}{s^2 + s + 4}, \quad G_2(s) = \frac{s + 2}{s + 5}.$$

Both systems are passive as confirmed by

```
G1 = tf([1,1,1],[1,1,4]);
isPassive(G1)
```

```
ans =

     1
```

```
G2 = tf([1,2],[1,5]);
isPassive(G2)
```

```
ans =

     1
```

The interconnected system is therefore passive.
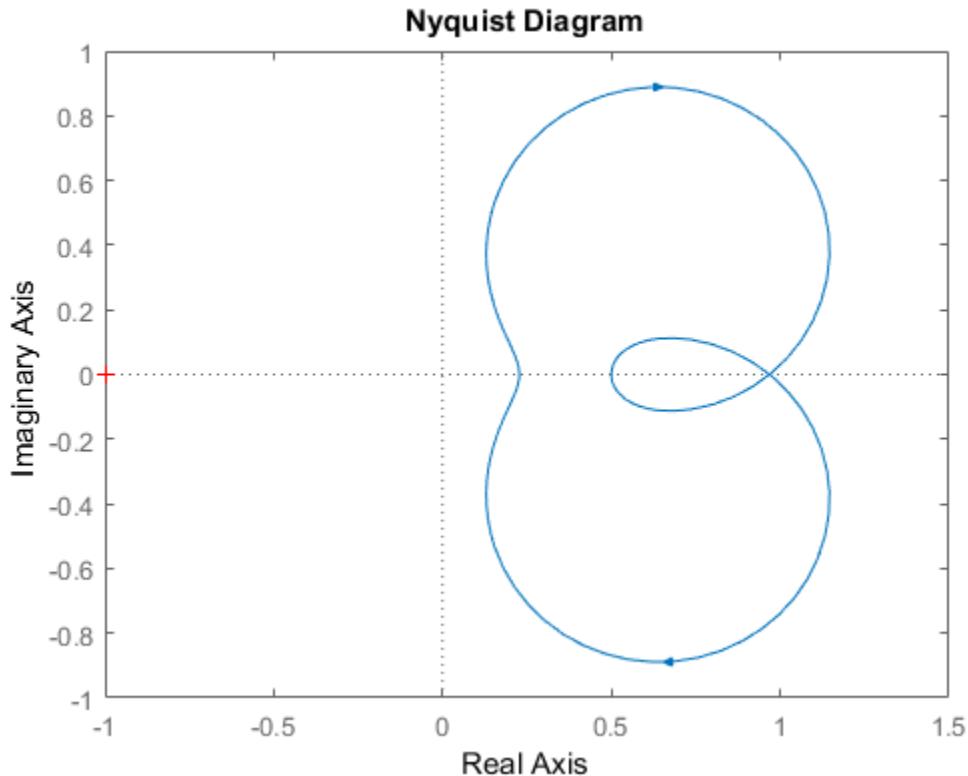
```
H = feedback(G1,G2);
isPassive(H)
```

```
ans =

     1
```

This is confirmed by verifying that the Nyquist plot of H is positive real.

```
nyquist(H)
```

Nyquist Diagram

Passivity Indices for Feedback Interconnection

There is a relationship between the passivity indices of G_1 and G_2 and the passivity indices of the interconnected system H. Let ν_1 and ν_2 denote the input passivity indices for G_1 and G_2, and let ρ_1 and ρ_2 denote the output passivity indices. If all these indices are positive, then the input passivity index ν and the output passivity index ρ for the feedback interconnection H satisfy

$$\nu \geq \frac{\nu_1 \rho_2}{\nu_1 + \rho_2}, \quad \rho \geq \rho_1 + \nu_2.$$

In other words, we can infer some minimum level of input and output passivity for the closed-loop system H from the input and output passivity indices of G_1 and G_2. For

details, see the paper by Zhu, F. and Xia, M and Antsaklis, P.J., "Passivity analysis and passivation of feedback systems using passivity indices," *American Control Conference* , 2014, pp. 1833-1838. Verify the lower bound for the input passivity index ν.

```
% Input passivity index for G1
nu1 = getPassiveIndex(G1,'input');
% Output passivity index for G2
rho2 = getPassiveIndex(G2,'output');
% Input passivity index for H
nu = getPassiveIndex(H,'input')
```

```
nu =

    0.1296
```

```
% Lower bound
nu1*rho2/(nu1+rho2)
```

```
ans =

   3.4864e-05
```

Similarly, verify the lower bound for the output passivity index of H.

```
% Output passivity index for G1
rho1 = getPassiveIndex(G1,'output');
% Input passivity index for G2
nu2 = getPassiveIndex(G2,'input');
% Output passivity index for H
rho = getPassiveIndex(H,'output')
```

```
rho =

    0.4482
```

```
% Lower bound
rho1+nu2
```

```
ans =
```

```
0.4001
```

See Also

`getPassiveIndex` | `isPassive`

Related Examples

- "About Passivity and Passivity Indices" on page 10-2
- "Parallel Interconnection of Passive Systems" on page 10-25
- "Series Interconnection of Passive Systems" on page 10-28
- "Passive Control with Communication Delays" on page 13-480

Control Design

PID Controller Design

PID Controller Design at the Command Line

This example shows how to design a PID controller for the plant given by:

$$sys = \frac{1}{(s+1)^3}.$$

As a first pass, create a model of the plant and design a simple PI controller for it.

```
sys = zpk([],[-1 -1 -1],1);
[C_pi,info] = pidtune(sys,'PI')
```

```
C_pi =

            1
  Kp + Ki * ---
            s

  with Kp = 1.14, Ki = 0.454

Continuous-time PI controller in parallel form.


info =

              Stable: 1
   CrossoverFrequency: 0.5205
          PhaseMargin: 60.0000
```
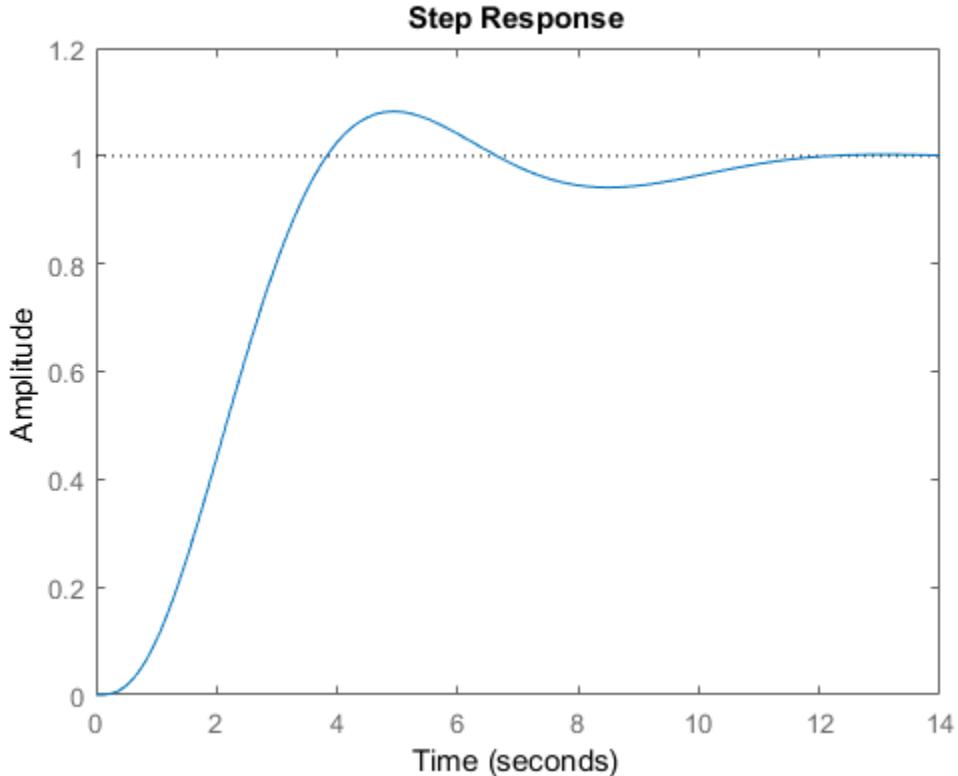
`C_pi` is a `pid` controller object that represents a PI controller. The fields of `info` show that the tuning algorithm chooses an open-loop crossover frequency of about 0.52 rad/s.

Examine the closed-loop step response (reference tracking) of the controlled system.

```
T_pi = feedback(C_pi*sys, 1);
step(T_pi)
```

Step Response

To improve the response time, you can set a higher target crossover frequency than the result that `pidtune` automatically selects, 0.52. Increase the crossover frequency to 1.0.

```
[C_pi_fast,info] = pidtune(sys,'PI',1.0)
```

```
C_pi_fast =

              1
  Kp + Ki * ---
              s

  with Kp = 2.83, Ki = 0.0495
```
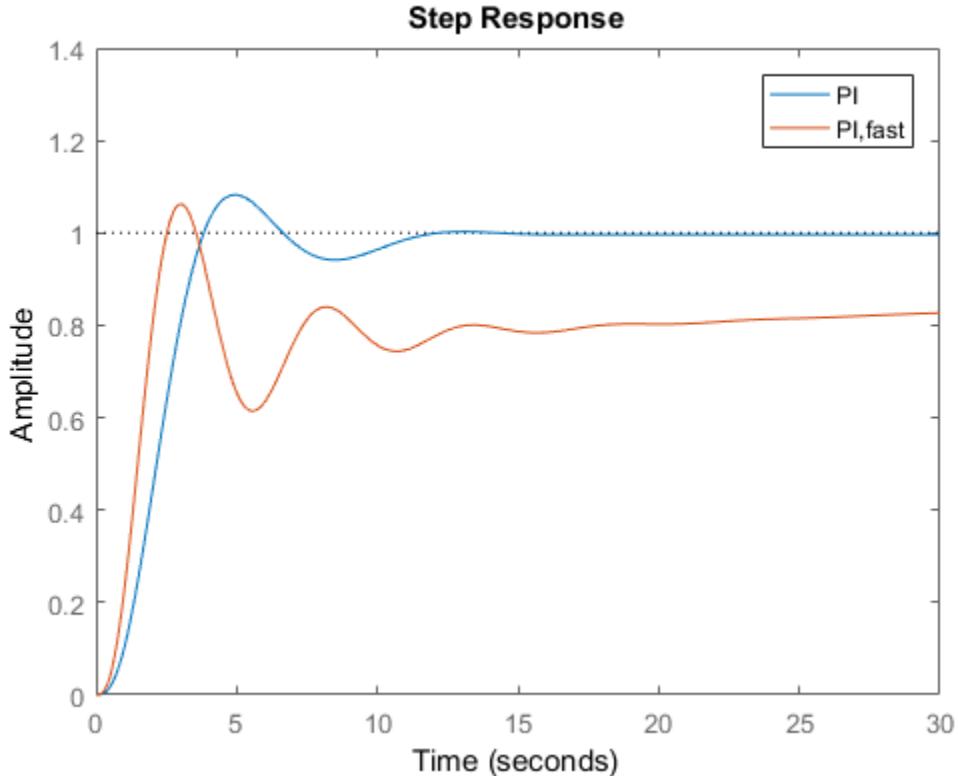
```
Continuous-time PI controller in parallel form.


info =

               Stable: 1
    CrossoverFrequency: 1
          PhaseMargin: 43.9973
```

The new controller achieves the higher crossover frequency, but at the cost of a reduced phase margin.

Compare the closed-loop step response with the two controllers.

```
T_pi_fast = feedback(C_pi_fast*sys,1);
step(T_pi,T_pi_fast)
axis([0 30 0 1.4])
legend('PI','PI,fast')
```

This reduction in performance results because the PI controller does not have enough degrees of freedom to achieve a good phase margin at a crossover frequency of 1.0 rad/s. Adding a derivative action improves the response.

Design a PIDF controller for `Gc` with the target crossover frequency of 1.0 rad/s.

```
[C_pidf_fast,info] = pidtune(sys,'PIDF',1.0)
```

```
C_pidf_fast =

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

```
  with Kp = 2.72, Ki = 0.985, Kd = 1.72, Tf = 0.00875

Continuous-time PIDF controller in parallel form.


info =

                Stable: 1
    CrossoverFrequency: 1
           PhaseMargin: 60.0000
```
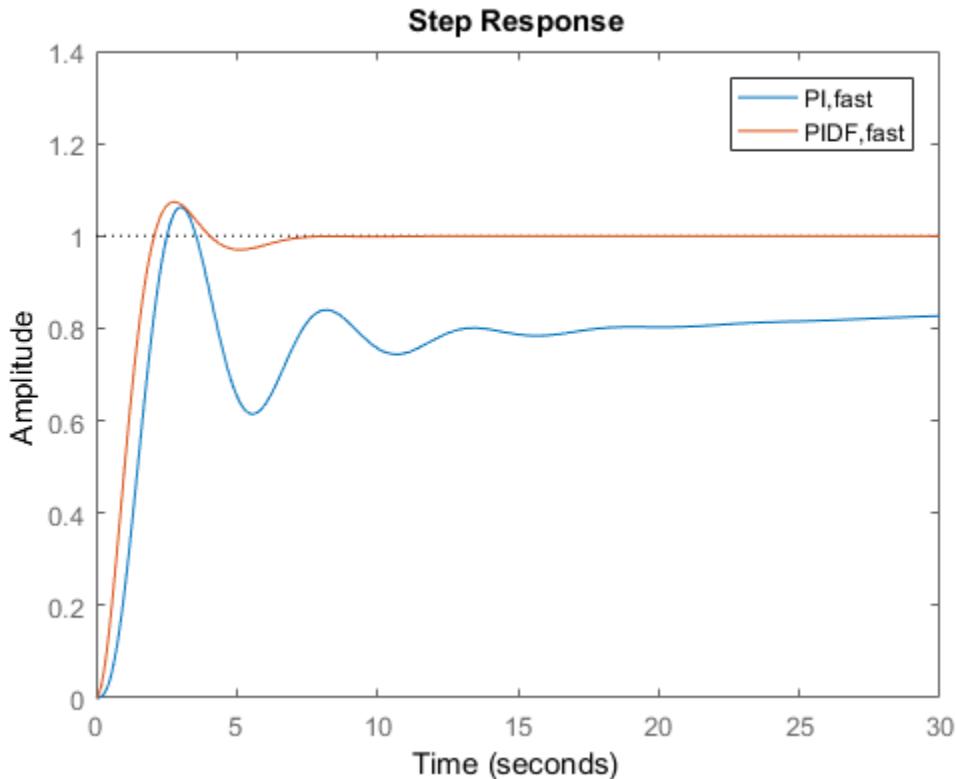
The fields of info show that the derivative action in the controller allows the tuning algorithm to design a more aggressive controller that achieves the target crossover frequency with a good phase margin.
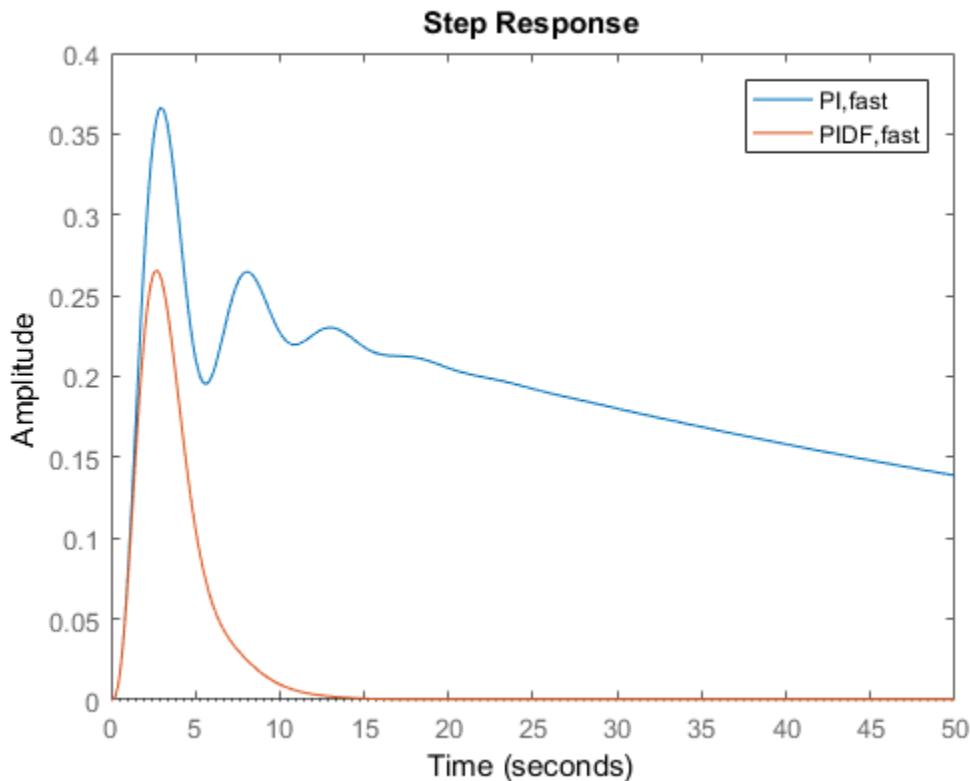
Compare the closed-loop step response and disturbance rejection for the fast PI and PIDF controllers.

```
T_pidf_fast =  feedback(C_pidf_fast*sys,1);
step(T_pi_fast, T_pidf_fast);
axis([0 30 0 1.4]);
legend('PI,fast','PIDF,fast');
```

You can compare the input (load) disturbance rejection of the controlled system with the fast PI and PIDF controllers. To do so, plot the response of the closed-loop transfer function from the plant input to the plant output.

```
S_pi_fast = feedback(sys,C_pi_fast);
S_pidf_fast = feedback(sys,C_pidf_fast);
step(S_pi_fast,S_pidf_fast);
axis([0 50 0 0.4]);
legend('PI,fast','PIDF,fast');
```

This plot shows that the PIDF controller also provides faster disturbance rejection.

See Also

`pid | pidtune`

Related Examples

- "Designing Cascade Control System with PI Controllers" on page 11-10
- "PID Controller Design for Fast Reference Tracking"

More About
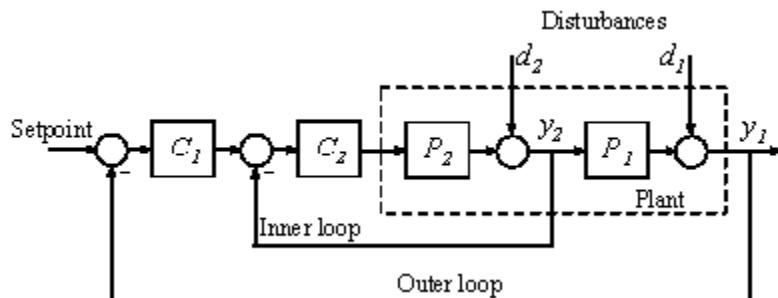
· "Choosing a PID Controller Design Tool"

Designing Cascade Control System with PI Controllers

This example shows how to design a cascade control loop with two PI controllers using the pidtune command.

Introduction to Cascade Control

Cascade control is mainly used to achieve fast rejection of disturbance before it propagates to the other parts of the plant. The simplest cascade control system involves two control loops (inner and outer) as shown in the block diagram below.

Controller **C1** in the outer loop is the primary controller that regulates the primary controlled variable **y1** by setting the set-point of the inner loop. Controller **C2** in the inner loop is the secondary controller that rejects disturbance **d2** locally before it propagates to **P1**. For a cascade control system to function properly, the inner loop must respond much faster than the outer loop.

In this example, you will design a single loop control system with a PI controller and a cascade control system with two PI controllers. The responses of the two control systems are compared for both reference tracking and disturbance rejection.

Plant

In this example, the inner loop plant P2 is

$$P2(s) = \frac{3}{s+2}$$
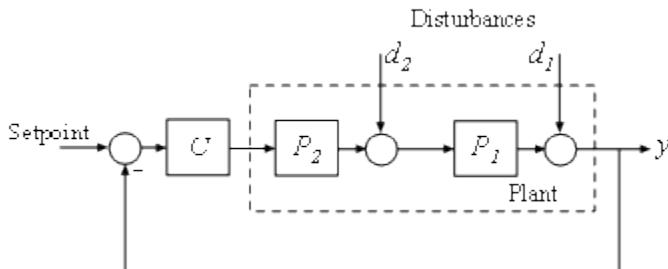
The outer loop plant P1 is

$$P1(s) = \frac{10}{(s+1)^3}$$

```
P2 = zpk([],-2,3);
P1 = zpk([],[-1 -1 -1],10);
```

Designing a Single Loop Control System with a PI Controller

Use **pidtune** command to design a PI controller in standard form for the whole plant model P = P1 * P2.

The desired open loop bandwidth is 0.2 rad/s, which roughly corresponds to the response time of 10 seconds.

```
% The plant model is P = P1*P2
P = P1*P2;
% Use a PID or PIDSTD object to define the desired controller structure
C = pidstd(1,1);
% Tune PI controller for target bandwidth is 0.2 rad/s
C = pidtune(P,C,0.2);
C


C =

             1      1
  Kp * (1 + ---- * ---)
             Ti     s

  with Kp = 0.0119, Ti = 0.849

Continuous-time PI controller in standard form
```

Designing a Cascade Control System with Two PI Controllers

The best practice is to design the inner loop controller **C2** first and then design the outer loop controller **C1** with the inner loop closed. In this example, the inner loop bandwidth

is selected as 2 rad/s, which is ten times higher than the desired outer loop bandwidth. In order to have an effective cascade control system, it is essential that the inner loop responds much faster than the outer loop.

Tune inner-loop controller C2 with open-loop bandwidth at 2 rad/s.

```
C2 = pidtune(P2,pidstd(1,1),2);
C2


C2 =

            1       1
  Kp * (1 + ---- * ---)
            Ti      s

  with Kp = 0.244, Ti = 0.134

Continuous-time PI controller in standard form
```

Tune outer-loop controller C1 with the same bandwidth as the single loop system.

```
% Inner loop system when the control loop is closed first
clsys = feedback(P2*C2,1);
% Plant seen by the outer loop controller C1 is clsys*P1
C1 = pidtune(clsys*P1,pidstd(1,1),0.2);
C1


C1 =

            1       1
  Kp * (1 + ---- * ---)
            Ti      s

  with Kp = 0.015, Ti = 0.716

Continuous-time PI controller in standard form
```
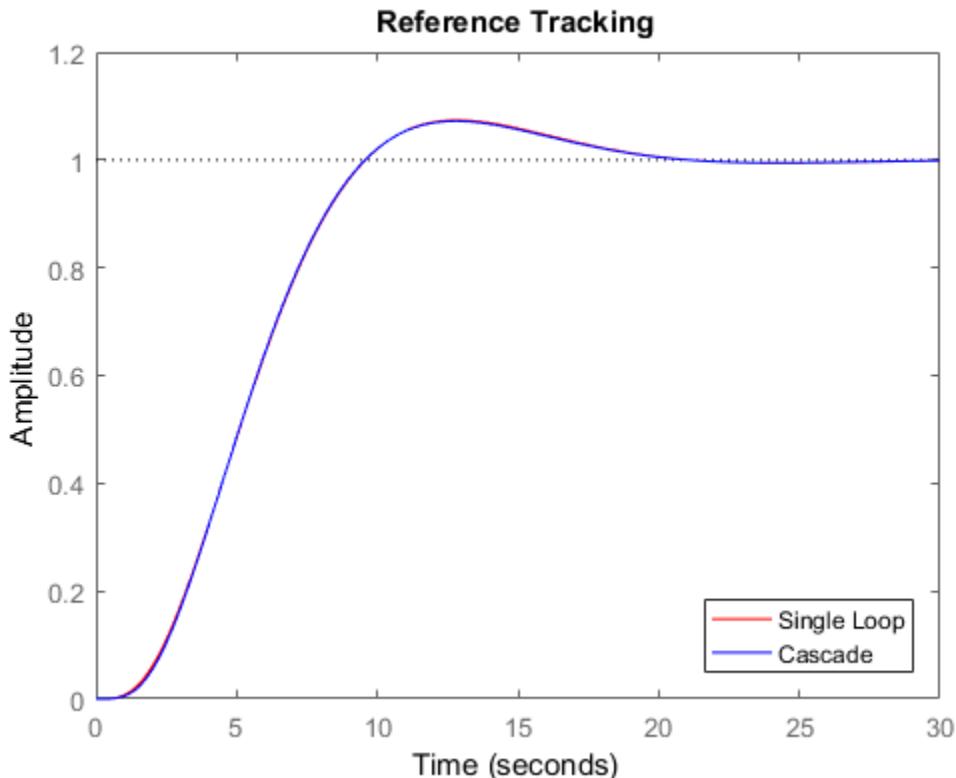
Performance Comparison

First, plot the step reference tracking responses for both control systems.

```
% single loop system for reference tracking
```
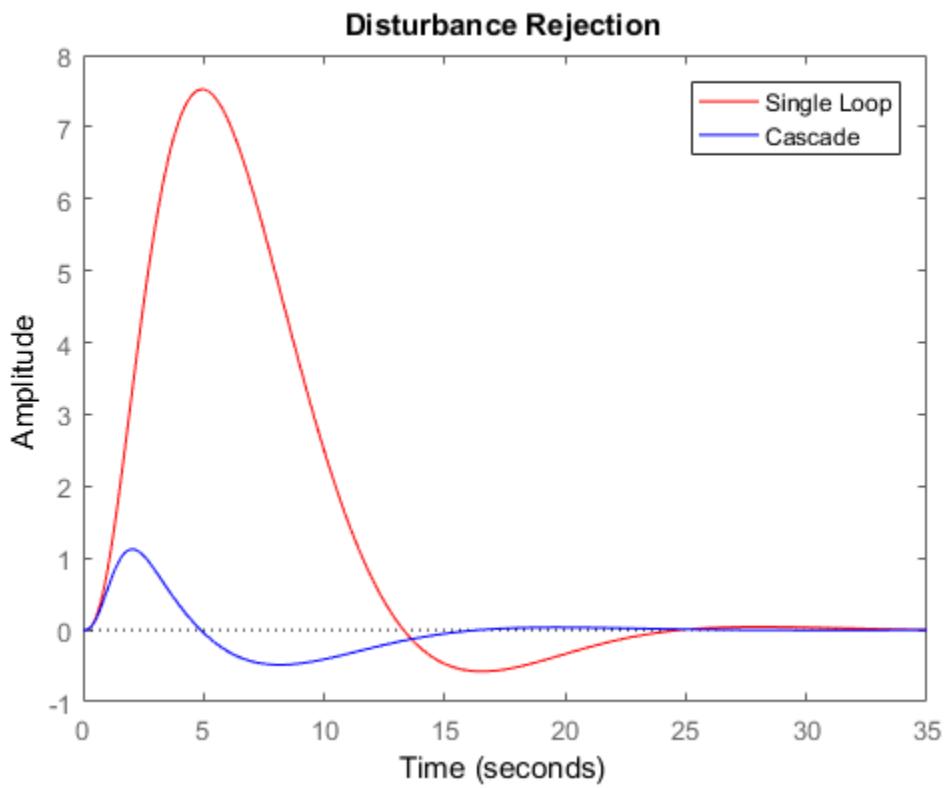
```matlab
sys1 = feedback(P*C,1);
sys1.Name = 'Single Loop';
% cascade system for reference tracking
sys2 = feedback(clsys*P1*C1,1);
sys2.Name = 'Cascade';
% plot step response
figure;
step(sys1,'r',sys2,'b')
legend('show','location','southeast')
title('Reference Tracking')
```


Secondly, plot the step disturbance rejection responses of d2 for both control systems.

```matlab
% single loop system for rejecting d2
```

```
sysd1 = feedback(P1,P2*C);
sysd1.Name = 'Single Loop';
% cascade system for rejecting d2
sysd2 = P1/(1+P2*C2+P2*P1*C1*C2);
sysd2.Name = 'Cascade';
% plot step response
figure;
step(sysd1,'r',sysd2,'b')
legend('show')
title('Disturbance Rejection')
```


Disturbance Rejection

From the two response plots you can conclude that the cascade control system performs much better in rejecting disturbance d2 while the set-point tracking performances are almost identical.

See Also
pidstd | pidtune

Related Examples
·   "PID Controller Design at the Command Line" on page 11-2
·   "Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (Command Line)"
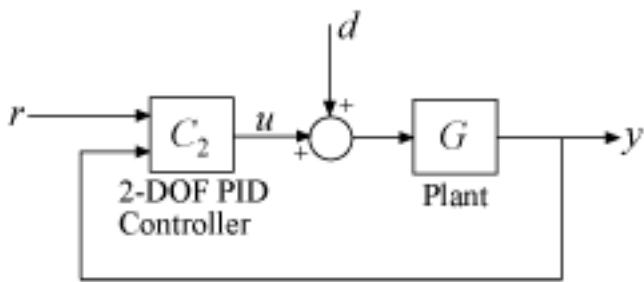
More About
·   "Choosing a PID Controller Design Tool"

Tune 2-DOF PID Controller (Command Line)

This example shows how to design a two-degree-of-freedom (2-DOF) PID controller at the command line. The example also compares the 2-DOF controller performance to the performance achieved with a 1-DOF PID controller.

2-DOF PID controllers include setpoint weighting on the proportional and derivative terms. Compared to a 1-DOF PID controller, a 2-DOF PID controller can achieve better disturbance rejection without significant increase of overshoot in setpoint tracking. A typical control architecture using a 2-DOF PID controller is shown in the following diagram.

For this example, design a 2-DOF controller for the plant given by:

$$G(s) = \frac{1}{s^2 + 0.5s + 0.1}.$$

Suppose that your target bandwidth for the system is 1.5 rad/s.

```
wc = 1.5;
G = tf(1,[1 0.5 0.1]);
C2 = pidtune(G,'PID2',wc)


C2 =

                        1
  u = Kp (b*r-y) + Ki --- (r-y) + Kd*s (c*r-y)
                        s

  with Kp = 1.26, Ki = 0.255, Kd = 1.38, b = 0.665, c = 0
```
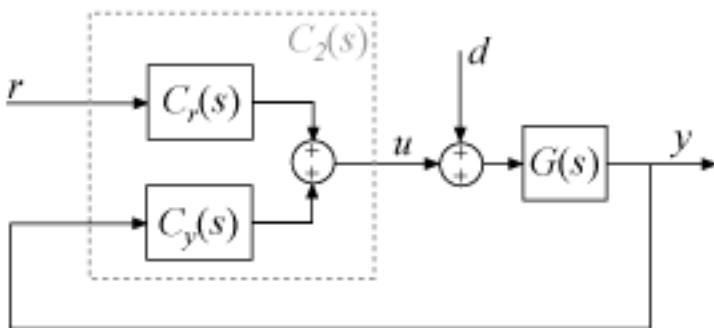
Continuous-time 2-DOF PID controller in parallel form.

Using the type string `'PID2'` causes `pidtune` to generate a 2-DOF controller, represented as a `pid2` object. The display confirms this result. The display also shows that `pidtune` tunes all controller coefficients, including the setpoint weights `b` and `c`, to balance performance and robustness.

To compute the closed-loop response, note that a 2-DOF PID controller is a 2-input, 1-output dynamic system. You can resolve the controller into two channels, one for the reference signal and one for the feedback signal, as shown in the diagram. (See "Continuous-Time 2-DOF PID Controller Representations" for more information.)

Decompose the controller into the components `Cr` and `Cy`, and use them to compute the closed-loop response from *r* to *y*.

```
C2tf = tf(C2);
Cr = C2tf(1);
Cy = C2tf(2);
T2 = Cr*feedback(G,Cy,+1);
```
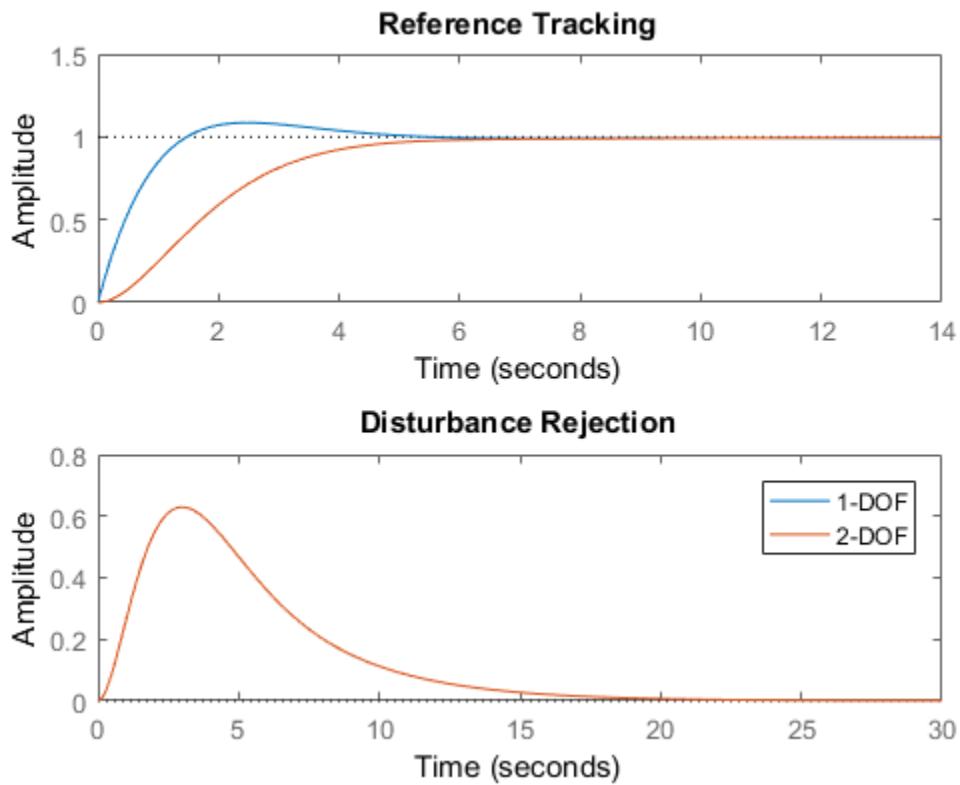
To examine the disturbance-rejection performance, compute the transfer function from *d* to *y*.

```
S2 = feedback(G,Cy,+1);
```

For comparison, design a 1-DOF PID controller with the same bandwidth and compute the corresponding transfer functions. Then compare the step responses.

11-17

```
C1 = pidtune(G,'PID',wc);
T1 = feedback(G*C1,1);
S1 = feedback(G,C1);

subplot(2,1,1)
stepplot(T1,T2)
title('Reference Tracking')
subplot(2,1,2)
stepplot(S1,S2)
title('Disturbance Rejection')
legend('1-DOF','2-DOF')
```


The plots show that adding the second degree of freedom eliminates the overshoot in the reference-tracking response without any cost to disturbance rejection. You can improve

disturbance rejection too using the `DesignFocus` option. This option causes `pidtune` to favor disturbance rejection over setpoint tracking.

```
opt = pidtuneOptions('DesignFocus','disturbance-rejection');
C2dr = pidtune(G,'PID2',wc,opt)


C2dr =

                        1
  u = Kp (b*r-y) + Ki --- (r-y) + Kd*s (c*r-y)
                        s

  with Kp = 1.72, Ki = 0.593, Kd = 1.25, b = 0, c = 0

Continuous-time 2-DOF PID controller in parallel form.
```
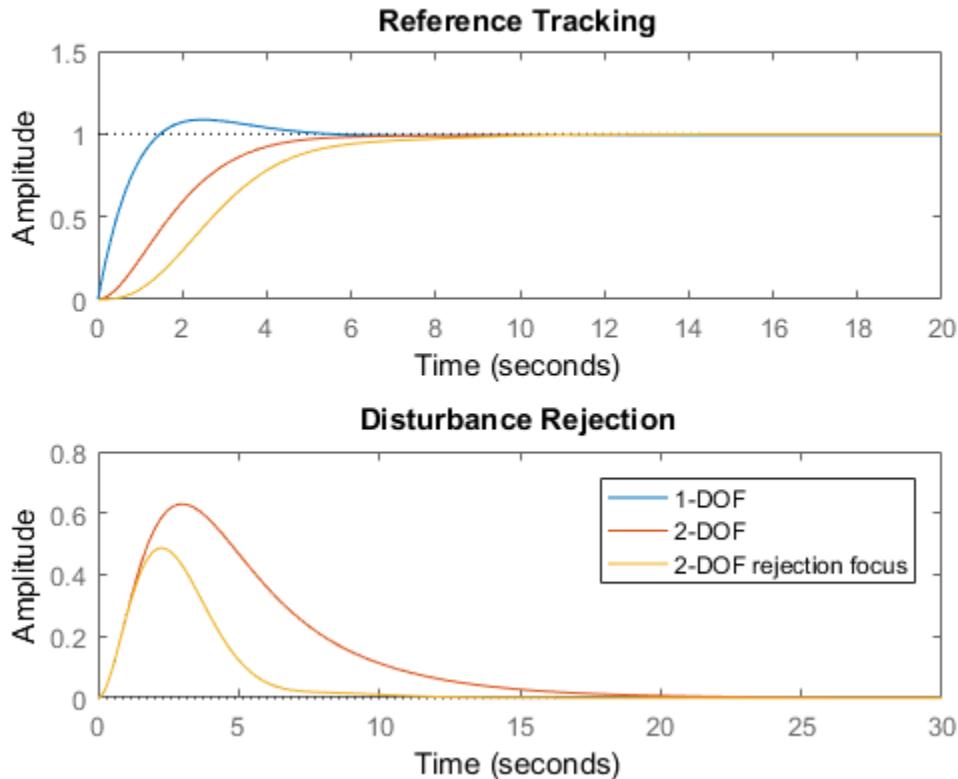
With the default balanced design focus, `pidtune` selects a `b` value between 0 and 1. For this plant, when you change design focus to favor disturbance rejection, `pidtune` sets `b` = 0 and `c` = 0. Thus, `pidtune` automatically generates an I-PD controller to optimize for disturbance rejection. (Explicitly specifying an I-PD controller without setting the design focus yields a similar controller.)

Compare the closed-loop responses with with the new controller.

```
C2dr_tf = tf(C2dr);
Cdr_r = C2dr_tf(1);
Cdr_y = C2dr_tf(2);
T2dr = Cdr_r*feedback(G,Cdr_y,+1);
S2dr = feedback(G,Cdr_y,+1);

subplot(2,1,1)
stepplot(T1,T2,T2dr)
title('Reference Tracking')
subplot(2,1,2)
stepplot(S1,S2,S2dr);
title('Disturbance Rejection')
legend('1-DOF','2-DOF','2-DOF rejection focus')
```

The plots show that the disturbance rejection is further improved compared to the balanced 2-DOF controller. This improvement comes with some sacrifice of reference-tracking performance, which is slightly slower. However, the reference-tracking response still has no overshoot.

Thus, using 2-DOF control can improve disturbance rejection without sacrificing as much reference tracking performance as 1-DOF control. These effects on system performance depend strongly on the properties of your plant. For some plants and some control

bandwidths, using 2-DOF control or changing the design focus has less or no impact on the tuned result.

See Also
pid2 | pidtune

Related Examples
- "Tune 2-DOF PID Controller (PID Tuner)" on page 11-22
- "Analyze Design in PID Tuner"
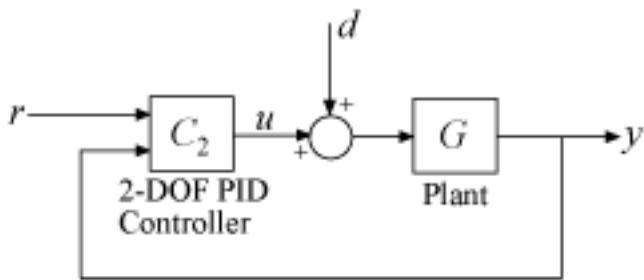
More About
- "Designing PID Controllers with PID Tuner"
- "Two-Degree-of-Freedom PID Controllers" on page 2-17

Tune 2-DOF PID Controller (PID Tuner)

This example shows how to design a two-degree-of-freedom (2-DOF) PID controller using PID Tuner. The example also compares the 2-DOF controller performance to the performance achieved with a 1-DOF PID controller.

In this example, you represent the plant as an "Numeric Models" on page 1-13. For information about using PID Tuner to tune a PID Controller (2DOF) block in a Simulink model, see "Design Two-Degree-of-Freedom PID Controllers" in the Simulink Control Design documentation.
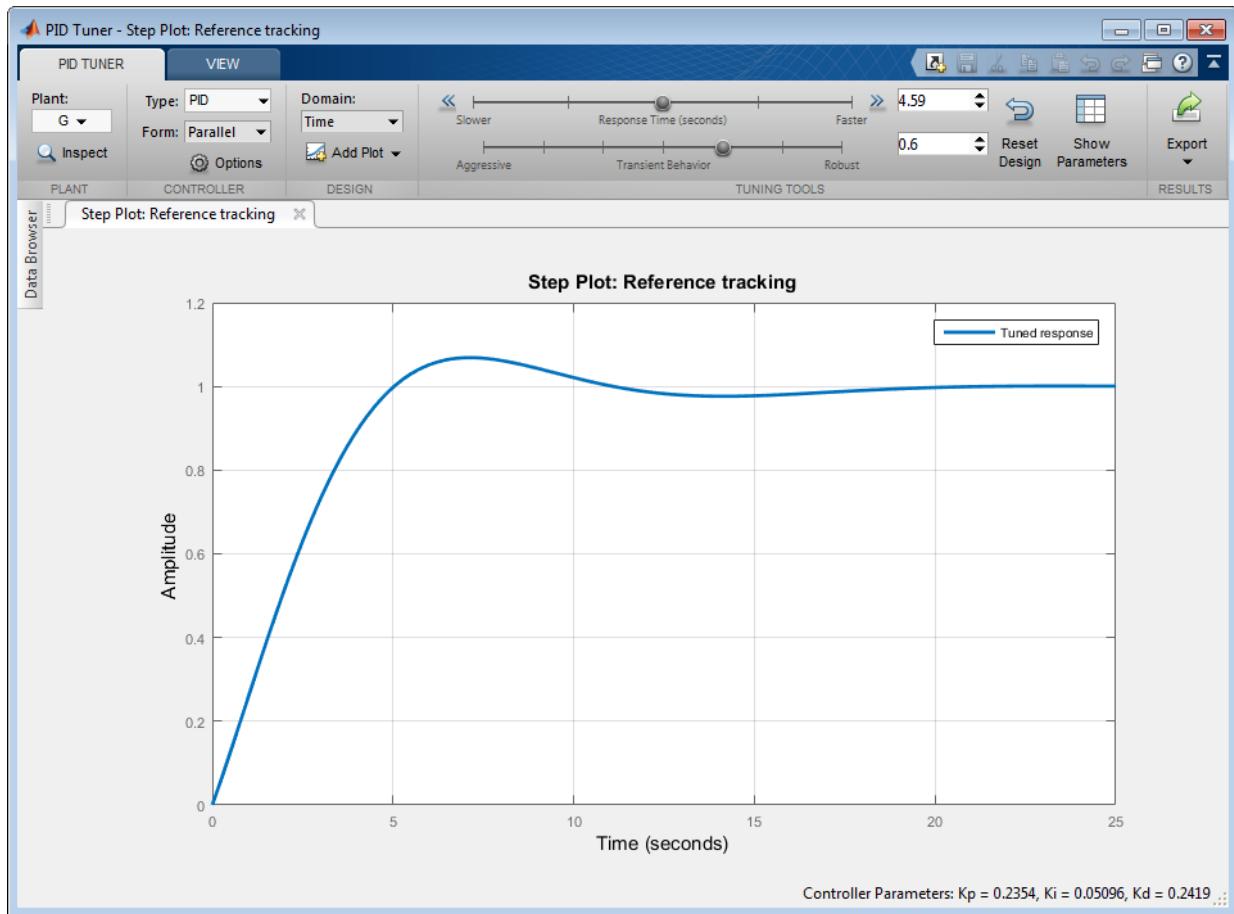
2-DOF PID controllers include setpoint weighting on the proportional and derivative terms. Compared to a 1-DOF PID controller, a 2-DOF PID controller can achieve better disturbance rejection without significant increase of overshoot in setpoint tracking. A typical control architecture using a 2-DOF PID controller is shown in the following diagram.
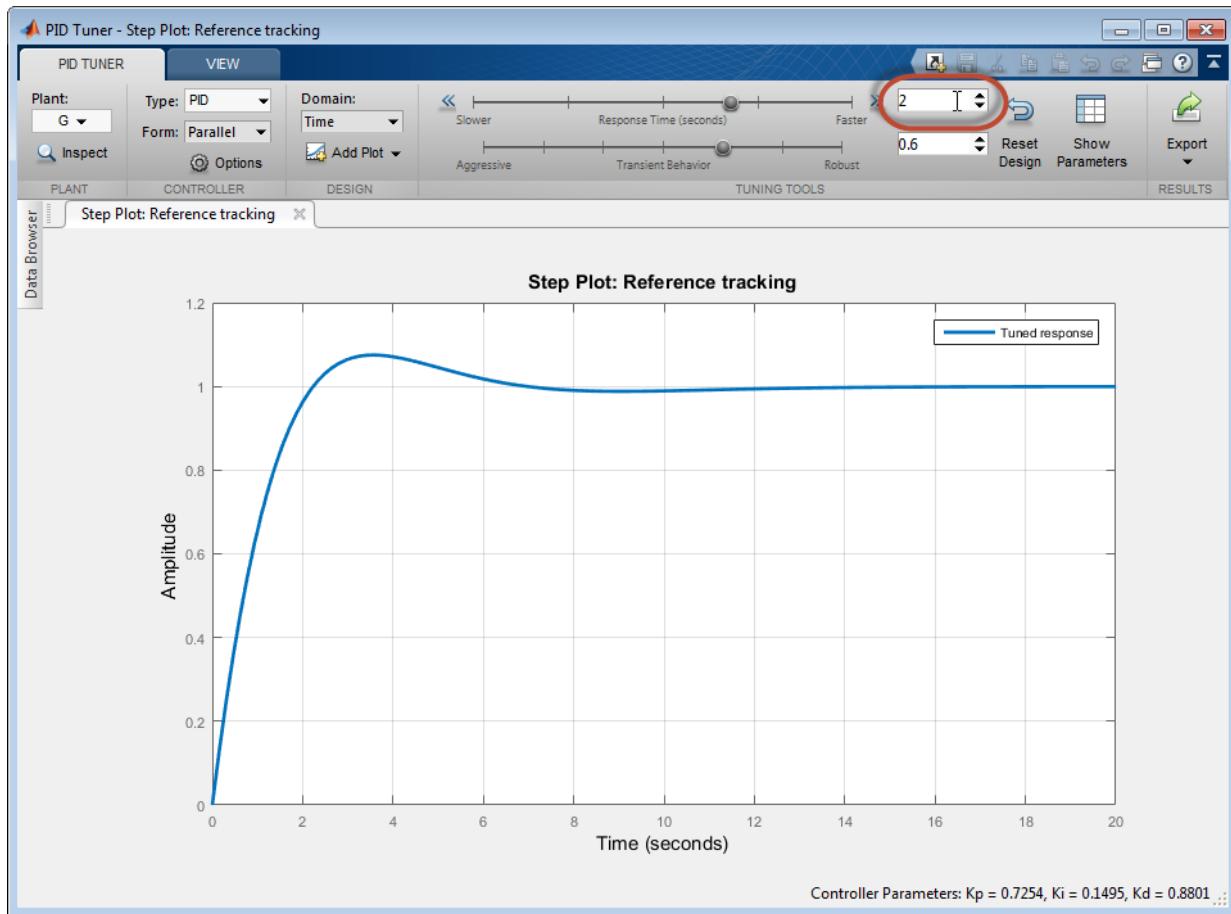
For this example, first design a 1-DOF controller for the plant given by:

$$G(s) = \frac{1}{s^2 + 0.5s + 0.1}.$$
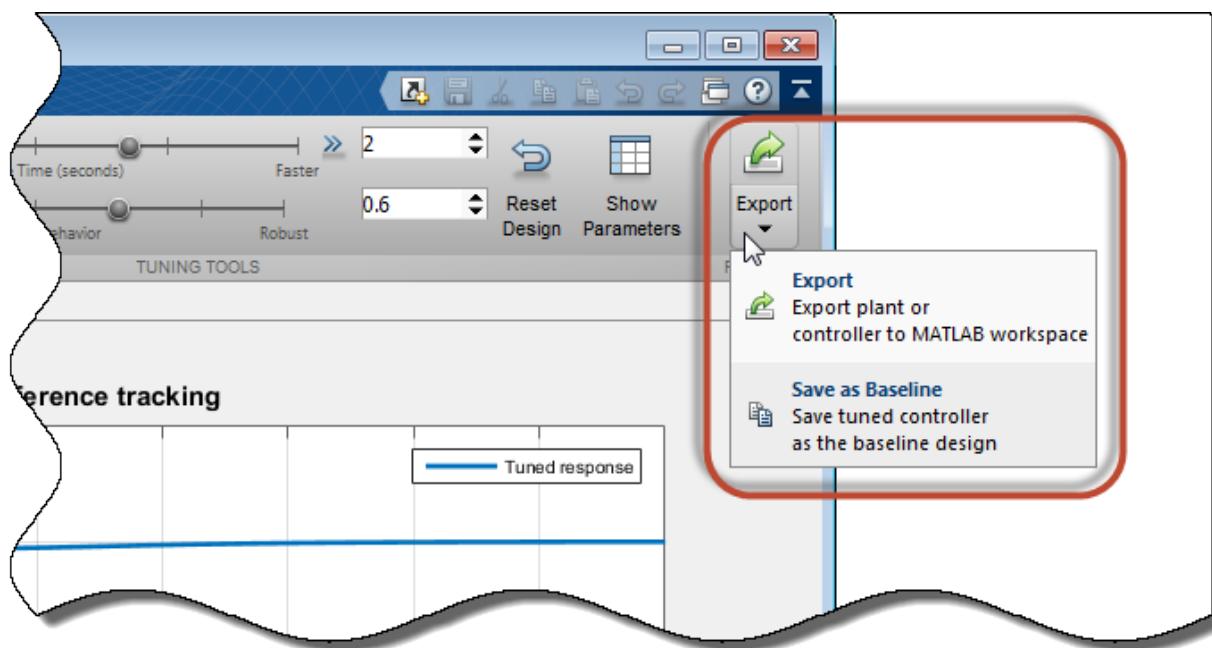
```
G = tf(1,[1 0.5 0.1]);
pidTuner(G,'PID')
```
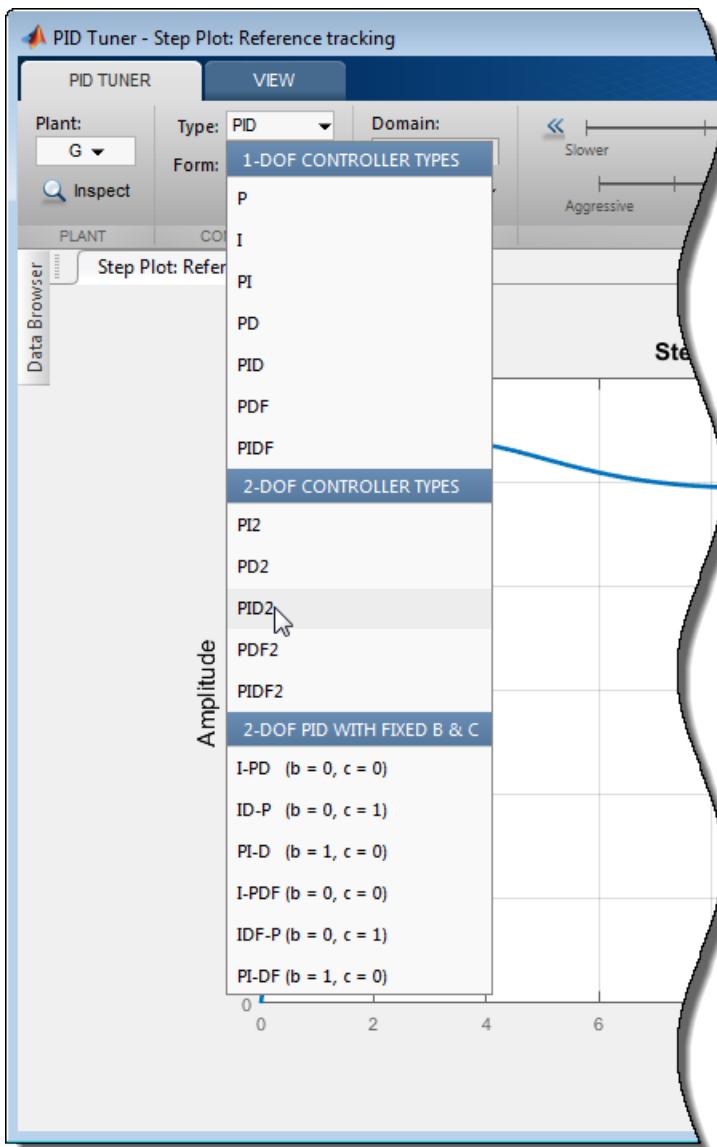
Suppose for this example that your application requires a faster response than the PID Tuner initial design. In the text box next to the **Response Time** slider, enter 2.

The resulting response is fast, but has a considerable amount of overshoot. Design a 2-DOF controller to improve the overshoot. First, set the 1-DOF controller as the baseline controller for comparison. Click the **Export** arrow ▼ and select `Save as Baseline`.
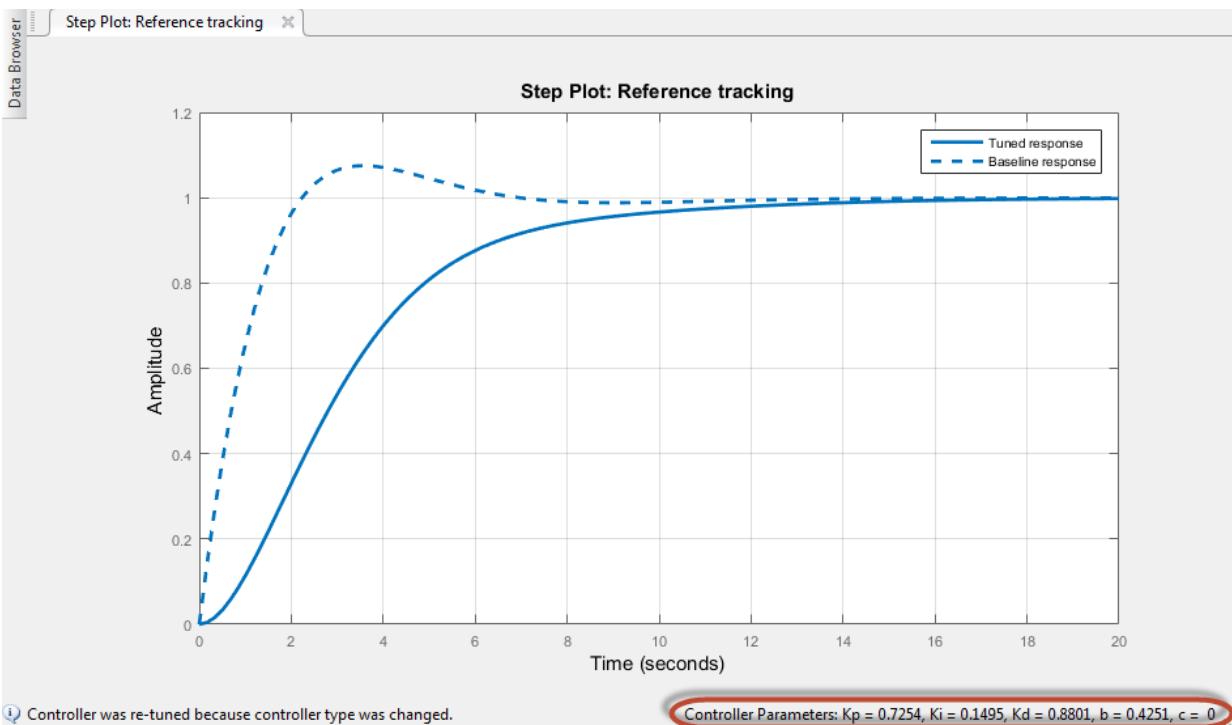
Design the 2-DOF controller. In the **Type** menu, select PID2.
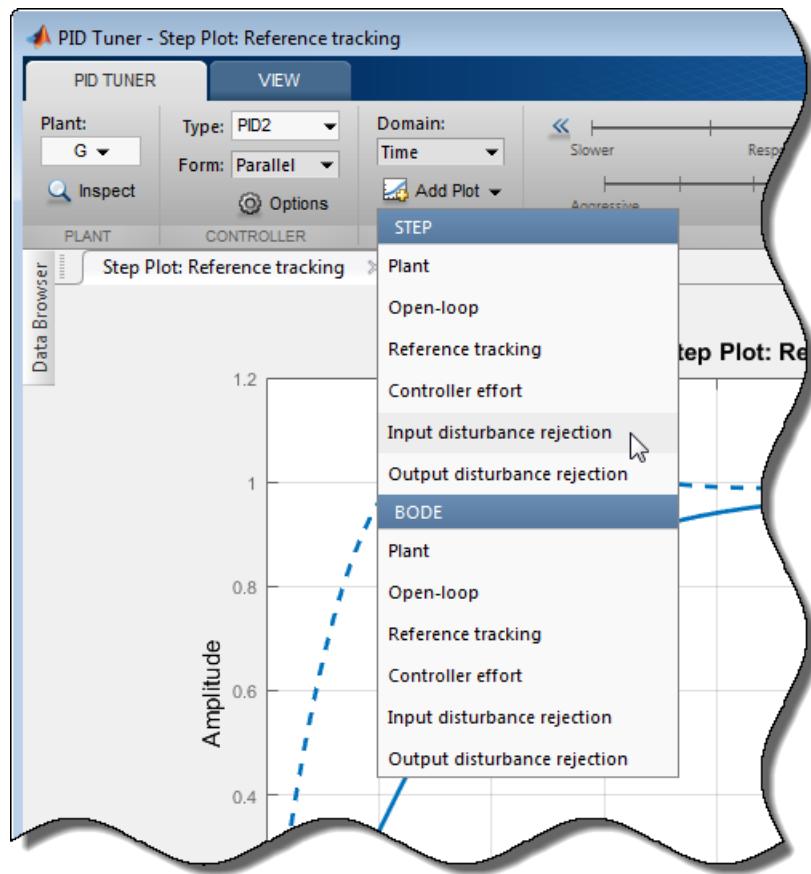
PID Tuner generates a 2-DOF controller with the same target response time. The controller parameters displayed at the bottom right show that PID Tuner tunes all
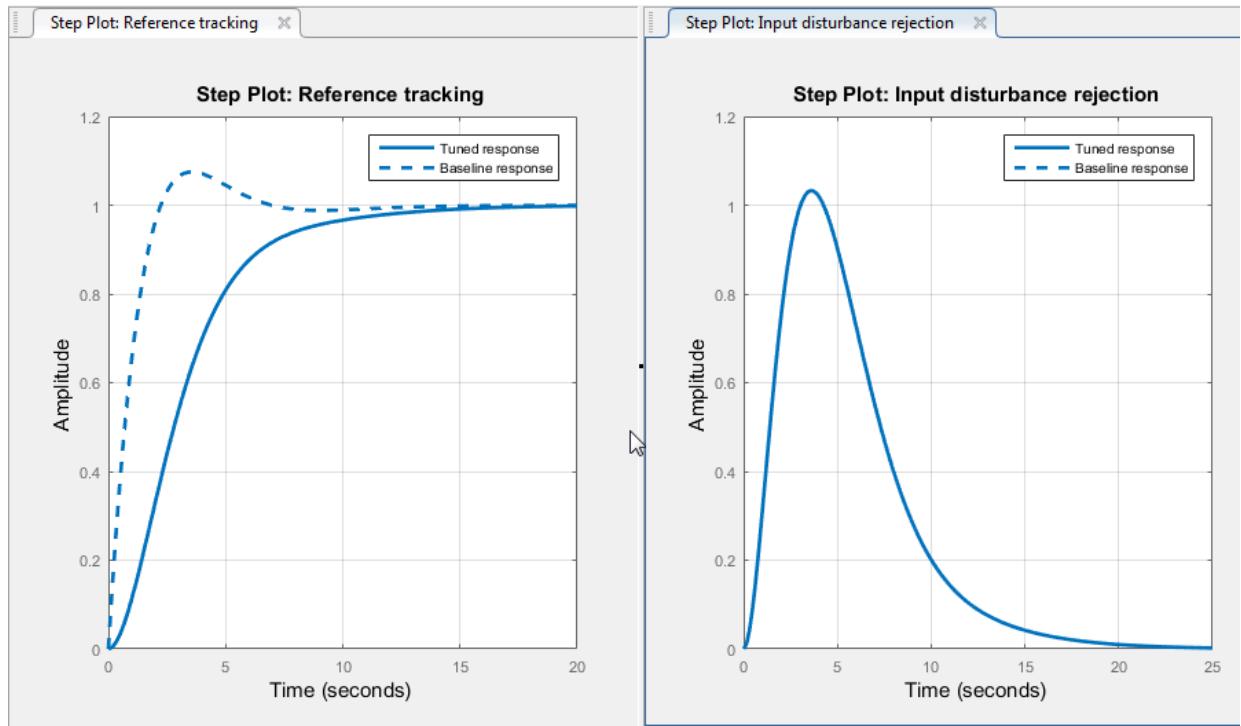
controller coefficients, including the setpoint weights b and c, to balance performance and robustness. Compare the 2-DOF controller performance (solid line) with the performance of the 1-DOF controller that you stored as the baseline (dotted line).

Adding the second degree of freedom eliminates the overshoot in the reference tracking response. Next, add a step response plot to compare the disturbance rejection performance of the two controllers. Select **Add Plot** > **Input Disturbance Rejection**.
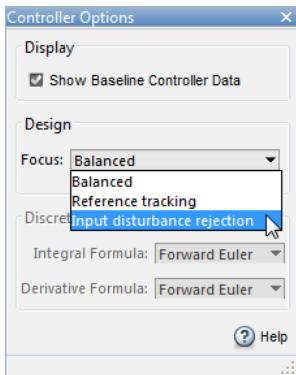
PID Tuner tiles the disturbance-rejection plot side by side with the reference-tracking plot.
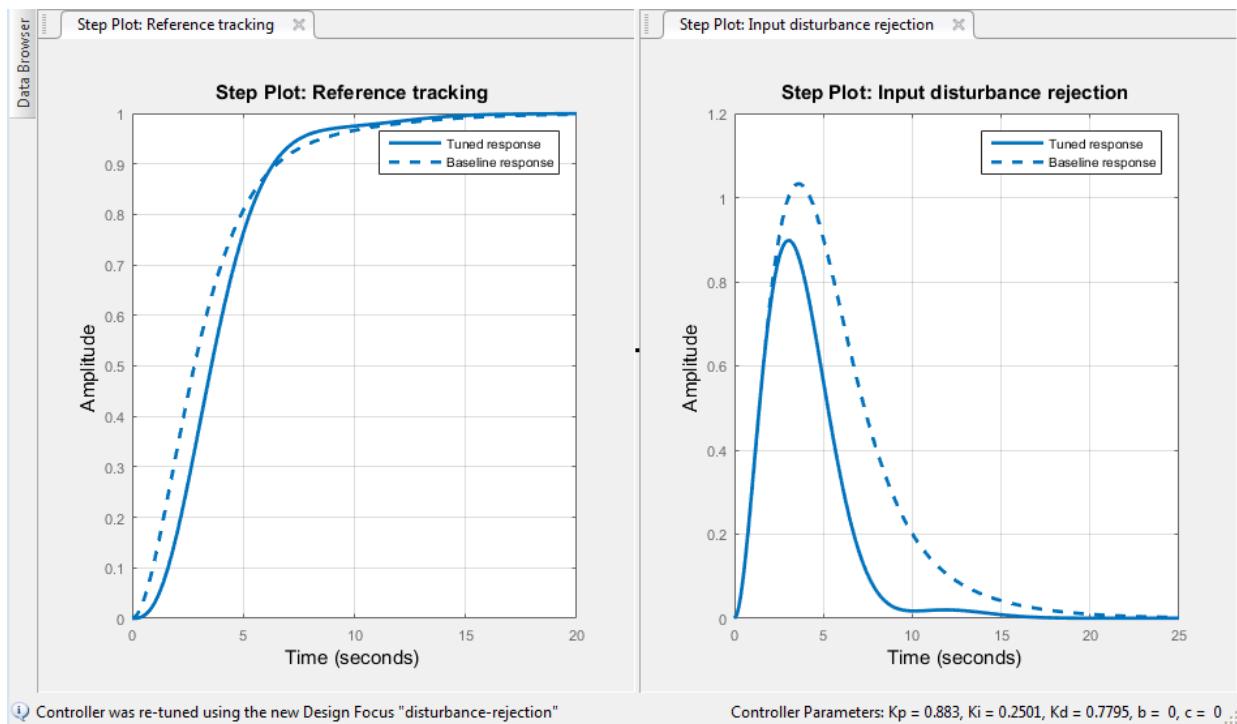
The disturbance-rejection performance is identical with both controllers. Thus, using a 2-DOF controller eliminates reference-tracking overshoot without any cost to disturbance rejection.

You can improve disturbance rejection too by changing the PID Tuner design focus. First, click the **Export** arrow ▼ and select Save as Baseline again to set the 2-DOF controller as the baseline for comparison.

Change the PID Tuner design focus to favor reference tracking without changing the response time or the transient-behavior coefficient. To do so, click ⚙ **Options**, and in the **Focus** menu, select Input disturbance rejection.

PID Tuner automatically retunes the controller coefficients with a focus on disturbance-rejection performance.

With the default balanced design focus, PID Tuner selects a b value between 0 and 1. For this plant, when you change design focus to favor disturbance rejection, PID Tuner sets b = 0 and c = 0. Thus, PID Tuner automatically generates an I-PD controller to optimize for disturbance rejection. (Explicitly specifying an I-PD controller without setting the design focus yields a similar controller.)

The response plots show that with the change in design focus, the disturbance rejection is further improved compared to the balanced 2-DOF controller. This improvement comes with some sacrifice of reference-tracking performance, which is slightly slower. However, the reference-tracking response still has no overshoot.

Thus, using 2-DOF control can improve disturbance rejection without sacrificing as much reference tracking performance as 1-DOF control. These effects on system performance depend strongly on the properties of your plant and the speed of your controller. For some plants and some control bandwidths, using 2-DOF control or changing the design focus has less or no impact on the tuned result.

See Also
pidTuner

Related Examples
- "Tune 2-DOF PID Controller (Command Line)" on page 11-16
- "Analyze Design in PID Tuner"

More About
- "Designing PID Controllers with PID Tuner"
- "Two-Degree-of-Freedom PID Controllers" on page 2-17

PID Controller Types for Tuning

In this section...
"Specifying PID Controller Type" on page 11-32
"1-DOF Controllers" on page 11-34
"2-DOF Controllers" on page 11-34
"2-DOF Controllers with Fixed Setpoint Weights" on page 11-35
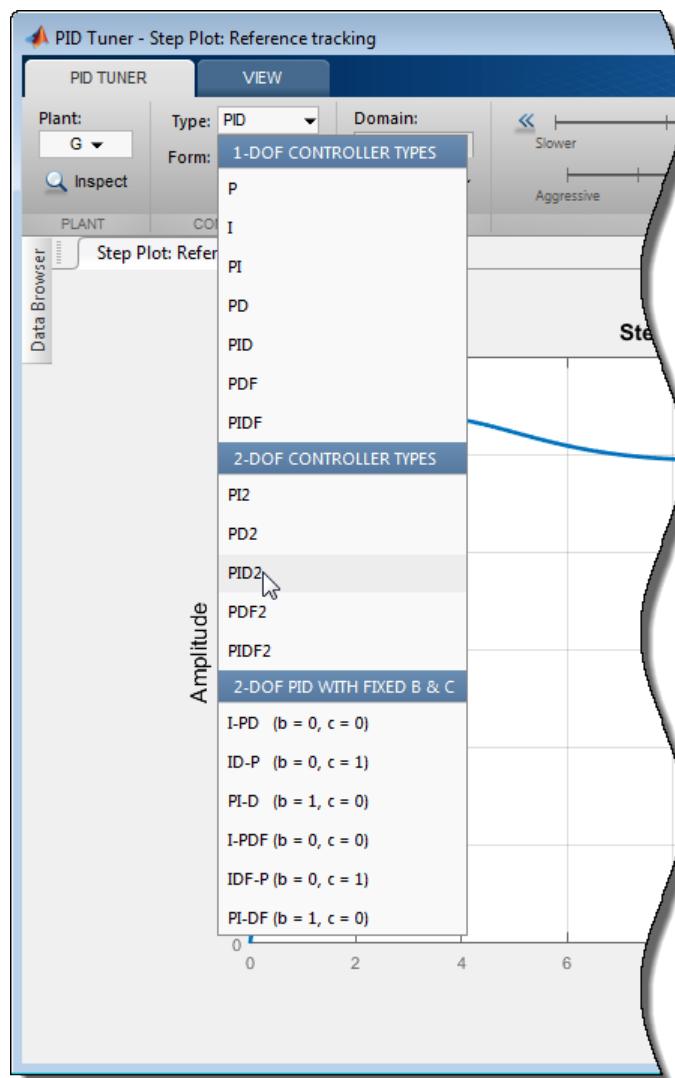
PID Tuner and the `pidtune` command can tune many PID and 2-DOF PID controller types. The term *controller type* refers to which terms are present in the controller action. For example, a PI controller has only a proportional and an integral term, while a PIDF controller contains proportional, integrator, and filtered derivative terms. This topic summarizes the types of PID controllers available for tuning with PID Tuner and `pidtune`.

Specifying PID Controller Type

To select the controller type, use one of these methods:

- For command-line tuning, provide the `type` argument to the `pidtune` command. For example, `C = pidtune(G,'PI')` tunes a PI controller for plant `G`.

- For tuning in PID Tuner:

  - Provide the `type` argument to the `pidTuner` command when you open PID Tuner. For example, `pidTuner(G,'PIDF2')` opens PID Tuner with an initial design that is a 2-DOF PID controller with a filter on the derivative term.

  - Provide the baseline-controller `Cbase` argument to the `pidTuner` command when you open PID Tuner. PID Tuner designs a controller of the same type as `Cbase`. For example, suppose `C0` is a `pid` controller object that has proportional and derivative action only (PD controller). Then, `pidTuner(G,C0)` opens PID Tuner with an initial design that is a PD controller.

  - In PID Tuner, use the **Type** menu to change controller types.

1-DOF Controllers

The following table summarizes the available 1-DOF PID controller types and provides representative controller formulas for parallel form. The standard-form and discrete-time formulas are analogous.

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
P	Proportional only	K_p	K_p
I	Integral only	$\dfrac{K_i}{s}$	$K_i \dfrac{T_s}{z-1}$
PI	Proportional and integral	$K_p + \dfrac{K_i}{s}$	$K_p + K_i \dfrac{T_s}{z-1}$
PD	Proportional and derivative	$K_p + K_d s$	$K_p + K_d \dfrac{z-1}{T_s}$
PDF	Proportional and derivative with first-order filter on derivative term	$K_p + \dfrac{K_d s}{T_f s + 1}$	$K_p + K_d \dfrac{1}{T_f + \dfrac{T_s}{z-1}}$
PID	Proportional, integral, and derivative	$K_p + \dfrac{K_i}{s} + K_d s$	$K_p + K_i \dfrac{T_s}{z-1} + K_d \dfrac{z-1}{T_s}$
PIDF	Proportional, integral, and derivative with first-order filter on derivative term	$K_p + \dfrac{K_i}{s} + \dfrac{K_d s}{T_f s + 1}$	$K_p + K_i \dfrac{T_s}{z-1} + K_d \dfrac{1}{T_f + \dfrac{T_s}{z-1}}$

2-DOF Controllers

PID Tuner can automatically design 2-DOF PID controller types with free setpoint weights. The following table summarizes the 2-DOF controller types in PID Tuner. The standard-form and discrete-time formulas are analogous. For more information about 2-

DOF PID controllers generally, see "Two-Degree-of-Freedom PID Controllers" on page 2-17.

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
PI2	2-DOF proportional and integral	$u = K_p (br - y) + \dfrac{K_i}{s}($	$u = K_p (br - y) + K_i \dfrac{T_s}{z-1}(r - y)$
PD2	2-DOF proportional and derivative	$u = K_p (br - y) + K_d s$	$u = K_p (br - y) + K_d \dfrac{z-1}{T_s}(cr - y)$
PDF2	2-DOF proportional and derivative with first-order filter on derivative term	$u = K_p (br - y) + K_d$	$u = K_p (br - y) + K_d \dfrac{1}{T_f + \dfrac{T_s}{z-1}}(cr$
PID2	2-DOF proportional, integral, and derivative	$u = K_p (br - y) + \dfrac{K_i}{s}($	$u = K_p (br - y) + K_i \dfrac{T_s}{z-1}(r - y) + k$
PIDF2	2-DOF proportional, integral, and derivative with first-order filter on derivative term	$u = K_p (br - y) + \dfrac{K_i}{s}($	$u = K_p (br - y) + K_i \dfrac{T_s}{z-1}(r - y) + k$

2-DOF Controllers with Fixed Setpoint Weights

With PID control, step changes in the reference signal can cause spikes in the control signal contributed by the proportional and derivative terms. By fixing the setpoint weights of a 2-DOF controller, you can mitigate the influence on the control signal exerted by changes in the reference signal. For example, consider the relationship between the inputs r (setpoint) and y (feedback) and the output u (control signal) of a continuous-time 2-DOF PID controller.

$$u = K_p (br - y) + \frac{K_i}{s}(r - y) + K_d s(cr - y)$$

If you set $b = 0$ and $c = 0$, then changes in the setpoint r do not feed through directly to either the proportional or the derivative terms in u. The $b = 0$, $c = 0$ controller is called an I-PD type controller. I-PD controllers are also useful for improving disturbance rejection.

Use PID Tuner to design the fixed-setpoint-weight controller types summarized in the following table. The standard-form and discrete-time formulas are analogous.

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
I-PD	2-DOF PID with $b = 0$, $c = 0$	$u = -K_p y + \dfrac{K_i}{s}(r$	$u = -K_p y + K_i \dfrac{T_s}{z-1}(r-y) - K$
I-PDF	2-DOF PIDF with $b = 0$, $c = 0$	$u = -K_p y + \dfrac{K_i}{s}(r$	$u = -K_p y + K_i \dfrac{T_s}{z-1}(r-y) - K$
ID-P	2-DOF PID with $b = 0$, $c = 1$	$u = -K_p y + \dfrac{K_i}{s}(r$	$u = -K_p y + K_i \dfrac{T_s}{z-1}(r-y) + K$
IDF-P	2-DOF PIDF with $b = 0$, $c = 1$	$u = -K_p y + \dfrac{K_i}{s}(r$	$u = -K_p y + K_i \dfrac{T_s}{z-1}(r-y) + K$
PI-D	2-DOF PID with $b = 1$, $c = 0$	$u = K_p (r-y) + \dfrac{K}{s}$	$u = K_p (r-y) + K_i \dfrac{T_s}{z-1}(r-y)$
PI-DF	2-DOF PIDF with $b = 1$, $c = 0$	$u = K_p (r-y) + \dfrac{K}{s}$	$u = K_p (r-y) + K_i \dfrac{T_s}{z-1}(r-y)$

See Also
pidtune | pidTuner

Related Examples

More About

12

Classical Control Design

Choosing a Control Design Approach

Control System Toolbox provides several approaches to tuning control systems. Use the following table to determine which approach best supports what you want to do.

	PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
Architecture	PID loops with unit feedback (1-DOF and 2-DOF)	Control systems having SISO controllers in common single-loop or multiloop configurations (see "Feedback Control Architectures" on page 12-20)	Any architecture, including any number of SISO or MIMO feedback loops
Control Design Approach	Automatically tune PID gains to balance performance and robustness	• Graphically tune poles and zeros on design plots, such as Bode, root locus, and Nichols • Automatically tune compensators using response optimization (Simulink Design Optimization™), LQG synthesis, or IMC tuning	Automatically tune controller parameters to meet design requirements you specify, such as setpoint tracking, stability margins, disturbance rejection, and loop shaping (see "Tuning Goals")
Analysis of Control System Performance	Time and frequency responses for reference tracking and disturbance rejection	Any combination of system responses	Any combination of system responses
Interface	• Graphical tuning using **PID**	Graphical tuning using Control System Designer	• Graphical tuning using Control System Tuner

	PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
	Tuner (see "Designing PID Controllers with PID Tuner") • Programmatic tuning using `pidtune` (see "PID Controller Design at the Command Line" on page 11-2)		• Programmatic tuning using `systune` (see "Programmatic Tuning")

More About

- "PID Controller Tuning"
- "Classical Control Design"
- "Tuning with Control System Tuner"
- "Programmatic Tuning"

Control System Designer Tuning Methods

Using **Control System Designer**, you can tune compensators using various graphical and automated tuning methods.

In this section...
"Graphical Tuning Methods" on page 12-4
"Automated Tuning Methods" on page 12-5
"Effective Plant for Tuning" on page 12-6
"Select a Tuning Method" on page 12-7

Graphical Tuning Methods

Use graphical tuning methods to interactively add, modify, and remove controller poles, zeros, and gains.

Tuning Method	Description	Useful For
Bode Editor	Tune your compensator to achieve a specific open-loop frequency response (loop shaping).	Adjusting open-loop bandwidth and designing to gain and phase margin specifications.
Closed-Loop Bode Editor	Tune your prefilter to improve closed-loop system response.	Improving reference tracking, input disturbance rejection, and noise rejection.
Root Locus Editor	Tune your compensator to produce closed-loop pole locations that satisfy your design specifications.	Designing to time-domain design specifications, such as maximum overshoot and settling time.
Nichols Editor	Tune your compensator to achieve a specific open-loop response (loop shaping), combining gain and phase information on a Nichols plot.	Adjusting open-loop bandwidth and designing to gain and phase margin specifications.

When using graphical tuning, you can modify the compensator either directly from the editor plots or using the compensator editor. A common design approach is to roughly tune your compensator using the editor plots, and then use the compensator editor to fine-tune the compensator parameters. For more information, see "Edit Compensator Dynamics" on page 12-90

The graphical tuning methods are not mutually exclusive. For example, you can tune your compensator using both the Bode editor and root locus editor simultaneously. This option is useful when designing to both time-domain and frequency-domain specifications.

For examples of graphical tuning, see the following:

- "Bode Diagram Design" on page 12-48
- "Root Locus Design" on page 12-62
- "Nichols Plot Design" on page 12-77

Automated Tuning Methods

Use automated tuning methods to automatically tune compensators based on your design specifications.

Tuning Method	Description	Requirements and Limitations
PID Tuning	Automatically tune PID gains to balance performance and robustness or tune controllers using classical PID tuning formulas.	Classical PID tuning formulas require a stable or integrating effective plant.
Optimization Based Tuning	Optimize compensator parameters using design requirements specified in graphical tuning and analysis plots.	Requires Simulink Design Optimization software. Tunes the parameters of a previously defined controller structure.
LQG Synthesis	Design a full-order stabilizing feedback controller as a linear-quadratic-Gaussian (LQG) tracker.	Maximum controller order depends on the effective plant dynamics.
Loop Shaping	Find a full-order stabilizing feedback controller with a specified open-loop bandwidth or shape.	Requires Robust Control Toolbox software. Maximum controller order depends on the effective plant dynamics.
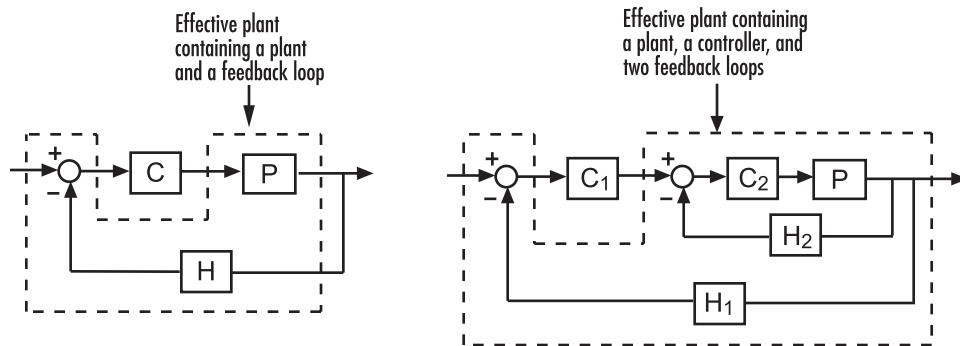
Tuning Method	Description	Requirements and Limitations
Internal Model Control (IMC) Tuning	Obtain a full-order stabilizing feedback controller using the IMC design method.	Assumes that your control system uses an IMC architecture that contains a predictive model of your plant dynamics. Maximum controller order depends on the effective plant dynamics.

A common design approach is to generate an initial compensator using PID tuning, LQG synthesis, loop shaping, or IMC tuning. You can then improve the compensator performance using either optimization-based tuning or graphical tuning.

For more information on automated tuning methods, see "Design Compensator Using Automated Tuning Methods" on page 12-96.

Effective Plant for Tuning

An *effective plant* is the system controlled by a compensator that contains all elements of the open loop in your model other than the compensator you are tuning. The following diagrams show examples of effective plants:
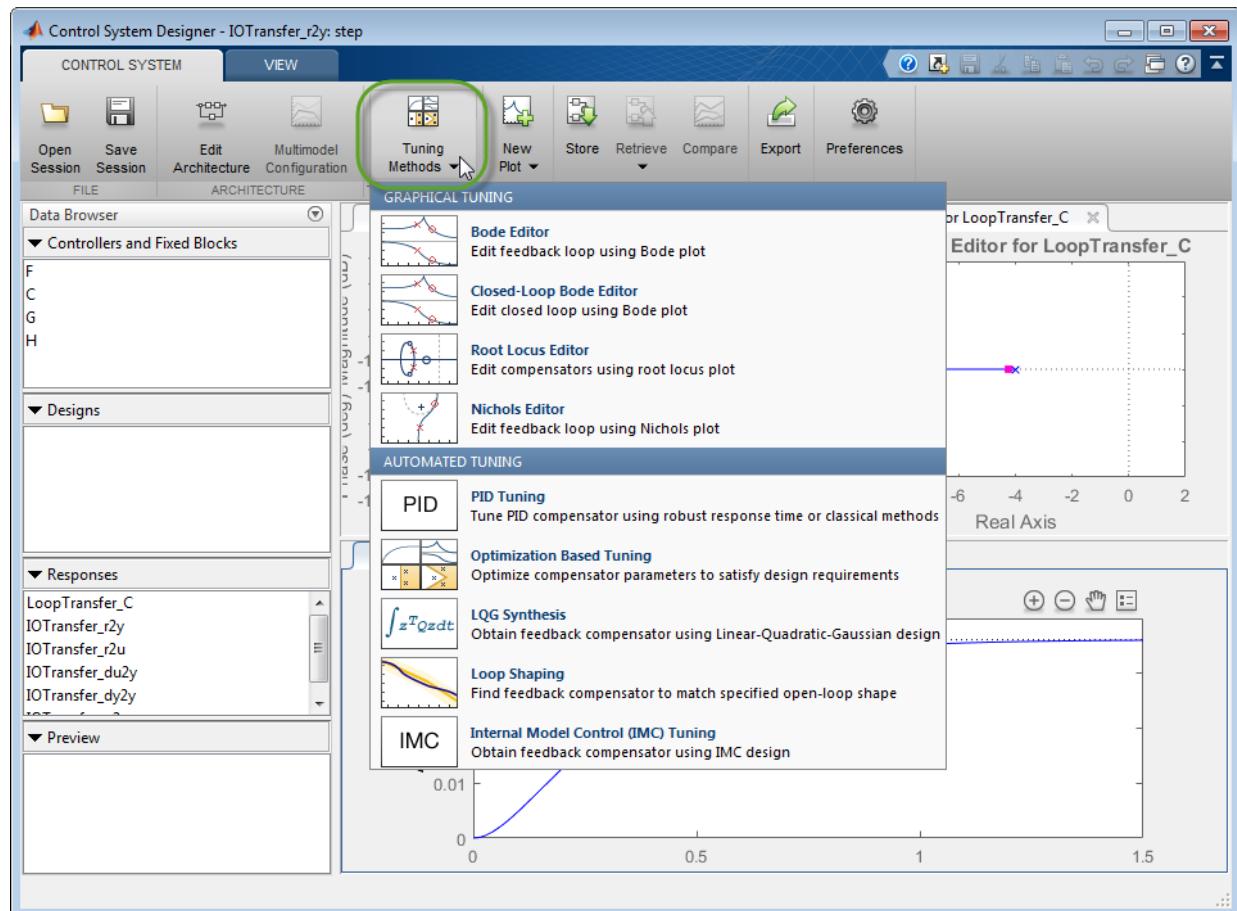
Knowing the properties of the effective plant seen by your compensator can help you understand which tuning methods work for your system. For example, some automated

tuning methods apply only to compensators whose open loops ($L = C\hat{P}$) have stable

effective plants ($\hat{P}$). Also, for tuning methods such as IMC and loop shaping, the maximum controller order depends on the dynamics of the effective plant.

Select a Tuning Method

To select a tuning method, in **Control System Designer**, click **Tuning Methods**.

See Also
Control System Designer
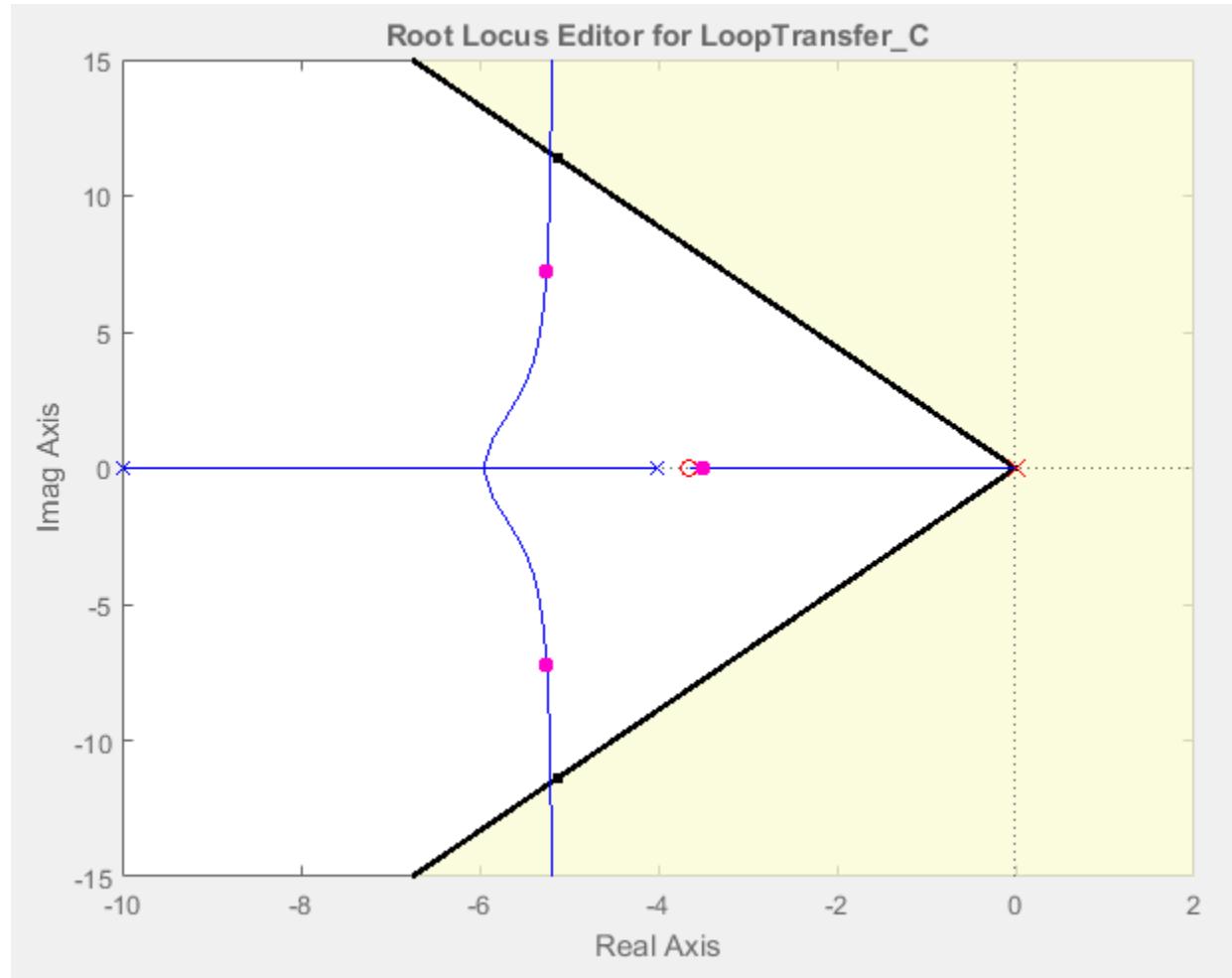
Related Examples

- "Bode Diagram Design" on page 12-48
- "Root Locus Design" on page 12-62
- "Nichols Plot Design" on page 12-77
- "Design Compensator Using Automated Tuning Methods" on page 12-96

Design Requirements

This topic describes time-domain and frequency-domain design requirements available in **Control System Designer**. Each requirement defines an exclusion region, indicated by a yellow shaded area. To satisfy a requirement, a response plot must remain outside of the associated exclusion region.

If you have Simulink Design Optimization software installed, you can use response optimization techniques to find a compensator that meets your specified design requirements. For examples of optimization-based control design using design requirements, see "Optimize LTI System to Meet Frequency-Domain Requirements" and "Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)".

For other **Control System Designer** tuning methods, you can use the specified design requirements as visual guidelines during the tuning process.

Add Design Requirements
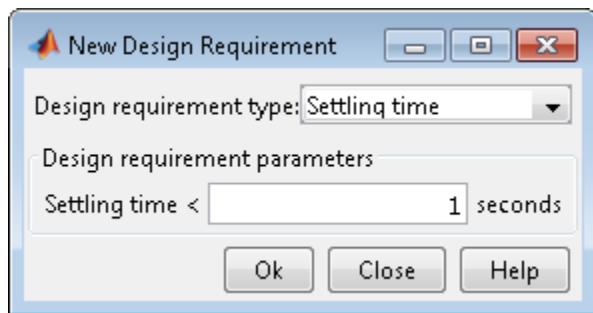
You can add design requirements either directly to existing plots or, when using optimization-based tuning, from the Response Optimization dialog box.

Add Requirements to Existing Plots

You can add design requirements directly to existing:

- Bode, root locus, and Nichols editor plots.
- Analysis plots:

  - Root locus plots and pole/zero maps
  - Bode diagrams
  - Nichols plots
  - Step and impulse responses

To add a design requirement to a plot, in **Control System Designer**, right-click the plot, and select **Design Requirements** > **New**.

In the New Design Requirement dialog box, in the **Design requirement type** drop-down list, select the type of requirement to add. You can select any valid requirement for the associated plot type.
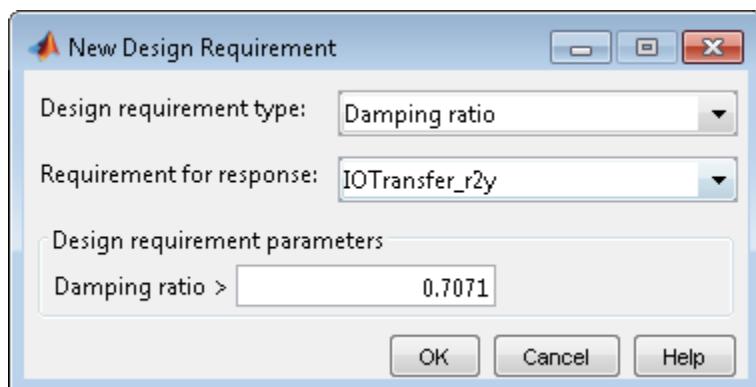
In the **Design requirement parameters** section, configure the requirement properties. Parameters are dependent on the type of requirement you select.

To create the specified requirement and add it to the plot, click **OK**.

Add Requirements from Response Optimization Dialog Box

When using optimization-based tuning, you can add design requirements from the Response Optimization dialog box.

To do so, click **Add new design requirement**.

In the New Design Requirement dialog box, select a **Design requirement type** from the drop-down list.

In the **Requirement for response** drop-down list, specify the response to which to apply the design requirement. You can select any response in **Data Browser**.

In the **Design requirement parameters** section, configure the requirement properties. Parameters are dependent on the type of requirement you select.

To create the specified design requirement, click **OK**. In the Response Optimization dialog box, on the **Design Requirements** tab, the new requirement is added to the table.

The app also adds the design requirement to a corresponding editor or analysis plot. The plot type used depends on the selected design requirement type.
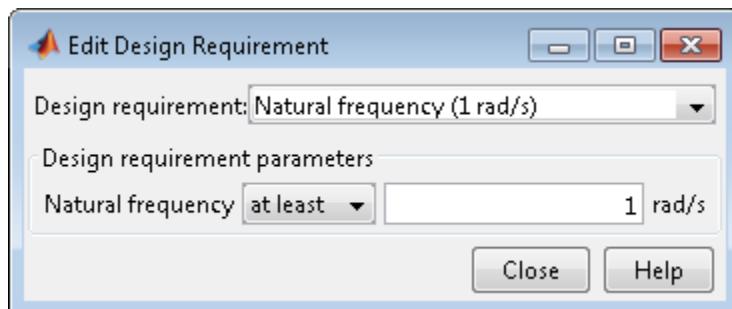
If the requirement is for a Bode, root locus, or Nichols plot and:

- A corresponding editor plot is open, the requirement is added to that plot.
- Only a corresponding analysis plot is open, the requirement is added to that plot.
- No corresponding plot is open, the requirement is added to a new Editor plot.

Otherwise, if the requirement is for a different plot type, the requirement is added to an appropriate analysis plot. For example, a `Step requirement bound` is added to a new step analysis plot.

Edit Design Requirements
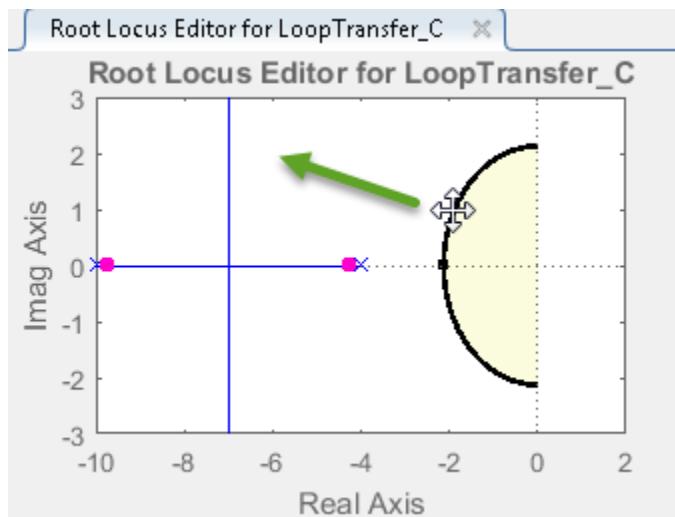
To edit an existing requirement, in **Control System Designer**, right-click the corresponding plot, and select **Design Requirements** > **Edit**.

In the Edit Design Requirement dialog box, in the **Design requirement** drop-down list, select a design requirement to edit. You can select any existing design requirement from the current plot.

In the **Design requirement parameters** section, specify the requirement properties. Parameters are dependent on the type of requirement you select. When you change a parameter, the app automatically updates the requirement display in the associated plot.

You can also interactively adjust design requirements by dragging the edges or vertices of the shaded exclusion region in the associated plot.

Root Locus and Pole-Zero Plot Requirements

Settling Time

Specifying a settling time for a continuous-time system adds a vertical boundary line to the root locus or pole-zero plot. This line represents pole locations associated with the specified settling time. This boundary is exact for a second-order system with no zeros. For higher order systems, the boundary is an approximation based on second-order dominant systems.

To satisfy this requirement, your system poles must be to the left of the boundary line.

For a discrete-time system, the design requirement boundary is a curved line centered on the origin. In this case, your system poles must be within the boundary line to satisfy the requirement.

Percent Overshoot

Specifying percent overshoot for a continuous-time system adds two rays to the plot that start at the origin. These rays are the locus of poles associated with the specified overshoot value. In the discrete-time case, the design requirement adds two curves originating at (1,0) and meeting on the real axis in the left-hand plane.

Note: The percent overshoot (p.o.) design requirement can be expressed in terms of the damping ratio, ζ:

$$p.o. = 100 \exp\left(-\frac{\pi\zeta}{\sqrt{1-\zeta^2}} \right)$$

Damping Ratio

Specifying a damping ratio for a continuous-time system adds two rays to the plot that start at the origin. These rays are the locus of poles associated with the specified overshoot value. This boundary is exact for a second-order system and, for higher order systems, is an approximation based on second-order dominant systems.

To meet this requirement, your system poles must be to the left of the boundary lines.

For discrete-time systems, the design requirement adds two curves originating at (1,0) and meeting on the real axis in the left-hand plane. In this case, your system poles must be within the boundary curves.

Natural Frequency

Specifying a natural frequency bound adds a semicircle to the plot that is centered around the origin. The radius of the semicircle equals the natural frequency.

If you specify a natural frequency lower bound, the system poles must remain outside this semicircle. If you specify a natural frequency upper bound, the system poles must remain within this semicircle.

Region Constraint
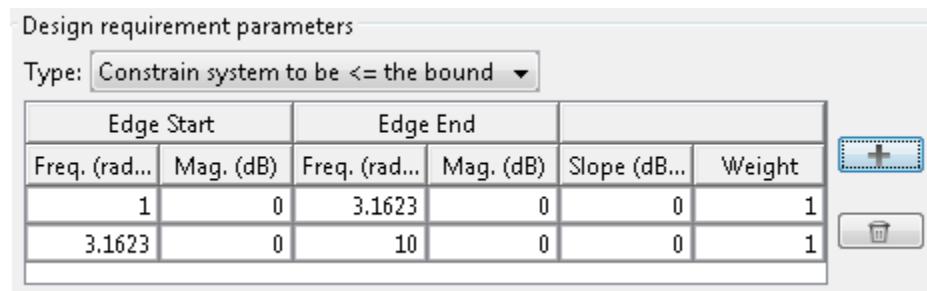
To specify a region constraint, define two or more vertices of a piece-wise linear boundary line. For each vertex, specify **Real** and **Imaginary** components. This requirement adds a shaded exclusion region on one side of the boundary line. To switch the exclusion region to the opposite side of the boundary, in the response plot, right-click the requirement, and select **Flip**.

To satisfy this requirement, your system poles must be outside of the exclusion region.

Open-Loop and Closed-Loop Bode Diagram Requirements

Upper Gain Limit

You can specify upper gain limits for both open-loop and closed-loop Bode responses.

Design requirement parameters

Type: Constrain system to be <= the bound ▼

Edge Start		Edge End			
Freq. (rad...	Mag. (dB)	Freq. (rad...	Mag. (dB)	Slope (dB...	Weight
1	0	3.1623	0	0	1
3.1623	0	10	0	0	1

A gain limit consists of one or more line segments. For the start and end points of each segment, specify a frequency, **Freq**, and magnitude, **Mag**. You can also specify the slope of the line segment in dB/decade. When you change the slope, the magnitude for the end point updates.

If you are using optimization-based tuning, you can assign a tuning **Weight** to each segment to indicate their relative importance.

In the **Type** drop-down list you can select whether to constrain the magnitude to be above or below the specified boundary.

Lower Gain Limit

You can specify lower gain limits in the same way as upper gain limits.

Gain and Phase Margin

You can specify a lower bound for the gain margin, the phase margin, or both. The specified bounds appear in text on the Bode magnitude plot.

Note: Gain and phase margin requirements are only applicable to open-loop Bode diagrams.

Open-Loop Nichols Plot Requirements

Phase Margin

Specify a minimum phase margin as a positive value. Graphically, **Control System Designer** displays this requirement as a region of exclusion along the 0 dB open-loop gain axis.

Gain Margin

Specify a minimum gain margin value. Graphically, **Control System Designer** displays this requirement as a region of exclusion along the -180 degree open-loop phase axis.

Closed-Loop Peak Gain
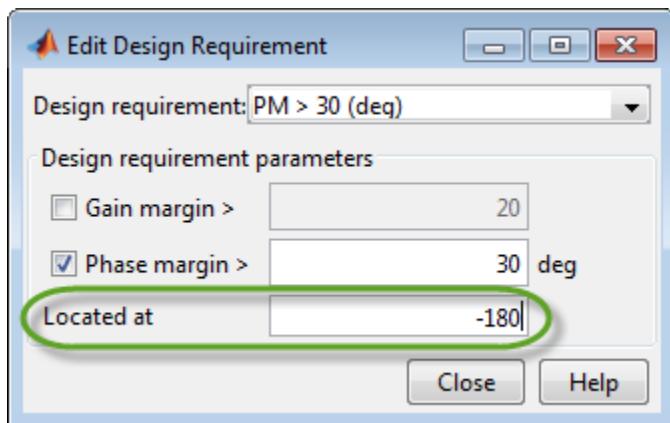
Specify a minimum closed-loop peak gain value. The specified dB value can be positive or negative. The design requirement follows the curves of the Nichols plot grid. As a best practice, have the grid on when using a closed-loop peak gain requirement.

Gain-Phase Design Requirement

To specify a gain-phase design requirement, define two or more vertices of a piece-wise linear boundary line. For each vertex, specify **Open-Loop phase** and **Open-Loop gain** values. This requirement adds a shaded exclusion region on one side of the boundary line. To switch the exclusion region to the opposite side of the boundary, in the Nichols plot, right-click the requirement, and select **Flip**.

Display Location

When editing a phase margin, gain margin, or closed-loop peak gain requirement, you can specify the display location as $-180 \pm k360$ degrees, where k is an integer value.
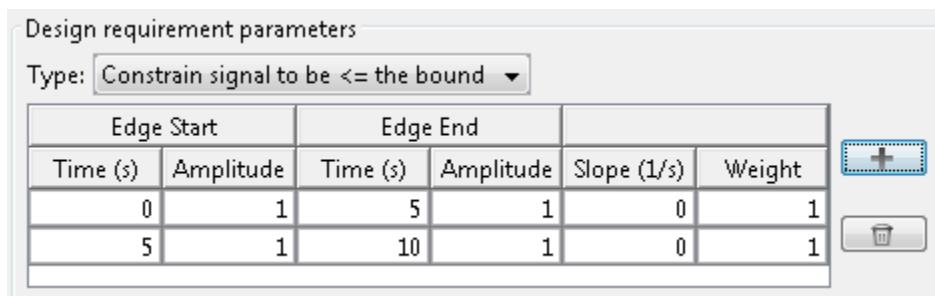
If you enter an invalid location, the closest valid location is selected. While displayed graphically at only one location, these requirements apply regardless of actual phase; that is, they are applied for all values of *k*.

Step and Impulse Response Requirements

Upper Time Response Bound

You can specify upper time response bounds for both step and impulse responses.

A time-response bound consists of one or more line segments. For the start and end points of each segment, specify a **Time** and **Amplitude** value. You can also specify the slope of the line segment. When you change the slope, the amplitude for the end point updates.

If you are using optimization-based tuning, you can assign a tuning **Weight** to each segment to indicate its relative importance.

In the **Type** drop-down list, you can select whether to constrain the response to be above or below the specified boundary.

Lower Time Response Bound

You can specify lower time response bounds for both step and impulse responses in the same way as upper gain limits.

Step Response Bound

For a step response plot, you can also specify a step response bound design requirement.

To define a step response bound requirement, specify the following step response parameters:

- **Final value** — Final steady-state value
- **Rise time** — Time required to reach the specified percentage, **% Rise**, of the **Final value**
- **Settling time** — Time at which the response enters and stays within the settling percentage, **% Settling**, of the **Final value**
- **% Overshoot** — Maximum percentage overshoot above the **Final value**
- **% Undershoot** — Maximum percentage undershoot below the **Initial value**

In **Control System Designer**, step response plots always use an **Initial value** and a **Step time** of 0
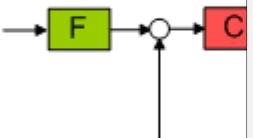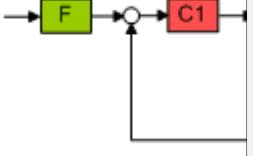
More About

- "Optimize LTI System to Meet Frequency-Domain Requirements"
- "Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)"

Feedback Control Architectures

When you open **Control System Designer** from MATLAB, you can select one of six possible control architecture configurations.

Configuration	Block Diagram	Features
1		• Single feedback loop • Compensator (**C**) and plant (**G**) in forward path • Sensor dynamics (**H**) in feedback path • Prefilter **F**
2		• Single feedback loop • Compensator (**C**) and sensor dynamics (**H**) in feedback path • Prefilter **F**
3		• Compensator (**C**) and plant (**G**) in forward path • Feedforward prefilter **F** for input disturbance attenuation • Sensor dynamics (**H**) in feedback path
4		• Nested multiloop architecture • Outer loop with compensator (**C1**) in forward path • Inner loop with compensator (**C2**) in feedback path • Sensor dynamics (**H**) in feedback path

Configuratio	Block Diagram	Features
5		• Standard Internal Model Control (IMC) architecture • Compensator (**C**) in forward path • Plant **G1** and plant predictive model **G2** • Disturbance model **Gd** • Prefilter **F**
6		• Cascaded multiloop architecture with the inner loop in the forward path of the outer loop. • Compensator (**C1**, **C2**) in the forward path and sensor dynamics (**H1**, **H2**) in the feedback path (both loops) • Prefilter **F**

If your control application does not match one of the supported control architectures, you can use block diagram algebra to convert your system to match an architecture. For an example of such an application, see "Design Multiloop Control System" on page 12-22.

Note: If you are unable to match your application to one of the supported control architectures, consider using the **Control System Tuner** app to design your control system.

See Also
Control System Designer | `sisoinit`

More About
• "Design Multiloop Control System" on page 12-22

Design Multiloop Control System
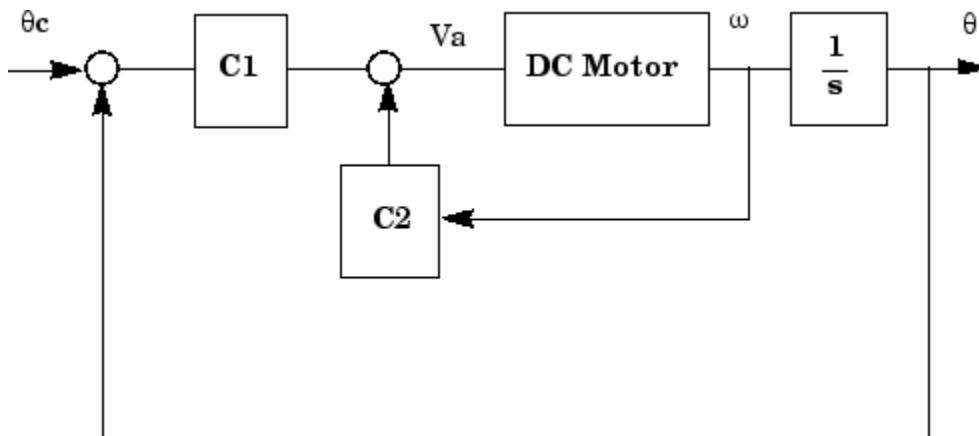
This example shows how to tune a multiloop DC motor position control system using **Control System Designer**.

In many applications, a single-loop control system is not feasible ue to your plant design or your design requirements. If you have a design with an inner and outer loop, you can use **Control System Designer** to design compensators for both loops.

The typical workflow is to tune the compensator for the inner loop first, by isolating the inner loop from the rest of the control system. Once the inner loop is satisfactorily tuned, tune the outer loop to achieve your desired closed-loop response.

System Model

For this example develop a position control system for a DC motor. A single-loop angular velocity controller is designed in "Bode Diagram Design" on page 12-48. To design an angular position controller, add an outer loop that contains an integrator.

Define a state-space plant model, as described in "SISO Example: The DC Motor".

```
% Deifne the motor parameters
R = 2.0
L = 0.5
Km = .015
Kb = .015
Kf = 0.2
```

```
J = 0.02
% Create the state-space model
A = [-R/L -Kb/L; Km/J -Kf/J]
B = [1/L; 0];
C = [0 1];
D = [0];
sys_dc = ss(A,B,C,D);
```

Design Objectives
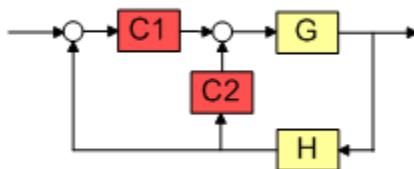
The design objective is to minimize the closed-loop step response settling time, while maintaining an inner-loop phase margin of at least 65 degrees with maximum bandwidth:

- Minimal closed-loop step response settling time.
- Inner-loop phase margin of at least 65 degrees.
- Maximium inner-loop bandwidth.

Match System To Control Architecture
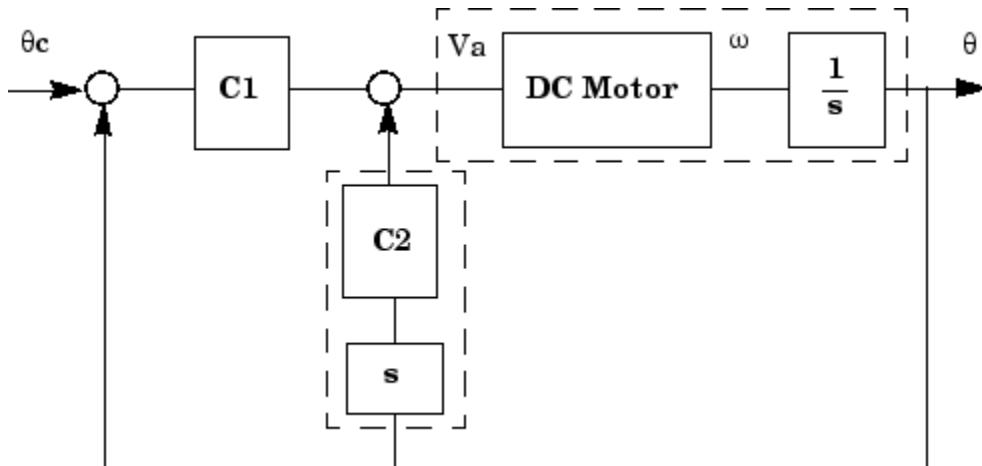
Control System Designer has six possible control architectures from which you can choose. For more information on these architectures, see "Feedback Control Architectures" on page 12-20.

For this example use Configuration 4, which has an inner and outer control loop.

Currently, the control system structure does not match Configuration 4. However, using block diagram algebra, you can modify the system model by adding:

- An integrator to the motor output to get the angular displacement.
- A differentiator to the inner-loop feedback path.

At the MATLAB command line, add the integrator to the motor plant model.

```
plant = sys_dc*tf(1,[1,0]);
```

Create an initial model of the inner-loop compensator that contains the feedback differentiator.

```
Cdiff = tf('s');
```
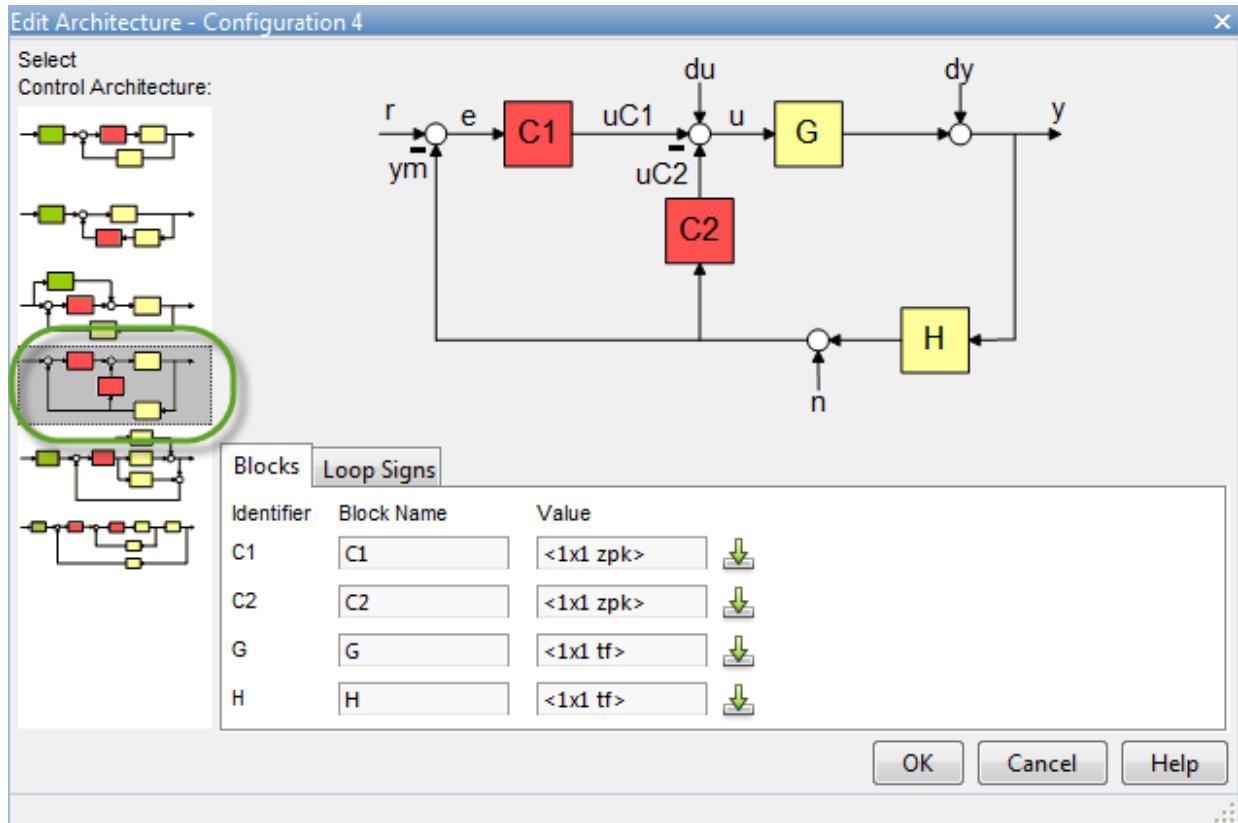
Define Control Architecture

Open **Control System Designer**.

```
controlSystemDesigner
```

In **Control System Designer**, on the **Control System** tab, click **Edit Architecture**.

In the Edit Architecture dialog box, under **Select Control Architecture**, click the fourth architecture.

Import the plant and controller models from the MATLAB workspace.

In the **Blocks** tab, for:

- Controller **C2**, specify a **Value** of `Cdiff`.
- Plant **G**, specify a **Value** of `plant`.

Click **OK**.

The app updates the control architecture and imports the specified models for the motor plant and the inner-loop controller.
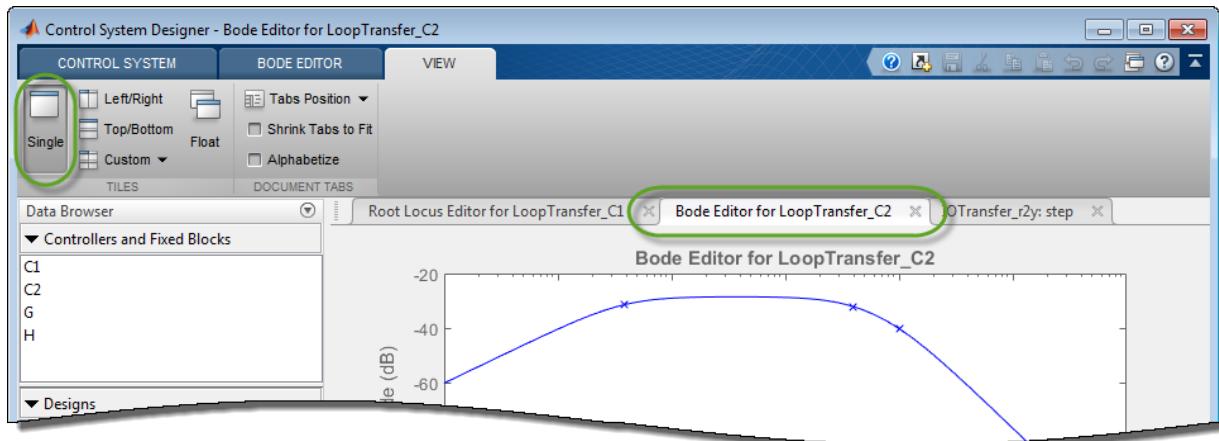
In **Control System Designer**, the following plots open:

- **Bode Editor for LoopTransfer_C1** — Open-loop Bode Editor for the outer loop

- **Root Locus Editor for LoopTransfer_C1** — Open-loop Root Locus Editor for the outer loop
- **Bode Editor for LoopTransfer_C2** — Open-loop Bode Editor for the inner loop
- **Root Locus Editor for LoopTransfer_C2** — Open-loop root Locus Editor for the inner loop
- **IOTransfer_r2y: step** — Overall closed-loop step response from input *r* to output *y*

For this example, close the **Bode Editor for LoopTransfer_C1** and **Root Locus Editor for LoopTransfer_C2** plots.
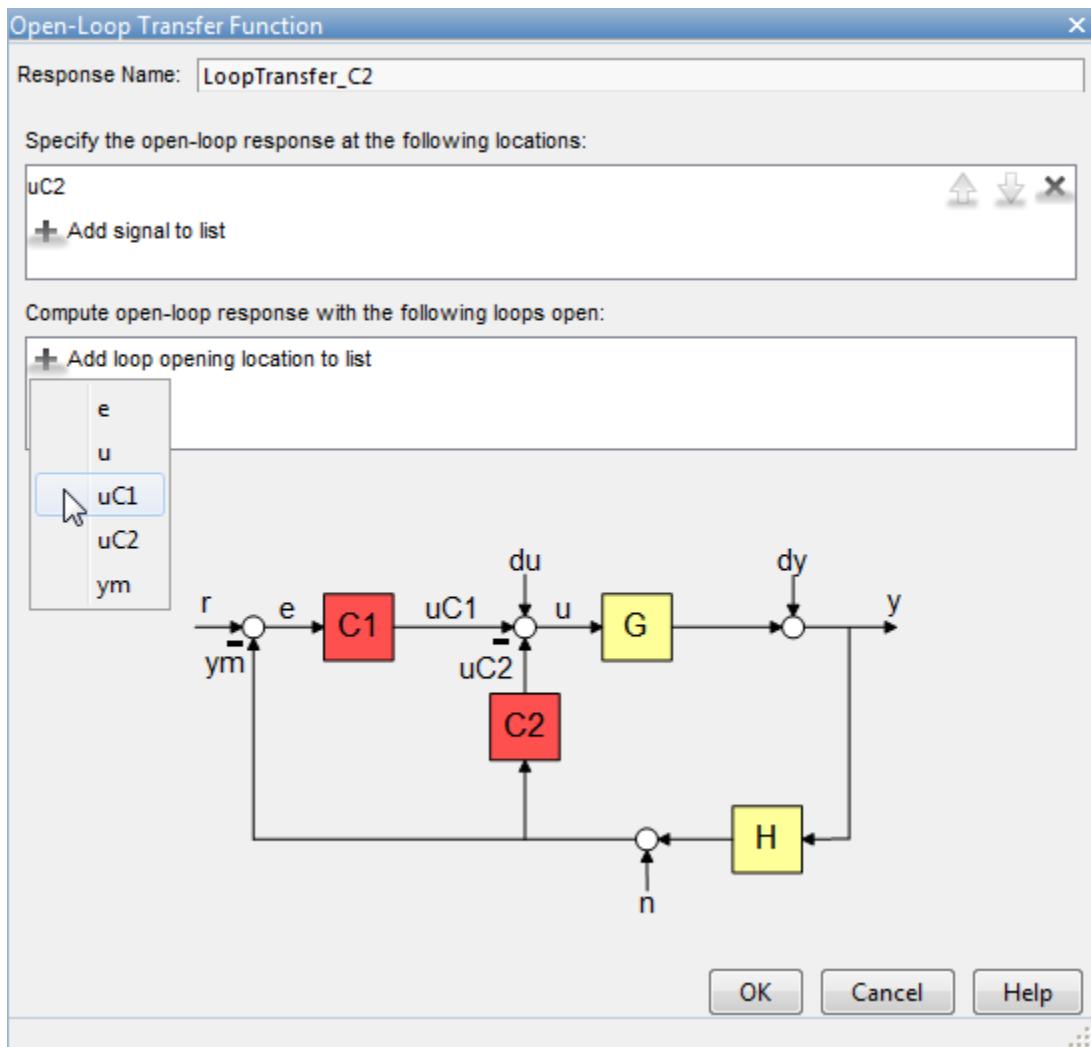
Since the inner loop is tuned first, configure the plots to view just the inner-loop Bode editor plot. On the **View** tab, click **Single**, and click **Bode Editor for LoopTransfer_C2**.

Isolate Inner Loop

To isolate the inner loop from the rest of the control system architecture, add a loop opening to the open-loop response of the inner loop. In the **Data Browser**, right-click `LoopTransfer_C2`, and select **Open Selection**.
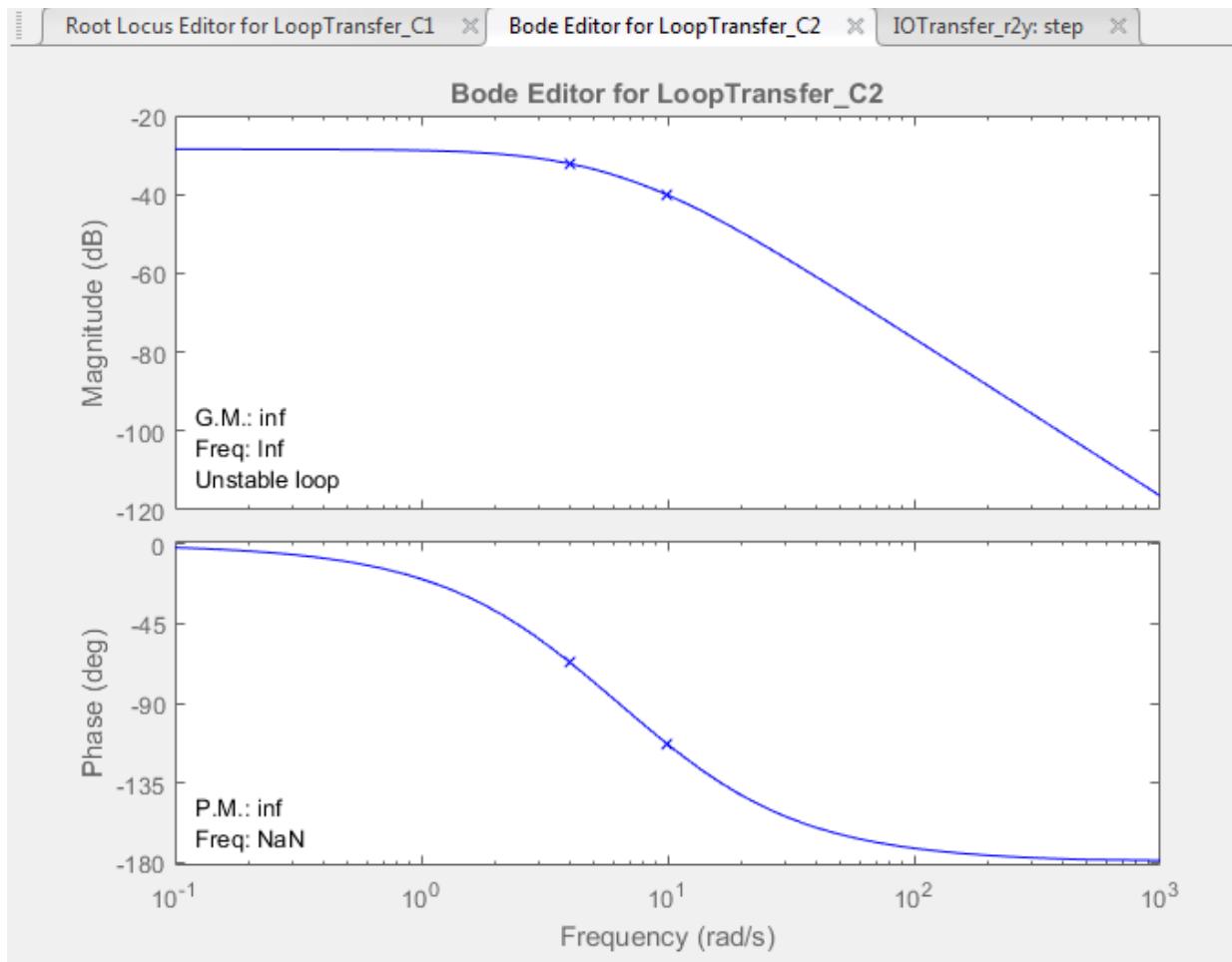
To add a loop opening at the output of outer-loop compensator, **C1**, in the Open-Loop Transfer Function dialog box, click ✚ **Add loop opening location to list**. Then, select **uC1**.

Click **OK**.

The app adds a loop opening at the selected location. This opening removes the effect of the outer control loop on the open-loop transfer function of the inner loop.
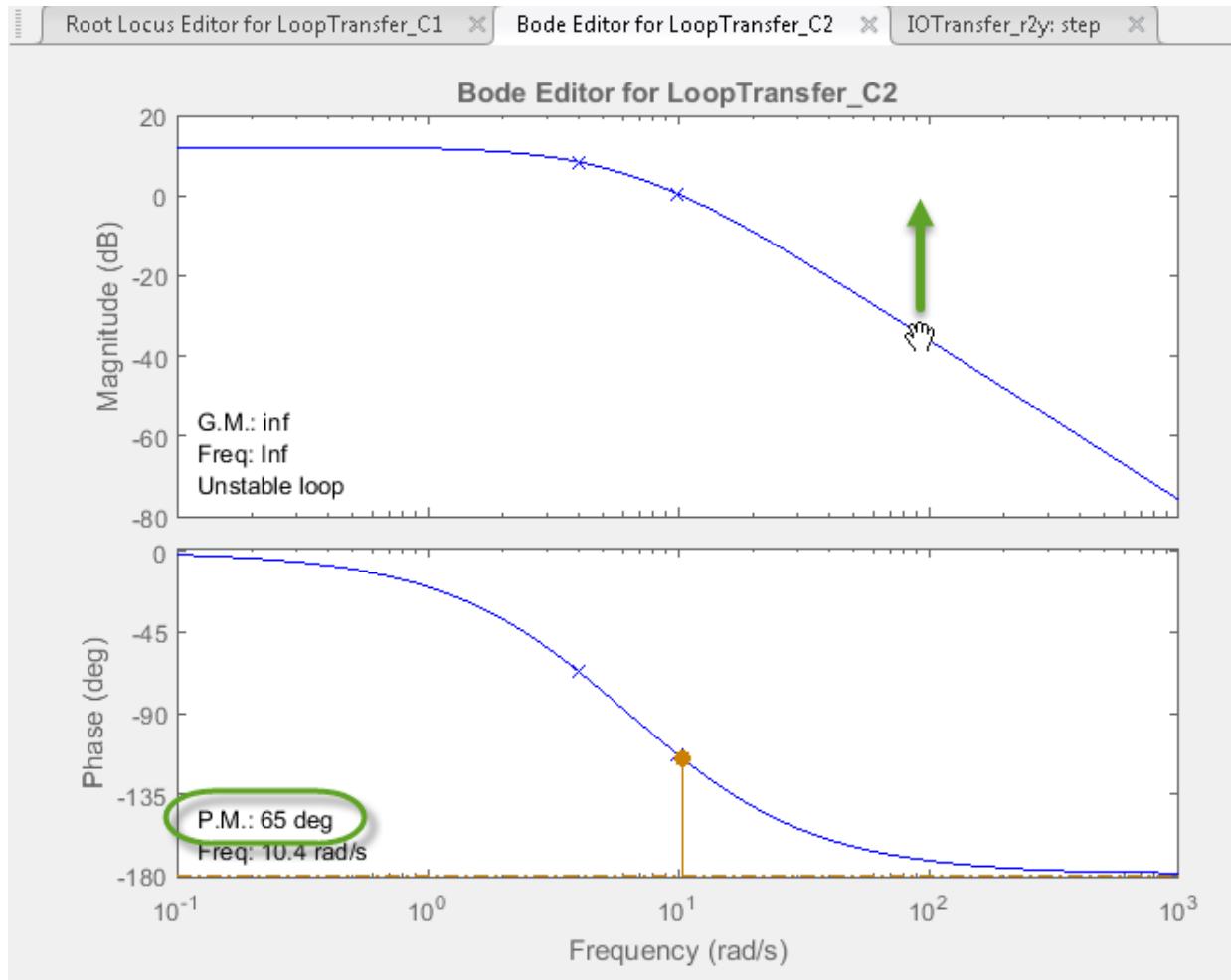
The **Bode Editor** response plot updates to reflect the new open-loop transfer function.

| Root Locus Editor for LoopTransfer_C1 ✕ | Bode Editor for LoopTransfer_C2 ✕ | IOTransfer_r2y: step ✕ |

Bode Editor for LoopTransfer_C2

G.M.: inf
Freq: Inf
Unstable loop

P.M.: inf
Freq: NaN

Tune Inner Loop

To increase the bandwidth of the inner loop, increase the gain of compensator **C2**.

In the **Bode Editor** plot, drag the magnitude response upward until the phase margin is 65 degrees. This corresponds to a compensator gain of 107. Increasing the gain further reduces the phase margin below 65 degrees.
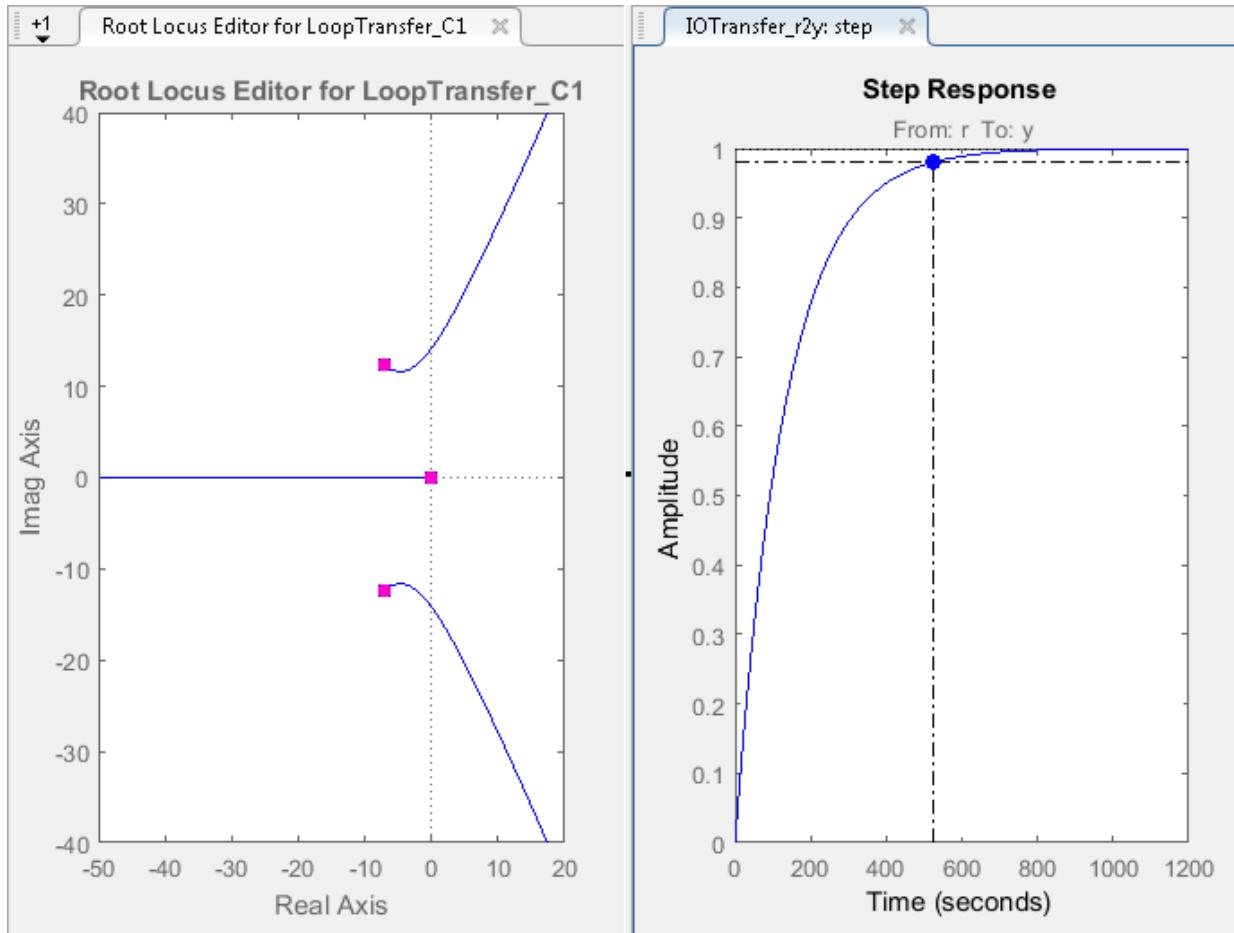
Alternatively, you can adjust the gain value using the compensator editor. For more information, see "Edit Compensator Dynamics" on page 12-90.

Tune Outer Loop

With the inner loop tuned, you can now tune the outer loop to reduce the closed-loop settling time.

In **Control System Designer**, on the **View** tab, select **Left/Right**. Arrange the plots to display the **Root Locus for LoopTransfer_C1** and **IOTransfer_r2y_step** plots simultaneously.
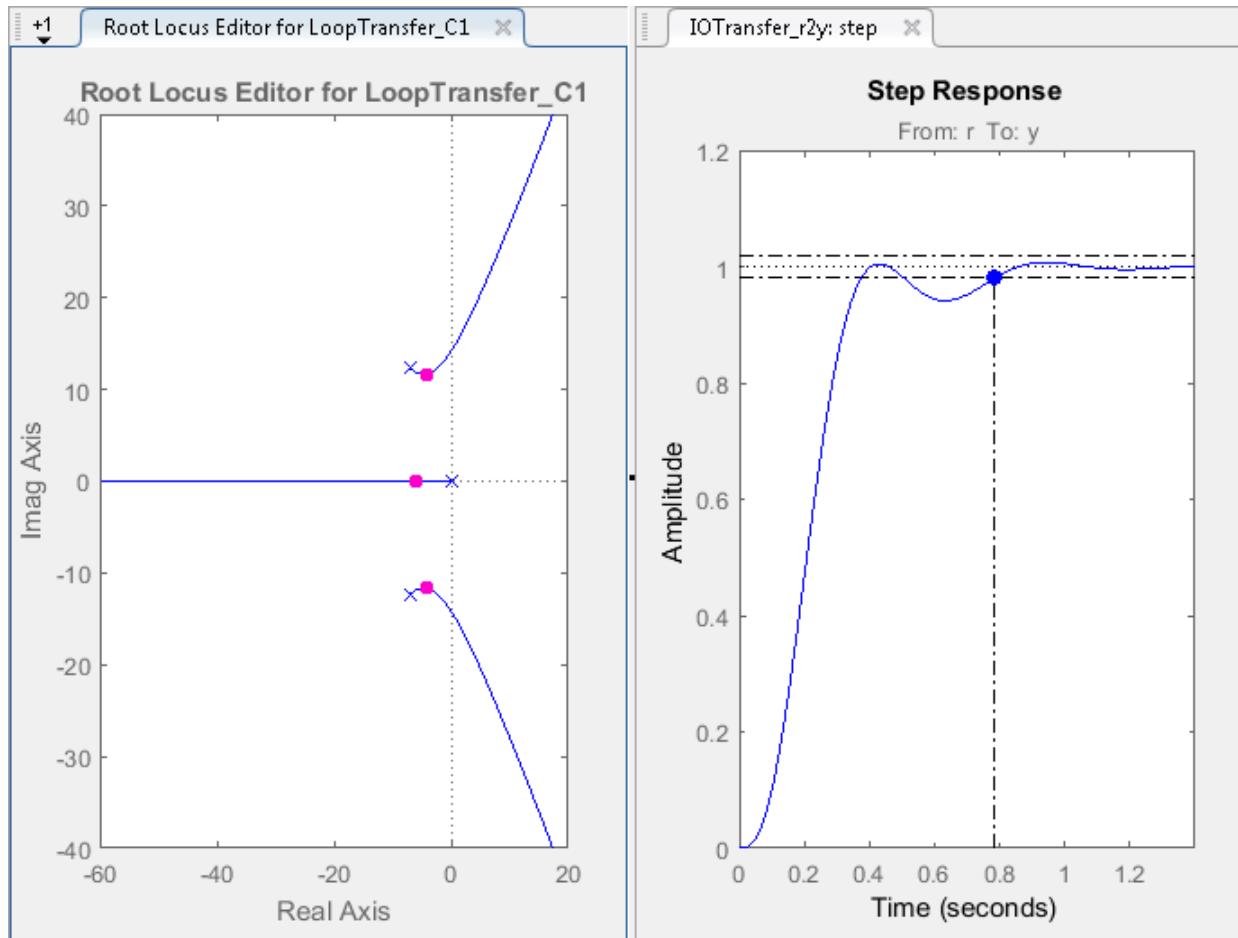
To view the current settling time, right-click in the step response plot and select **Characteristics** > **Settling Time**.

The current closed-loop settling time is greater than 500 seconds.

In the **Root Locus Editor**, increase the gain of compensator **C1**. As the gain increases, the complex pole pair moves toward a slower time constant and the real pole moves

toward a faster time constant. A gain of **600** produces a good compromise between rise time and settling time.

With a closed-loop settling time below 0.8 seconds and an inner-loop phase margin of 65 degrees, the design satisfies the design requirements.

See Also

Control System Designer

More About

- "Feedback Control Architectures" on page 12-20

Multimodel Control Design

This example shows how to design a compensator for a set of plant models using **Control System Designer**.

Typically, the dynamics of a system are not known exactly and may vary. For example, system dynamics can vary because of:

- Parameter value variations caused by manufacturing tolerances — For example, the resistance value of a resistor is typically within a range about the nominal value, 5 Ω +/– 5%.
- Operating conditions — For example, aircraft dynamics change based on altitude and speed.

Any controller you design for such a system must satisfy the design requirements for all potential system dynamics.

In this section...
"Control Design Overview" on page 12-33
"Model Arrays" on page 12-34
"Nominal Model" on page 12-36
"Frequency Grid" on page 12-38
"Design Controller for Multiple Plant Models" on page 12-38

Control Design Overview

To design a controller for a system with varying dynamics:

1. Sample the variations.
2. Create an LTI model for each sample.
3. Create an array of sampled LTI models.
4. Design a controller for a nominal representative model from the array.
5. Analyze the controller design for all models in the array.
6. If the controller design does not satisfy the requirements for all the models, specify a different nominal model and redesign the controller.

Model Arrays

In **Control System Designer**, you can specify multiple models for any plant or sensor in the current control architecture using an array of LTI models (see "Model Arrays" on page 2-96). If you specify model arrays for more than one plant or sensor, the lengths of the arrays must match.

Create Model Arrays

To create arrays for multimodel control design, you can:

- Create multiple LTI models using the `tf`, `ss`, `zpk`, or `frd` commands.

```
% Specify model parameters.
m = 3;
b = 0.5;
k = 8:1:10;
T = 0.1:.05:.2;
% Create an array of LTI models.
for ct = 1:length(k);
    G(:,:,ct) = tf(1,[m,b,k(ct)]);
end
```

- Create an array of LTI models using the `stack` command.

```
% Create individual LTI models.
G1 = tf(1, [1 1 8]);
G2 = tf(1, [1 1 9]);
G3 = tf(1, [1 1 10]);
% Combine models in an array.
G = stack(1,G1,G2,G3);
```
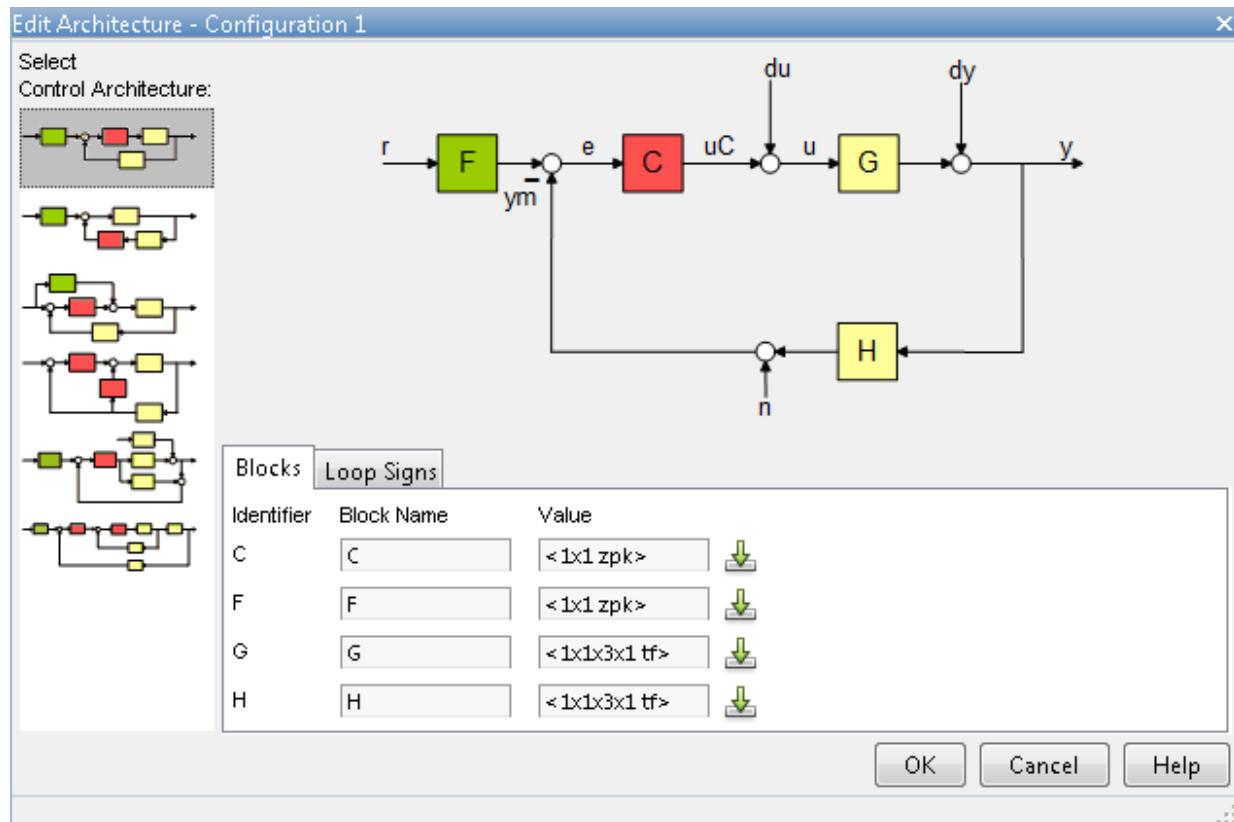
- Perform batch linearizations at multiple operating points. Then export the computed LTI models to create an array of LTI models. See the example Reference Tracking of a DC Motor with Parameter Variations. (Requires Simulink Control Design software)

- Sample an uncertain state-space (`uss`) model using `usample`.

- Compute a `uss` model from a Simulink model. Then use `usubs` or `usample` to create an array of LTI models. See "Obtain Uncertain State-Space Model from Simulink Model". (Requires Robust Control Toolbox software)

- Specify a core Simulink block to linearize to a `uss` or `ufrd` model. See "Specify Uncertain Linearization for Core or Custom Simulink Blocks". (Requires Simulink Control Design and Robust Control Toolbox software)

Import Model Arrays to Control System Designer

To import models as arrays, you can pass them as input arguments when opening **Control System Designer** from the MATLAB command line. For more information, see Control System Designer.

You can also import model arrays into **Control System Designer** when configuring the control architecture. In the Edit Architecture dialog box:

· In the **Value** text box, specify the name of an LTI model from the MATLAB workspace.

· To import block data from the MATLAB workspace or from a MAT-file in your current working directory, click ⬇.

Nominal Model

What Is a Nominal Model?

The *nominal model* is a representative model in the array of LTI models that you use to design the controller in **Control System Designer**. Use the editor and analysis plots to visualize and analyze the effect of the controller on the remaining plants in the array.

You can select any model in the array as your nominal model. For example, you can choose a model that:

- Represents the expected nominal operating point of your system.
- Is an average of the models in the array.
- Represents a worst-case plant.
- Lies closest to the stability point.
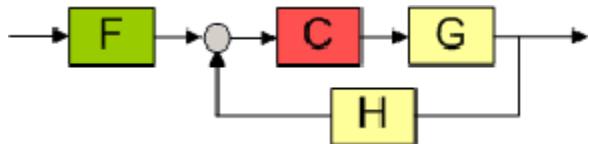
Tip You can plot and analyze the open-loop dynamics of the system on a Bode plot to determine which model to choose as nominal.

Specify Nominal Model
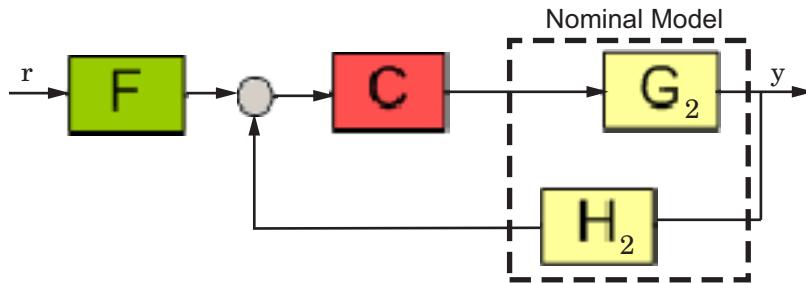
To select a nominal model from the array of LTI models, in **Control System Designer**, click **Multimodel Configuration**. Then, in the Multimodel Configuration dialog box, select a **Nominal model index**. The default index is 1.

For each plant or sensor that is defined as a model array, the app selects the model at the specified index as the nominal model. Otherwise, the app uses scalar expansion to apply the single LTI model for all model indices.

For example, for the following control architecture:

if *G* and *H* are both three-element arrays and the nominal model index is 2, the software uses the second element in both the arrays to compute the nominal model:
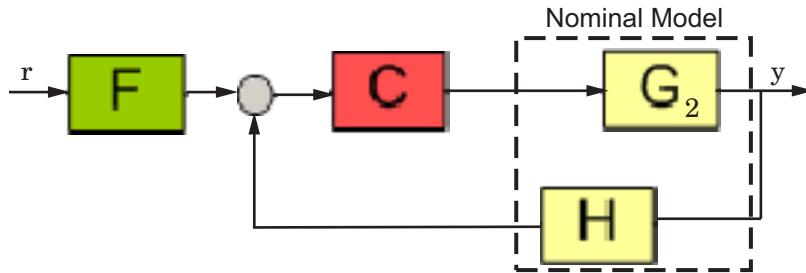
The nominal response from *r* to *y* is:

$$T = \frac{CG_2}{1 + CG_2 H_2}$$

The app also computes and plots the responses showing the effect of *C* on the remaining pairs of plant and sensor models — $G_1 H_1$ and $G_3 H_3$.

If only *G* is an array of LTI models, and the specified nominal model is 2, then the control architecture for nominal response is:

In this case, the nominal response from *r* to *y* is:

$$T = \frac{CG_2}{1 + CG_2 H}$$

The app also computes and plots the responses showing the effect of *C* on the remaining pairs of plant and sensor model — $G_1 H$ and $G_3 H$.

Frequency Grid

The frequency response of a system is computed at a series of frequency values, called a *frequency grid*. By default, **Control System Designer** computes a logarithmically equally spaced grid based on the dynamic range of each model in the array.

Specify a custom frequency grid when:

- The automatic grid has more points than you require. To improve computational efficiency, specify a less dense grid spacing.
- The automatic grid is not sufficiently dense within a particular frequency range. For example, if the response does not capture the resonant peak dynamics of an underdamped system, specify a more dense grid around the corner frequency.
- You are only interested in the response within specific frequency ranges. To improve computational efficiency, specify a grid that covers only the frequency ranges of interest.

For more information on specifying logarithmically spaced vectors, see `logspace`.

Note: Modifying the frequency grid does not affect the frequency response computation for the nominal model. The app always uses the **Auto select** option to compute the nominal model frequency response.

Design Controller for Multiple Plant Models

This example shows how to design a compensator for a set of plant models using **Control System Designer**.

1 **Create Array of Plant Models**

   Create an array of LTI plant models using the `stack` command.

   ```
   % Create an array of LTI models to model plant (G) variations.
   G1 = tf(1,[1 1 8]);
   G2 = tf(1,[1 1 9]);
   G3 = tf(1,[1 1 10]);
   G = stack(1,G1,G2,G3);
   ```

2 **Create Array of Sensor Models**
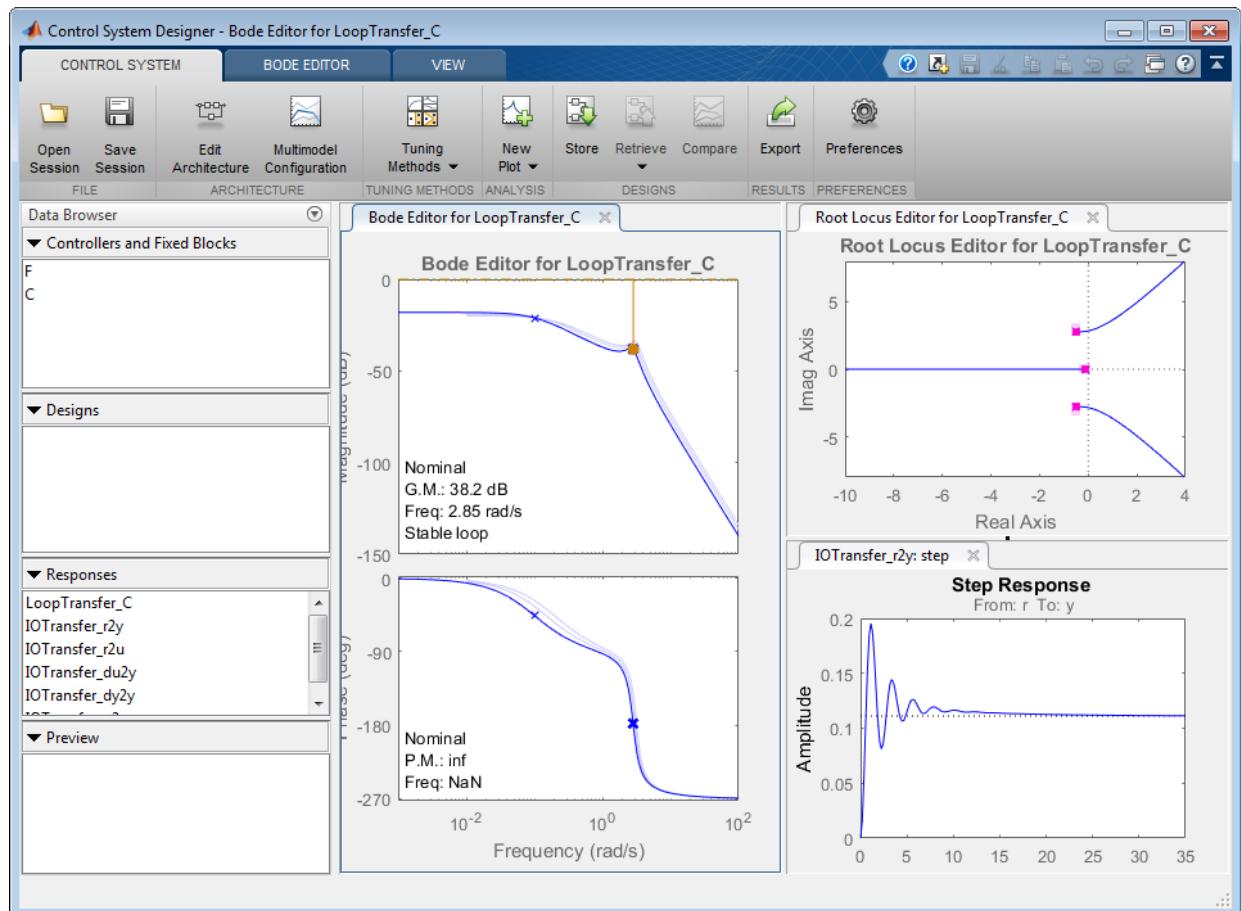
   Similarly, create an array of sensor models.

```
H1 = tf(1,[1/0.1,1]);
H2 = tf(1,[1/0.15,1]);
H3 = tf(1,[1/0.2,1]);
H = stack(1,H1,H2,H3);
```

3   Open Control System Designer

Open **Control System Designer**, and import the plant and sensor model arrays.
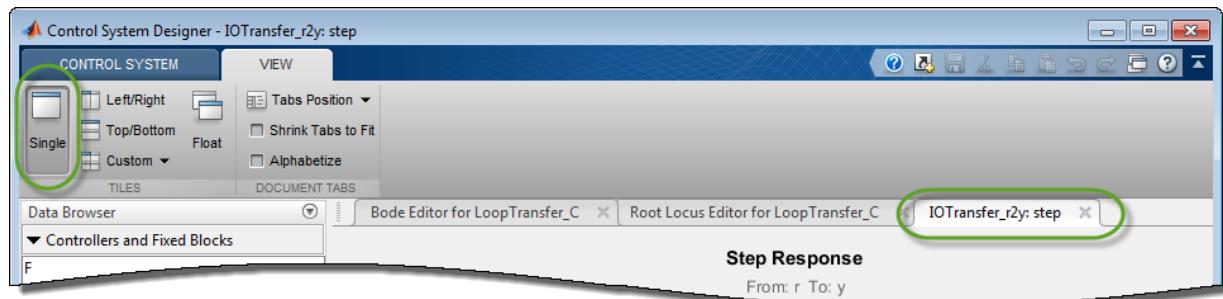
```
controlSystemDesigner(G,1,H)
```
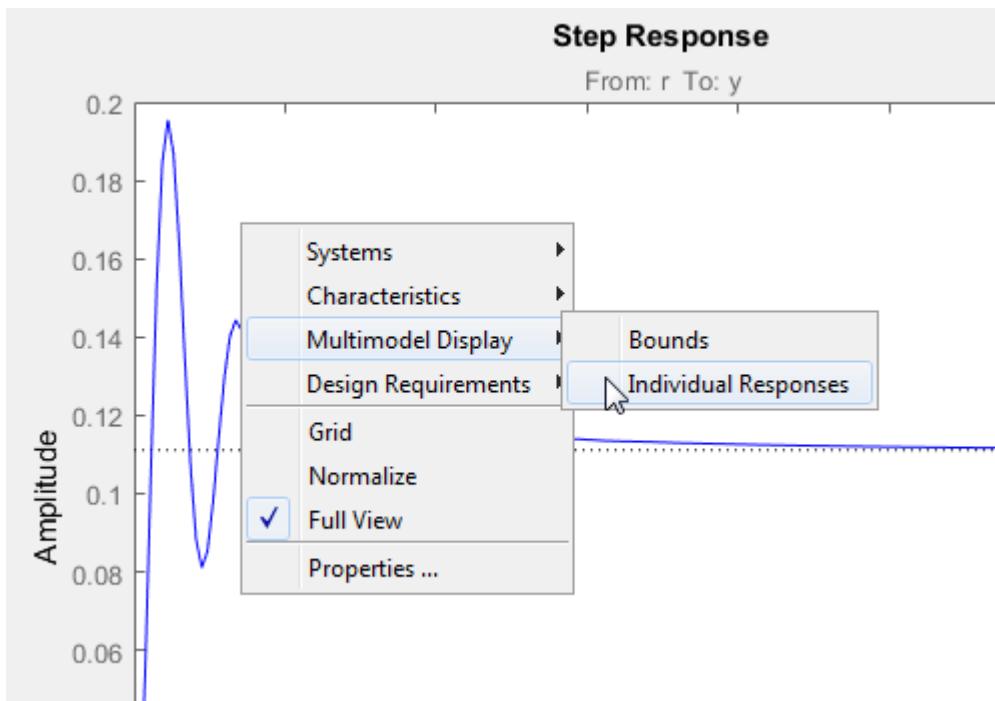
The app opens and imports the plant and sensor model arrays.

4 Configure Analysis Plot

To view the closed-loop step response in a larger plot, in **Control System Designer**, on the **View** tab, click **Single**. Then, click **IOTransfer_r2y: step**.
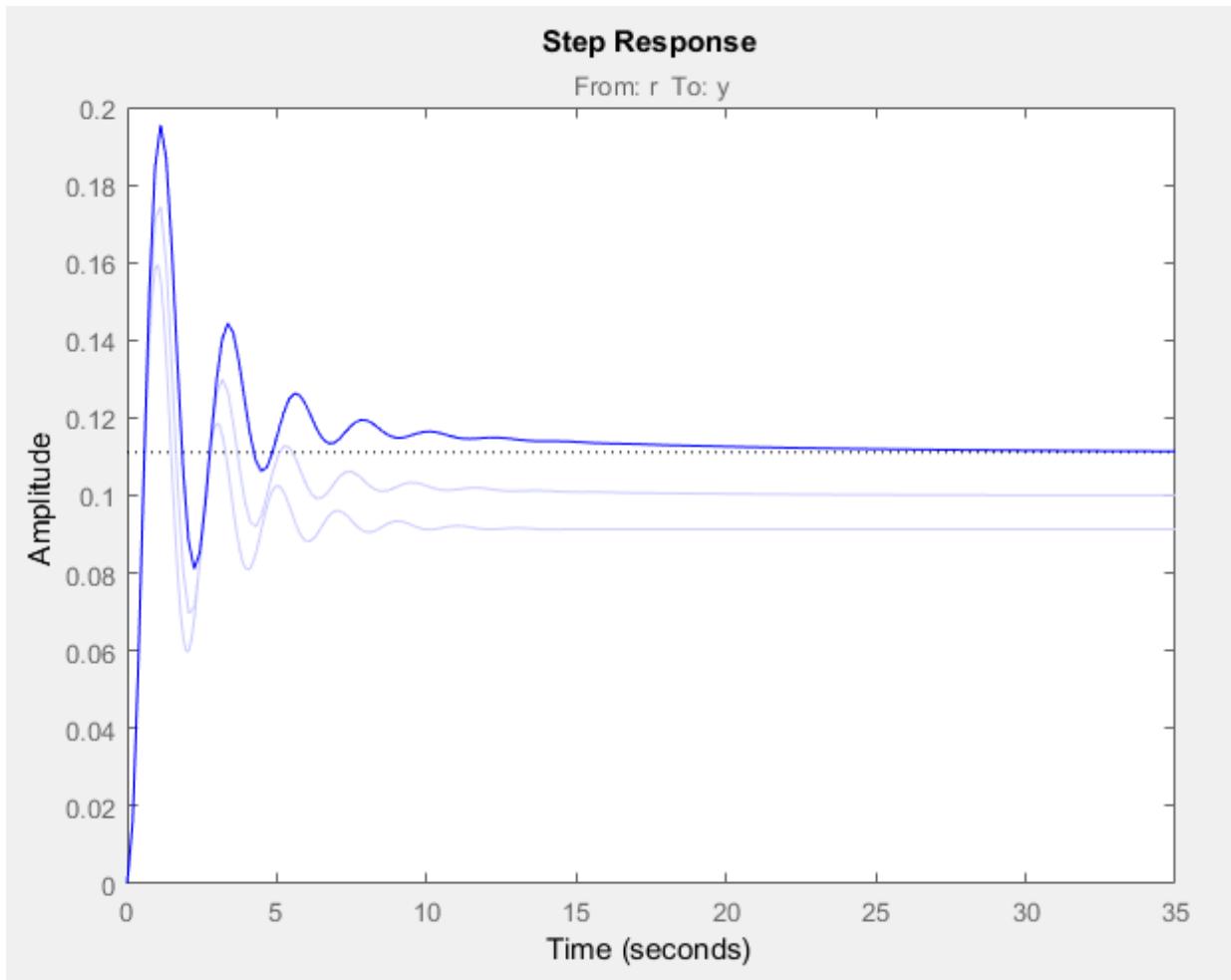
By default the step response shows only the nominal response. To display the individual responses for the other model indices, right-click the plot area, and select **Multimodel Configuration** > **Individual Responses**.

Note: To view an envelope of all model responses, right-click the plot area, and select
Multimodel Configuration > Bounds
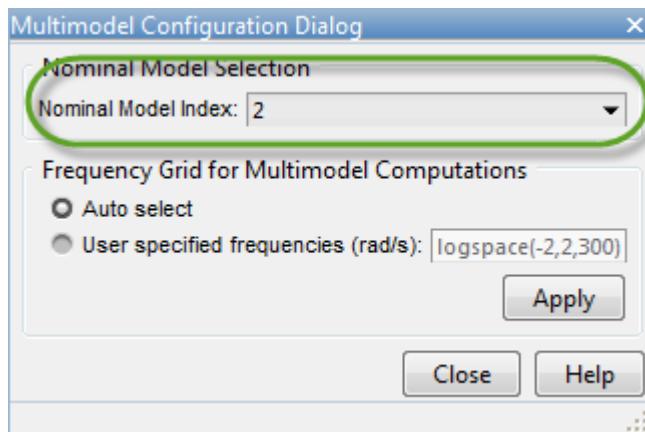
The plot updates to display the responses for the other models.
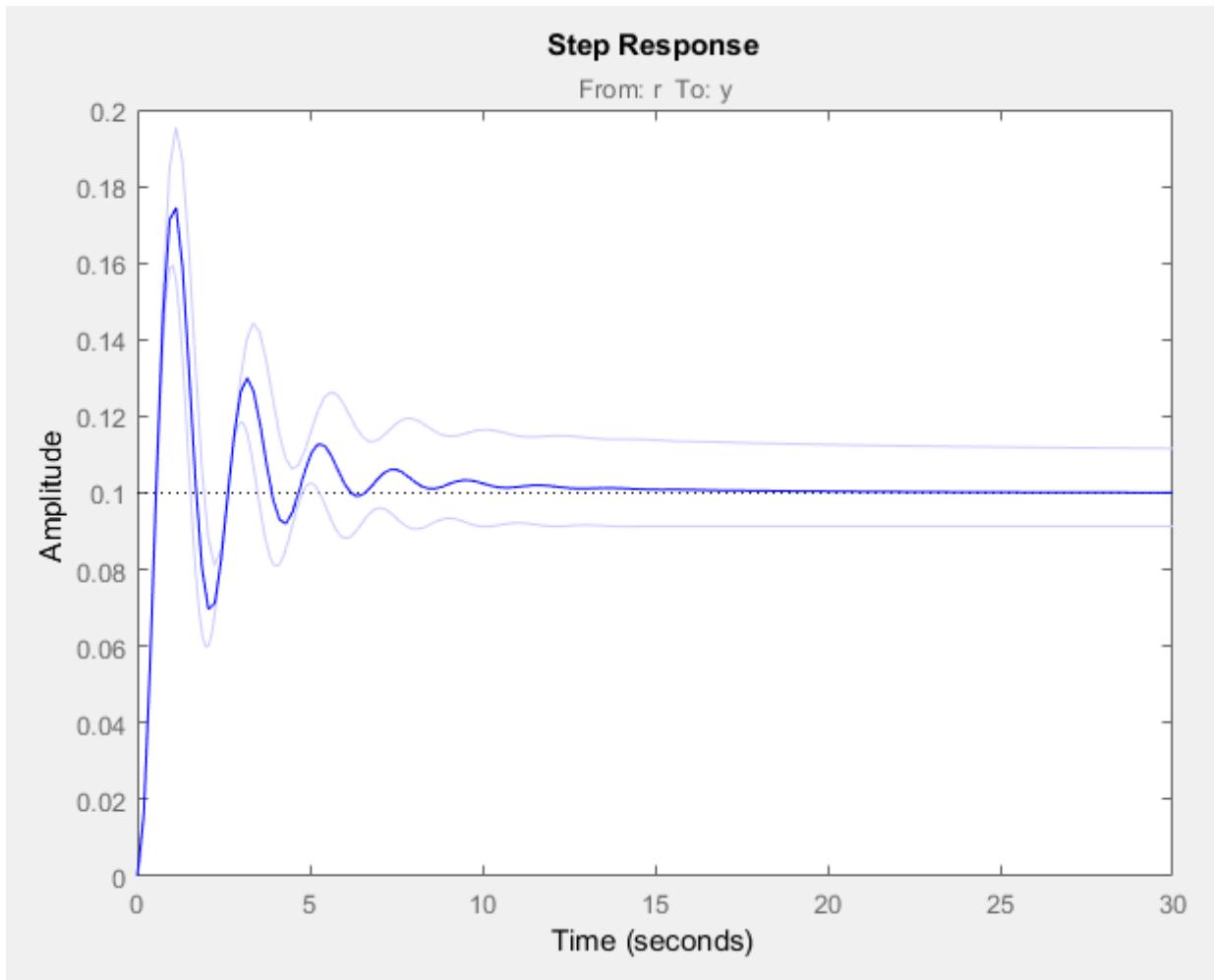
5   Select Nominal Model

On the **Control System** tab, click **Multimodel Configuration**.

In the Mutimodel Configuration dialog box, specify a **Nominal Model Index** of 2.

Click **Close**.
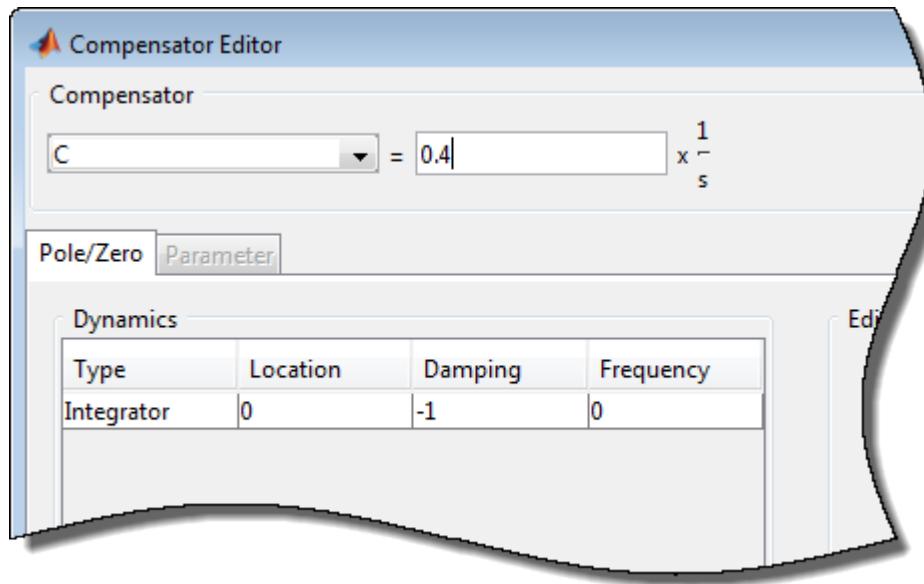
The selected nominal model corresponds to the average system response.

6 **Design Compensator**
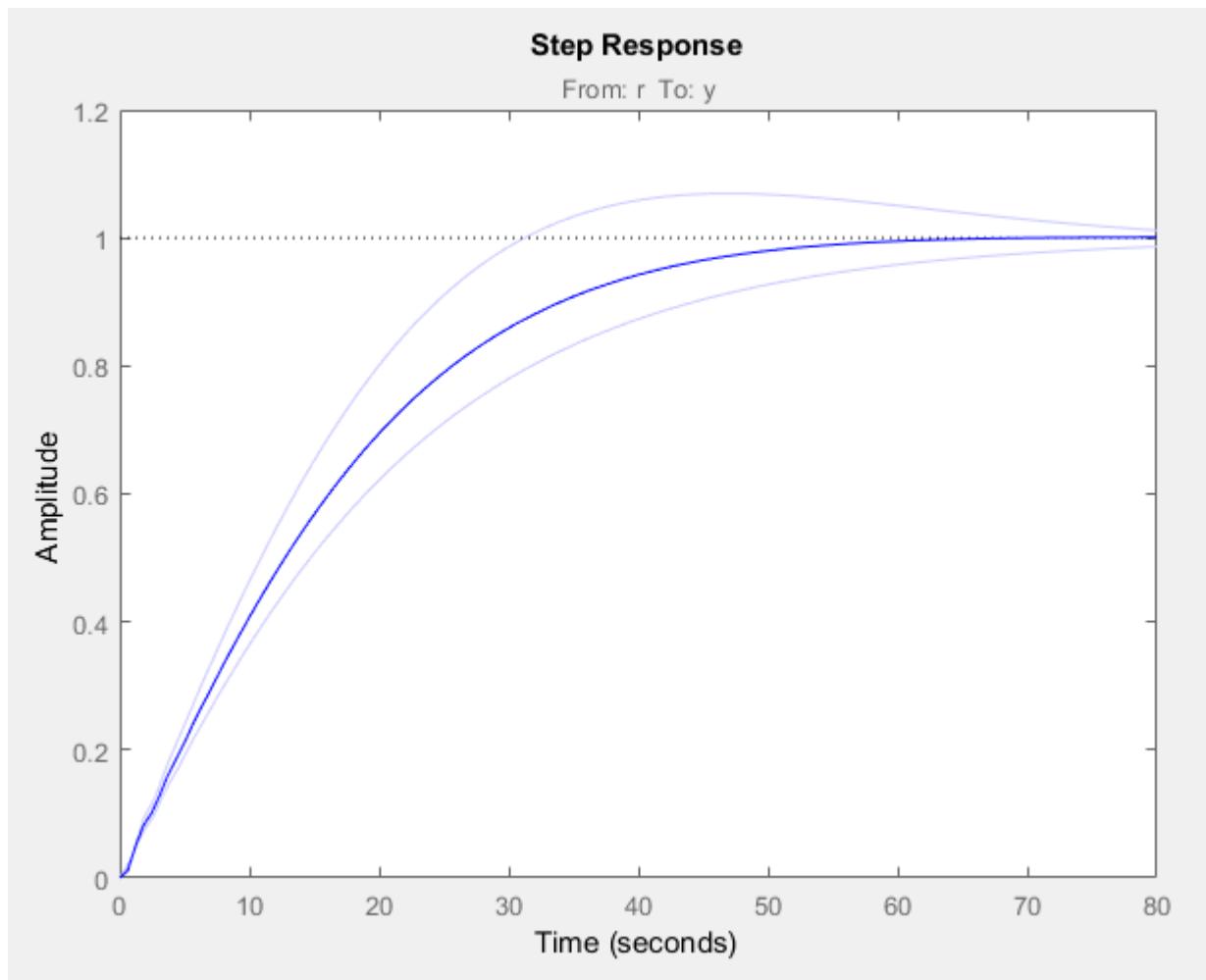
To design a compensator using the nominal model, you can use any of the supported "Control System Designer Tuning Methods" on page 12-4.

For this example, use the Compensator Editor to manually specify the compensator dynamics. Add an integrator to the compensator and set the compensator gain to `0.4`. For more information, see "Edit Compensator Dynamics" on page 12-90.

7   **Analyze Results**

The tuned controller produces a step response with minimal overshoot for the nominal models and a worst-case overshoot less than 10%.

See Also
Control System Designer

Related Examples
- "Model Arrays" on page 2-96

• "Control System Designer Tuning Methods" on page 12-4

Bode Diagram Design

Bode diagram design is an interactive graphical method of modifying a compensator to achieve a specific open-loop response (loop shaping). To interactively shape the open-loop response using **Control System Designer** , use the **Bode Editor**. In the editor, you can adjust the open-loop bandwidth and design to gain and phase margin specifications.

To adjust the loop shape, you can add poles and zeros to your compensator and adjust their values directly in the **Bode Editor**, or you can use the Compensator Editor. For more information, see "Edit Compensator Dynamics" on page 12-90.

For information on all of the tuning methods available in **Control System Designer**, see "Control System Designer Tuning Methods" on page 12-4.

Tune Compensator For DC Motor Using Bode Diagram Graphical Tuning

This example shows how to design a compensator for a DC motor using Bode diagram graphical tuning techniques.

Plant Model and Requirements

The transfer function of the DC motor plant, as described in "SISO Example: The DC Motor", is:

$$G = \frac{1.5}{s^2 + 14s + 40.02}$$

For this example, the design requirements are:

- Rise time of less than 0.5 seconds
- Steady-state error of less than 5%
- Overshoot of less than 10%
- Gain margin greater than 20 dB
- Phase margin greater than 40 degrees

Open Control System Designer

At the MATLAB command line, create a transfer function model of the plant, and open **Control System Designer** in the Bode Editor configuration.
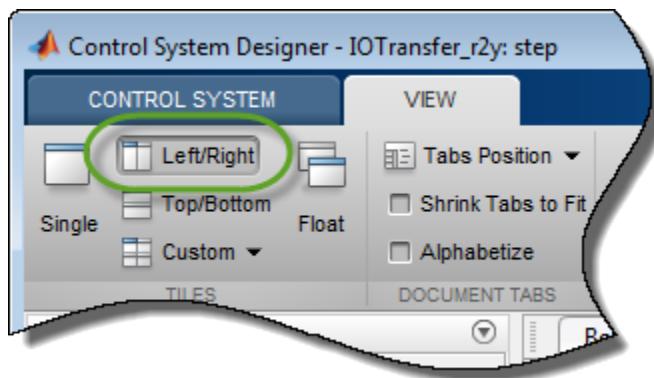
```
G = tf(1.5,[1 14 40.02]);
```

```
controlSystemDesigner('bode',G);
```

The app opens and imports G as the plant model for the default control architecture, **Configuration 1**.

In the app, the following response plots open:

- Open-loop **Bode Editor** for the LoopTransfer_C response. This response is the open-loop transfer function *GC*, where *C* is the compensator and *G* is the plant.
- **Step Response** for the IOTransfer_r2y response. This response is the input-output transfer function for the overall closed-loop system.

Tip To open the open-loop **Bode Editor** when **Control System Designer** is already open, on the **Control System** tab, in the **Tuning Methods** drop-down list, select **Bode Editor**. In the Select Response to Edit dialog box, select an existing response to plot, or create a New Open-Loop Response.

To view the open-loop frequency response and closed-loop step response simultaneously, on the **Views** tab, click **Left/Right**.
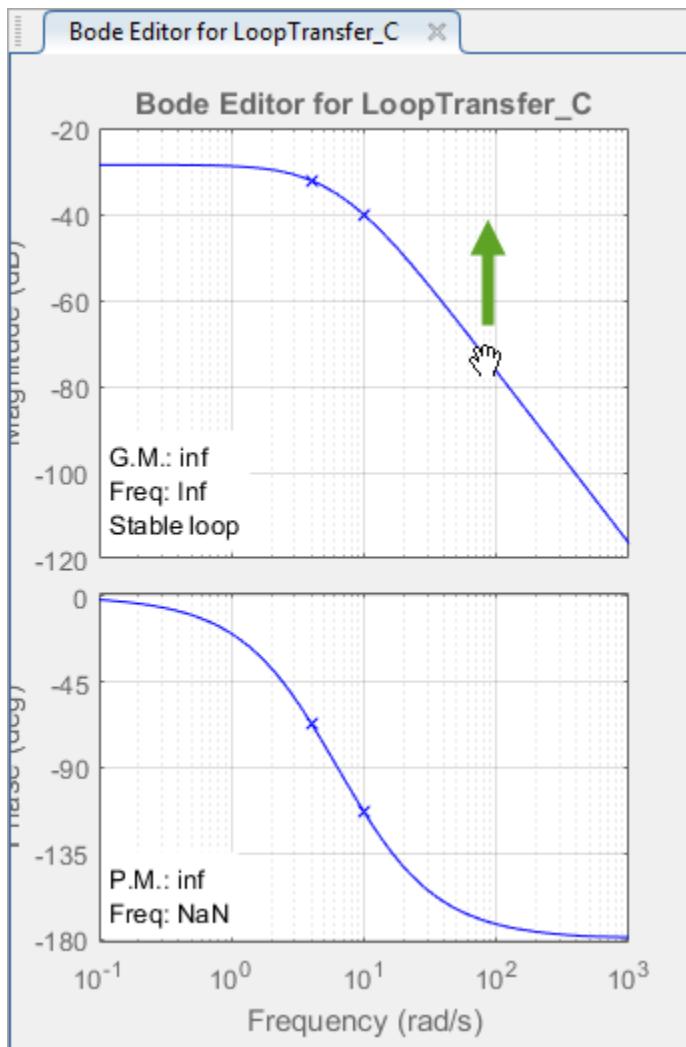
The app displays the **Bode Editor** and **Step Response** plots side-by-side.

Adjust Bandwidth

Since the design requires a rise time less than 0.5 seconds, set the open-loop DC crossover frequency to about 3 rad/s. To a first-order approximation, this crossover frequency corresponds to a time constant of 0.33 seconds.
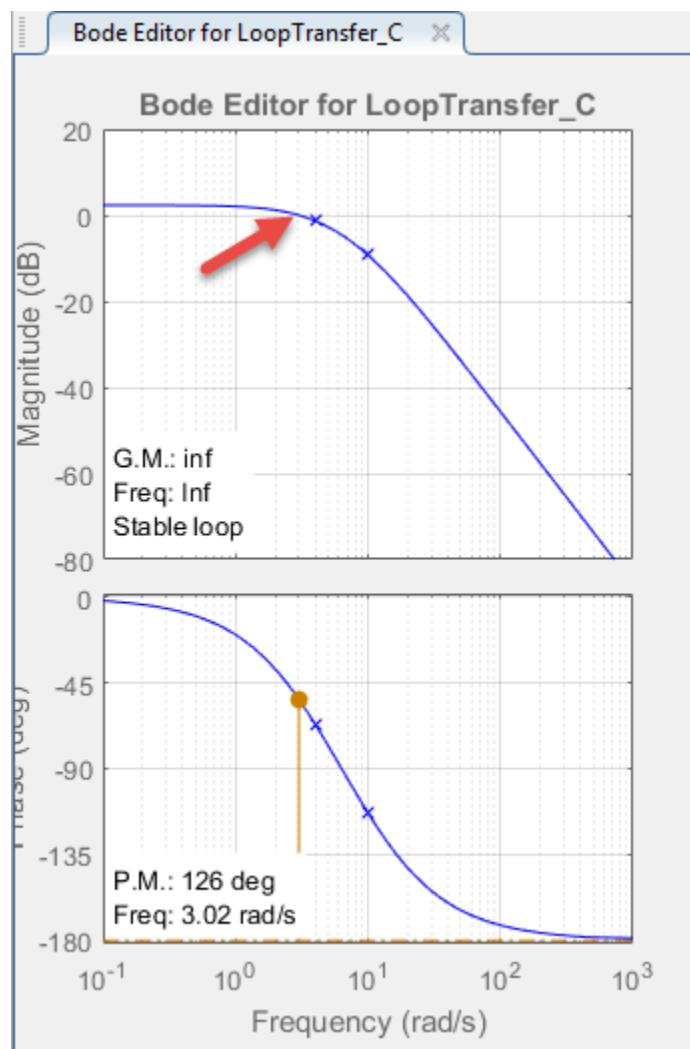
To make the crossover easier to see, turn on the plot grid. Right-click the **Bode Editor** plot area, and select **Grid**. The app adds a grid to the Bode response plots.

To adjust the crossover frequency increase the compensator gain. In the **Bode Editor** plot, in the **Magnitude** response plot, drag the response upward. Doing so increases the gain of the compensator.

As you drag the magnitude plot, the app computes the compensator gain and updates the response plots.

Drag the magnitude response upward until the crossover frequency is about 3 rad/s.

View Step Response Characteristics

To add the rise time to the **Step Response** plot, right-click the plot area, and select **Characteristics** > **Rise Time**.

To view the rise time, move the cursor over the rise time indicator.

The rise time is around 0.23 seconds, which satisfies the design requirements.

Similarly, to add the peak response to the **Step Response** plot, right-click the plot area, and select **Characteristics** > **Peak Response**.

The peak overshoot is around 3.5%.

Add Integrator To Compensator

To meet the 5% steady-state error requirement, eliminate steady-state error from the closed-loop step response by adding an integrator to your compensator. In the **Bode Editor** right-click in the plot area, and select **Add Pole/Zero** > **Integrator**.

Adding an integrator produces zero steady-state error. However, changing the compensator dynamics also changes the crossover frequency, increasing the rise time. To reduce the rise time, increase the crossover frequency to 3 rad/s.

Adjust Compensator Gain

To return the crossover frequency to 3 rad/s, increase the compensator gain further. Right-click the **Bode Editor** plot area, and select **Edit Compensator**.

In the Compensator Editor dialog box, in the **Compensator** section, specify a gain of 99, and press **Enter**.

The response plots update automatically.

The rise time is around 0.4 seconds, which satisfies the design requirements. However, the peak overshoot is around 32%. A compensator consisting of a gain and an integrator is not sufficient to meet the design requirements. Therefore, the compensator requires additional dynamics.

Add Lead Network to Compensator

In the **Bode Editor**, review the gain margin and phase margin for the current compensator design. The design requires a gain margin greater than 20 dB and phase margin greater than 40 degrees. The current design does not meet either of these requirements.

To increase the stability margins, add a lead network to the compensator.
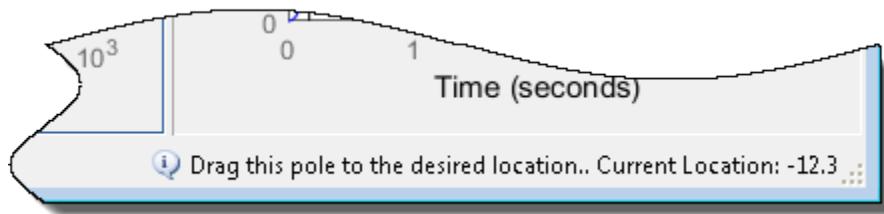
In the **Bode Editor**, right-click and select **Add Pole/Zero > Lead**.

To specify the location of the lead network pole, click on the magnitude response. The app adds a real pole (red **X**) and real zero (red **0**) to the compensator and to the **Bode Editor** plot.
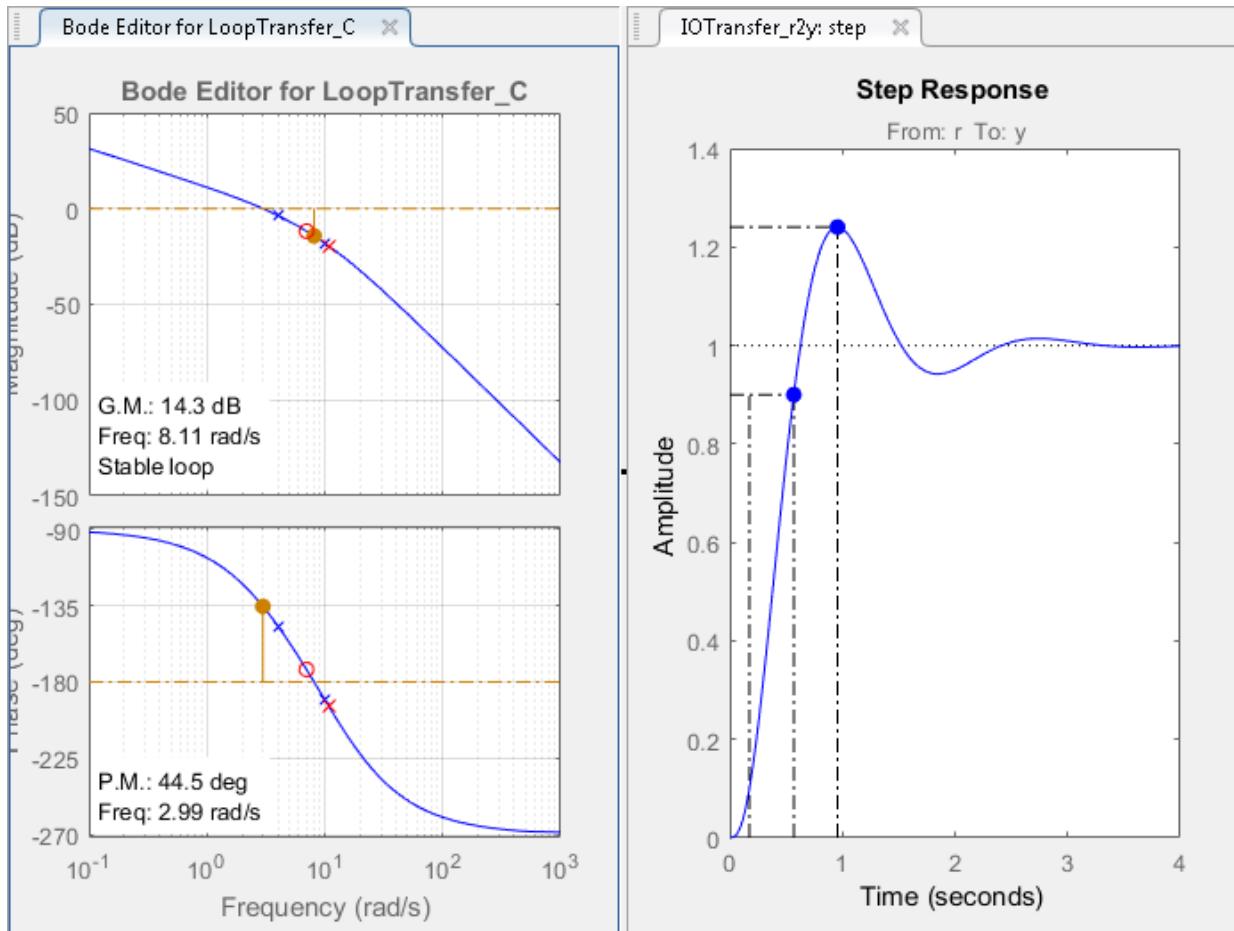
In the **Bode Editor**, drag the pole and zero to change their locations. As you drag them, the app updates the pole/zero values and updates the response plots.

To decrease the magnitude of a pole or zero, drag it towards the left. Since the pole and zero are on the negative real axis, dragging them to the left moves them closer to the origin in the complex plane.

Tip As you drag a pole or zero, the app displays the new value in the status bar, on the right side.

As an initial estimate, drag the zero to a location around -7 and the pole to a location around -11.

The phase margin meets the design requirements; however, the gain margin is still too low.

Edit Lead Network Pole and Zero

To improve the controller performance, tune the lead network parameters.

In the Compensator Editor dialog box, in the **Dynamics** section, click the **Lead** row.

In the **Edit Selected Dynamics** section, in the **Real Zero** text box, specify a location of -4.3, and press **Enter**. This value is near the slowest (left-most) pole of the DC motor plant.

In the **Real Pole** text box, specify a value of -28, and press **Enter**.

When you modify a lead network parameters, the **Compensator** and response plots update automatically.

In the app, in the **Bode Editor**, the gain margin of 20.5 just meets the design requirement.

To add robustness to the system, in the Compensator Editor dialog box, decrease the compensator gain to 550, and press **Enter**. The gain margin increases to 21.8, and the response plots update.

In **Control System Designer**, in the response plots, compare the system performance to the design requirements. The system performance characteristics are:

- Rise time is 0.445 seconds.
- Steady-state error is zero.
- Overshoot is 3.39%.

- Gain margin is 21.8 dB.
- Phase margin is 65.6 degrees.

The system response meets all of the design requirements.

See Also

bodeplot | Control System Designer
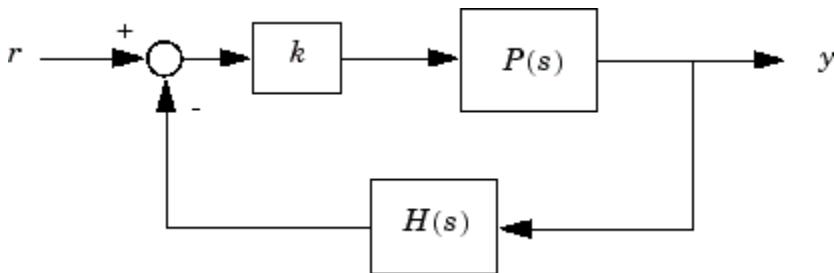
More About

- "Edit Compensator Dynamics" on page 12-90
- "Control System Designer Tuning Methods" on page 12-4
- "Root Locus Design" on page 12-62
- "Nichols Plot Design" on page 12-77

Root Locus Design

Root locus design is a common control system design technique in which you edit the compensator gain, poles, and zeros in the root locus diagram.

As the open-loop gain, k, of a control system varies over a continuous range of values, the root locus diagram shows the trajectories of the closed-loop poles of the feedback system. For example, in the following tracking system:

$P(s)$ is the plant, $H(s)$ is the sensor dynamics, and k is an adjustable scalar gain The closed-loop poles are the roots of

$$q(s) = 1 + kP(s)H(s)$$

The root locus technique consists of plotting the closed-loop pole trajectories in the complex plane as k varies. You can use this plot to identify the gain value associated with a desired set of closed-loop poles.

Tune Electrohydraulic Servomechanism Using Root Locus Graphical Tuning
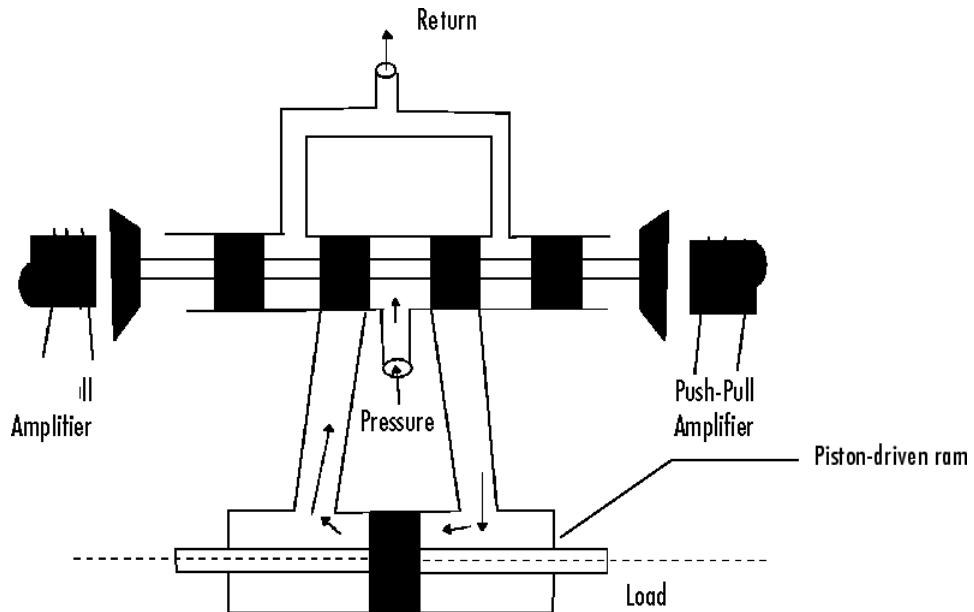
This example shows how to design a compensator for an electrohydraulic servomechanism using root locus graphical tuning techniques.

Plant Model

A simple version of an electrohydraulic servomechanism model consists of
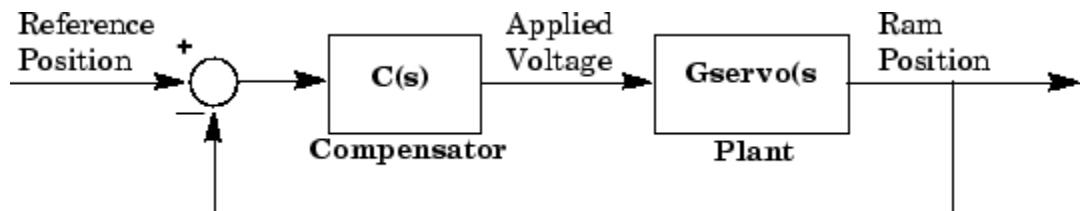
- A push-pull amplifier (a pair of electromagnets)
- A sliding spool in a vessel of high-pressure hydraulic fluid
- Valve openings in the vessel to allow for fluid to flow

- A central chamber with a piston-driven ram to deliver force to a load
- A symmetrical fluid return vessel

The force on the spool is proportional to the current in the electromagnet coil. As the spool moves, the valve opens, allowing the high-pressure hydraulic fluid to flow through the chamber. The moving fluid forces the piston to move in the opposite direction of the spool. For more information on this model, including the derivation of a linearized model, see [1].

You can use the input voltage to the electromagnet to control the ram position. When measurements of the ram position are available, you can use feedback for the ram position control, as shown in the following, where *Gservo* represents the servomechanism:

Design Requirements

For this example, tune the compensator, *C*(*s*) to meet the following closed-loop step response requirements:

- The 2% settling time is less than 0.05 seconds.
- The maximum overshoot is less than 5%.

Open Control System Designer
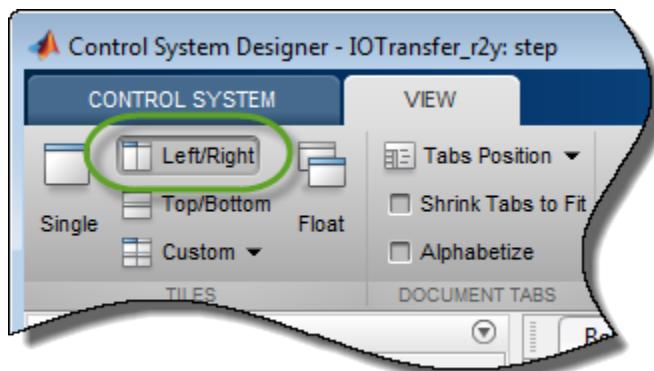
At the MATLAB command line, load a linearized model of the servomechanism, and open **Control System Designer** in the root locus editor configuration.

```
load ltiexamples Gservo
controlSystemDesigner('rlocus',Gservo);
```
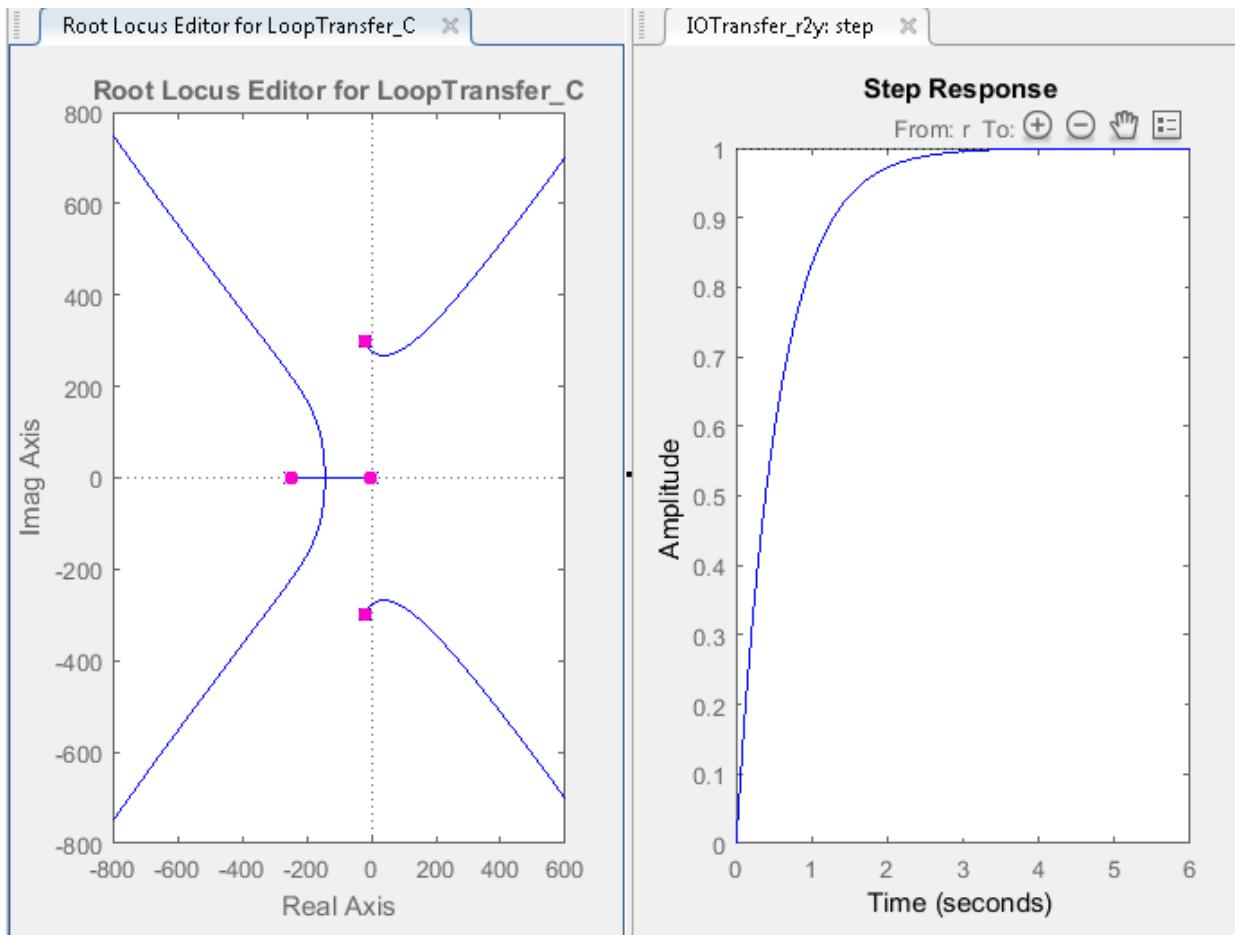
The app opens and imports Gservo as the plant model for the default control architecture, **Configuration 1**.

In **Control System Designer**, a **Root Locus Editor** plot and input-output **Step Response** open.

To view the open-loop frequency response and closed-loop step response simultaneously, on the **Views** tab, click **Left/Right**.
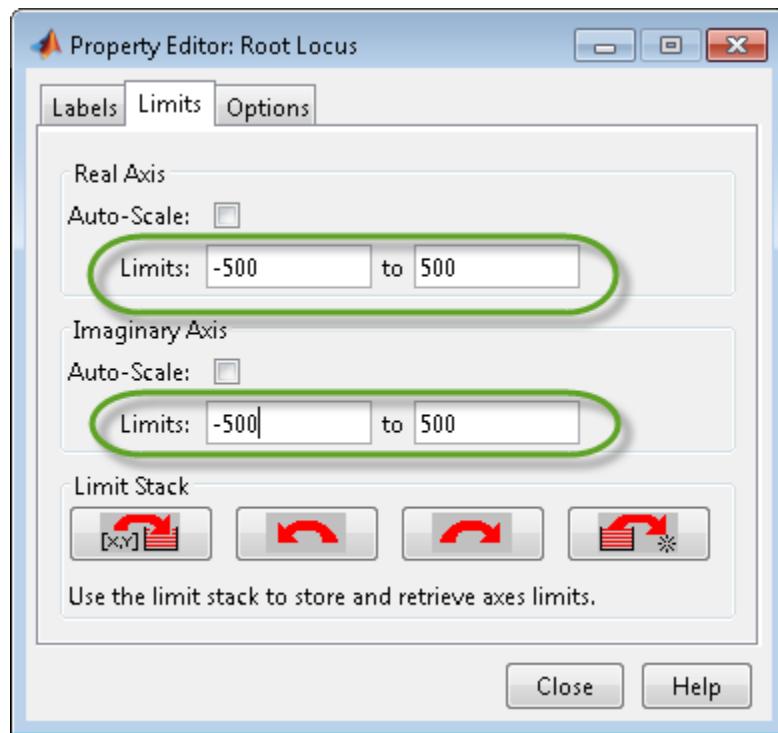
The app displays **Bode Editor** and **Step Response** plots side-by-side.

In the closed-loop step response plot, the rise time is around two seconds, which does not satisfy the design requirements.

To make the root locus diagram easier to read, zoom in. In the **Root Locus Editor**, right-click the plot area and select **Properties**.

In the Property Editor dialog box, on the **Limits** tab, specify **Real Axis** and **Imaginary Axis** limits from -500 to 500.
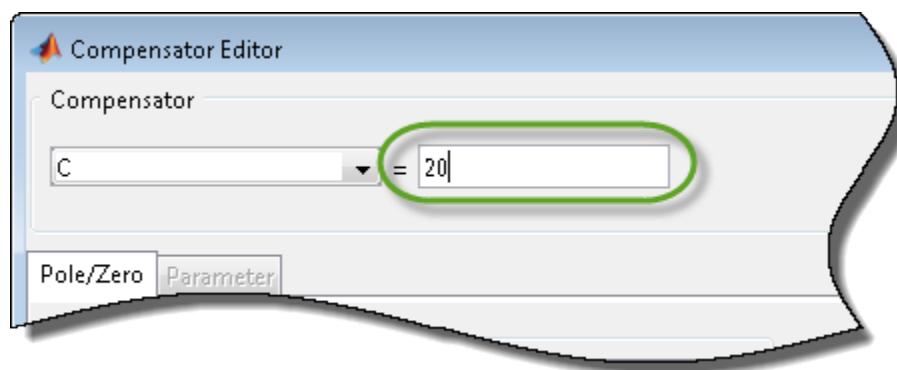
Click **Close**.
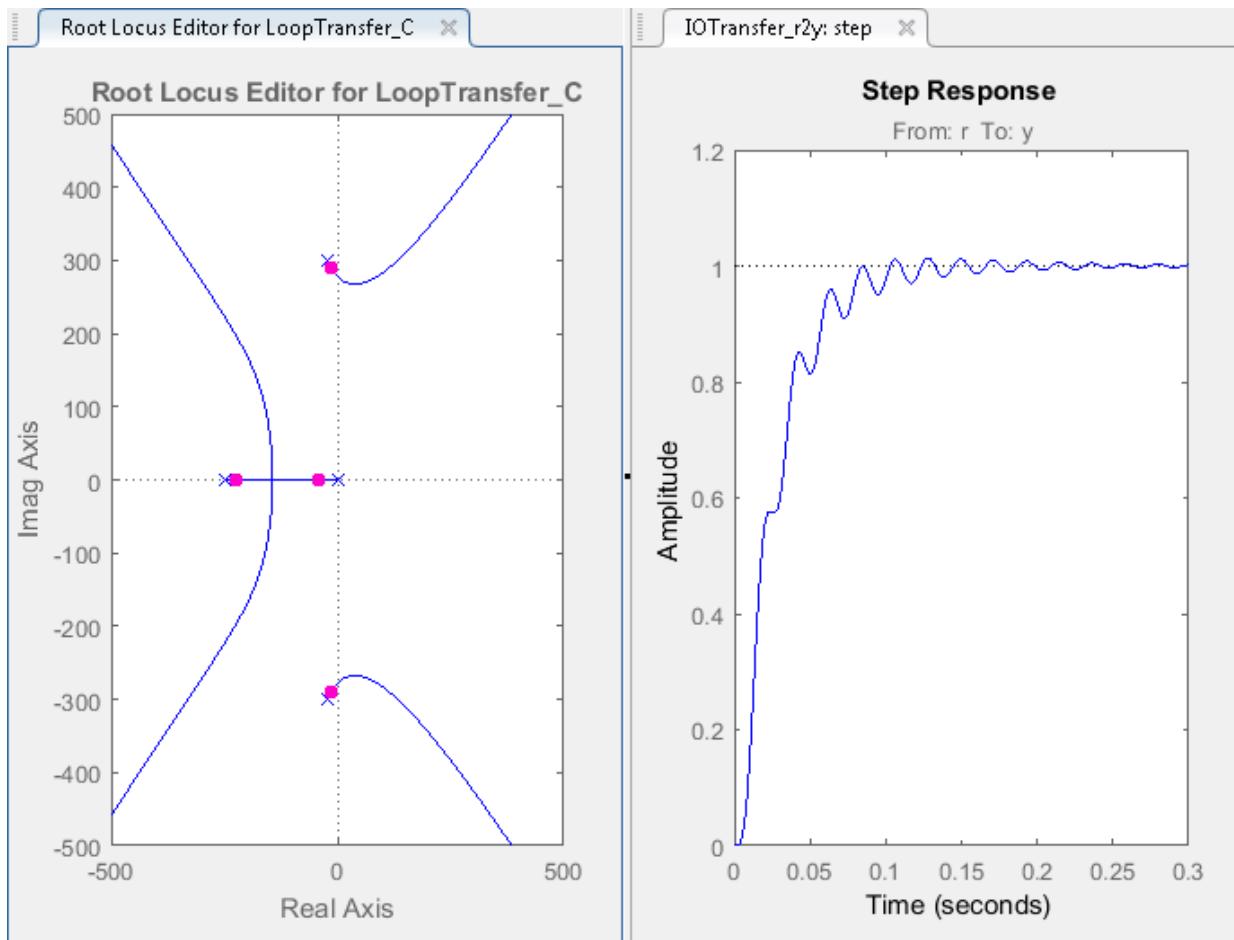
Increase Compensator Gain

To create a faster response, increase the compensator gain. In the **Root Locus Editor**, right-click the plot area and select **Edit Compensator**.

In the Compensator Editor dialog box, specify a gain of 20.

In the **Root Locus Editor** plot, the closed-loop pole locations move to reflect the new gain value. Also, the **Step Response** plot updates.
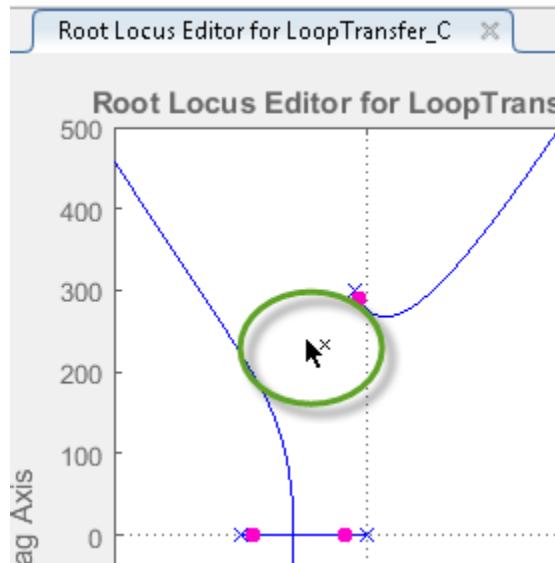
The closed-loop response does not satisfy the settling time requirement and exhibits unwanted ringing.

Increasing the gain makes the system underdamped and further increases lead to instability. Therefore, to meet the design requirements, you must specify additional compensator dynamics. For more information on adding and editing compensator dynamics, see "Edit Compensator Dynamics" on page 12-90.
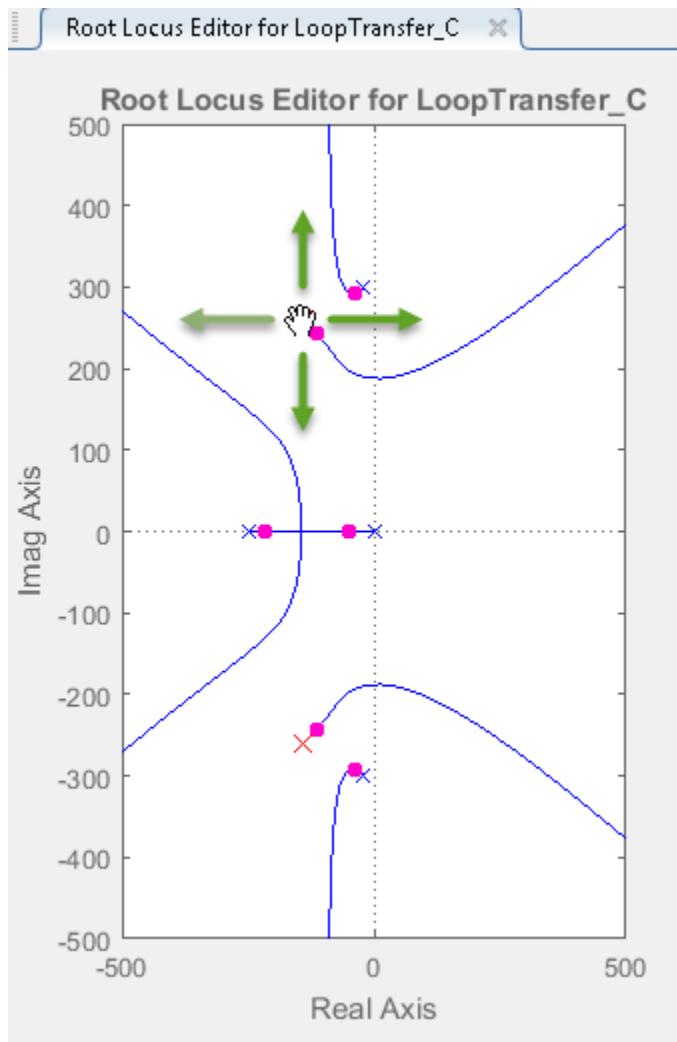
Add Poles to Compensator

To add a complex pole pair to the compensator, in the **Root Locus Editor**, right-click the plot area and select **Add Pole/Zero** > **Complex Pole**. Click the plot area where you want to add one of the complex poles.
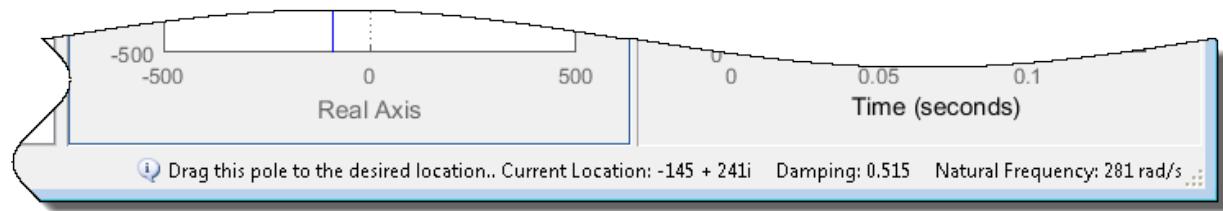
The app adds the complex pole pair to the root locus plot as red X's, and updates the step response plot.

In the **Root Locus Editor**, drag the new poles to locations near $-140 \pm 260i$. As you drag one pole, the other pole updates automatically.
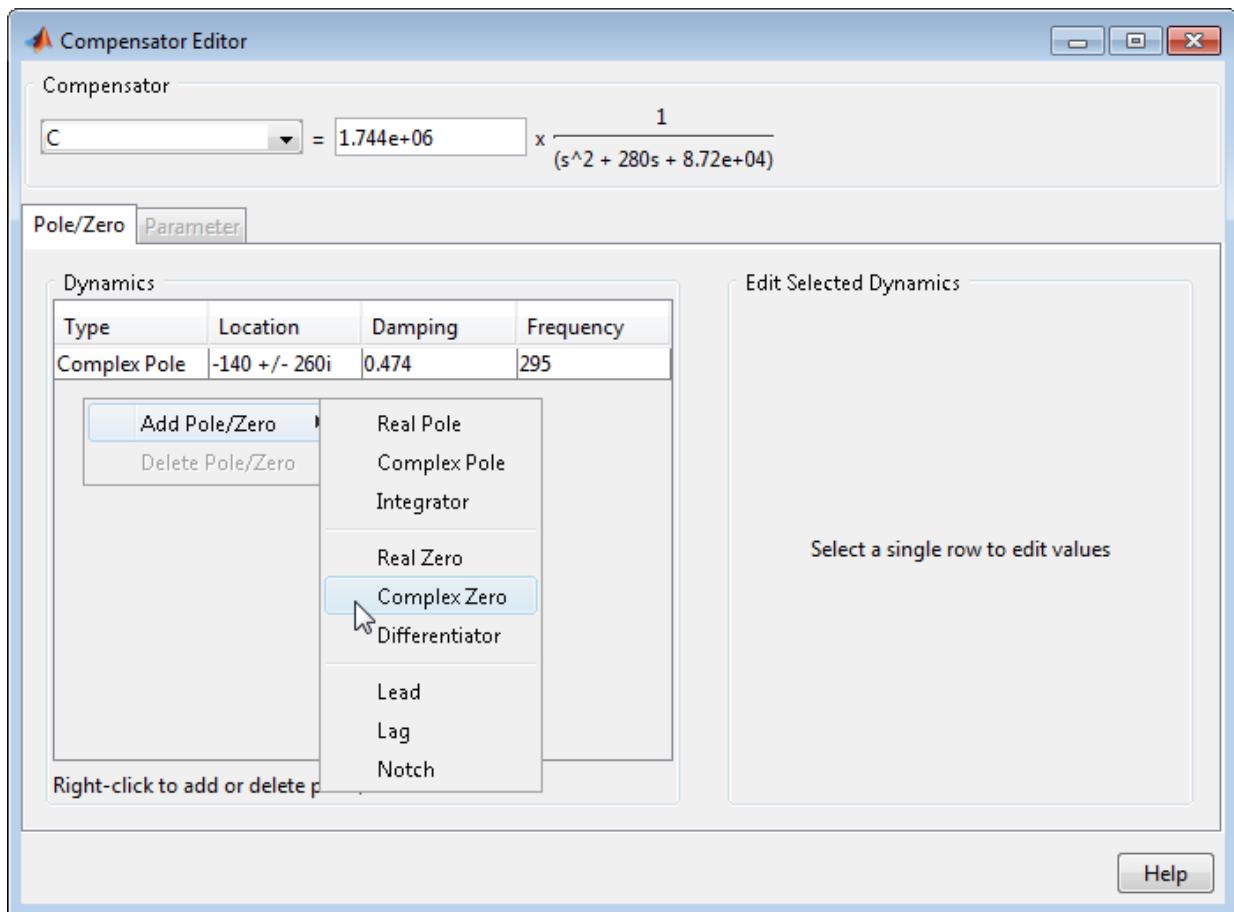
Tip As you drag a pole or zero, the app displays the new value in the status bar, on the right side.

-500
-500   0   500
Real Axis

0
0   0.05   0.1
Time (seconds)

ⓘ Drag this pole to the desired location.. Current Location: -145 + 241i   Damping: 0.515   Natural Frequency: 281 rad/s
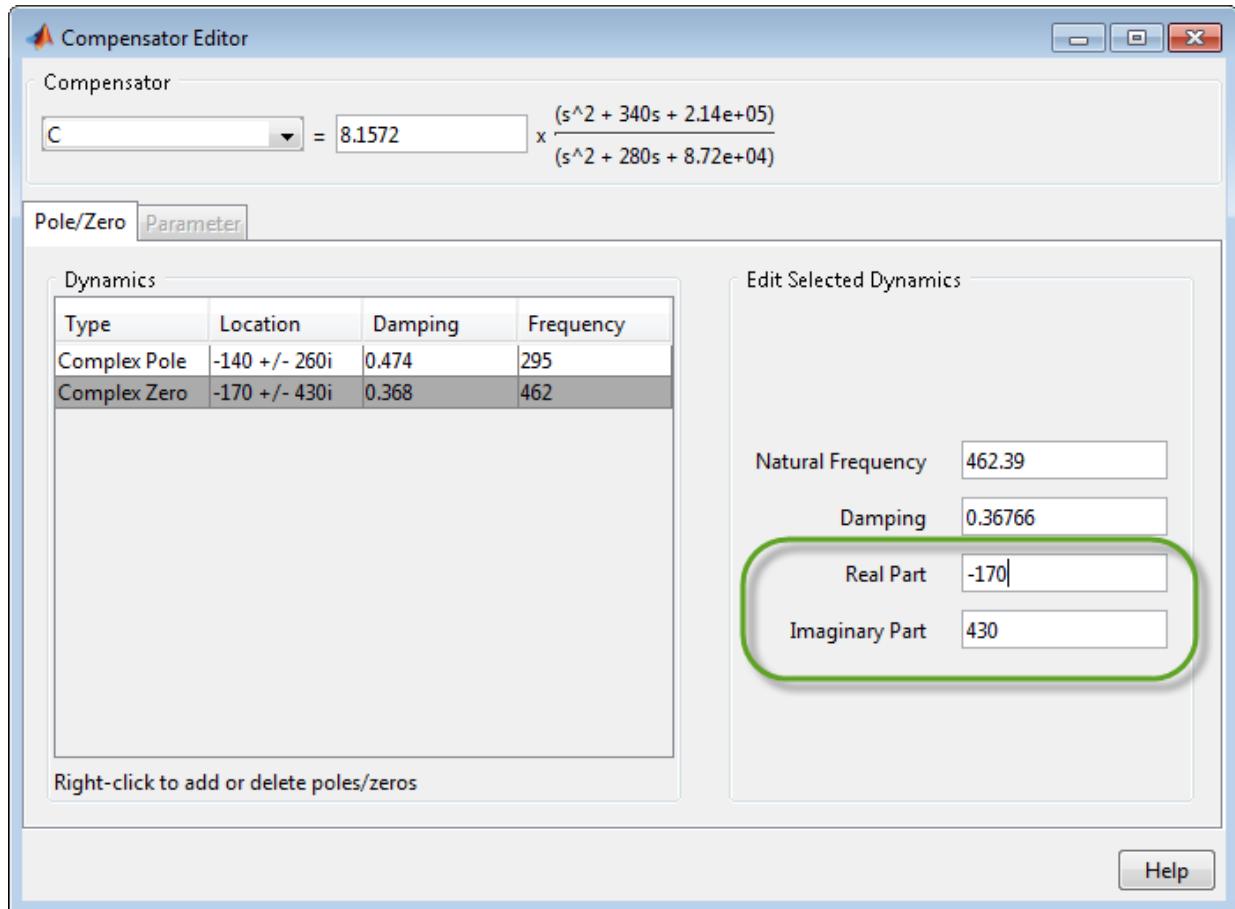
Add Zeros to Compensator

To add a complex zero pair to your compensator, in the Compensator Editor dialog box, right-click the **Dynamics** table, and select **Add Pole/Zero** > **Complex Zero**
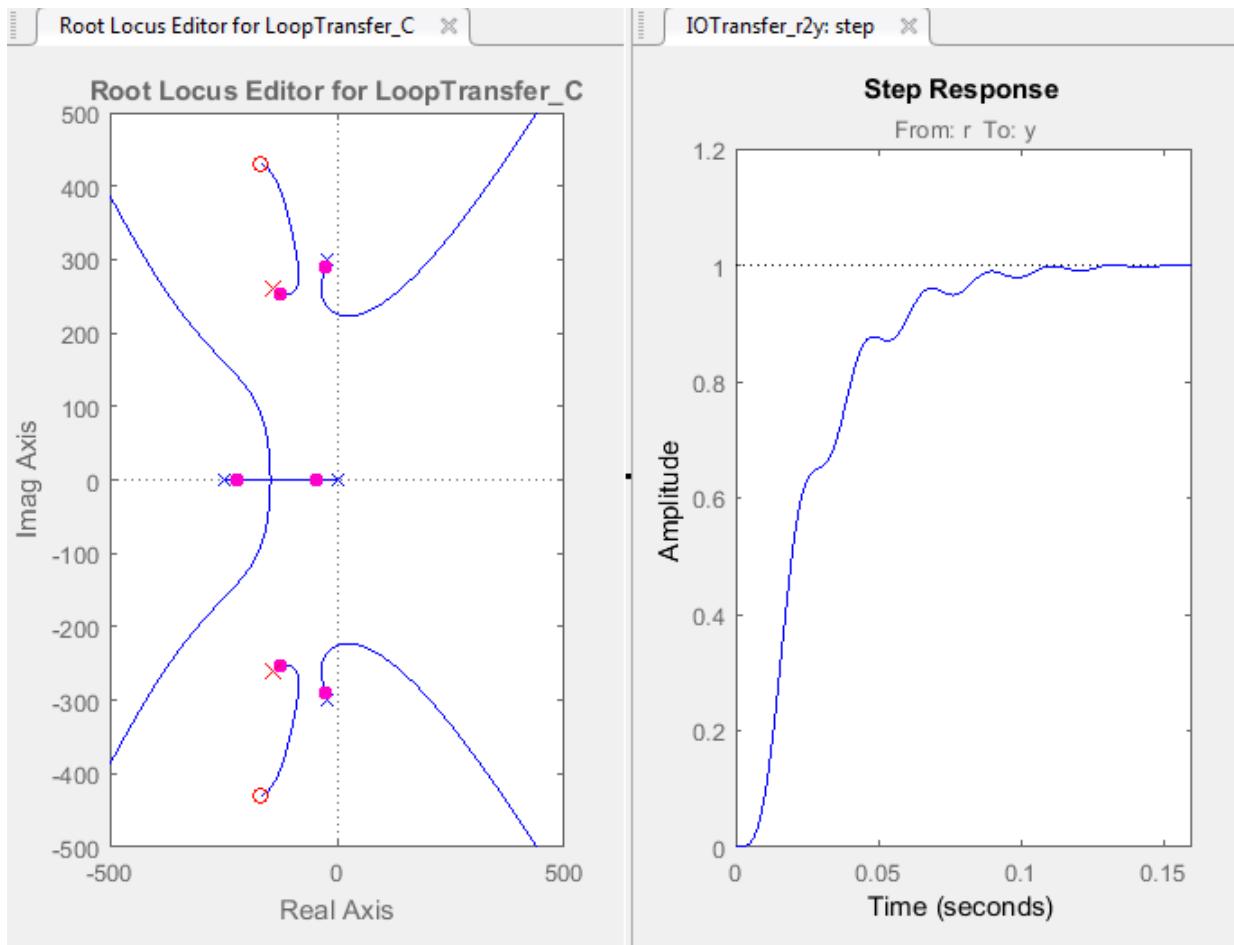
The app adds a pair of complex zeros at $-1 \pm i$ to your compensator

In the **Dynamics** table, click the **Complex Zero** row. Then in the **Edit Selected Dynamics** section, specify a **Real Part** of -170 and an **Imaginary Part** of 430.

The compensator and response plots automatically update to reflect the new zero locations.

In the **Step Response** plot, the settling time is around 0.1 seconds, which does not satisfy the design requirements.

Adjust Pole and Zero Locations
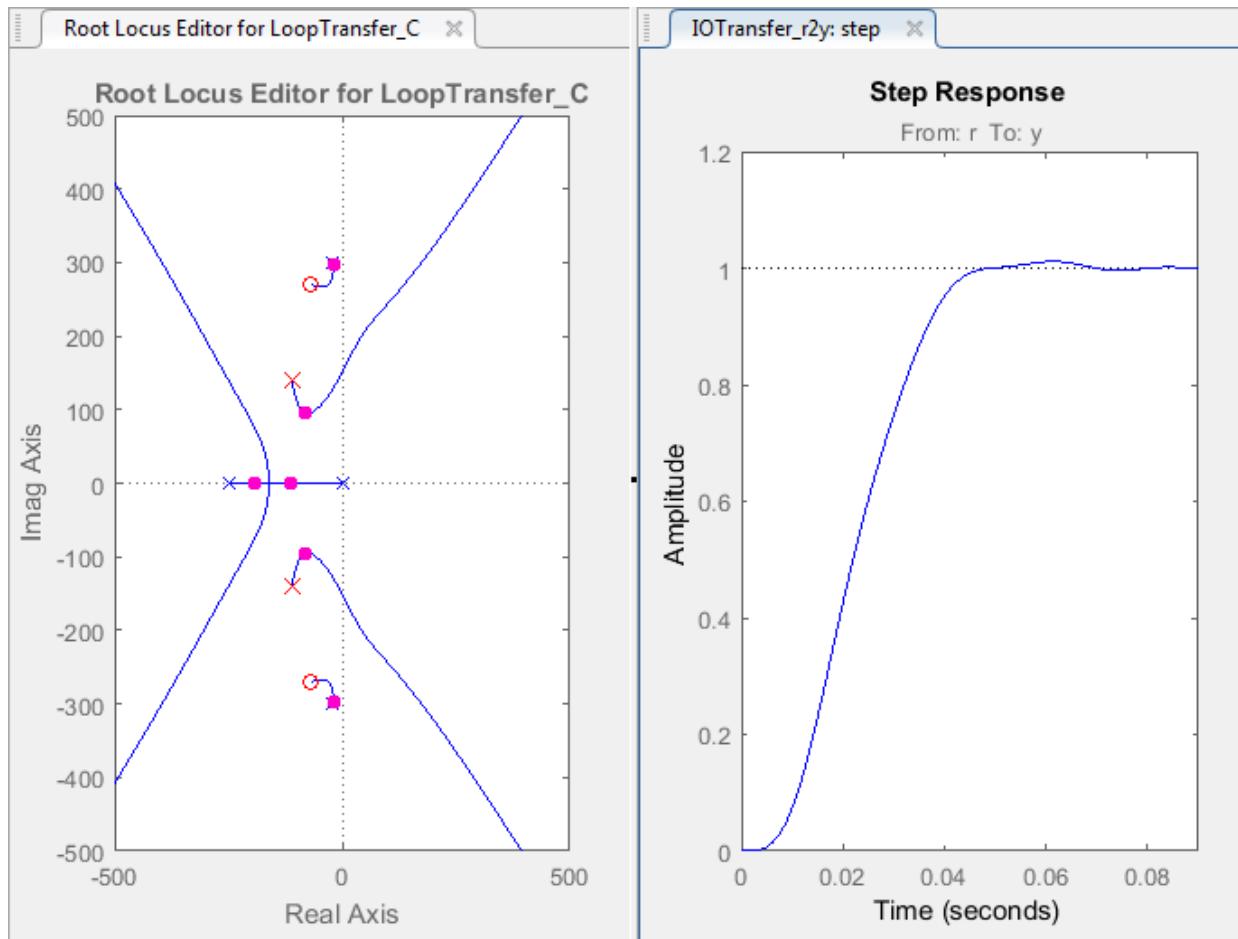
The compensator design process can involve some trial and error. Adjust the compensator gain, pole locations and, zero locations until you meet the design criteria.

One possible compensator design that satisfies the design requirements is:

- Compensator gain of 10
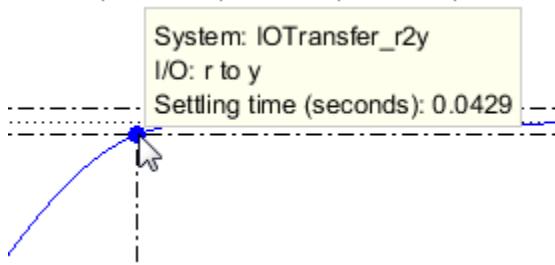
- Complex poles at $-110 \pm 140i$
- Complex zeros at $-70 \pm 270i$

In the Compensator Editor dialog box, configure your compensator using these values. In the **Step Response** plot, the settling time is around 0.05 seconds.

To verify the exact settling time, right-click the **Step Response** plot area and select **Characteristics > Settling Time**. A settling time indicator appears on the response plot.

To view the settling time, move the cursor over the settling time indicator.

The settling time is about 0.043 seconds, which satisfies the design requirements.

References

[1] Clark, R. N. *Control System Dynamics*, Cambridge University Press, 1996.

See Also

Control System Designer | `rlocusplot`

More About

- "Edit Compensator Dynamics" on page 12-90
- "Control System Designer Tuning Methods" on page 12-4
- "Bode Diagram Design" on page 12-48
- "Nichols Plot Design" on page 12-77

Nichols Plot Design

Nichols plot design is an interactive graphical method of modifying a compensator to achieve a specific open-loop response (loop shaping). Unlike "Bode Diagram Design" on page 12-48, Nichols plot design uses Nichols plots to view the open-loop frequency response. Nichols plots combine gain and phase information into a single plot, which is useful when you are designing to gain and phase margin specifications. You can also use the Nichols plot grid lines to estimate the closed-loop response (see `ngrid`). For more information on Nichols plots, see `nicholsplot`.

Tune Compensator For DC Motor Using Nichols Plot Graphical Design

This example shows how to design a compensator for a DC motor using Nichols plot graphical tuning techniques.

Plant Model and Requirements

The transfer function of the DC motor plant, as described in "SISO Example: The DC Motor", is:

$$G = \frac{1.5}{s^2 + 14s + 40.02}$$

For this example, the design requirements are:

- Rise time of less than 0.5 seconds
- Steady-state error of less than 5%
- Overshoot of less than 10%
- Gain margin greater than 20 dB
- Phase margin greater than 40 degrees

Open Control System Designer

At the MATLAB command line, create a transfer function model of the plant, and open **Control System Designer** in the **Nichols Editor** configuration.
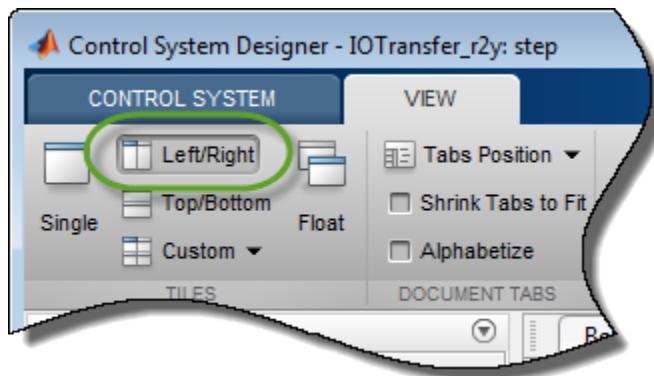
```
G = tf(1.5,[1 14 40.02]);
controlSystemDesigner('bode',G);
```

The app opens and imports G as the plant model for the default control architecture, **Configuration 1**.

In the app, the following response plots open:

- Open-loop **Nichols Editor** for the LoopTransfer_C response. This response is the open-loop transfer function *GC*, where *C* is the compensator and *G* is the plant.

- **Step Response** for the IOTransfer_r2y response. This response is the input-output transfer function for the overall closed-loop system.

Tip To open the open-loop **Nichols Editor** when **Control System Designer** is already open, on the **Control System** tab, in the **Tuning Methods** drop-down list, select **Nichols Editor**. In the Select Response to Edit dialog box, select an existing response to plot, or create a New Open-Loop Response.

To view the open-loop frequency response and closed-loop step response simultaneously, on the **Views** tab, click **Left/Right**.
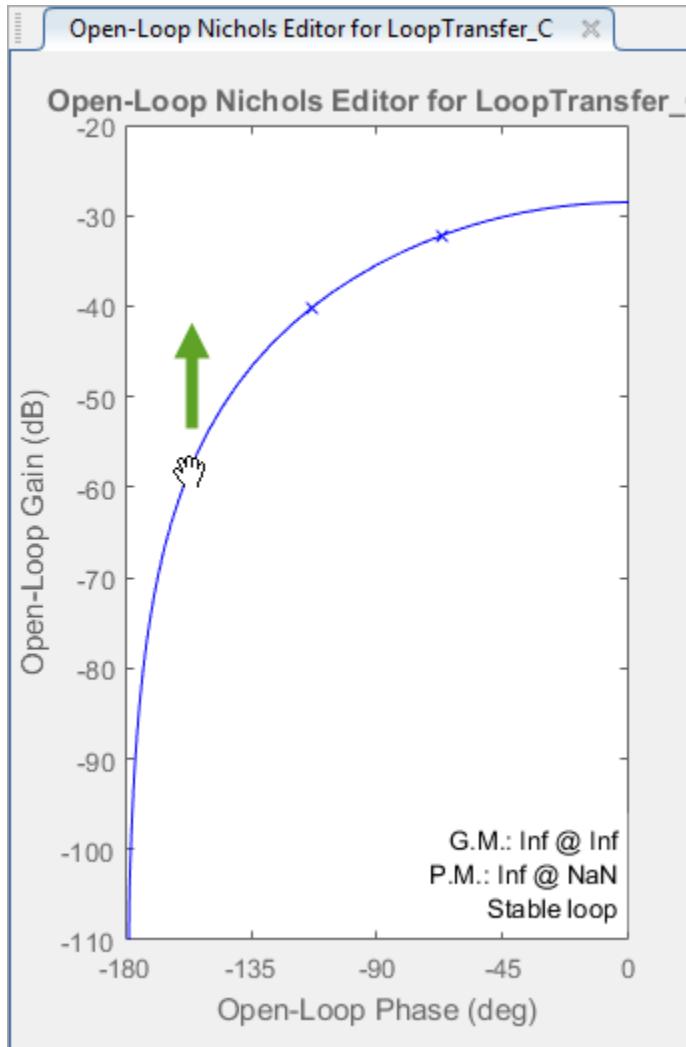
The app displays the **Nichols Editor** and **Step Response** plots side-by-side.
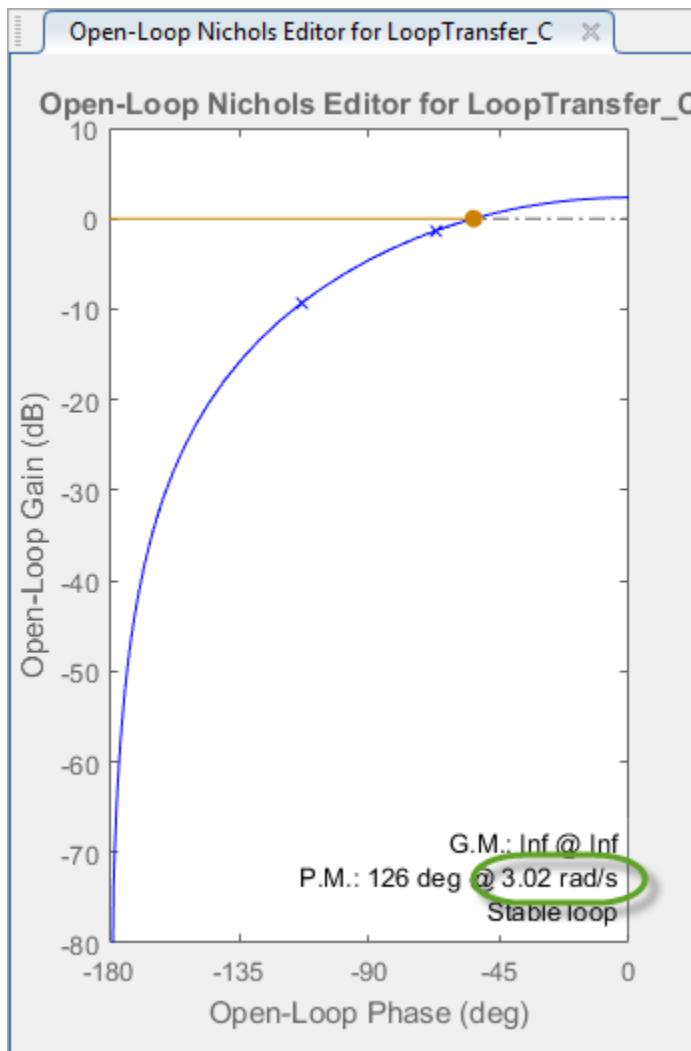
Adjust Bandwidth

Since the design requires a rise time less than 0.5 seconds, set the open-loop DC crossover frequency to about 3 rad/s. To a first-order approximation, this crossover frequency corresponds to a time constant of 0.33 seconds.

To adjust the crossover frequency increase the compensator gain. In the **Nichols Editor**, drag the response upward. Doing so increases the gain of the compensator.

As you drag the Nichols plot, the app computes the compensator gain and updates the response plots.
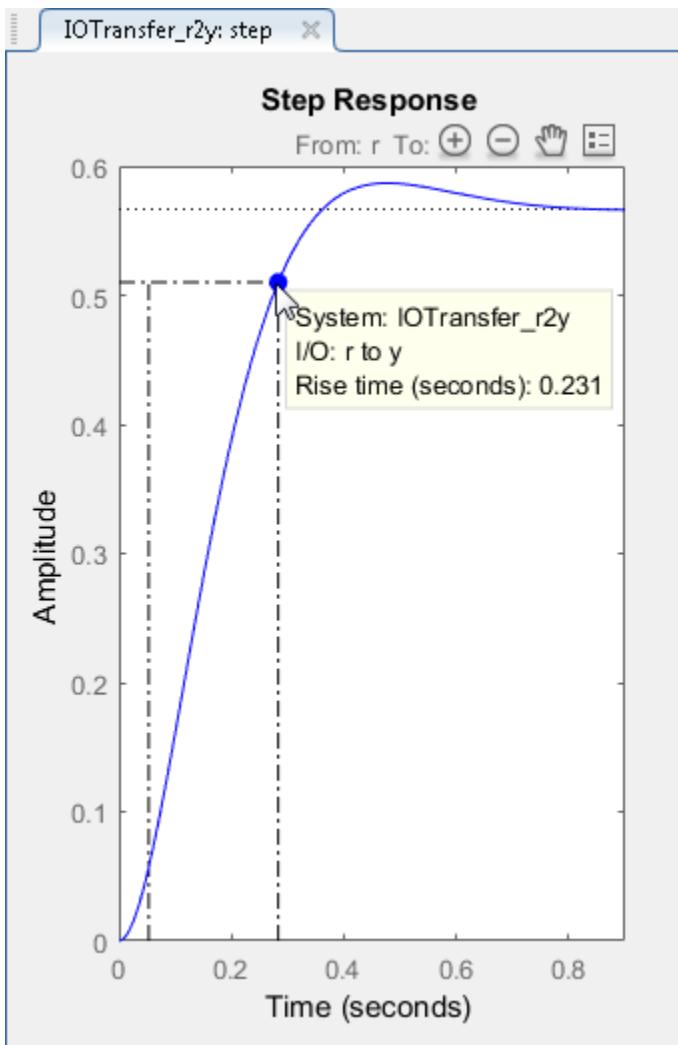
Drag the Nichols plot upward until the crossover frequency is about 3 rad/s.

View Step Response Characteristics

To add the rise time to the **Step Response** plot, right-click the plot area, and select **Characteristics** > **Rise Time**.

To view the rise time, move the cursor over the rise time indicator.

The rise time is around 0.23 seconds, which satisfies the design requirements.

Similarly, to add the peak response to the **Step Response** plot, right-click the plot area, and select **Characteristics** > **Peak Response**.

The peak overshoot is around 3.5%.

Add Integrator To Compensator

To meet the 5% steady-state error requirement, eliminate steady-state error from the closed-loop step response by adding an integrator to your compensator. In the **Nichols Editor** right-click in the plot area, and select **Add Pole/Zero > Integrator**.

Adding an integrator produces zero steady-state error. However, changing the compensator dynamics also changes the crossover frequency, increasing the rise time. To reduce the rise time, increase the crossover frequency to 3 rad/s.

Adjust Compensator Gain

To return the crossover frequency to 3 rad/s, increase the compensator gain further. Right-click the **Nichols Editor** plot area, and select **Edit Compensator**.

In the Compensator Editor dialog box, in the **Compensator** section, specify a gain of 99, and press **Enter**.

The response plots update automatically.

The rise time is around 0.4 seconds, which satisfies the design requirements. However, the peak overshoot is around 32%. A compensator consisting of a gain and an integrator is not sufficient to meet the design requirements. Therefore, the compensator requires additional dynamics.

Add Lead Network to Compensator
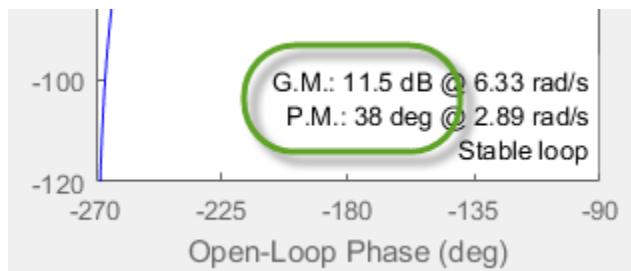
In the **Nichols Editor**, review the gain margin and phase margin for the current compensator design. The design requires a gain margin greater than 20 dB and phase

margin greater than 40 degrees. The current design does not meet either of these requirements.

To increase the stability margins, add a lead network to the compensator.
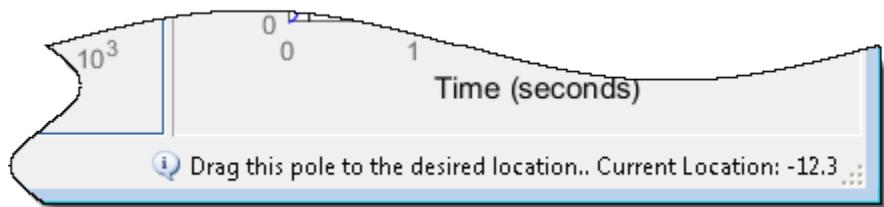
In the **Nichols Editor**, right-click and select **Add Pole/Zero** > **Lead**.

To specify the location of the lead network pole, click on the magnitude response. The app adds a real pole (red X) and real zero (red O) to the compensator and to the **Nichols Editor** plot.

In the **Nichols Editor**, drag the pole and zero to change their locations. As you drag them, the app updates the pole/zero values and updates the response plots.

To decrease the magnitude of a pole or zero, drag it towards the left. Since the pole and zero are on the negative real axis, dragging them to the left moves them closer to the origin in the complex plane.

Tip As you drag a pole or zero, the app displays the new value in the status bar, on the right side.

As an initial estimate, drag the zero to a location around -7 and the pole to a location around -11.

The phase margin meets the design requirements; however, the gain margin is still too low.

Edit Lead Network Pole and Zero

To improve the controller performance, tune the lead network parameters.

In the Compensator Editor dialog box, in the **Dynamics** section, click the **Lead** row.

In the **Edit Selected Dynamics** section, in the **Real Zero** text box, specify a location of -4.3, and press **Enter**. This value is near the slowest (left-most) pole of the DC motor plant.

In the **Real Pole** text box, specify a value of -28, and press **Enter**.

When you modify a lead network parameters, the **Compensator** and response plots update automatically.

In the app, in the **Nichols Editor**, the gain margin of 20.5 just meets the design requirement.

To add robustness to the system, in the Compensator Editor dialog box, decrease the compensator gain to 550, and press **Enter**. The gain margin increases to 21.8, and the response plots update.

In **Control System Designer**, in the response plots, compare the system performance to the design requirements. The system performance characteristics are:

- Rise time is 0.445 seconds.
- Steady-state error is zero.
- Overshoot is 3.39%.

- Gain margin is 21.8 dB.
- Phase margin is 65.6 degrees.

The system response meets all of the design requirements.

See Also

Control System Designer | `nicholsplot`

More About

- "Edit Compensator Dynamics" on page 12-90
- "Control System Designer Tuning Methods" on page 12-4
- "Bode Diagram Design" on page 12-48
- "Root Locus Design" on page 12-62

Edit Compensator Dynamics

In this section...
"Compensator Editor" on page 12-90
"Graphical Compensator Editing" on page 12-92
"Poles and Zeros" on page 12-93
"Lead and Lag Networks" on page 12-93
"Notch Filters" on page 12-94

Using **Control System Designer**, you can manually edit compensator dynamics to achieve your design goals. In particular, you can adjust the compensator gain, and you can add the following compensator dynamics:
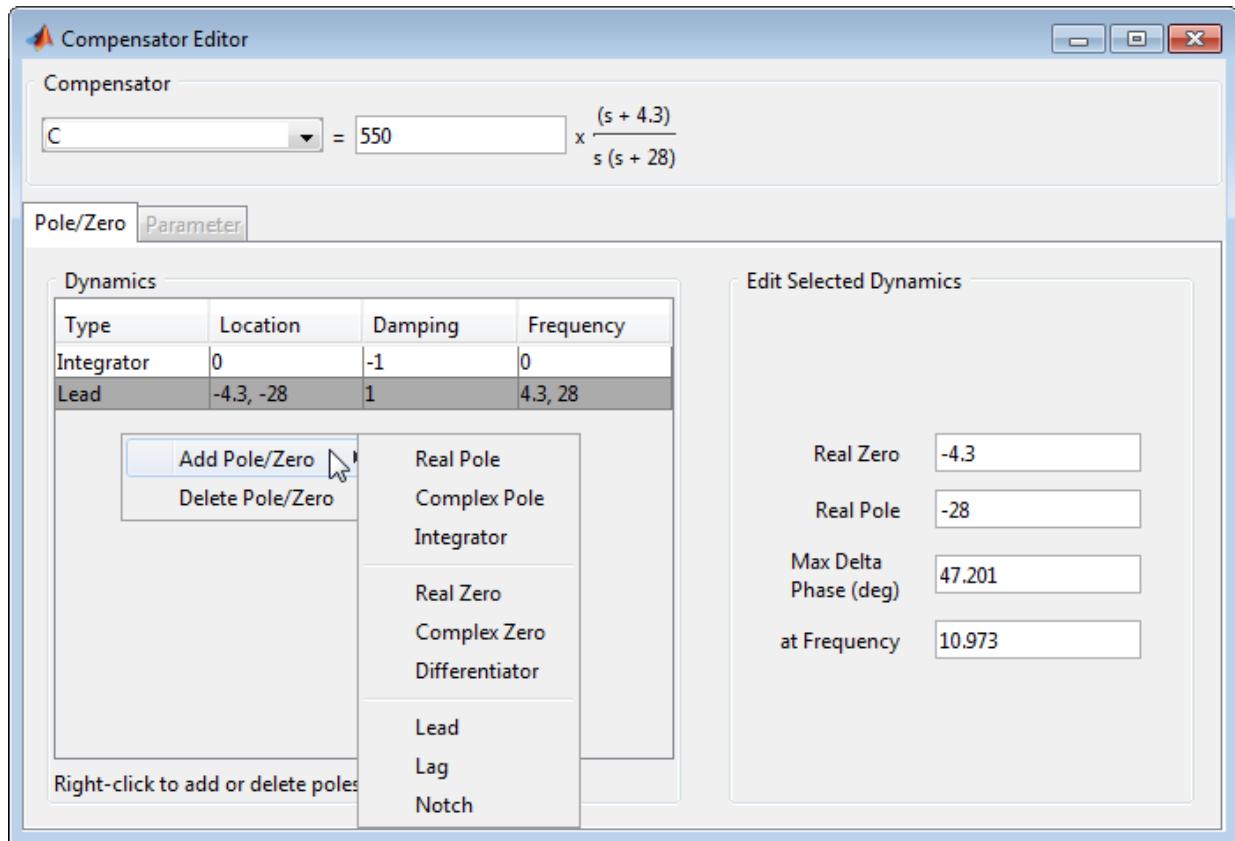
- Real and complex poles, including integrators
- Real and complex zeros, including differentiators
- Lead and lag networks
- Notch filters

You can add dynamics and modify compensator parameters using the Compensator Editor or using the graphical **Bode Editor**, **Root Locus Editor**, or **Nichols Editor** plots.

Compensator Editor

To open the Compensator Editor dialog box, in **Control System Designer**, in an editor plot area, right-click and select **Edit Compensator**. Alternatively, in the **Data Browser**, in the **Controllers** section, right-click the compensator you want to edit and click **Open Selection**.

To add poles and zeros to your compensator, right-click in the **Dynamics** table and, under **Add Pole/Zero**, select the type of pole/zero you want to add.

The app adds a pole or zero of the selected type with default parameters.

To edit a pole or zero, in the **Dynamics** table, click on the pole/zero type you want to edit. Then, in the **Edit Selected Dynamics** section, in the text boxes, specify the pole and zero locations.
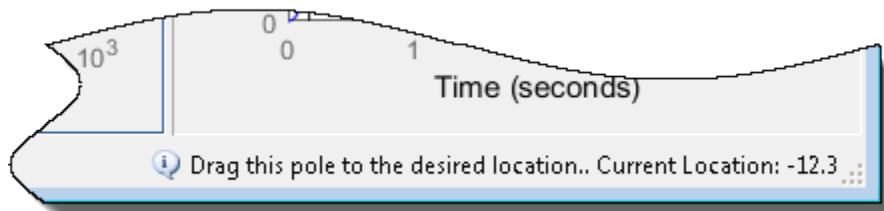
To delete poles and zeros, in the **Dynamics** table, click on the pole/zero type you want to delete. Then, right-click and select **Delete Pole/Zero**.

Graphical Compensator Editing
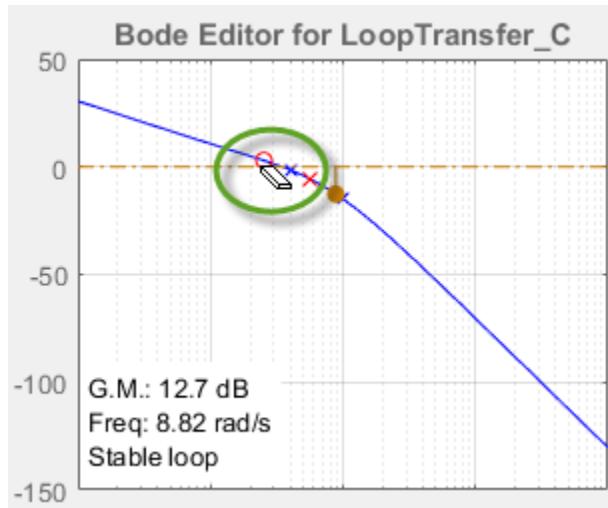
You can also add and adjust poles and zeros directly from Bode Editor, Root Locus Editor, or Nichols Editor plots. Use this method to roughly place poles and zeros in the correct area before fine-tuning their locations using the Compensator Editor.

To add poles and zeros directly from an editor plot, right-click the plot area and, under **Add Pole/Zero**, select the type of pole/zero you want to add. In the editor plot, the app displays the editable compensator poles and zeros as red X's and 0's respectively.

In the editor plots, you can drag poles and zeros to adjust their locations. As you drag a pole or zero, the app displays the new value in the status bar, on the right side.

To delete a pole or zero, right-click the plot area and select **Delete Pole/Zero**. Then, in the editor plot, click the pole or zero you want to delete.

Poles and Zeros

You can add the following poles and zeros to your compensator:

- Real pole/zero — Specify the pole/zero location on the real axis
- Complex poles/zeros — Specify complex conjugate pairs by:
    - Setting the real and imaginary parts directly.
    - Setting the natural frequency, ω_n, and damping ratio, ξ.
- Integrator — Add a pole at the origin to eliminate steady-state error for step inputs and DC inputs.
- Differentiator — Add a zero at the origin.

Lead and Lag Networks

You can add lead networks, lag networks, and combination lead-lag networks to your compensator.

Network Type	Description	Use This To
Lead	One pole and one zero on the negative real axis, with the zero having a lower natural frequency	- Increase stability margins - Increase system bandwidth - Reduce rise time
Lag	One pole and one zero on the negative real axis, with the pole having a lower natural frequency	- Reduce high-frequency gain - Increase phase margin - Improve steady-state accuracy
Lead-Lag	A combination of a lead network and a lag network	Combine the effects of lead and lag networks

To add a lead-lag network, add separate lead and lag networks.

To configure a lead or lag network for your compensator, use one of the following options:

- Specify the pole and zero locations. Placing the pole and zero further apart increases the amount of phase angle change.

- Specify the maximum amount of phase angle change and the frequency at which this change occurs. The app automatically computes the pole and zero locations.

When graphically changing pole and zero locations for a lead or lag compensator, in the editor plot, you can drag the pole and zeros independently.

Notch Filters

If you know that your system has disturbances at a particular frequency, you can add a notch filter to attenuate the gain of the system at that frequency. The notch filter transfer function is:

$$\frac{s^2 + 2\xi_1 \omega_n s + \omega_n^2}{s^2 + 2\xi_2 \omega_n s + \omega_n^2}$$
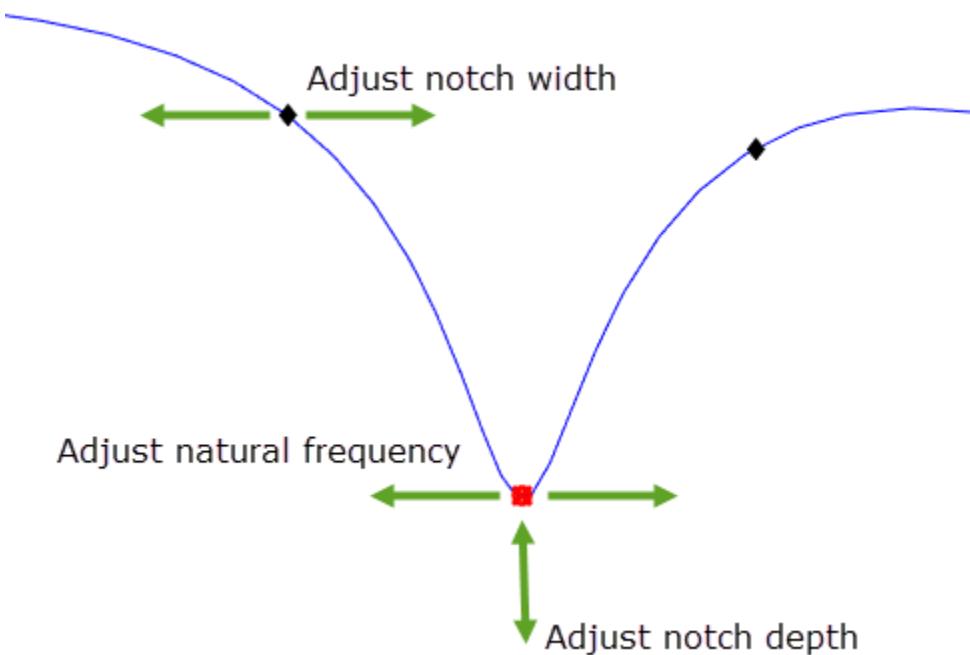
where

- ω_n is the natural frequency of the notch.
- The ratio ξ_2/ξ_1 sets the depth of the notch.

To configure a notch filter for your compensator, in the Compensator Editor dialog box, you can specify the:

- **Natural Frequency** — Attenuated frequency
- **Notch Depth** and **Notch Width**
- **Damping** for the complex poles and zeros of the transfer function.

When graphically editing a notch filter, in the Bode Editor, you can drag the bottom of the notch to adjust ω_n and the notch depth. To adjust the width of the notch without changing ω_n or the notch depth, you can drag the edges of the notch.

See Also

Control System Designer

More About

- "Bode Diagram Design" on page 12-48
- "Root Locus Design" on page 12-62
- "Nichols Plot Design" on page 12-77

Design Compensator Using Automated Tuning Methods
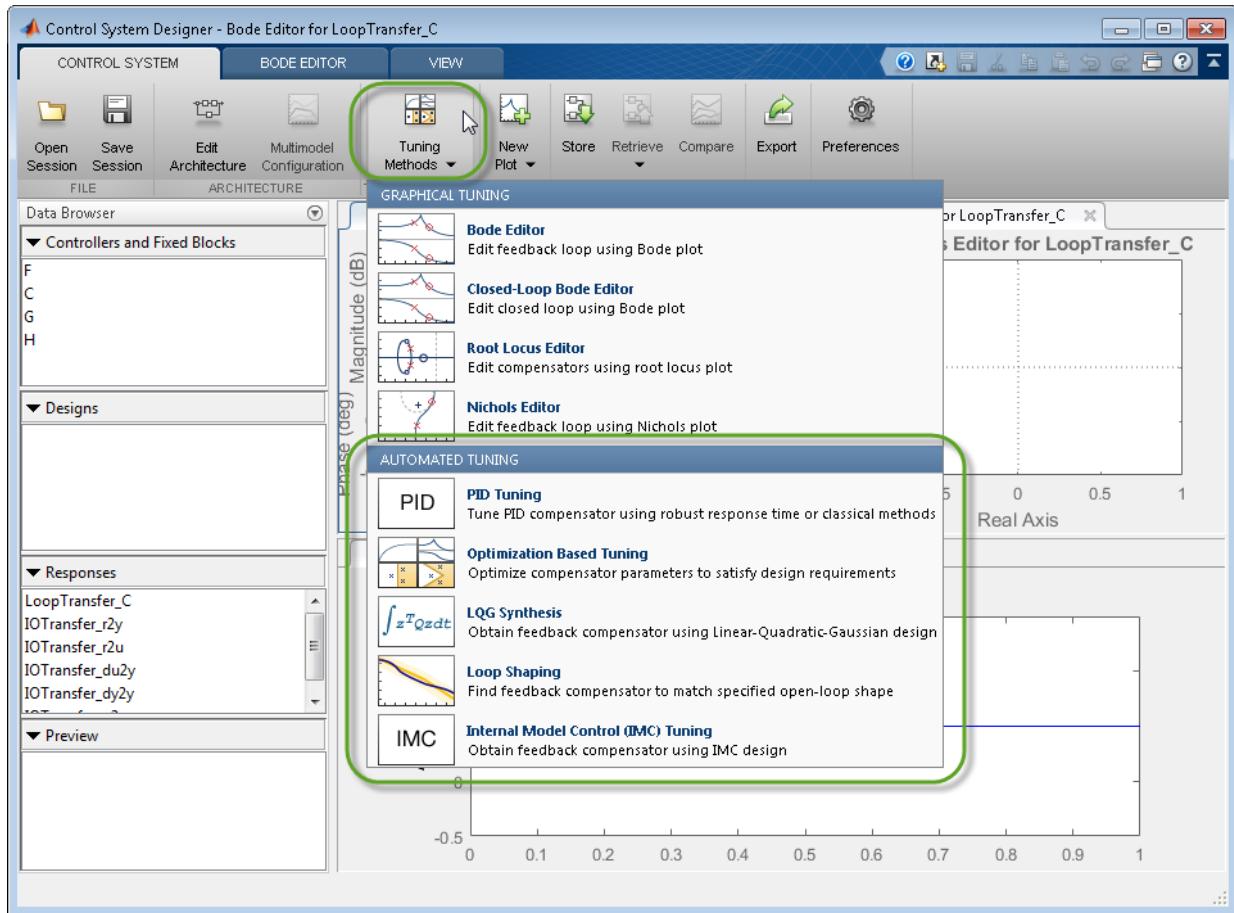
This example shows how to tune a compensator using automated tuning methods in **Control System Designer**.

In this section...
"Select Tuning Method" on page 12-96
"Select Compensator and Loop to Tune" on page 12-98
"PID Tuning" on page 12-99
"Optimization-Based Tuning" on page 12-105
"LQG Synthesis" on page 12-108
"Loop Shaping" on page 12-109
"Internal Model Control Tuning" on page 12-110

Select Tuning Method

To select an automated tuning method, in **Control System Designer**, click **Tuning Methods**.
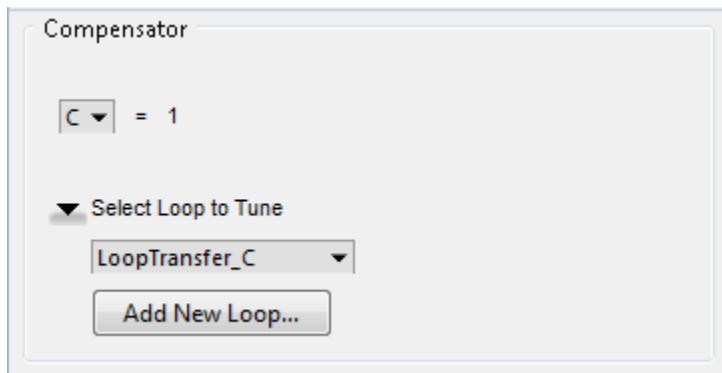
Select one of the following tuning methods:

- PID Tuning — Tune PID gains to balance performance and robustness or use classical tuning formulas.

- Optimization Based Tuning — Optimize compensator parameters using design requirements implemented in graphical tuning and analysis plots (requires Simulink Design Optimization software).

- LQG Synthesis — Design a full-order stabilizing feedback controller as a linear-quadratic-Gaussian (LQG) tracker.

- `Loop Shaping` — Find a full-order stabilizing feedback controller with a specified open-loop bandwidth or shape (requires Robust Control Toolbox software).
- `Internal Model Control (IMC) Tuning` — Obtain a full-order stabilizing feedback controller using the IMC design method.

Select Compensator and Loop to Tune

In the dialog box for your selected tuning method, in the **Compensator** section, select the compensator and loop to tune.

- **Compensator** — Select a compensator to tune from the drop-down list. The app displays the current compensator transfer function.
- **Select Loop to Tune** — Select an existing open-loop transfer function to tune from the drop-down list. You can select any open-loop transfer function from the **Data Browser** that includes the selected compensator in series
- **Add New Loop** — Create a new loop to tune. In the Open-Loop Transfer Function dialog box, select signals and loop openings to configure the loop transfer function.
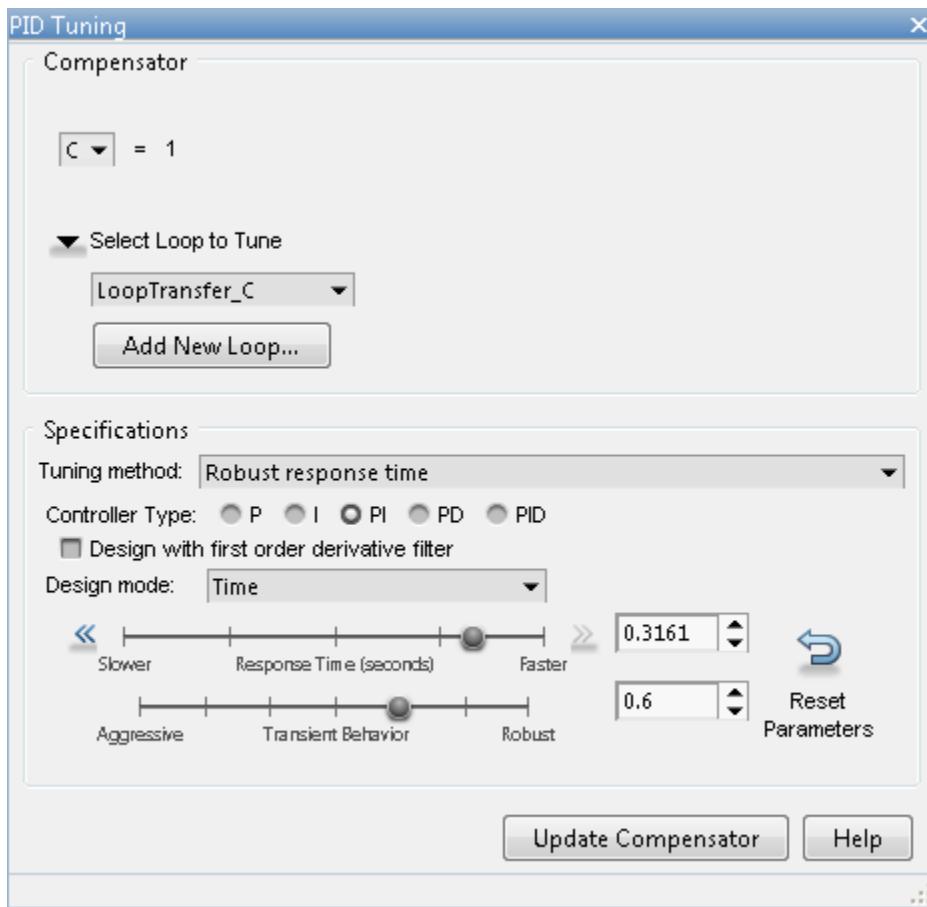
Note: For optimization-based tuning, you do not specify the compensator and loop to tune in this way. Instead, you define the compensator structure and select compensator and prefilter parameters to optimize. For more information, see "Select Tunable Compensator Elements".

PID Tuning

Using **Control System Designer**, you can automatically tune any of the following PID controller types:

- P — Proportional-only control
- I — Integral-only control
- PI — Proportional-integral control
- PD — Proportional-derivative control
- PDF — Proportional-derivative control with a low-pass filter on the derivative term
- PID — Proportional-integral-derivative control
- PIDF — Proportional-integral-derivative control with a low-pass filter on the derivative term

To open the PID Tuning dialog box, in **Control System Designer**, click **Tuning Methods**, and select **PID Tuning**.
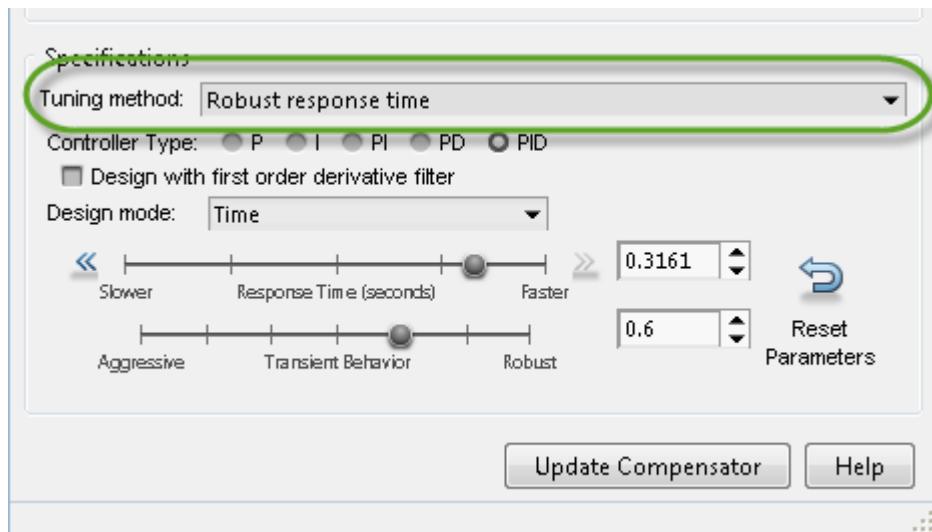
Robust Response Time

The robust response time algorithm automatically tunes PID parameters to balance performance and robustness. Using the robust response time method, you can:

- Tune all parameters for any type of PID controller.
- Design for plants that are stable, unstable, or integrating.

To tune your compensator using this method:

1   In the PID Tuning dialog box, in the **Specifications** section, in the **Tuning method** drop-down list, select Robust response time.
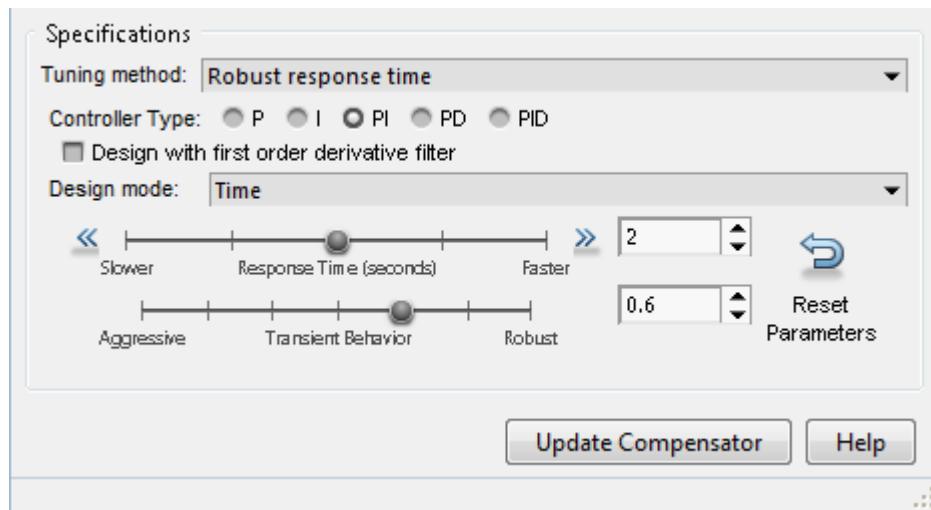
2 Select a **Controller type**. If you choose **PD** or **PID**, check **Design with first order derivative filter** to design a PDF or PIDF controller, respectively.
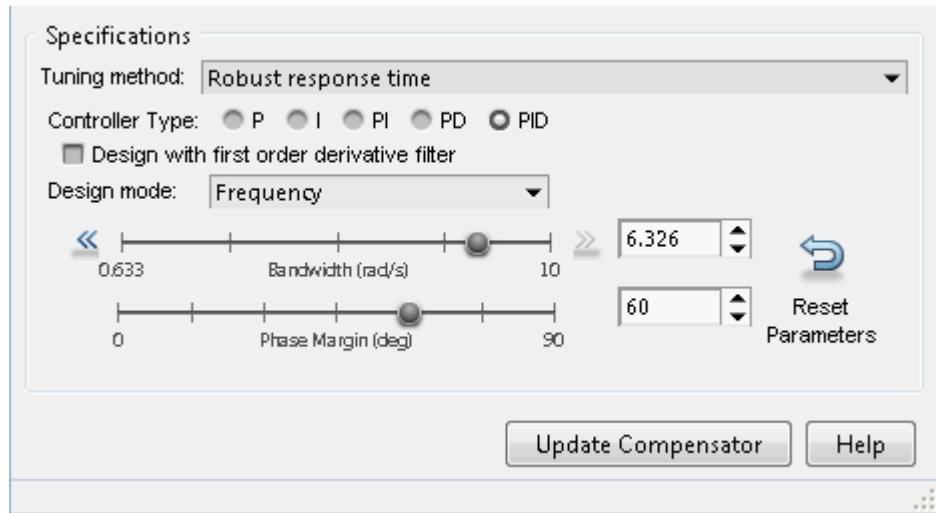
Tip Adding derivative action to the controller gives the algorithm more freedom to achieve both adequate phase margin and faster response time.

---

3 In the **Design mode** drop-down list, select one of the following:

- Time — Specify controller performance using time-domain parameters.

- **Response Time** — Specify a faster or slower controller response time. To modify the response time by a factor of ten, use the left or right arrows.

- **Transient Behavior** — Specify the controller transient behavior. You can make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.

- Frequency — Specify controller performance using frequency-domain parameters.

- **Bandwidth** — Specify the closed-loop bandwidth of the control system. To produce a faster response time, increase the bandwidth. To modify the bandwidth by a factor of ten, use the left or right arrows.

- **Phase Margin** — Specify a target phase margin for the system. To reduce overshoot and create a more robust controller, increase the phase margin.

4   To apply the specified controller design to the selected compensator, click **Update Compensator**.

Note: If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

---

5   By default, the app automatically computes controller parameters for balanced performance and robustness. To revert to these default parameters at any time, click **Reset Parameters**.
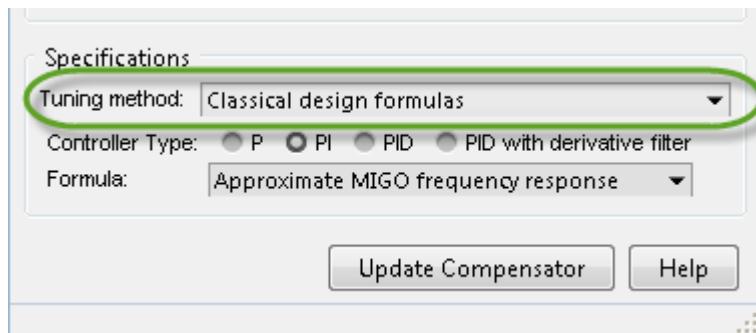
Classical Design Formulas

You can use classical PID design formulas to tune P, PI, PID, and PIDF controllers. These design formulas:

- Require a stable or integrating plant. For more information about the effective plant seen by the compensator, see "Effective Plant for Tuning" on page 12-6.

- Cannot tune the derivative filter. If you select a PIDF controller, the classical design methods set the filter time constant to $Td/10$, where Td is the tuned derivative time.

To tune your compensator using a classical method:
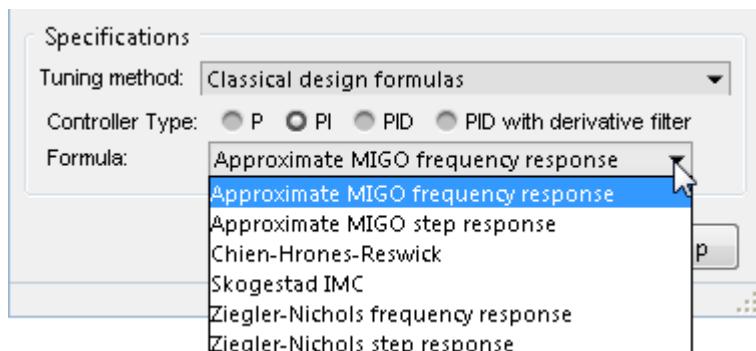
1   In the PID Tuning dialog box, in the **Specifications** section, in the **Tuning method** drop-down list, select `Classical design formulas`.

2   Select a **Controller type**.

Tip Adding derivative action to the compensator gives the algorithm more freedom to achieve both adequate phase margin and faster response time.

3   In the **Formula** drop-down list, select a classical design formula.

- `Approximate MIGO frequency response` — Compute controller parameters using closed-loop, frequency-domain, approximate M-constrained integral gain optimization (see [1]).

- `Approximate MIGO step response` — Compute controller parameters using open-loop, time-domain, approximate M-constrained integral gain optimization (see [1]).

- `Chien-Hrones-Reswick` — Approximate the plant as a first-order model with a time delay, and compute PID parameters using a Chien-Hrones-Reswick lookup table for 0% overshoot and disturbance rejection (see [2]).

- `Skogestad IMC` — Approximate the plant as a first-order model with a time delay, and compute PID parameters using Skogestad design rules (see [3]).

> **Note:** This method is different from selecting "Internal Model Control Tuning" on page 12-110 as the full-order compensator tuning method.

- `Ziegler-Nichols frequency response` — Compute controller parameters from a Ziegler-Nichols lookup table, based on the ultimate gain and frequency of the system (see [2]).

- `Ziegler-Nichols step response` — Approximate the plant as a first-order model with a time delay, and compute PID parameters using the Ziegler-Nichols design method (see [2]).

4 Apply the specified controller design to the selected compensator. Click **Update Compensator**.

> **Note:** If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

Optimization-Based Tuning
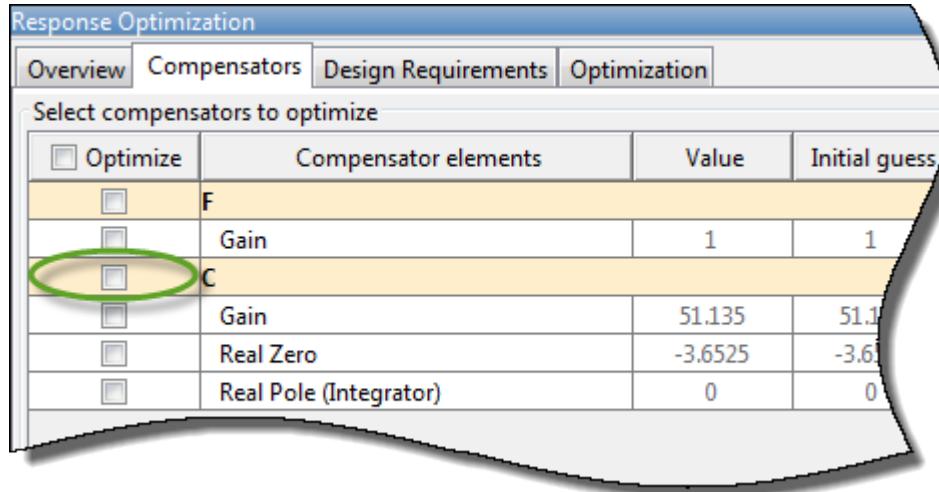
Optimization-based tuning is available only if you have Simulink Design Optimization software installed. You can use this method to design control systems for LTI models by optimizing controller parameters.

Note: Optimization-based tuning only changes the values of controller parameters and not the controller structure itself. For information on adding or removing compensator elements, see "Edit Compensator Dynamics" on page 12-90.

To design a controller using optimization-based tuning:

1   Define the structure of the compensators you want to tune. Typically, you design an initial controller either manually or using a different automated tuning method.

2   Open the Response Optimization dialog box. In **Control System Designer**, click **Tuning Methods**, and select **Optimization-Based Tuning**.

3   Select compensator parameters to optimize. On the **Compensators** tab, in the **Optimize** column, select the compensator elements to tune.

   You can optimize elements for any compensator listed in the **Data Browser**. To select all elements for a given compensator, select the check box that corresponds to that compensator.
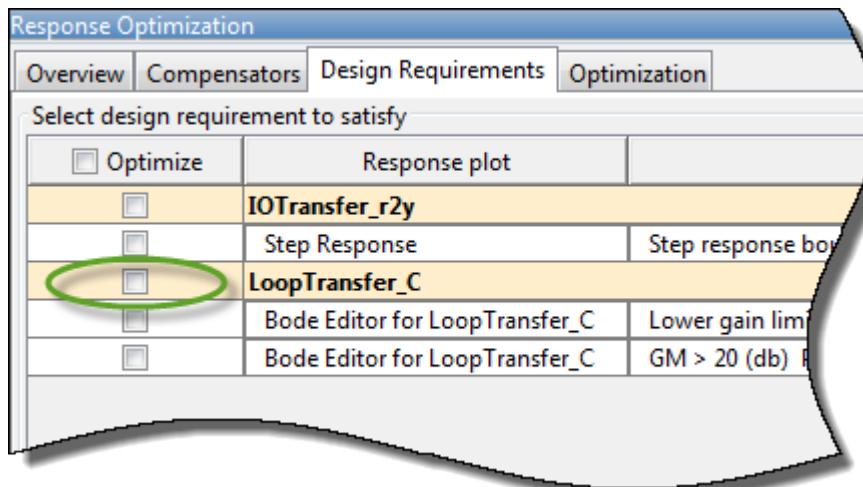
   Any elements that you do not select in the **Optimize** column remain at their current values during optimization.

4   For each compensator element, specify:

- **Initial guess** — Starting point for the optimization algorithm. To use the current element **Value** as the **Initial guess**, click a row in the table, and click **Use Value as Initial guess**.

- **Minimum** and **Maximum** bounds on the element value. The optimization constrains the search results to the specified range.

- **Typical value** scaling factor for normalizing the compensator elements.

5 On the **Design Requirements** tab, in the **Optimize** column, select the design requirements to satisfy during optimization.

Each design requirement is associated with a plot of a specific response in the **Data Browser**. To select all requirements for a given response, select the corresponding check box.

For information on adding and editing design requirements, see "Design Requirements" on page 12-9.
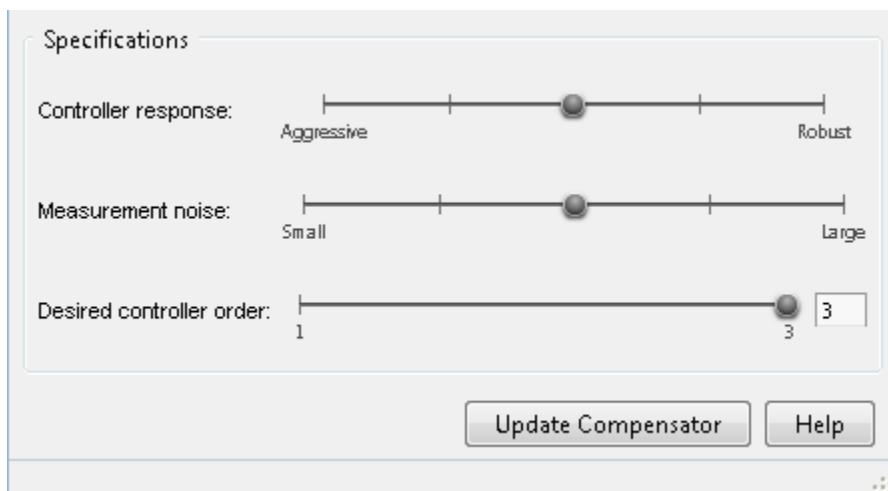
6 (optional) Configure optimization options. On the **Optimization** tab, click **Optimization options**.

7 Click **Start Optimization**.

For examples of optimization-based tuning, see "Optimize LTI System to Meet Frequency-Domain Requirements" and "Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)".

LQG Synthesis

Linear-quadratic-Gaussian (LQG) control is a technique for designing optimal dynamic regulators and setpoint trackers. This technique allows you to trade off performance and control effort, and to take into account process disturbances and measurement noise.

LQG synthesis generates a full-order feedback controller that guarantees closed-loop stability. The designed controller contains an integrator, which guarantees zero steady-state error for plants without a free differentiator.

To design an LQG controller:

1   Open the LQG Synthesis dialog box. In **Control System Designer**, click **Tuning Methods**, and select **LQG Synthesis**.

2   Specify the transient behavior of the controller using the **Controller response** slider. You can make the controller more aggressive at disturbance rejection or more robust against plant uncertainty. If you believe your model is accurate and that the manipulated variable has a large enough range, an aggressive controller is preferable.

3   Specify an estimate of the level of output measurement noise for your application using the **Measurement noise** slider. To produce a more robust controller, specify a larger noise estimate.

4   Specify your controller order preference using the **Desired controller order** slider. The maximum controller order is dependent on the effective plant dynamics.

5   Apply the specified controller design to the selected compensator. Click **Update Compensator**.

Note: If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

For an example of LQG synthesis using **Control System Designer**, see "Design LQG Tracker Using Control System Designer" on page 12-168.

Loop Shaping

If you have Robust Control Toolbox software installed, you can use loop shaping to design SISO compensators in **Control System Designer**. Loop shaping generates a stabilizing feedback controller to match, as closely as possible, a target loop shape. You can specify this loop shape as a bandwidth or an open-loop frequency response.

To design a controller using loop shaping:

1   Open the Loop Shaping dialog box. In **Control System Designer**, click **Tuning Methods**, and select **Loop Shaping**.

2   Select one of the following tuning preferences:

   · **Target bandwidth** — Specify a **Target open-loop bandwidth**, ω_b , to produce

     a loop shape of the specified bandwidth over an integrator, $\frac{\omega_b}{s}$.

   · **Target loop shape** — Specify the **Target open-loop shape** as a `tf`, `ss`, or `zpk` object. To limit the frequencies over which to match the target loop shape, specify the **Frequency range for loop shaping** as a two-element row vector.

3   Specify your controller order preference using the **Desired controller order** slider. The maximum controller order is dependent on the effective plant dynamics.

4 Apply the specified controller design to the selected compensator. Click **Update Compensator**.

Note: If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.
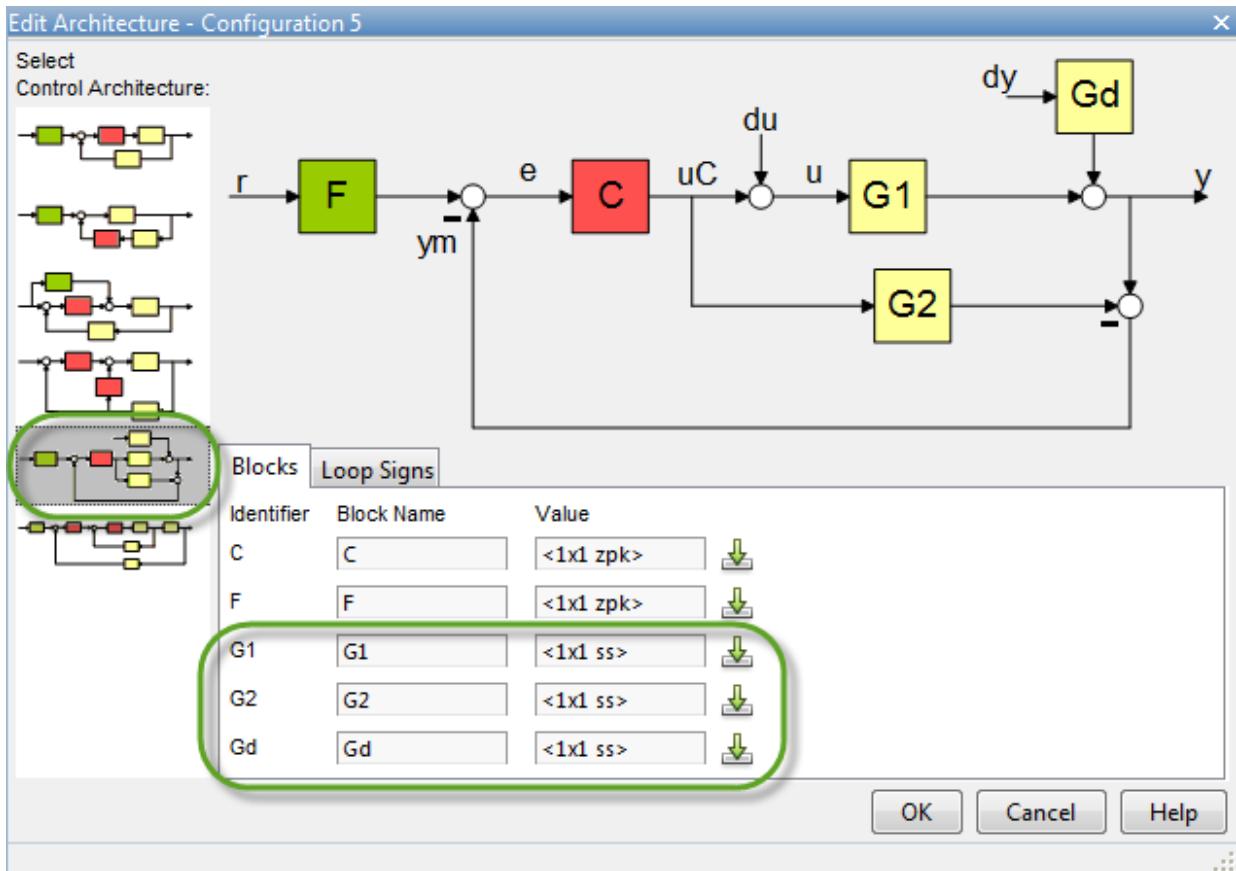
Internal Model Control Tuning

Internal model control (IMC) uses a predictive model of the plant dynamics to compute control actions.. IMC design generates a full-order feedback controller that guarantees closed-loop stability when there is no model error. The tuned compensator also contains an integrator, which guarantees zero steady-state offset for plants without a free differentiator. You can use this tuning method for both stable and unstable plants.

To design an IMC controller:
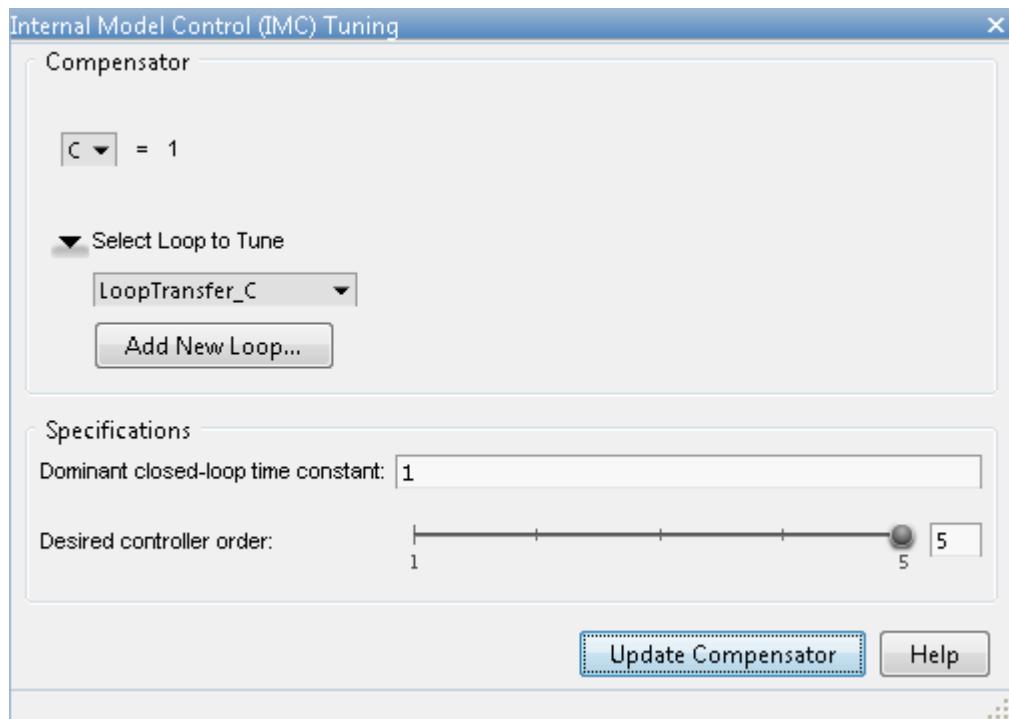
1 Select and configure the IMC control architecture. In **Control System Designer**, click **Edit Architecture**.

In the Edit Architecture dialog box, select the fifth control architecture, and import the plant model, **G1**, predictive model, **G2**, and disturbance model **Gd**.

Click **OK**.

2 Open the Internal Model Control (IMC) Tuning dialog box. In **Control System Designer**, click **Tuning Methods**, and select **Internal Model Control (IMC) Tuning**.

3. Specify a **Dominant closed-loop time constant**. The default value is 5% of the open-loop settling time. In general, increasing this value slows down the closed-loop system and makes it more robust.

4. Specify your controller order preference using the **Desired controller order** slider. The maximum controller order is dependent on the effective plant dynamics.

5. Apply the specified controller design to the selected compensator. Click **Update Compensator**.

Note: If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

For an example of IMC tuning, see "Design Internal Model Controller for Chemical Reactor Plant" on page 12-153.

References

[1] Åström, K. J. and Hägglund, T. "Replacing the Ziegler-Nichols Tuning Rules." Chapter 7 in *Advanced PID Control*, Research Triangle Park, NC: Instrumentation, Systems, and Automation Society, 2006, pp. 233–270.

[2] Åström, K. J. and Hägglund, T. "Ziegler-Nichols' and Related Methods." Section 6.2 in *Advanced PID Control*, Research Triangle Park, NC: Instrumentation, Systems, and Automation Society, 2006, pp. 167–176.

[3] Skogestad, S., "Simple analytic rules for model reduction and PID controller tuning." *Journal of Process Control*, Vol. 13, No. 4, 2003, pp. 291–309.

See Also

Control System Designer

More About

- "Control System Designer Tuning Methods" on page 12-4
- "Design Internal Model Controller for Chemical Reactor Plant" on page 12-153
- "Design LQG Tracker Using Control System Designer" on page 12-168

Analyze Designs Using Response Plots

This example shows how to use the plotting tools in **Control System Designer** to analyze your control system designs. There are two types of **Control System Designer** plots:

- Analysis plots — Use these plots to visualize your system performance and display response characteristics.
- Editor plots — Use these plots to visualize your system performance and interactively tune your compensator dynamics using graphical tuning methods.

In this section...
"Analysis Plots" on page 12-114
"Editor Plots" on page 12-117
"Plot Characteristics" on page 12-118
"Plot Tools" on page 12-119
"Design Requirements" on page 12-121
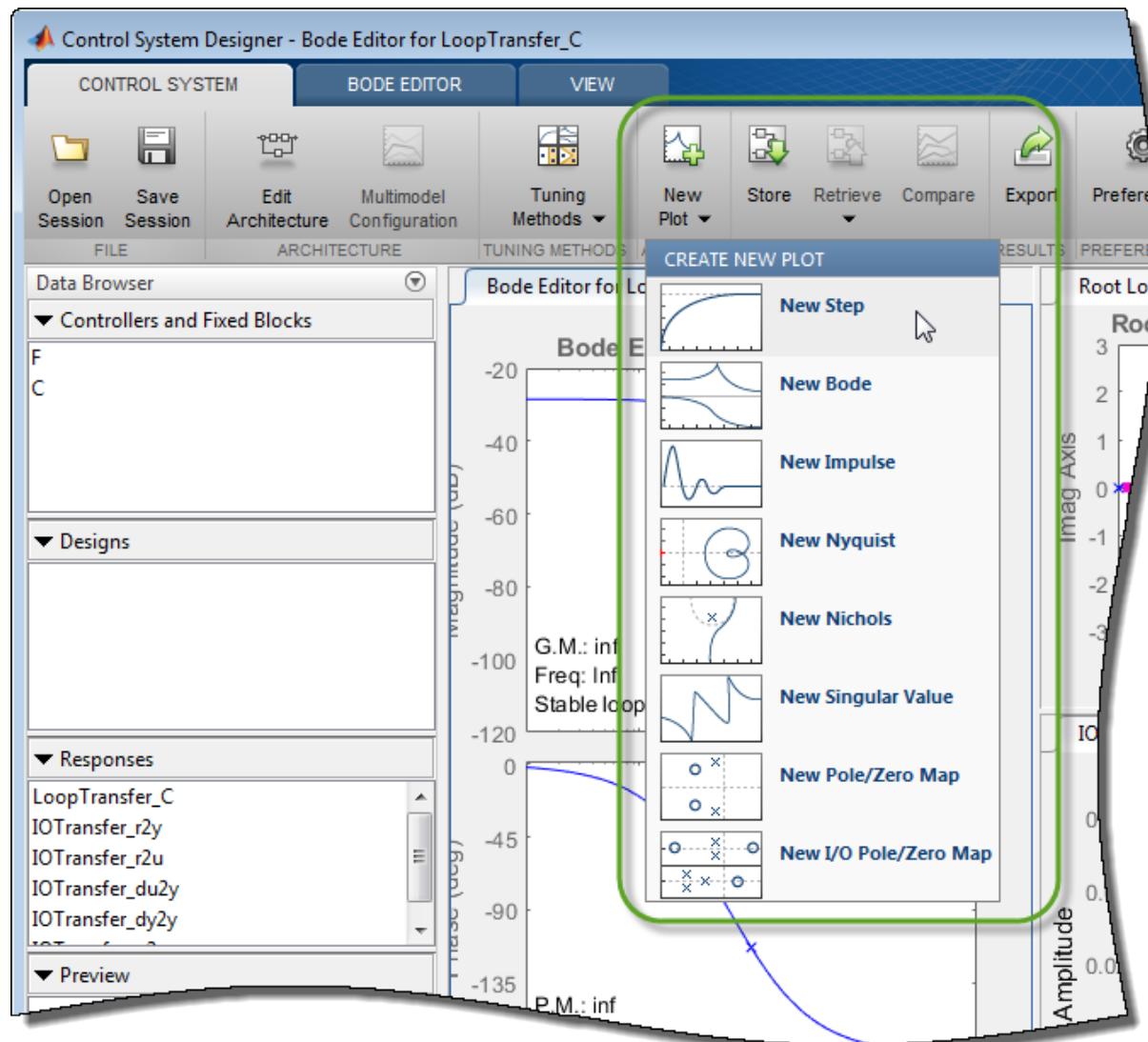
Analysis Plots

Use analysis plots to visualize your system performance and display response characteristics. You can also use these plots to compare multiple control system designs. For more information, see "Compare Performance of Multiple Designs" on page 12-124.

To create a new analysis plot, in **Control System Designer**, on the **Control System** tab, click **New Plot**, and select the type of plot to add.

In the new plot dialog box, specify an existing or new response to plot.

Note: Using analysis plots, you can compare the performance of multiple designs stored in the **Data Browser**. For more information, see "Compare Performance of Multiple Designs" on page 12-124.

Plot Existing Response

To plot an existing response, in the **Select Response to Plot** drop-down list, select an existing response from the **Data Browser**. The details for the selected response are displayed in the text box.

To plot the selected response, click **Plot**.

Plot New Response

To plot a new response, specify the following:

- **Select Response to Plot** — Select the type of response to create.

  - `New Input-Output Transfer Response` — Create a transfer function response for specified input signals, output signals, and loop openings.

  - `New Open-Loop Response` — Create an open-loop transfer function response at a specified location with specified loop openings.

  - `New Sensitivity Transfer Response` — Create a sensitivity response at a specified location with specified loop openings.

- **Response Name** — Enter a name in the text box.

- Signal selection boxes — Specify signals as inputs, outputs, or loop openings by clicking ➕. If you open **Control System Designer** from:

  - MATLAB — Select a signal using the **Architecture** block diagram for reference.

  - Simulink — Select an existing signal from the current control system architecture, or add a signal from the Simulink model.
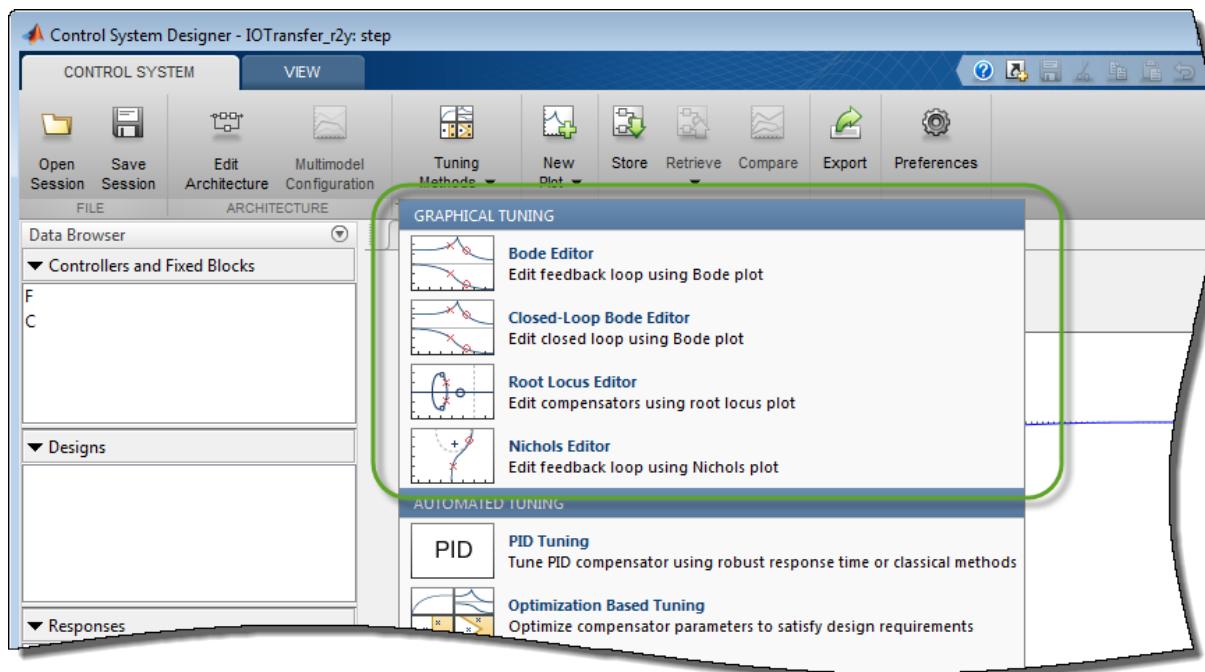
  Use ⬆, ⬇, and ✖ to reorder and delete signals.

To add the specified response to the **Data Browser** and create the selected plot, click **Plot**.

Editor Plots

Use editor plots to visualize your system performance and interactively tune your compensator dynamics using graphical tuning methods.

To create a new editor plot, in **Control System Designer**, on the **Control System** tab, click **Tuning Methods**, and select one of the **Graphical Tuning** methods.

For examples of graphical tuning using editor plots, see:
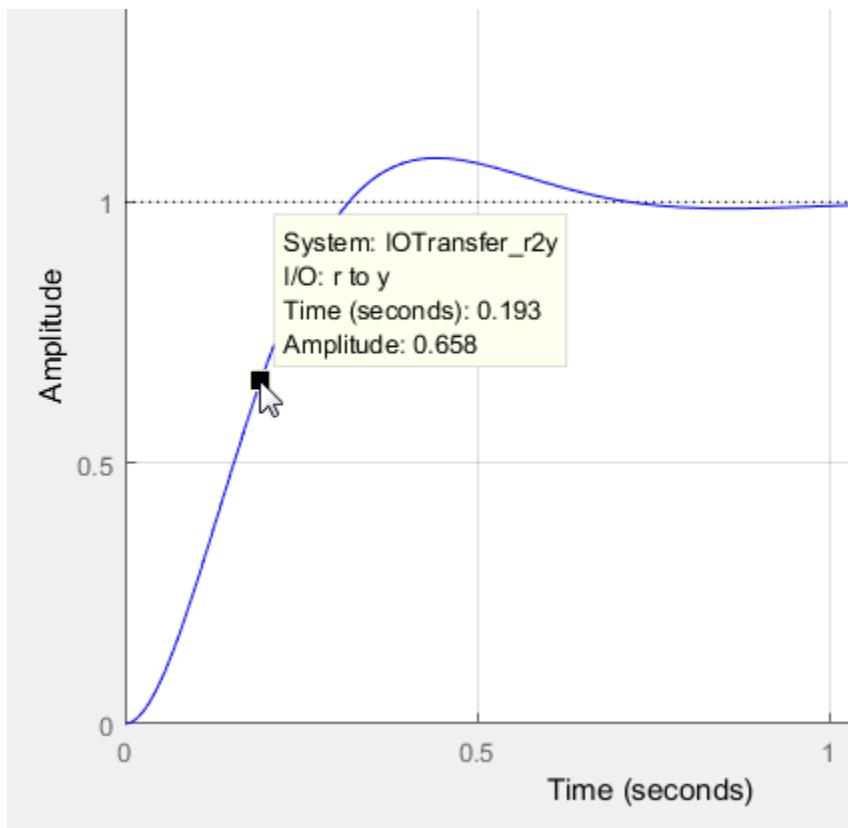
- "Bode Diagram Design" on page 12-48
- "Root Locus Design" on page 12-62
- "Nichols Plot Design" on page 12-77

For more information on interactively editing compensator dynamics, see "Edit Compensator Dynamics" on page 12-90.
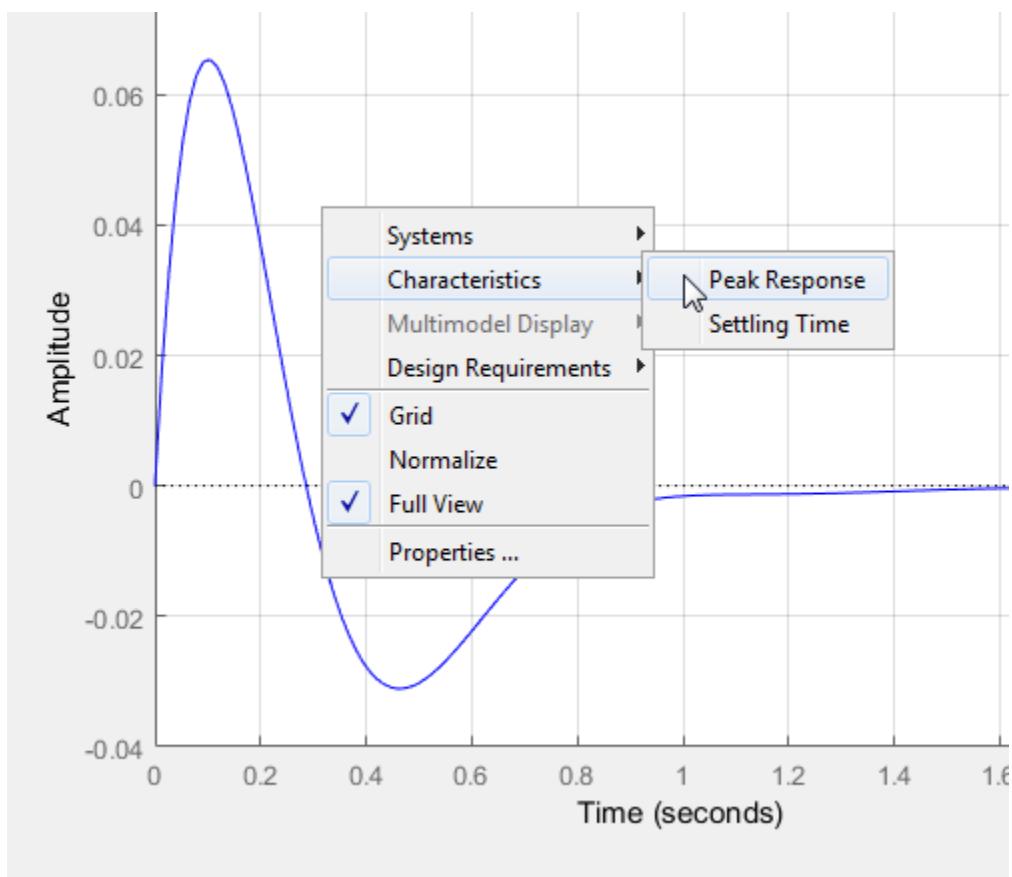
Plot Characteristics

On any analysis plot in **Control System Designer**:

- To see response information and data values, click a line on the plot.
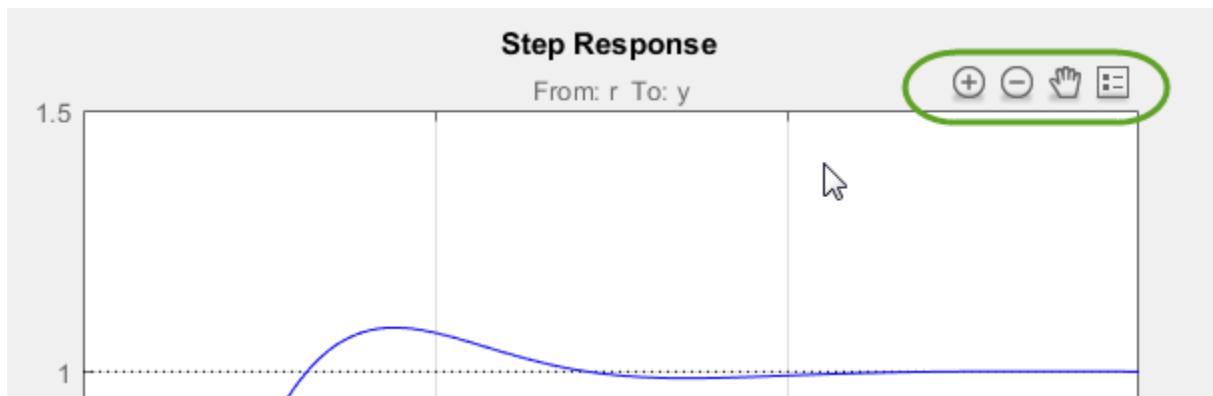
- To view system characteristics, right-click anywhere on the plot, as described in "System Characteristics on Response Plots" on page 8-10.
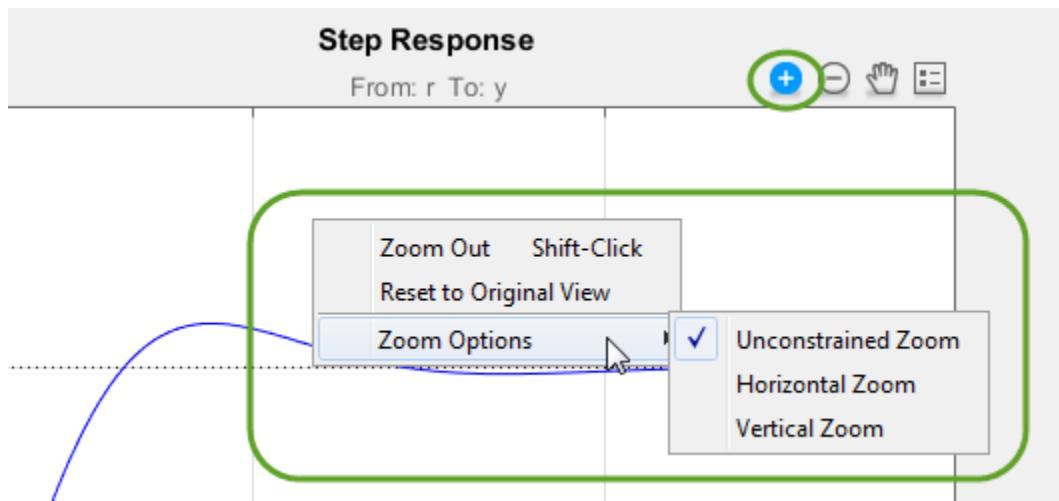
Plot Tools

Mouse over any analysis plot to access plot tools at the upper right corner of the plot.
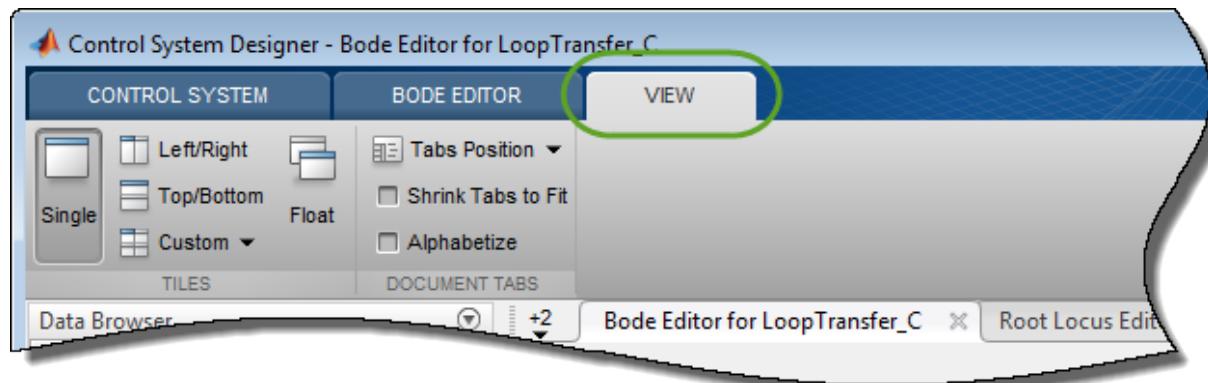
- and — Zoom in and zoom out. Click to activate, and drag the cursor over the region to zoom. The zoom icon turns dark when zoom is active. Right-click while zoom is active to access additional zoom options. Click the icon again to deactivate.

- — Pan. Click to activate, and drag the cursor across the plot area to pan. The pan icon turns dark when pan is active. Right-click while pan is active to access additional pan options. Click the icon again to deactivate.
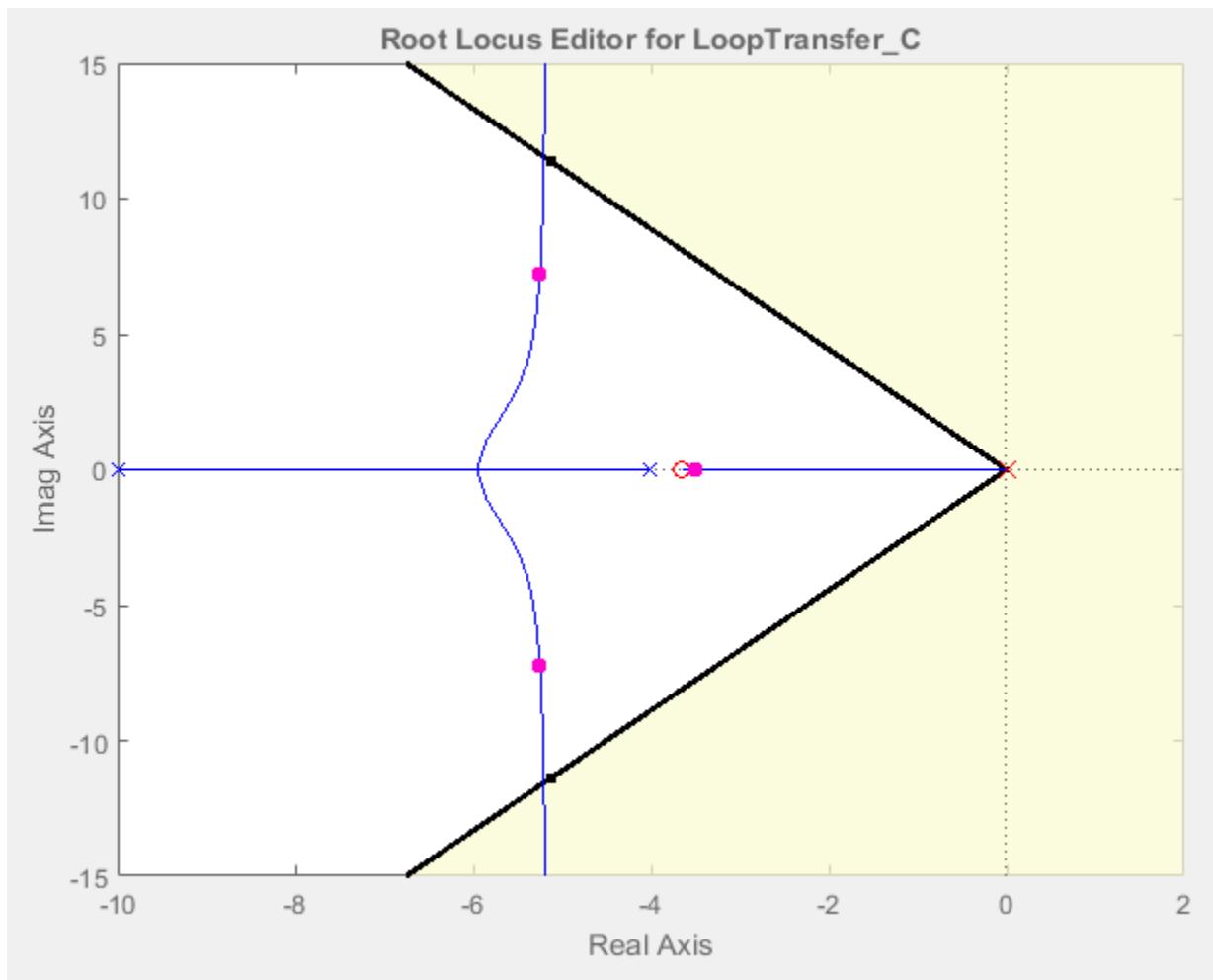
- — Legend. By default, the plot legend is inactive. To toggle the legend on and off, click this icon. To move the legend, drag it to a new location on the plot.

To change the way plots are tiled or sorted, use the options on the **View** tab.
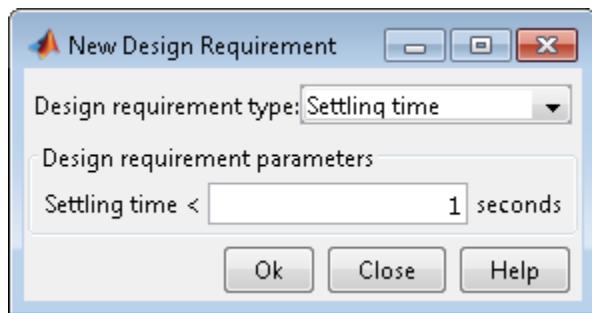
Design Requirements

You can add graphical representations of design requirements to any editor or analysis plots. These requirements define shaded exclusion regions in the plot area.

Use these regions as guidelines when analyzing and tuning your compensator designs. To meet a design requirement, your response plots must remain outside of the corresponding shaded area.

To add design requirements to a plot, right-click anywhere on the plot and select **Design Requirements > New**.

In the New Design Requirement dialog box, specify the **Design requirement type**, and define the **Design requirement parameters**. Each type of design requirement has a different set of parameters to configure. For more information on adding design requirements to analysis and editor plots, see "Design Requirements" on page 12-9.

More About

- "Control System Designer Tuning Methods" on page 12-4
- "Compare Performance of Multiple Designs" on page 12-124
- "Design Requirements" on page 12-9

Compare Performance of Multiple Designs
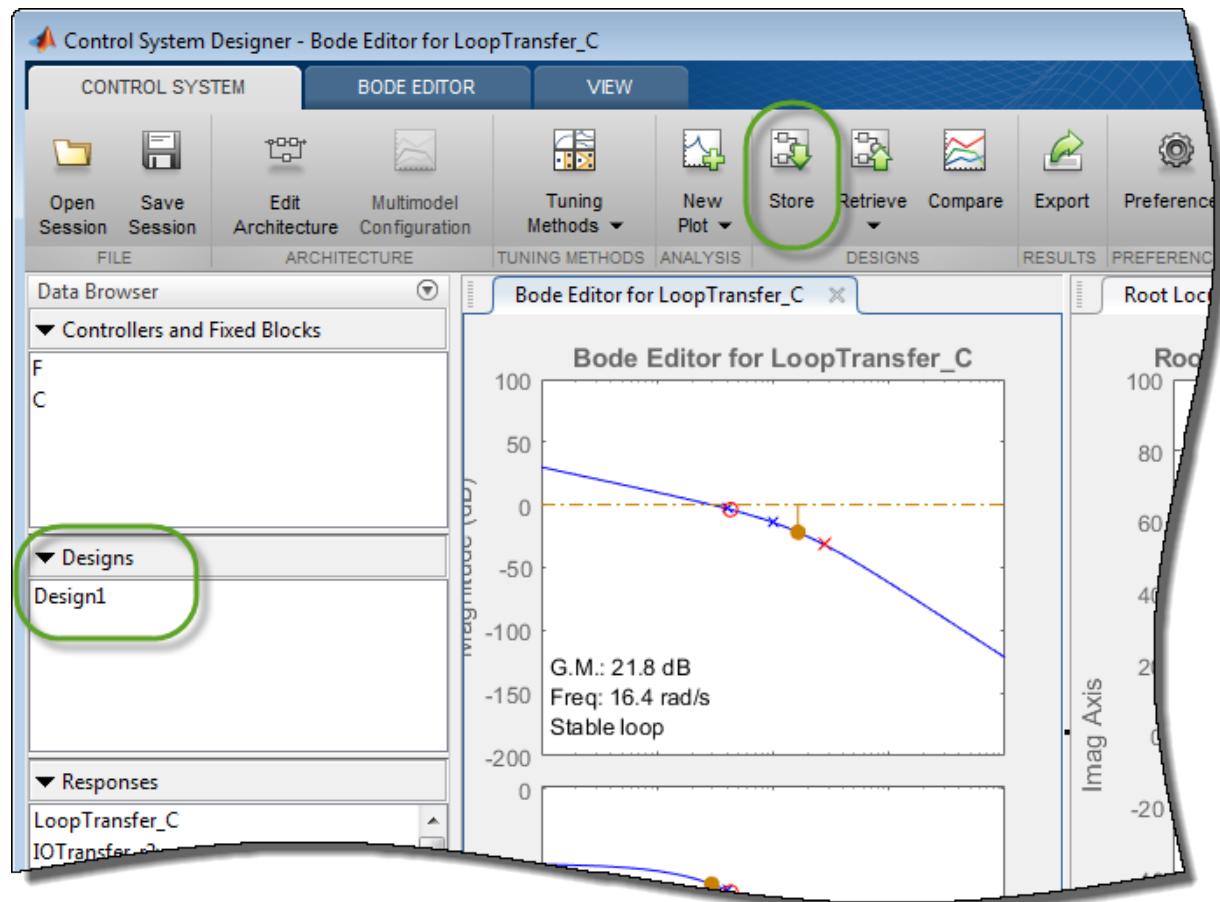
This example shows how to compare the performance of two different control system designs. Such comparison is useful, for example, to see the effects of different tuning methods or compensator structures.

Store First Design

In this example, the first design is the compensator tuned graphically in "Bode Diagram Design" on page 12-48.

After tuning the compensator with this first tuning method, store the design in **Control System Designer**.

On the **Control System** tab, in the **Designs** section, click ⬚ **Store**. The stored design appears in the **Data Browser** in the **Designs** area.

To rename the stored design, in the **Data Browser**, double-click the design, and specify a new name.

Compute New Design

On the **Control System** tab, tune the compensator using a different tuning method.
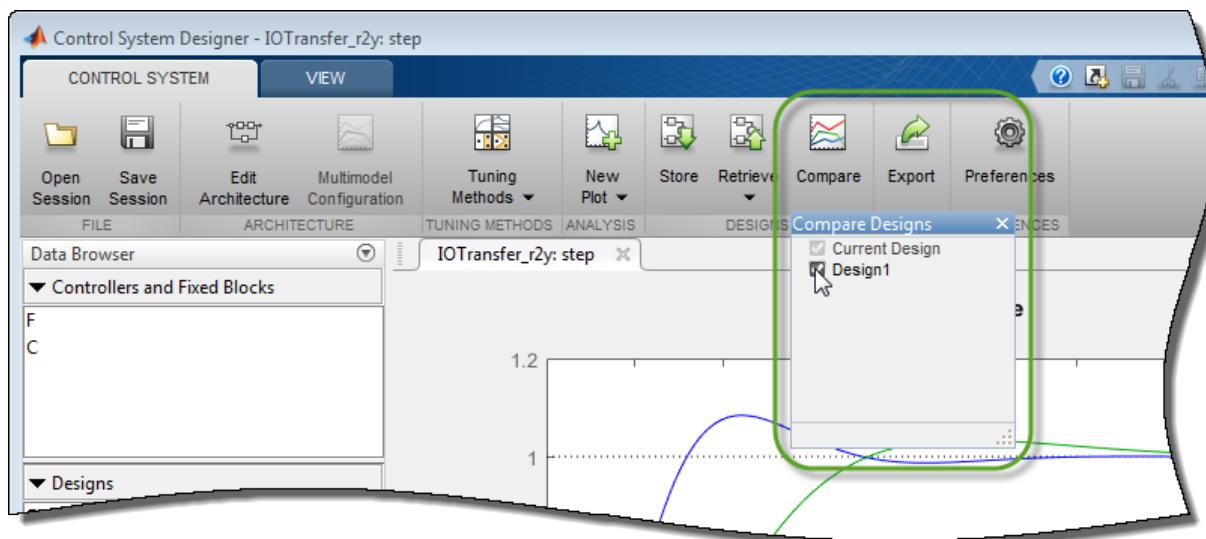
Under **Tuning Methods**, select PID Tuning.

To design a controller with the default Robust response time specifications, in the PID Tuning dialog box, click **Update Compensator**.
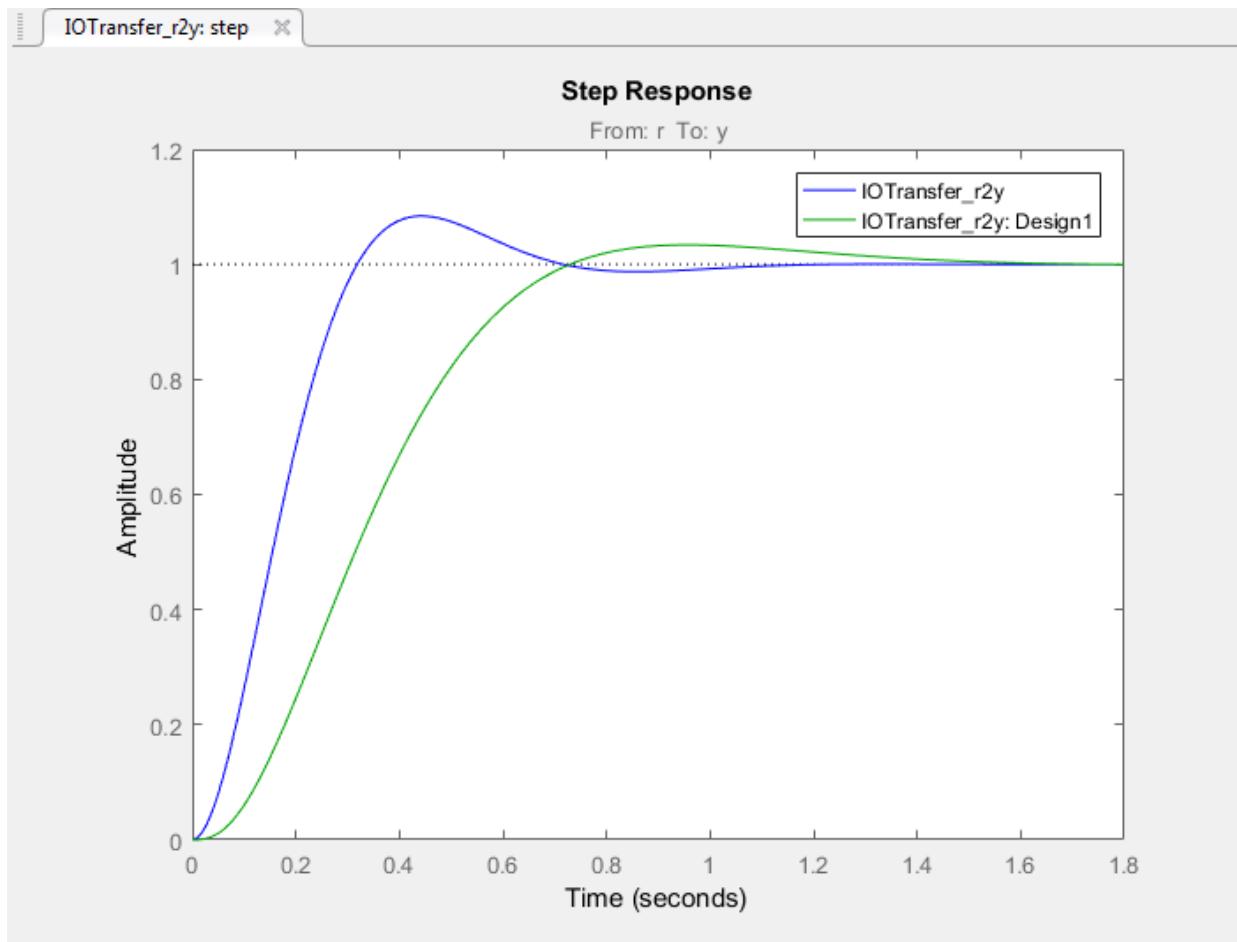
Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design.

On the **Control System** tab, click **Compare**.
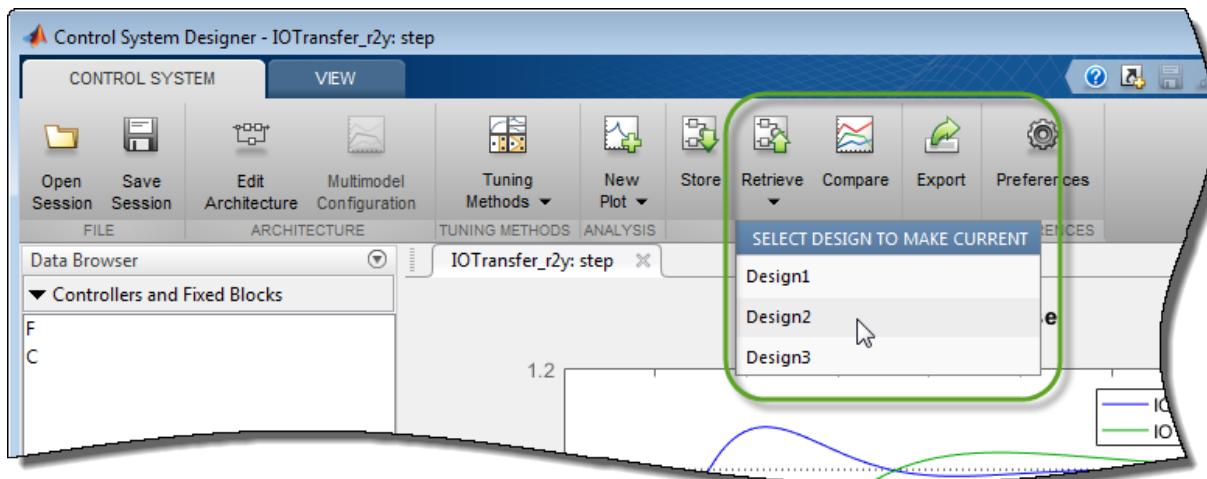
In the **Compare Designs** dialog box, the current design is checked by default. To compare a design with the current design, check the corresponding box. All analysis plots update to reflect the checked designs. The blue trace corresponds to the current design. Refer to the plot legend to identify the responses for other designs.

Restore Previously Saved Design

Under some conditions, it is useful to restore a previously stored design. For example, when designing a compensator for a Simulink model, you can write the current compensator values to the model (see "Update Simulink Model and Validate Design"). To test a stored compensator in your model, first restore the stored design as the current design.

To do so, in **Control System Designer**, click ⬛ **Retrieve**. Select the stored design that you want to make current.

Note: The retrieved design overwrites the current design. If necessary, store the current design before retrieving a previously stored design.

More About
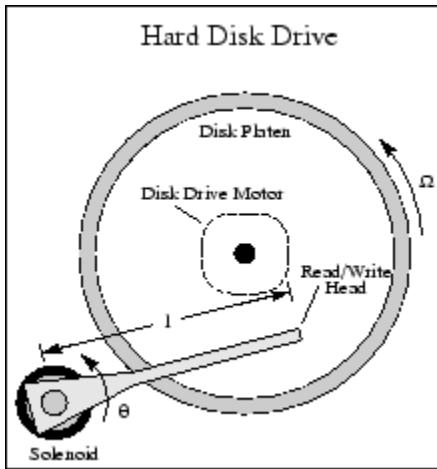
· "Analyze Designs Using Response Plots" on page 12-114
· "Control System Designer Tuning Methods" on page 12-4

Design Hard-Disk Read/Write Head Controller

In this section...
"Overview of this Case Study" on page 12-129
"Creating the Read/Write Head Model" on page 12-129
"Model Discretization" on page 12-130
"Adding a Compensator Gain" on page 12-132
"Adding a Lead Network" on page 12-133
"Design Analysis" on page 12-136

Overview of this Case Study

This case study demonstrates the ability to perform classical digital control design by going through the design of a computer hard-disk read/write head position controller.

Creating the Read/Write Head Model

Using Newton's law, a simple model for the read/write head is the differential equation

$$J\frac{d^2\theta}{dt^2} + C\frac{d\theta}{dt} + K\theta = K_i i$$

where J is the inertia of the head assembly, C is the viscous damping coefficient of the bearings, K is the return spring constant, K_i is the motor torque constant, θ is the angular position of the head, and i is the input current.

Taking the Laplace transform, the transfer function from i to θ is

$$H(s) = \frac{K_i}{Js^2 + Cs + K}$$

Using the values $J = 0.01$ kg m^2, $C = 0.004$ Nm/(rad/sec), $K = 10$ Nm/rad, and $K_i = 0.05$ Nm/rad, form the transfer function description of this system. At the MATLAB prompt, type

```
J = .01; C = 0.004; K = 10; Ki = .05;
num = Ki;
den = [J C K];
H = tf(num,den)
```

These commands produce the following result.

```
Transfer function:
      0.05
---------------------
0.01 s^2 + 0.004 s + 10
```
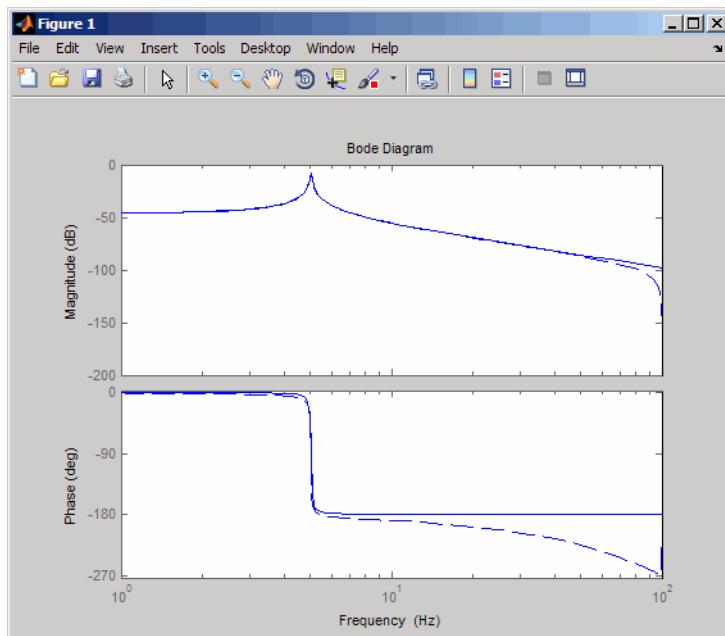
Model Discretization

The task here is to design a digital controller that provides accurate positioning of the read/write head. The design is performed in the digital domain. First, discretize the continuous plant. Because our plant will be equipped with a digital-to-analog converter (with a zero-order hold) connected to its input, use c2d with the 'zoh' discretization method. Type

```
Ts = 0.005;     % sample time = 0.005 second
Hd = c2d(H,Ts,'zoh')
Transfer function:
6.233e-05 z + 6.229e-05
---------------------
 z^2 - 1.973 z + 0.998
```
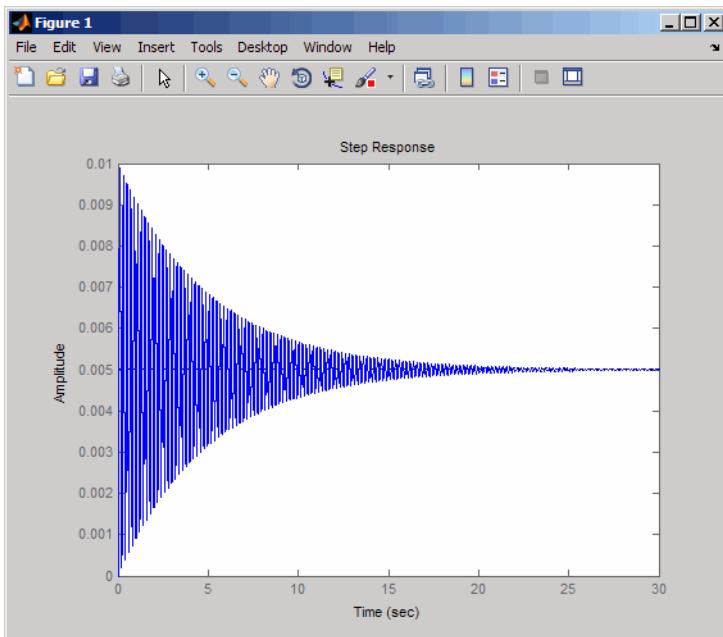
```
Sample time: 0.005
```

You can compare the Bode plots of the continuous and discretized models with

```
bodeplot(H,'-',Hd,'--')
```


To analyze the discrete system, plot its step response, type

```
step(Hd)
```

The system oscillates quite a bit. This is probably due to very light damping. You can check this by computing the open-loop poles. Type
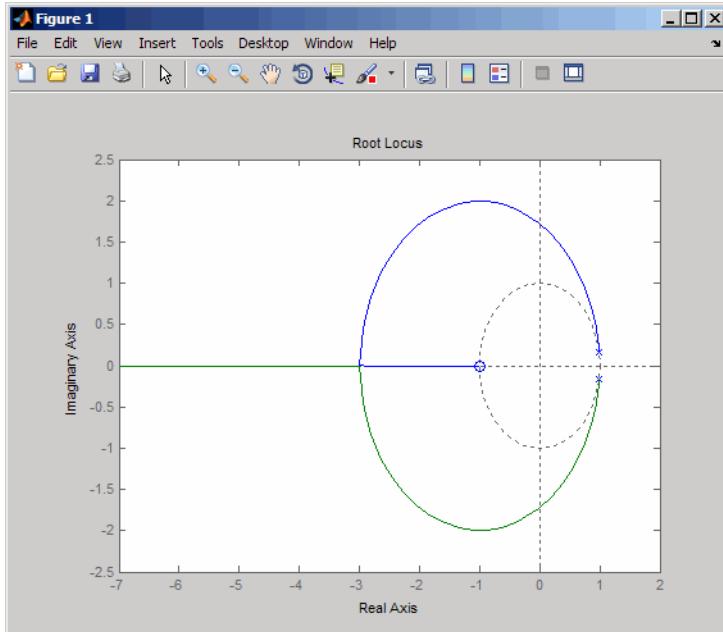
```
% Open-loop poles of discrete model
damp(Hd)
```

Pole	Magnitude	Damping	Frequency (rad/seconds)	Time Constant (seconds)
9.87e-01 + 1.57e-01i	9.99e-01	6.32e-03	3.16e+01	5.00e+00
9.87e-01 - 1.57e-01i	9.99e-01	6.32e-03	3.16e+01	5.00e+00

Pole	Magnitude	Damping	Frequency (rad/seconds)	Time Constant (seconds)
9.87e-01 + 1.57e-01i	9.99e-01	6.32e-03	3.16e+01	5.00e+00
9.87e-01 - 1.57e-01i	9.99e-01	6.32e-03	3.16e+01	5.00e+00

The poles have very light equivalent damping and are near the unit circle. You need to design a compensator that increases the damping of these poles.

Adding a Compensator Gain

The simplest compensator is just a gain, so try the root locus technique to select an appropriate feedback gain.

```
rlocus(Hd)
```


As shown in the root locus, the poles quickly leave the unit circle and go unstable. You need to introduce some lead or a compensator with some zeros.
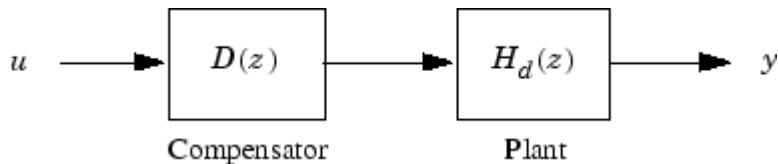
Adding a Lead Network

Try the compensator

$$D(z) = \frac{z + \alpha}{z + b}$$

with $\alpha = -0.85$ and $b = 0$.

The corresponding open-loop model
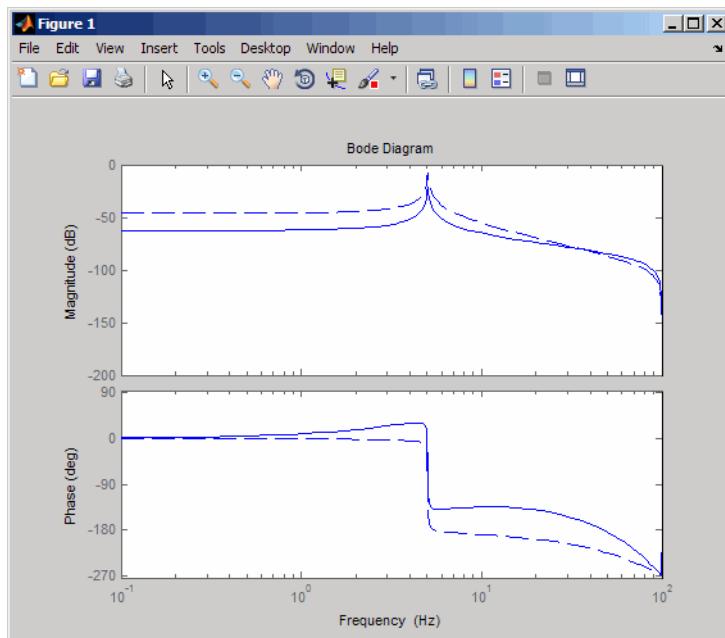
is obtained by the series connection

```
D = zpk(0.85,0,1,Ts)
oloop = Hd * D
```

Now see how this compensator modifies the open-loop frequency response.

```
bodeplot(Hd,'--',oloop,'-')
```

The plant response is the dashed line and the open-loop response with the compensator is the solid line.
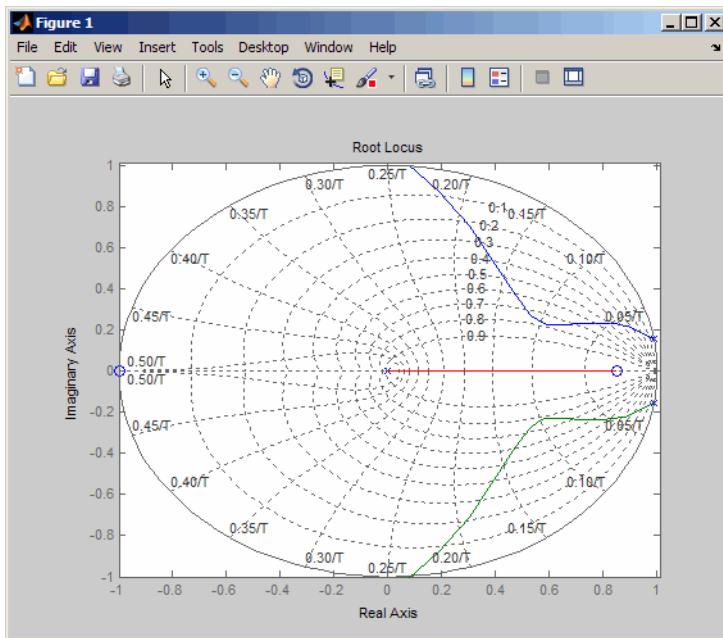
The plot above shows that the compensator has shifted up the phase plot (added lead) in the frequency range $\omega > 10$ rad/sec.
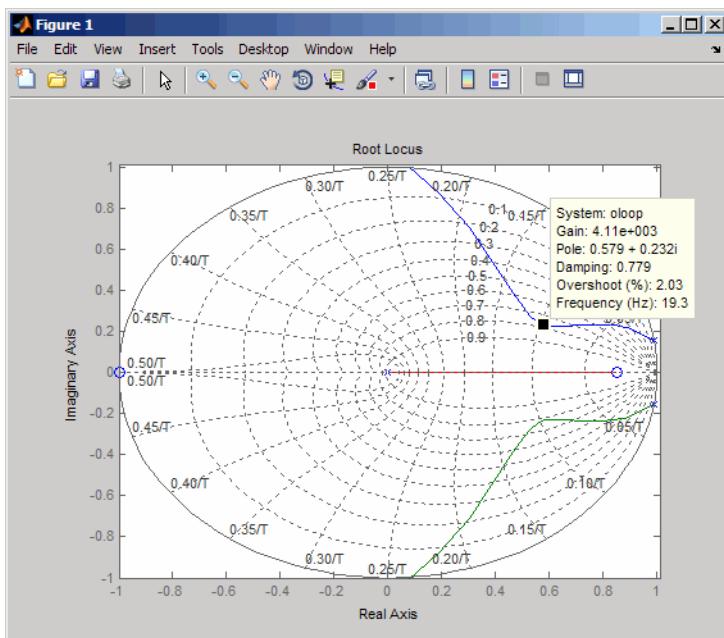
Now try the root locus again with the plant and compensator as open loop.

```
rlocus(oloop)
zgrid
```

Open the **Property Editor** by right-clicking in the plot away from the curve. On the **Limits** page, set the *x*- and *y*-axis limits from -1 to 1.01. This figure shows the result.
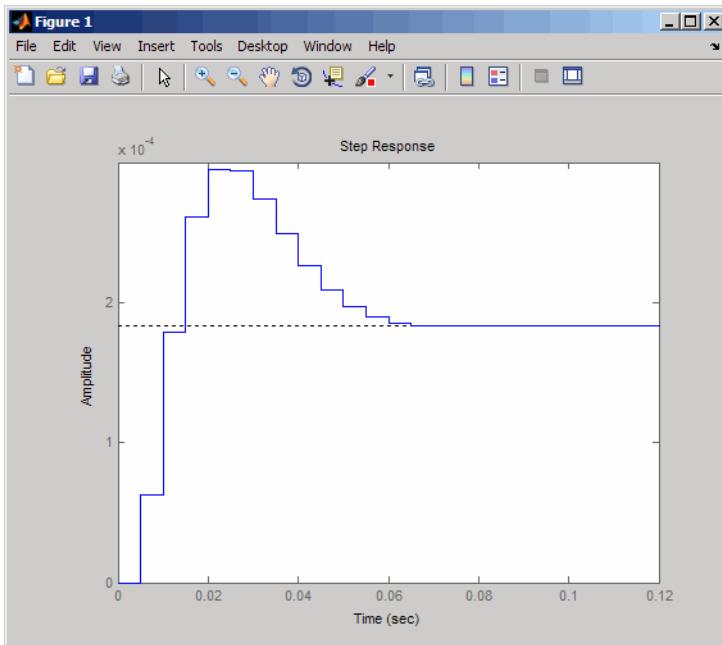
This time, the poles stay within the unit circle for some time (the lines drawn by `zgrid` show the damping ratios from $\zeta = 0$ to 1 in steps of 0.1). Use a data marker to find the point on the curve where the gain equals 4.111e+03. This figure shows the data marker at the correct location.

Design Analysis

To analyze this design, form the closed-loop system and plot the closed-loop step response.

```
K = 4.11e+03;
cloop = feedback(oloop,K);
step(cloop)
```
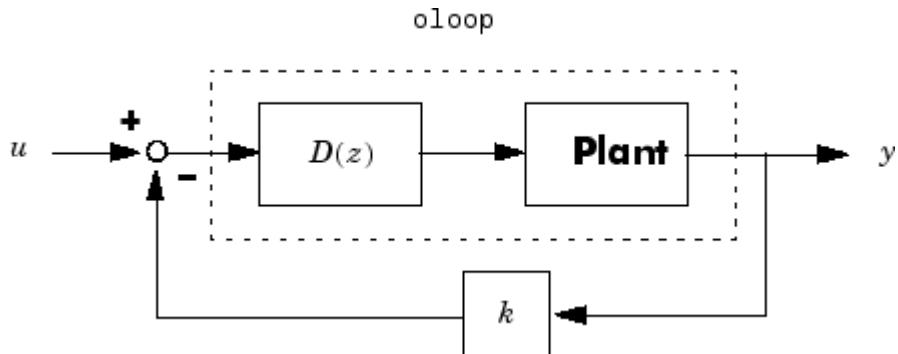
This response depends on your closed loop set point. The one shown here is relatively fast and settles in about 0.07 seconds. Therefore, this closed loop disk drive system has a seek time of about 0.07 seconds. This is slow by today's standards, but you also started with a very lightly damped system.

Now look at the robustness of your design. The most common classical robustness criteria are the gain and phase margins. Use the function `margin` to determine these margins. With output arguments, `margin` returns the gain and phase margins as well as the corresponding crossover frequencies. Without output argument, `margin` plots the Bode response and displays the margins graphically.

To compute the margins, first form the unity-feedback open loop by connecting the compensator $D(z)$, plant model, and feedback gain k in series.

```
olk = K * oloop;
```

Next apply `margin` to this open-loop model. Type

```
[Gm,Pm,Wcg,Wcp] = margin(olk);
Margins = [Gm Wcg Pm Wcp]
Margins =

    3.7987  296.7978   43.2031  106.2462

Margins =

    3.7987  296.7978   43.2031  106.2462
```
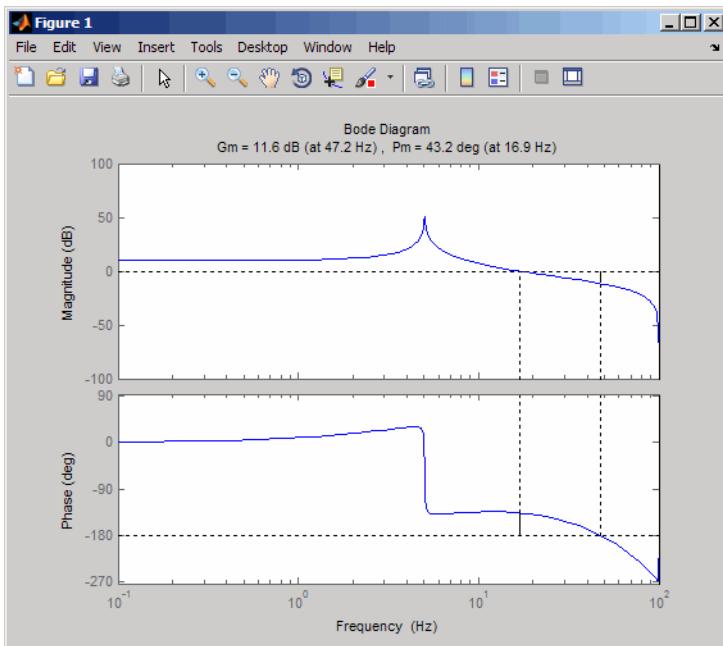
To obtain the gain margin in dB, type

```
20*log10(Gm)
ans =
   11.5926
```

You can also display the margins graphically by typing

```
margin(olk)
```

The command produces the plot shown below.

This design is robust and can tolerate a 11 dB gain increase or a 40 degree phase lag in the open-loop system without going unstable. By continuing this design process, you may be able to find a compensator that stabilizes the open-loop system and allows you to reduce the seek time.

See Also
`bodeplot` | `feedback` | `margin` | `rlocus`

Design Compensator for Plant Model with Time Delays

This example shows how to design a compensator for a plant with time delays using Control System Designer.

Analysis and Design of Feedback Systems with Time Delays

When working with time delay systems it is advantageous to work with analysis and design tools that directly support time delays so that performance and stability can be evaluated exactly. However, many control design techniques and algorithms cannot directly handle time delays. A common workaround consists of replacing delays by their Pade approximations (all-pass filters). Because this approximation is only valid at low frequencies, it is important to choose the right approximation order and check the approximation validity.

Control System Designer provides a variety of design and analysis tools. Some of these tools support time delays exactly while others support time delays indirectly through approximations. Use these tools to design compensators for your control system and visualize the compromises made when using approximations.

Plant Model

For this example, which uses a unity feedback configuration, the plant model has a time delay:

$$G(s) = e^{-0.5s} \frac{1}{s+1}$$

Create the plant model.

```
G = tf(1,[1,1],'InputDelay',0.5);
```

Tools that Support Time Delays
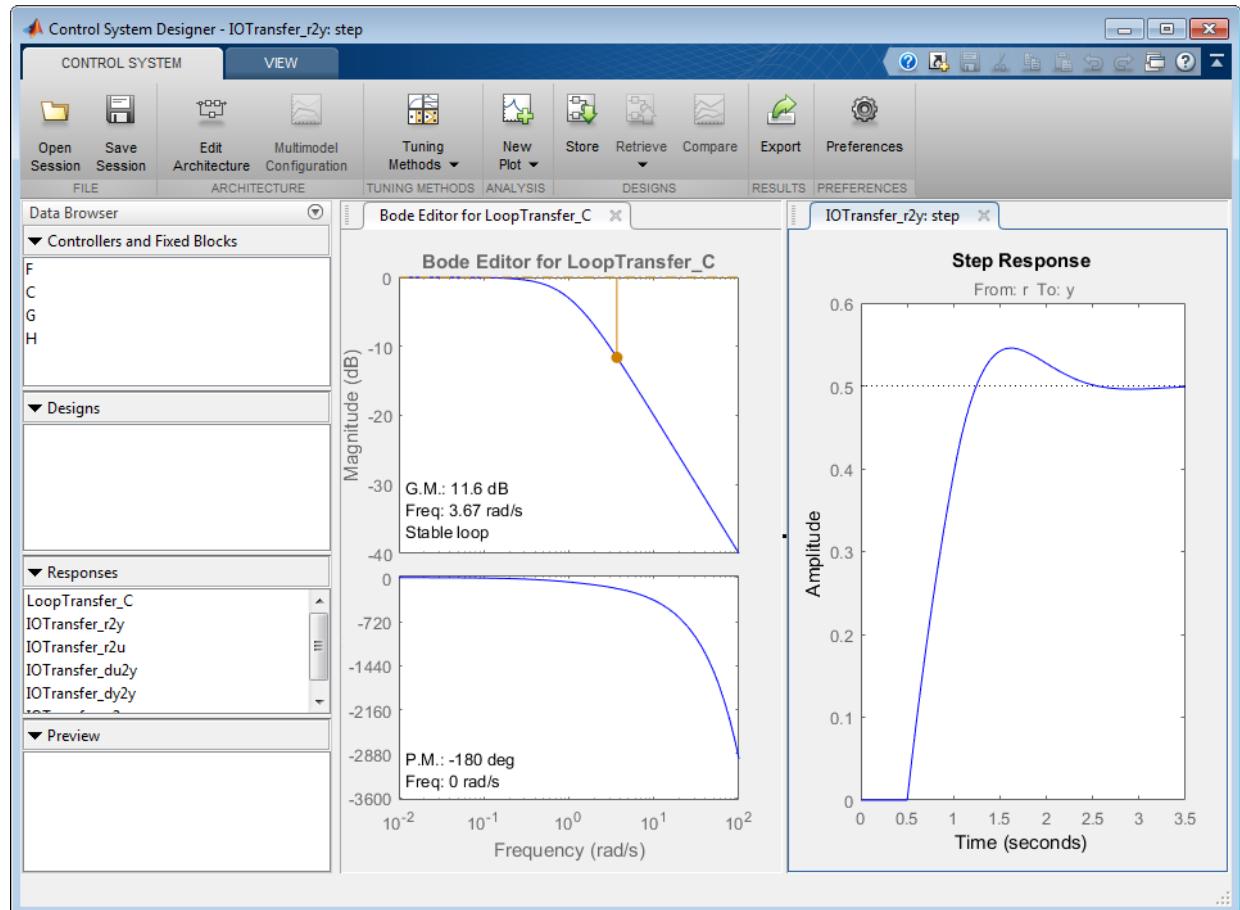
In the app, the following tools support time delays directly:

- Bode and Nichols Editors

- Time Response Plots
- Frequency Response Plots
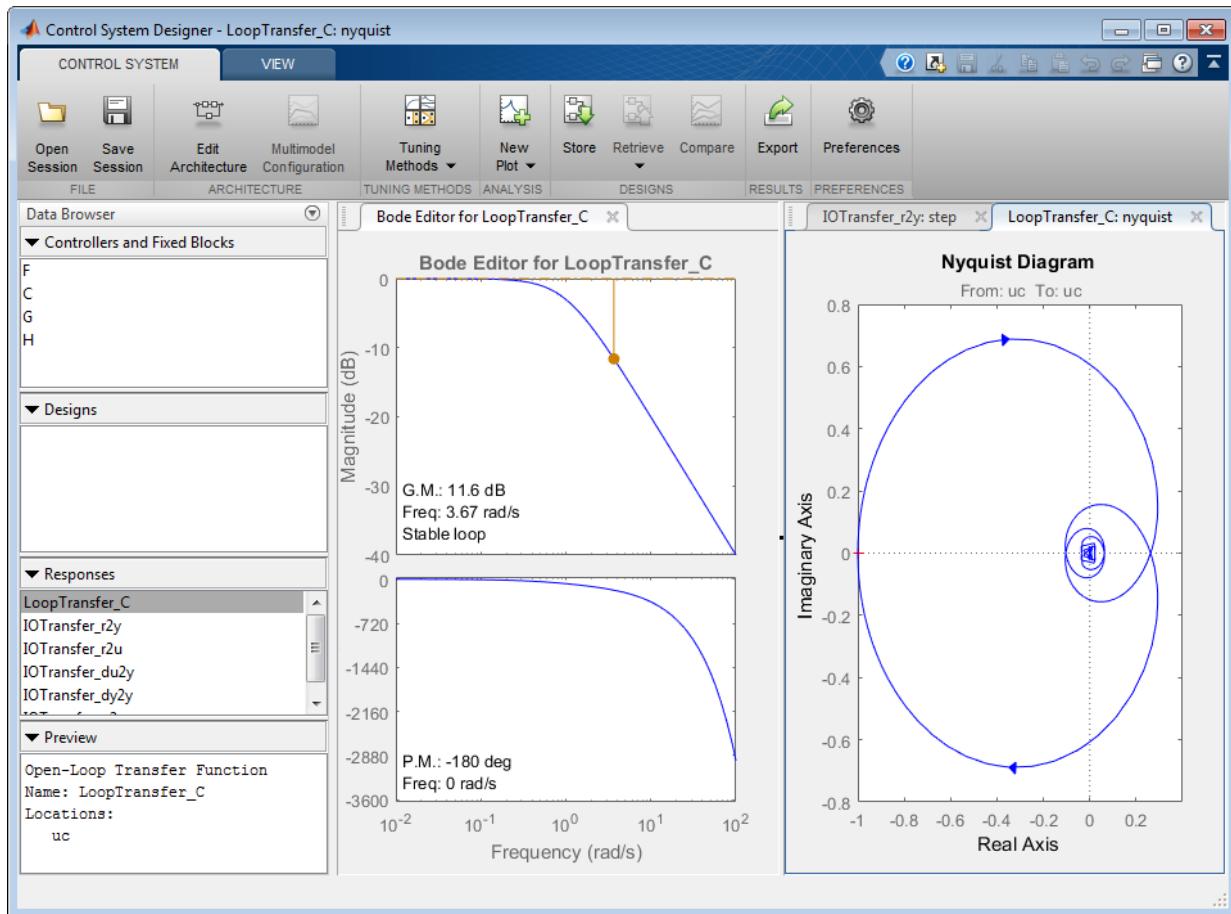
Open Control System Designer, importing the plant model and using a Bode editor configuration.

```
controlSystemDesigner({'bode'},G)
```


The phase response of the Bode plot shows the roll-off effect from the exact representation of the delay. The beginning of the step response shows an exact representation of the 0.5 second delay.

Open a Nyquist plot of the open-loop response. In the **Data Browser**, right-click `LoopTransfer_C`, and select **Plot > nyquist**.

The Nyquist response wrapping around the origin in a spiral fashion is the result of the exact representation of the time delay.

Tools that Approximate Time Delays
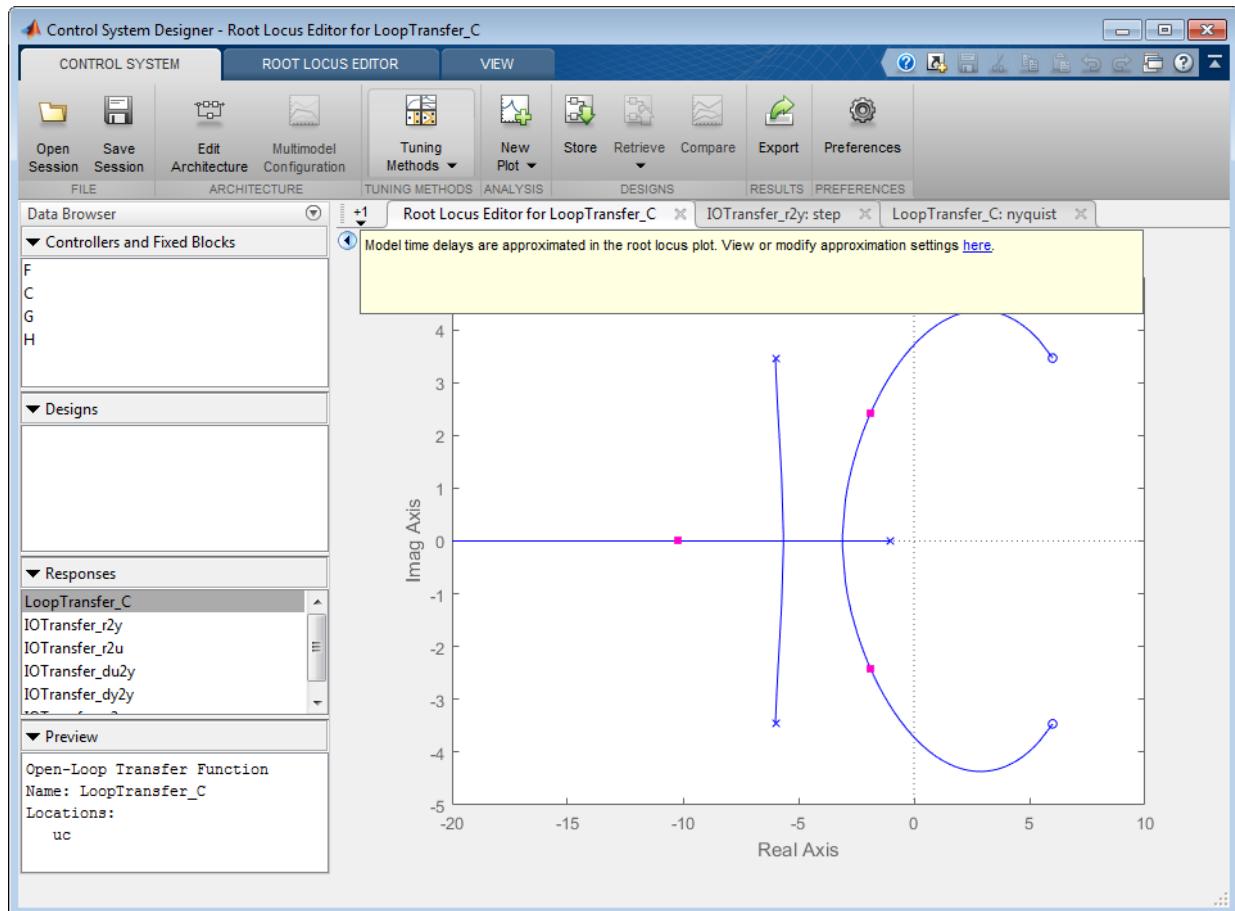
In the app, the following tools approximate time delays:

· Root Locus Editor
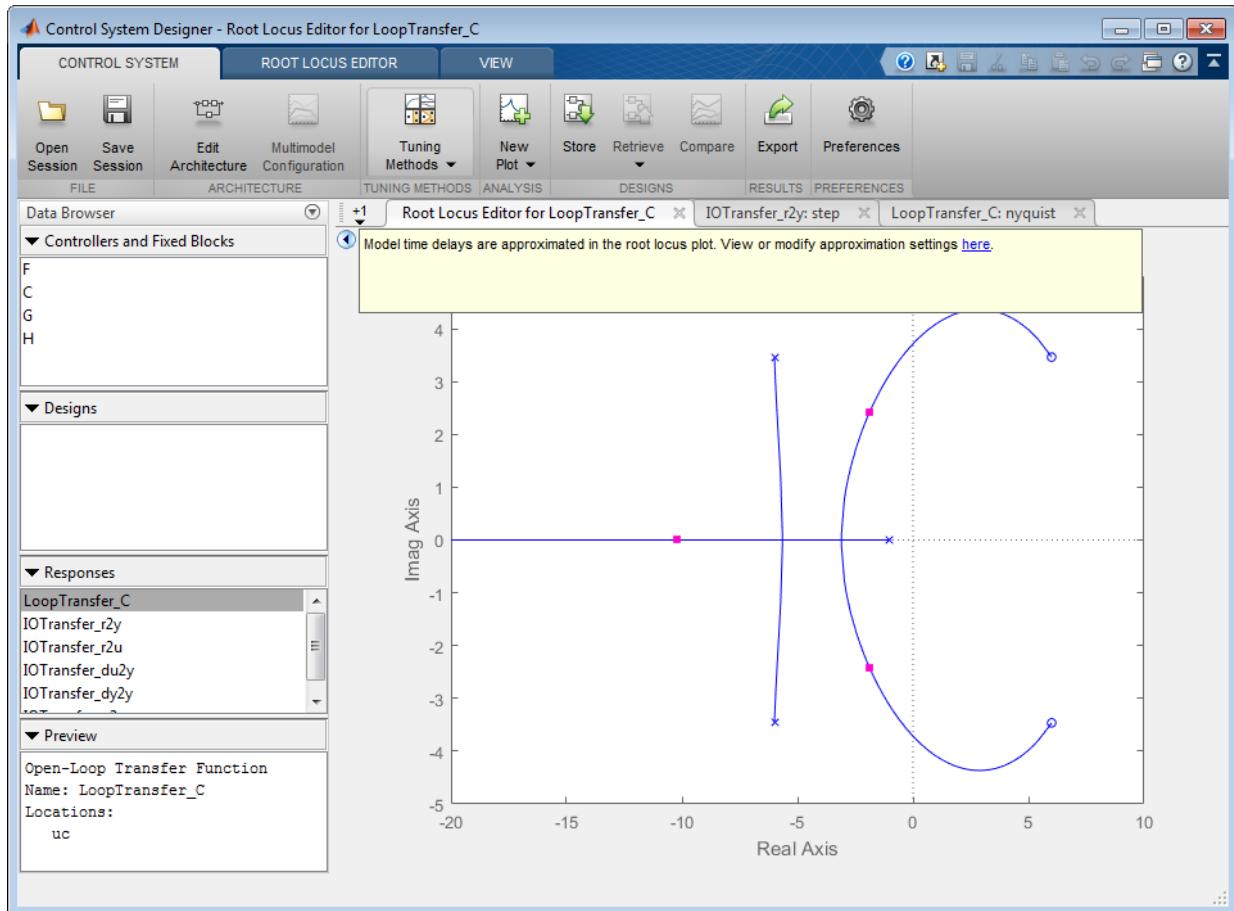
- Pole/Zero Plots
- Many of the automated tuning methods

When using approximations, the results are not exact and depend on the validity of the approximation. Each tool in Control System Designer provides a warning pane to indicate when time-delays are approximated.

Open a root locus editor plot for the open-loop response. Click **Tuning Methods**, and select **Root Locus Editor**. In the Select Response to Edit dialog box, click **Plot**.
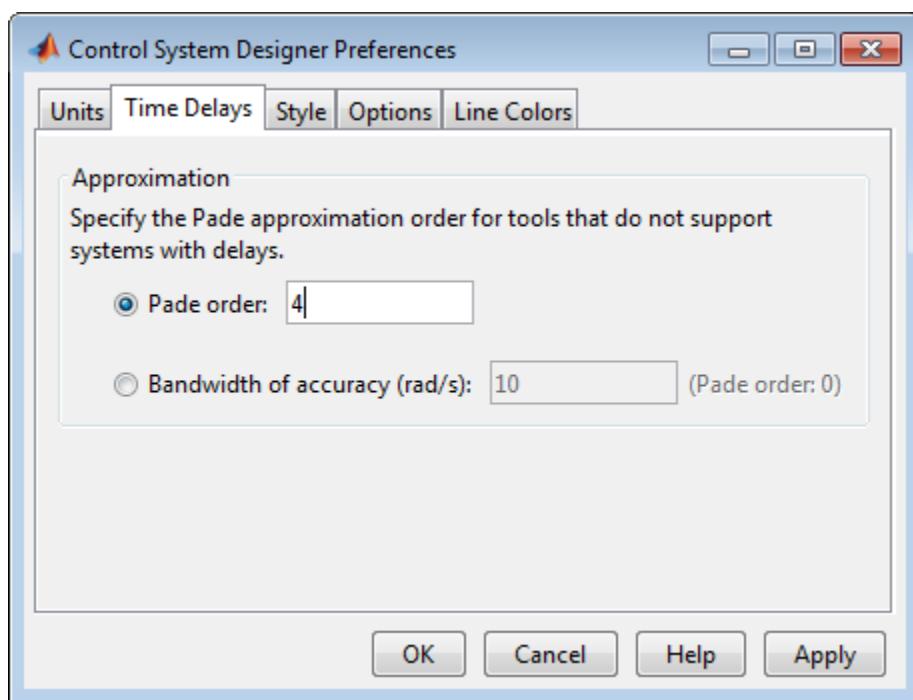
The Root Locus Editor shows a notification that the plot is using a time delay approximation. This notification can be minimized by clicking on the arrow icon to the left.
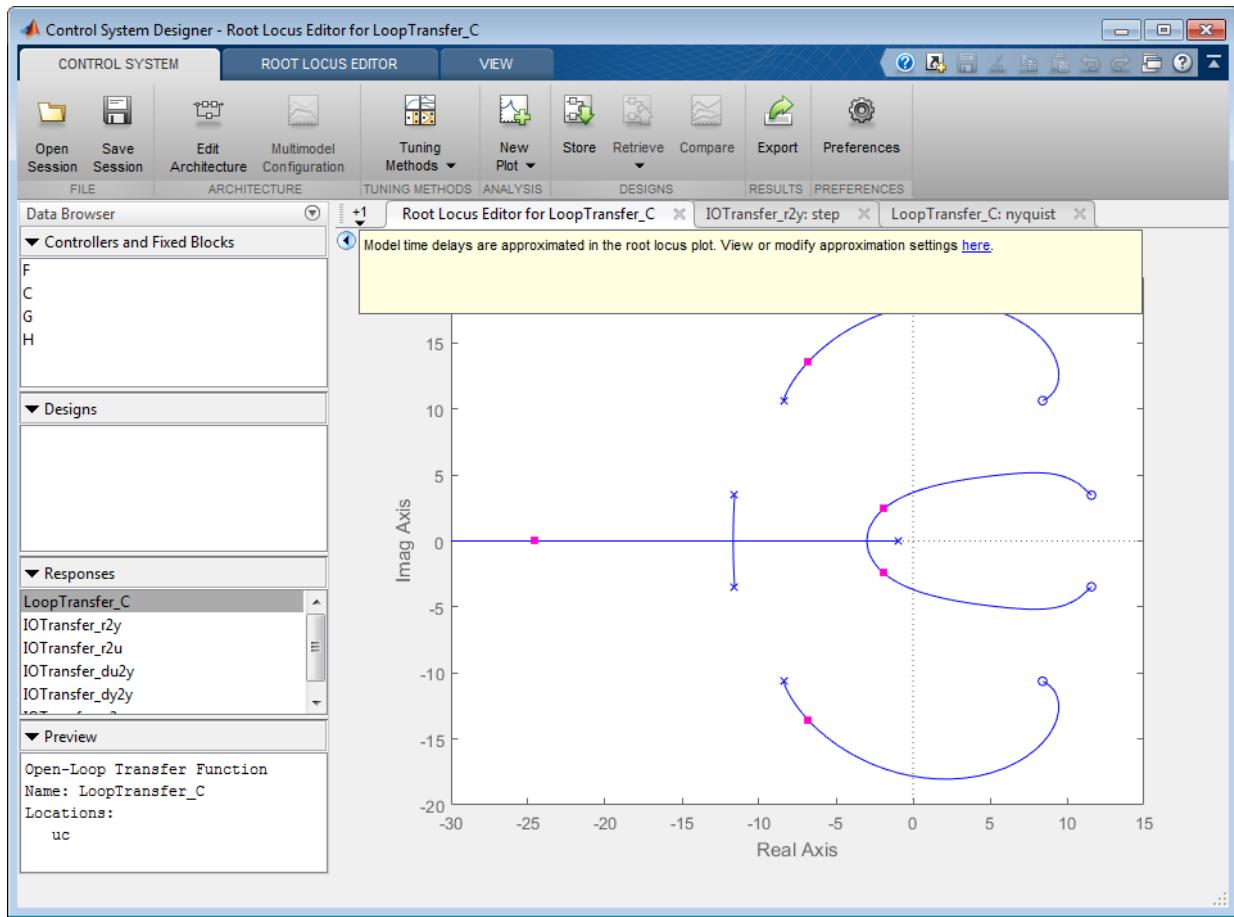
Change Approximation Settings

To change the approximation settings, click the hyperlink in the notification. In the Control System Designer Preferences dialog box, on the **Time Delays** tab, specify a **Pade order** of 4. Aternatively, you can set the bandwidth over which you want the approximation to be accurate.

The higher-order Pade approximation adds poles and zeros to the root locus plot.

See Also

Control System Designer

Design Compensator for Systems Represented by Frequency Response Data

This example shows how to design a compensator for a plant model defined by frequency response data (FRD) using Control System Designer.

Acquire Frequency Response Data (FRD) Plant Model

Non-parametric representations of plant models, such as frequency response data, are often used for analysis and control design. These FRD models are typically obtained from:

1) Signal analyzer hardware that performs frequency domain measurements on systems.

2) Non-parametric estimation techniques using the systems time response data. You can use the following products to estimate FRD models:

Simulink® Control Design™:

- Function: frestimate
- Example: "Frequency Response Estimation Using Simulation-Based Techniques".

Signal Processing Toolbox™:
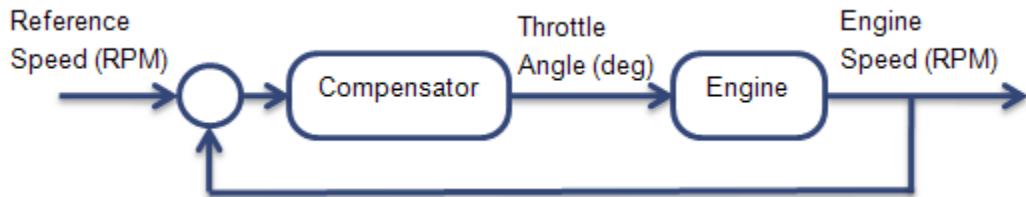
- Function: tfestimate.

System Identification Toolbox™:

- Functions: etfe, spa, spafdr

FRD Model and Design Requirements

In this example, design an engine speed controller that actuates the engine throttle angle:

The frequency response of the engine is already estimated. Load and view the data.

```
load FRDPlantDemoData.mat
AnalyzerData


AnalyzerData =

          Response: [594x1 double]
         Frequency: [594x1 double]
    FrequencyUnits: 'rad/s'
```

Create an FRD model object:

```
FRDPlant = frd(AnalyzerData.Response,AnalyzerData.Frequency,...
    'Unit',AnalyzerData.FrequencyUnits);
```

The design requirements are:
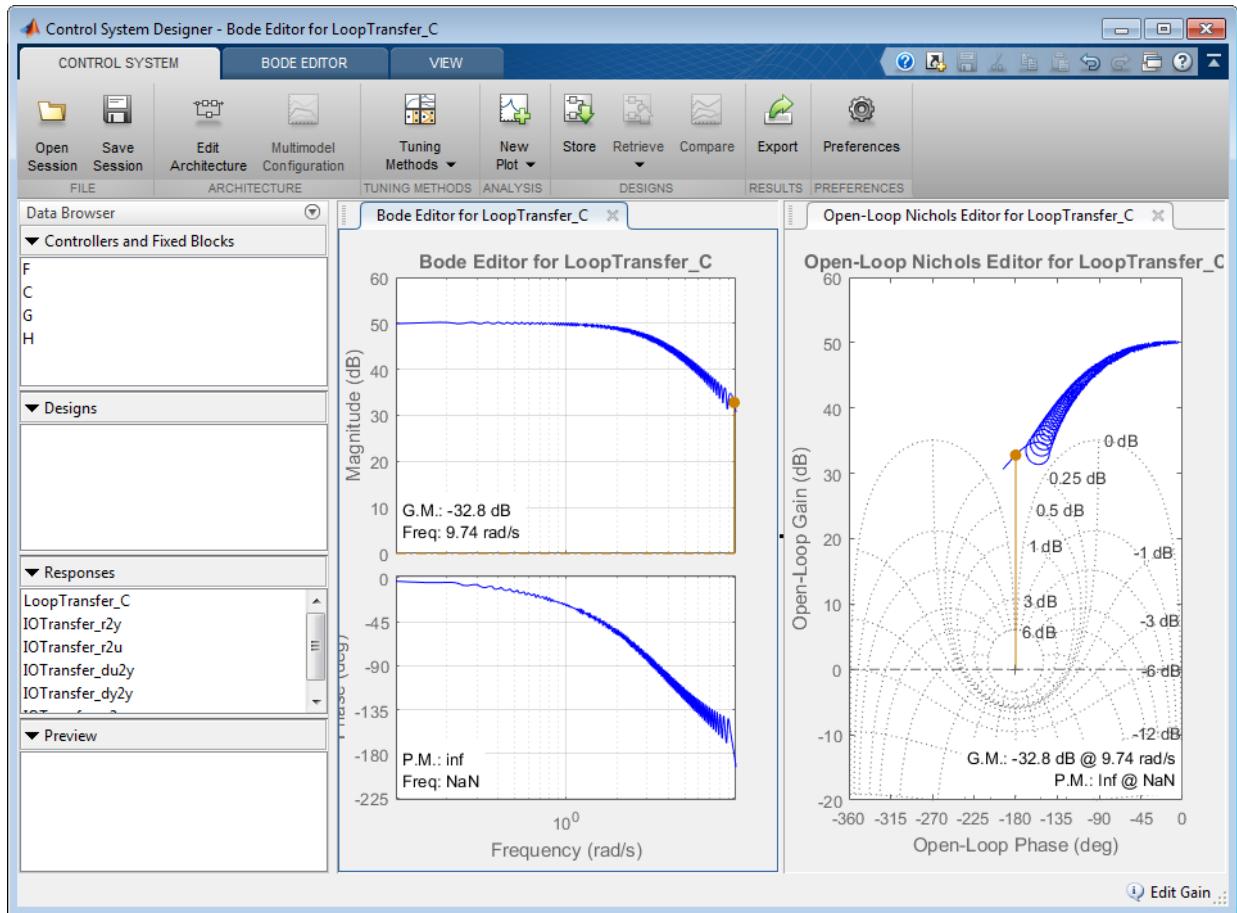
- Zero steady-state error for step reference speed changes
- Phase margin greater than 60 degrees
- Gain margin greater than 20 dB.

Design Compensator

Open Control System Designer.

```
controlSystemDesigner({'bode','nichols'},FRDPlant)|
```

The Control System Designer opens with both Bode and Nichols open-loop editors.

You can design the compensator by shaping the open-loop frequency response in either the Bode editor or Nichols editor. In these editors, interactively modify the gain, poles, and zeros of the compensator.

To satisfy the tracking requirement of zero steady-state error, add an integrator to the compensator. Right-click the Bode editor plot area, and select **Add Pole/Zero > Integrator**.
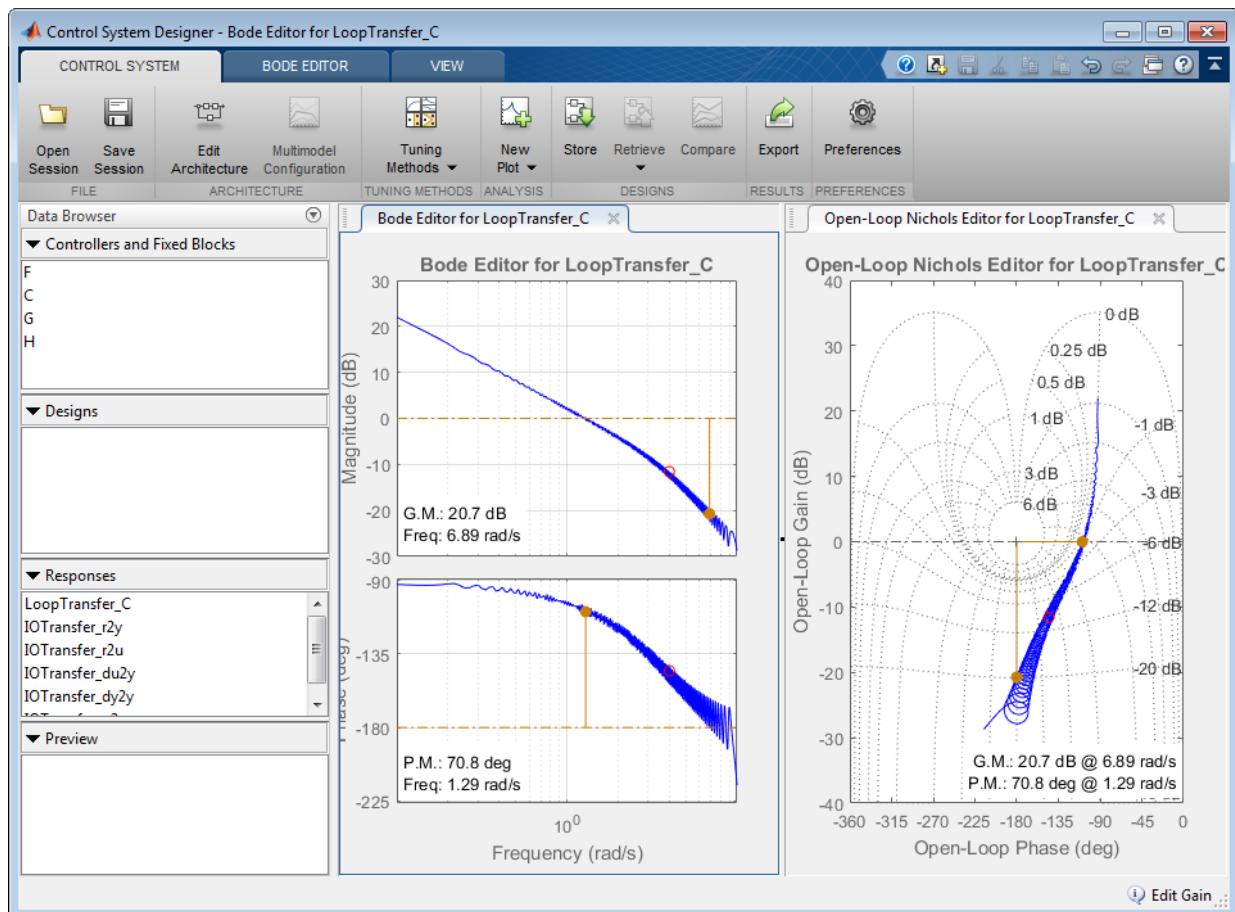
To meet the gain and phase margin requirements, add a zero to the compensator. Right-click the Bode editor plot area, and select **Add Pole/Zero > Real Zero**. Modify

the location of the zero and the gain of the compensator until you satisfy the margin requirements.

One possible design that satisfies the design requirements is:

$$C(s) = \frac{0.001(s+4)}{s}.$$

This compensator design, which is a PI controller, achieves a 20.7 dB gain margin and a 70.8 degree phase margin.
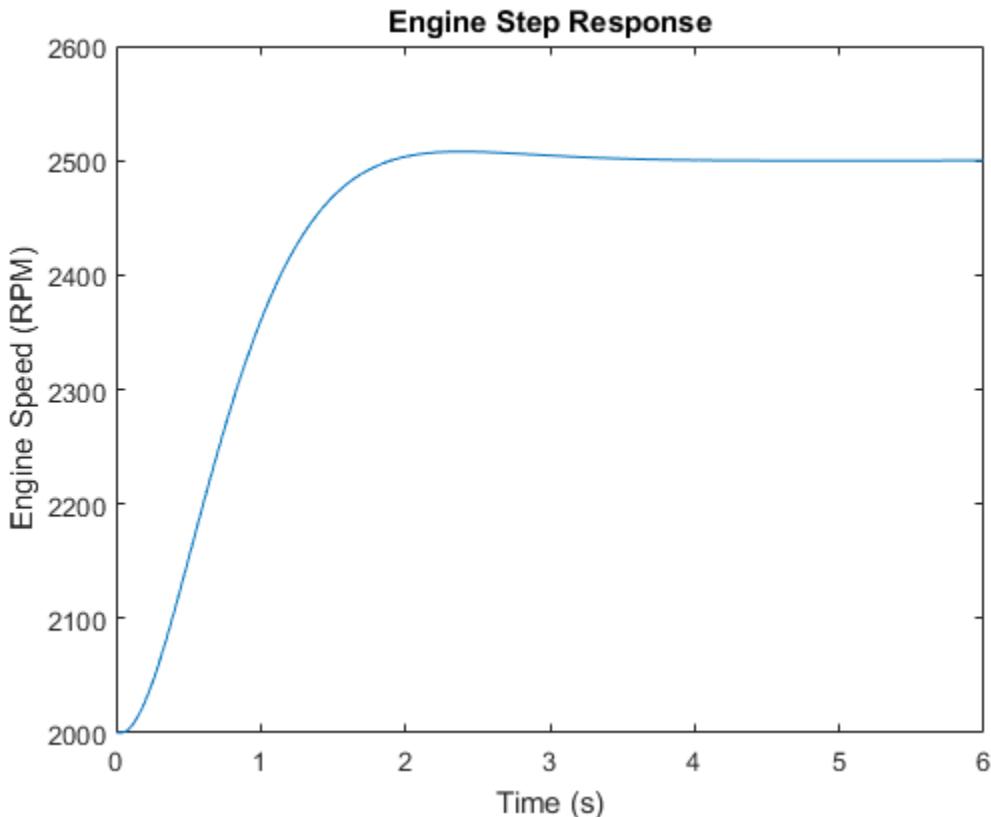
Export the designed compensator to the workspace. Click **Export**.

Validate the Design

Validate the controller performance by simulating the engine response using a nonlinear model in Simulink®. For this example, the validation simulation results are in `EngineStepResponse`.

Plot the response of the engine to a reference speed change from 2000 to 2500 RPM:

```
plot(EngineStepResponse.Time,EngineStepResponse.Speed)
title('Engine Step Response')
xlabel('Time (s)')
ylabel('Engine Speed (RPM)')
```

The response shows zero steady-state error and well-behaved transients with the following metrics.

```
stepinfo(EngineStepResponse.Speed,EngineStepResponse.Time)


ans =

        RiseTime: 1.1048
     SettlingTime: 1.7194
      SettlingMin: 2.4501e+03
      SettlingMax: 2.5078e+03
         Overshoot: 0.3127
        Undershoot: 0
              Peak: 2.5078e+03
          PeakTime: 2.3853
```
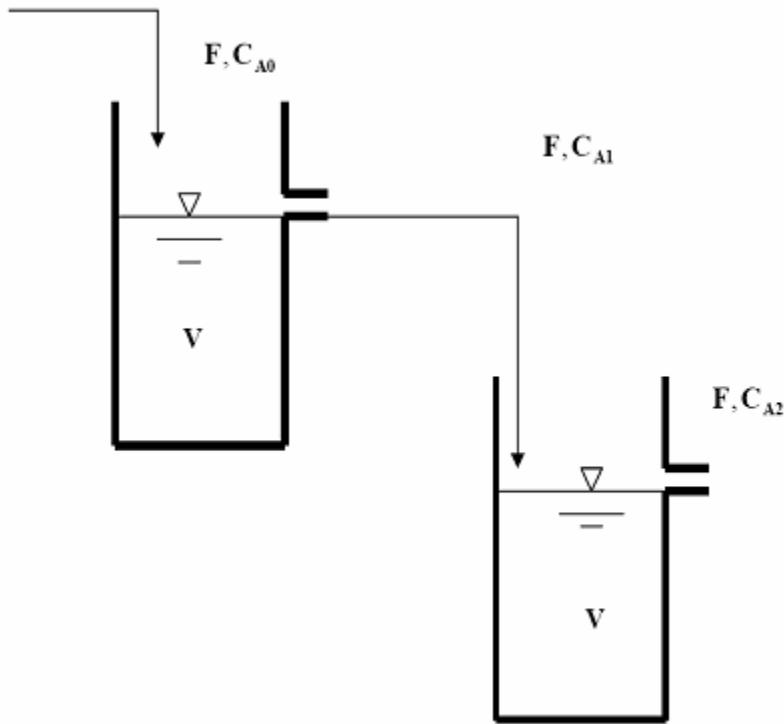
See Also
Control System Designer

Design Internal Model Controller for Chemical Reactor Plant

This example shows how to design a compensator in an IMC structure for series chemical reactors, using Control System Designer. Model-based control systems are often used to track setpoints and reject load disturbances in process control applications.

Plant Model

The plant for this example is a chemical reactor system, comprised of two well-mixed tanks.

The reactors are isothermal and the reaction in each reactor is first order on component A:

$$r_A = -kC_A$$

Material balance is applied to the system to generate a dynamic model of the system. The tank levels are assumed to stay constant because of the overflow nozzle and hence there is no level control involved.

For details about this plant, see Example 3.3 in Chapter 3 of "Process Control: Design Processes and Control systems for Dynamic Performance" by Thomas E. Marlin.

The following differential equations describe the component balances:

$$V\frac{dC_{A1}}{dt} = F(C_{A0} - C_{A1}) - VkC_{A1}$$

$$V\frac{dC_{A2}}{dt} = F(C_{A1} - C_{A2}) - VkC_{A2}$$

At steady state,

$$\frac{dC_{A1}}{dt} = 0$$

$$\frac{dC_{A2}}{dt} = 0$$

the material balances are:

$$F^*(C_{A0}^* - C_{A1}^*) - VkC_{A1}^* = 0$$

$$F^*(C_{A1}^* - C_{A2}^*) - VkC_{A2}^* = 0$$

where C_{A0}^*, $C_{A1}*$, and $C_{A2}*$ are steady-state values.

Substitute, the following design specifications and reactor parameters:

$$F^* = 0.085 \ mole/min$$

$$C_{A0}^* = 0.925 \ mol/min$$

$$V = 1.05 \ m^3$$

$$k = 0.04 \ min^{-1}$$

The resulting steady-state concentrations in the two reactors are:

$$C_{A1}^* = KC_{A0}^* = 0.6191 mol/m^3$$

$$C_{A2}^* = K^2 C_{A0}^* = 0.4144 mol/m^3$$

where

$$K = \frac{F^*}{F*+Vk} = 0.6693$$

For this example, design a controller to maintain the outlet concentration of reactant from the second reactor, C_{A2}^*, in the presence of any disturbance in feed concentration, C_{A0}. The manipulated variable is the the molar flowrate of the reactant, F, entering the first reactor.

Linear Plant Models
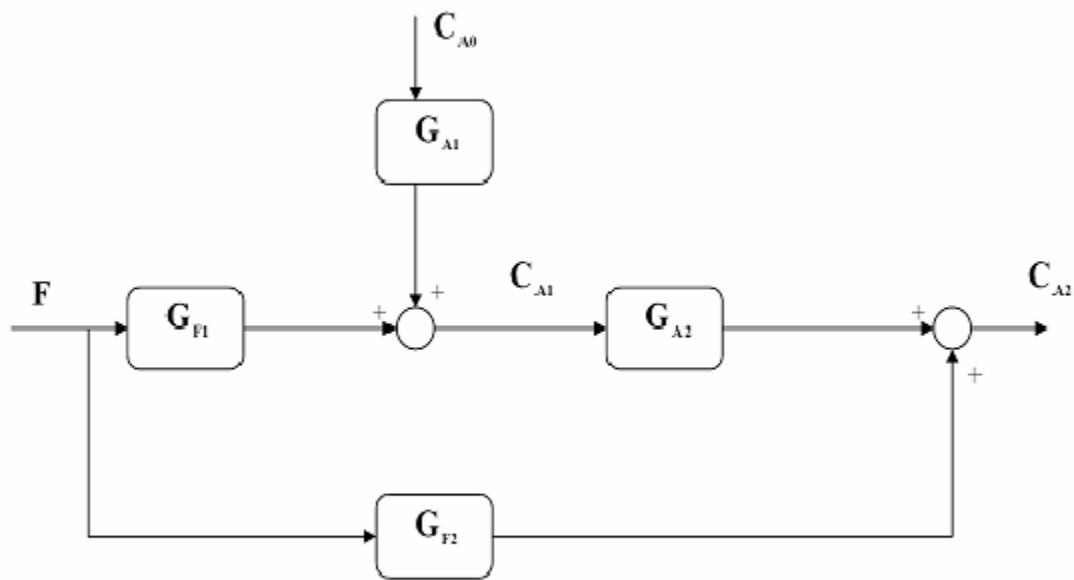
In this control design problem, the plant model is

$$\frac{C_{A2}(s)}{F(s)}$$

and the disturbance model is

$$\frac{C_{A0}(s)}{C_{A2}(s)}$$

This chemical process can be represented in the following block diagram:

where

$$G_{A1} = \frac{C_{A1}(s)}{C_{A0}(s)} = \frac{0.6693}{8.2677s + 1}$$

$$G_{F1} = \frac{C_{A1}(s)}{F(s)} = \frac{2.4087}{8.2677s + 1}$$

$$G_{A2} = \frac{C_{A2}(s)}{C_{A1}(s)} = \frac{0.6693}{8.2677s + 1}$$

$$G_{F2} = \frac{C_{A2}(s)}{F(s)} = \frac{1.6118}{8.2677s + 1}$$

Based on the block diagram, obtain the plant and disturbance models as follows:

$$\frac{C_{A2}(s)}{F(s)} = G_{F1}G_{A2} + G_{F2} = \frac{13.3259s + 3.2239}{(8.2677s + 1)^2}$$

$$\frac{C_{A2}}{C_{A0}} = G_{A1}G_{A2} = \frac{0.4480}{(8.2677s + 1)^2}$$

Create the plant model at the command line:
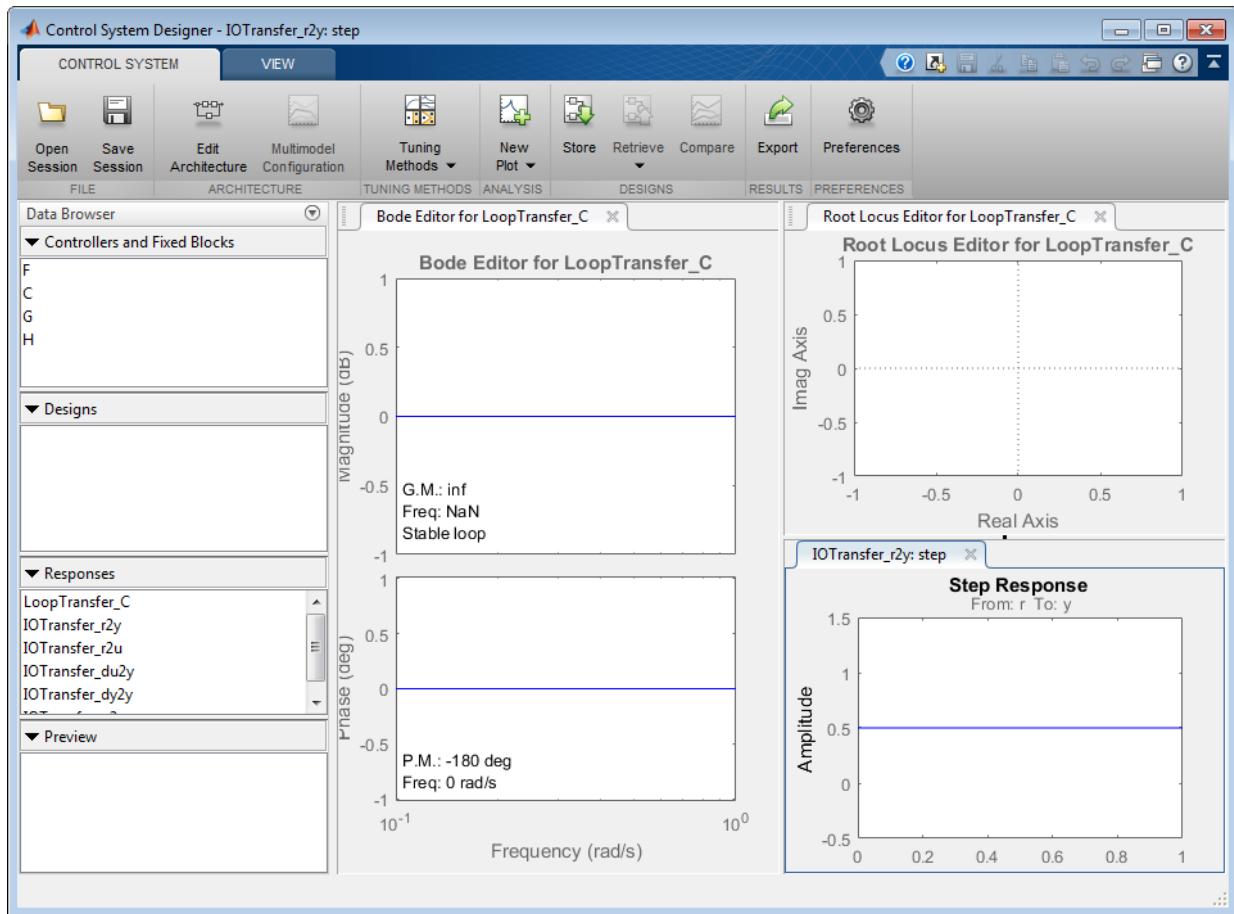
```
s = tf('s');
G1 = (13.3259*s+3.2239)/(8.2677*s+1)^2;
G2 = G1;
Gd = 0.4480/(8.2677*s+1)^2;
```

G1 is the real plant used in controller evaluation. G2 is an approximation of the real plant and it is used as the predictive model in the IMC structure. G2 = G1 means that there is no model mismatch. Gd is the disturbance model.
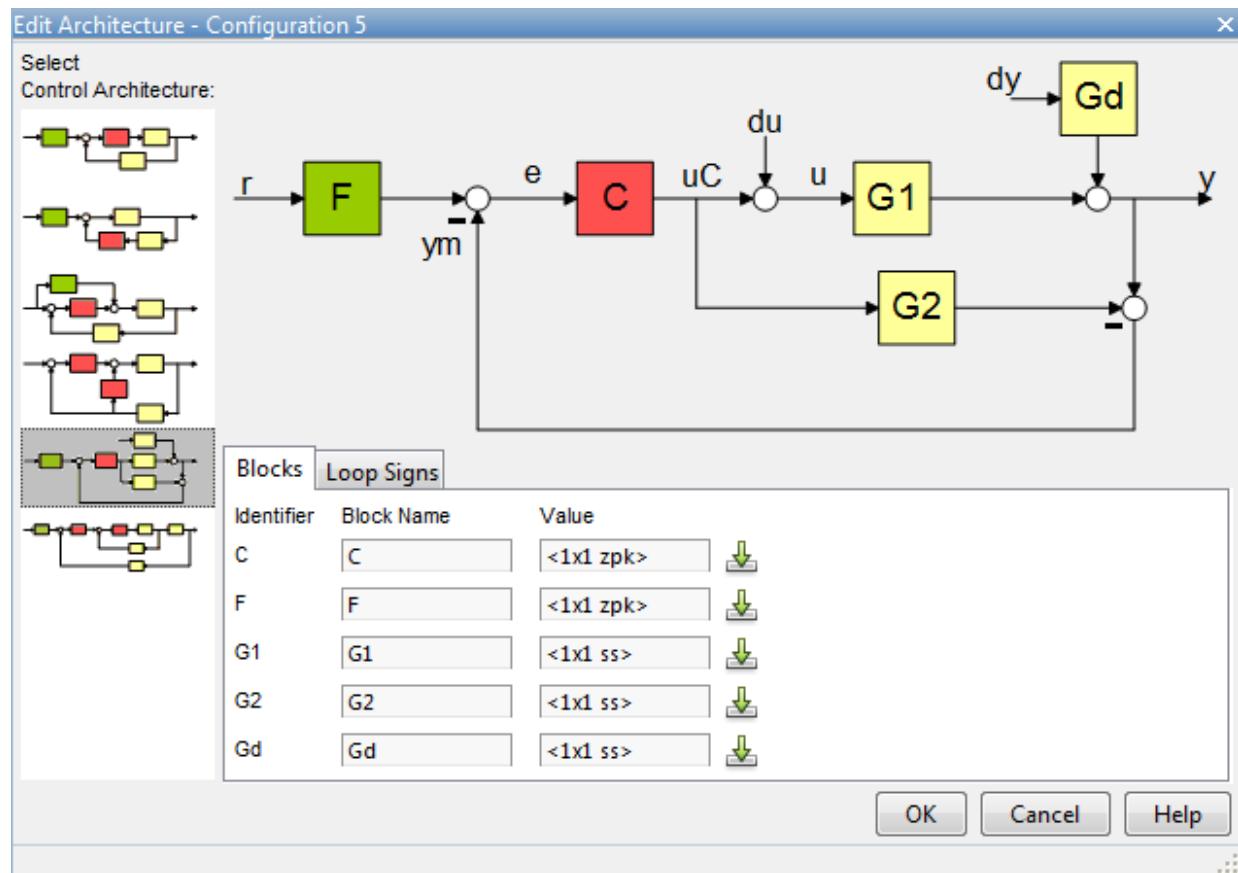
Define IMC Strucutre in Control System Designer

Open Control System Designer.
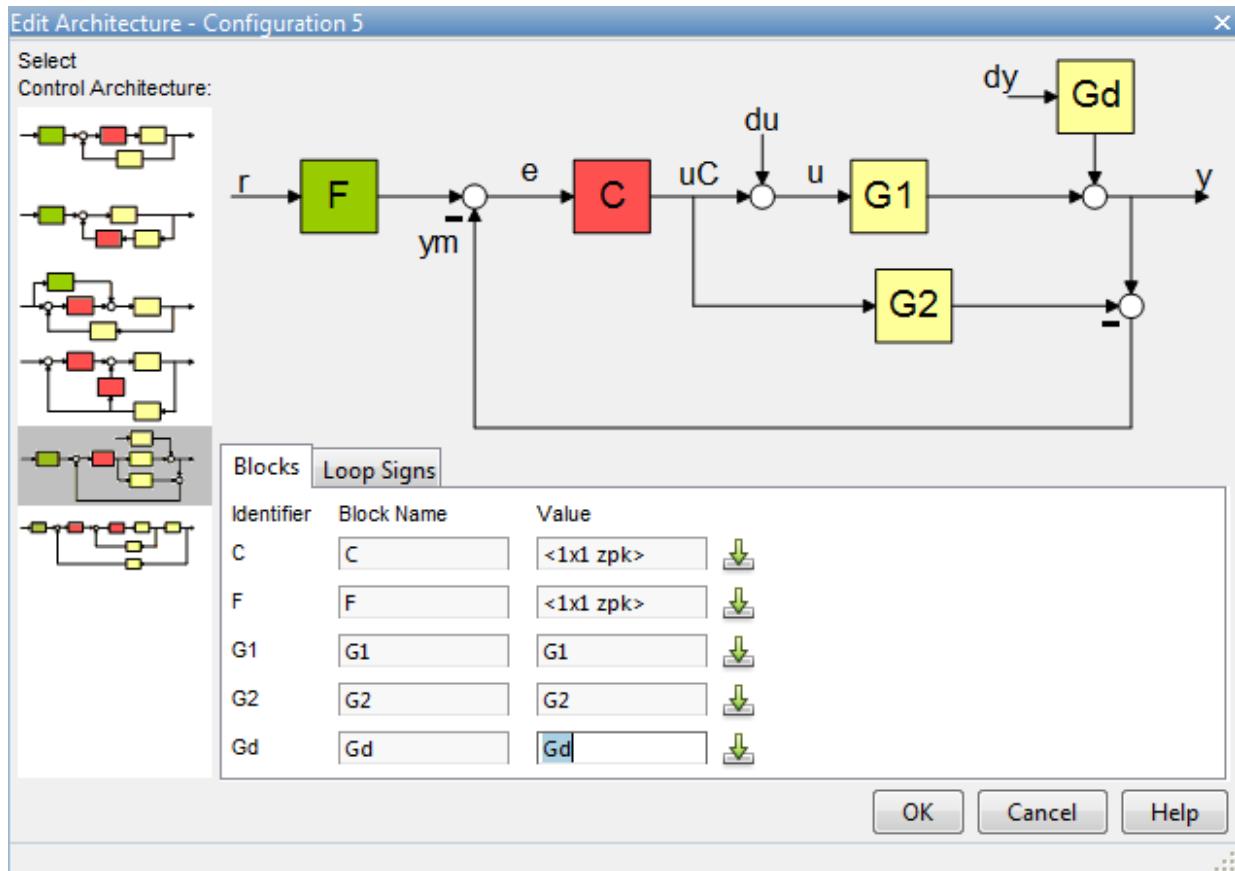
```
controlSystemDesigner
```

Select the IMC control architecture. In Control System Designer, click **Edit Architecture**. In the Edit Architecture dialog box, select Configuration 5.
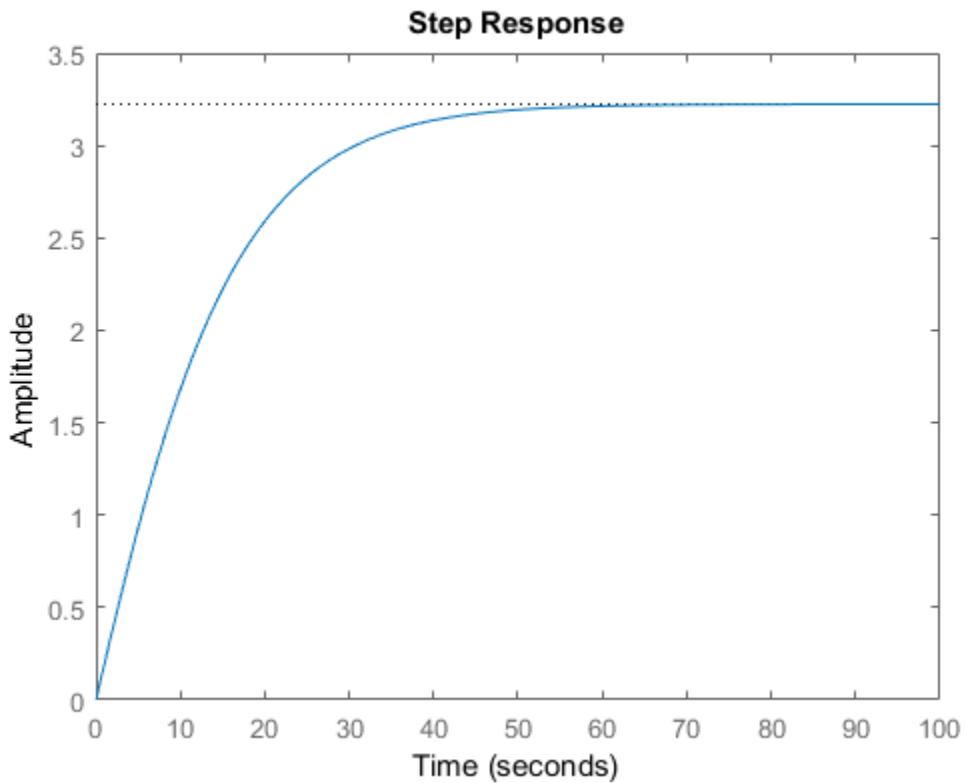
Load the system data. For **G1**, **G2**, and **Gd**, specify a model **Value**.
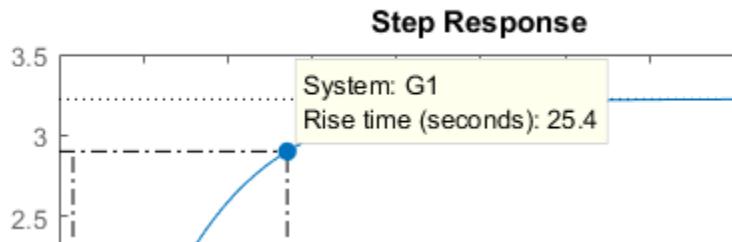
Tune Compensator

Plot the open-loop step response of `G1`.

```
step(G1)
```
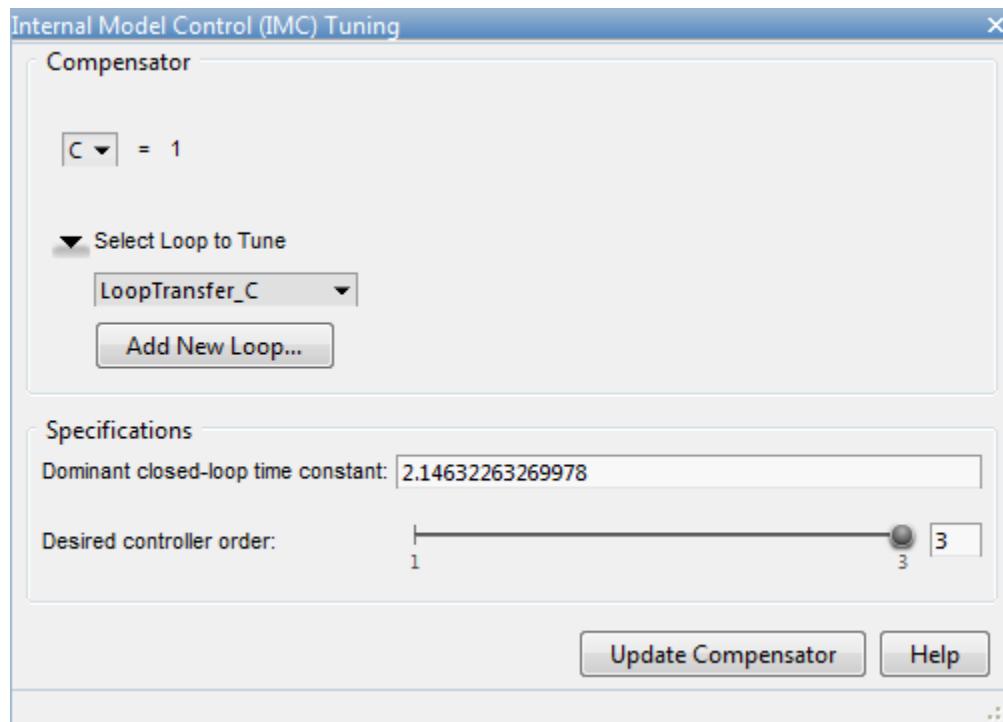
Right-click the plot and select **Characteristics > Rise Time** submenu. Click the blue rise time marker.
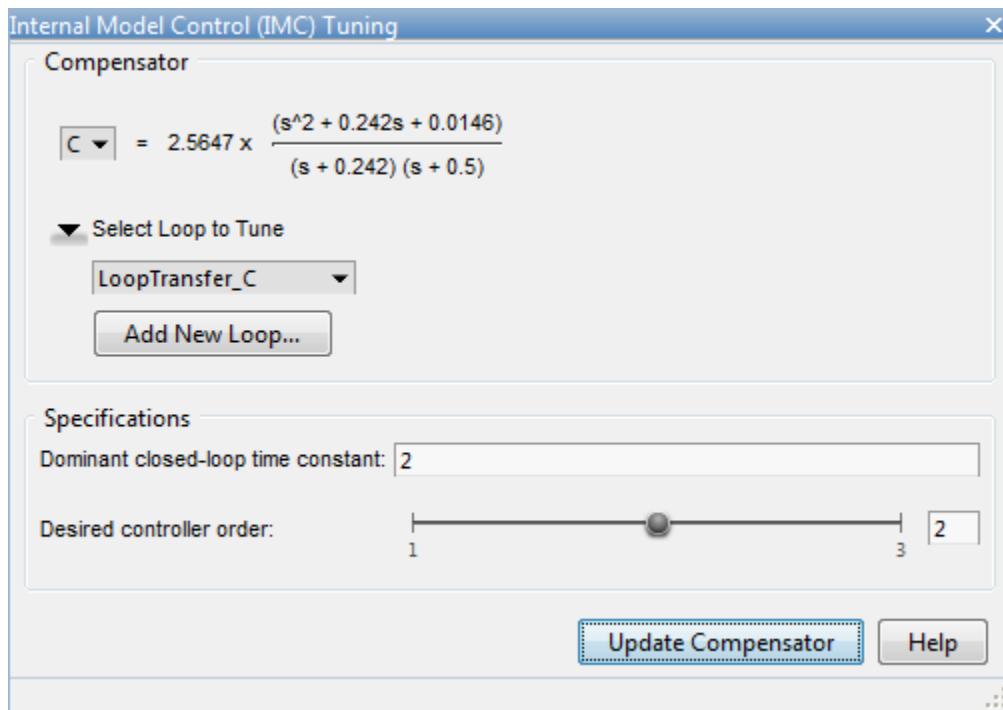
The rise time is about 25 seconds and we want to tune the IMC compensator to achieve a faster closed-loop response time.
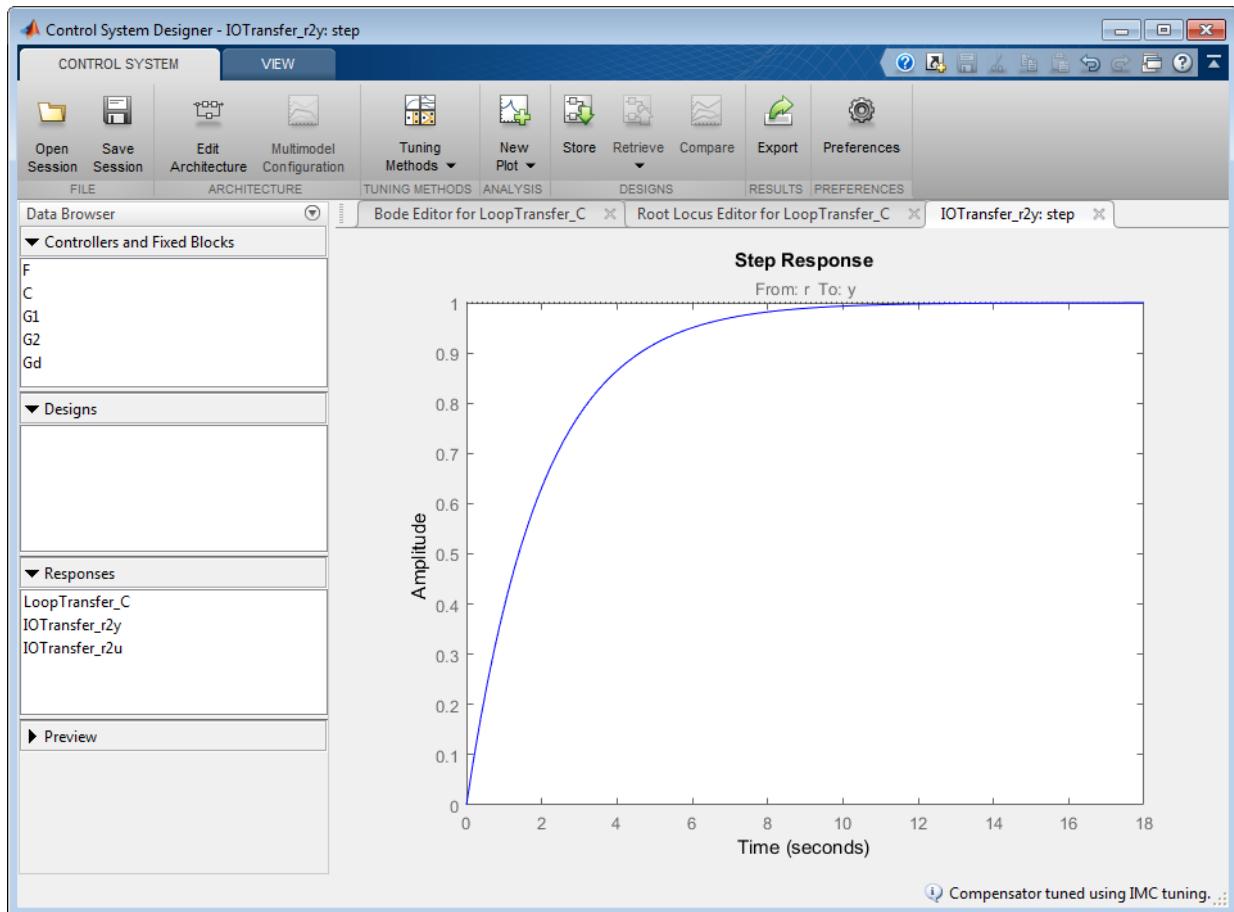
To tune the IMC compensator, click the **Tuning Methods**, and select **Internal Model Control (IMC) Tuning**.

Select a closed-loop time constant of 2 and specify 2 as the desired compensator order. Click **Update Compensator**.

To view the closed-loop step response, in Control System Designer, double-click the **IOTransfer_r2y:step** plot tab.
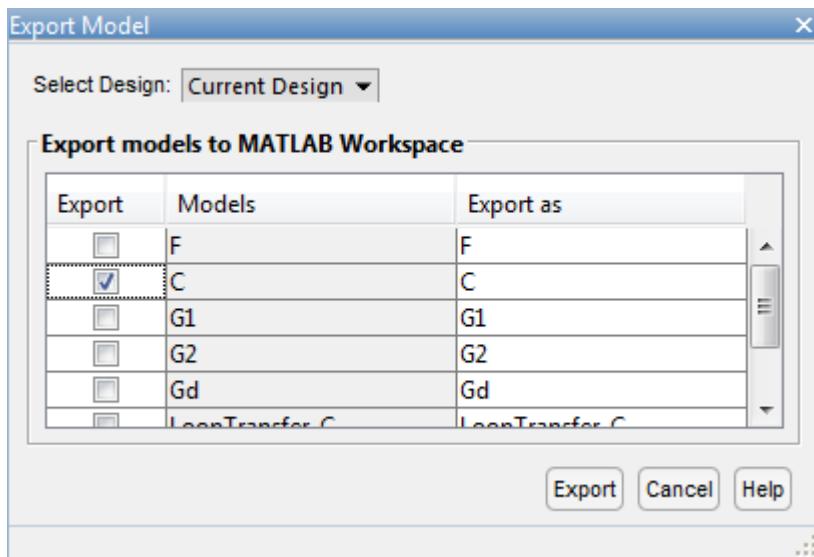
Control Performance with Model Mismatch

When designing the controller, we assumed G1 was equal to G2. In practice, they are often different, and the controller needs to be robust enough to track setpoints and reject disturbances.

Create model mismatches between G1 and G2 and examine the control performance at the MATLAB command line in the presence of both setpoint change and load disturbance.

Export the IMC Compensator to the MATLAB workspace. Click the Export. In the Export Model dialog box, select compensator model **C**.

Click **Export**.

Convert the IMC structure to a classic feeback control structure with the controller in the feedforward path and unit feedback.

```
C = zpk([-0.121 -0.121],[-0.242, -0.466],2.39);
C_new = feedback(C,G2,+1)


C_new =

                2.39 (s+0.121)^4
  -------------------------------------------
  (s-0.0001594) (s+0.121) (s+0.1213) (s+0.2419)

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

Define the following plant models:

• No Model Mismatch:

```
G1p = (13.3259*s+3.2239)/(8.2677*s+1)^2;
```

- **G1** time constant changed by 5%:
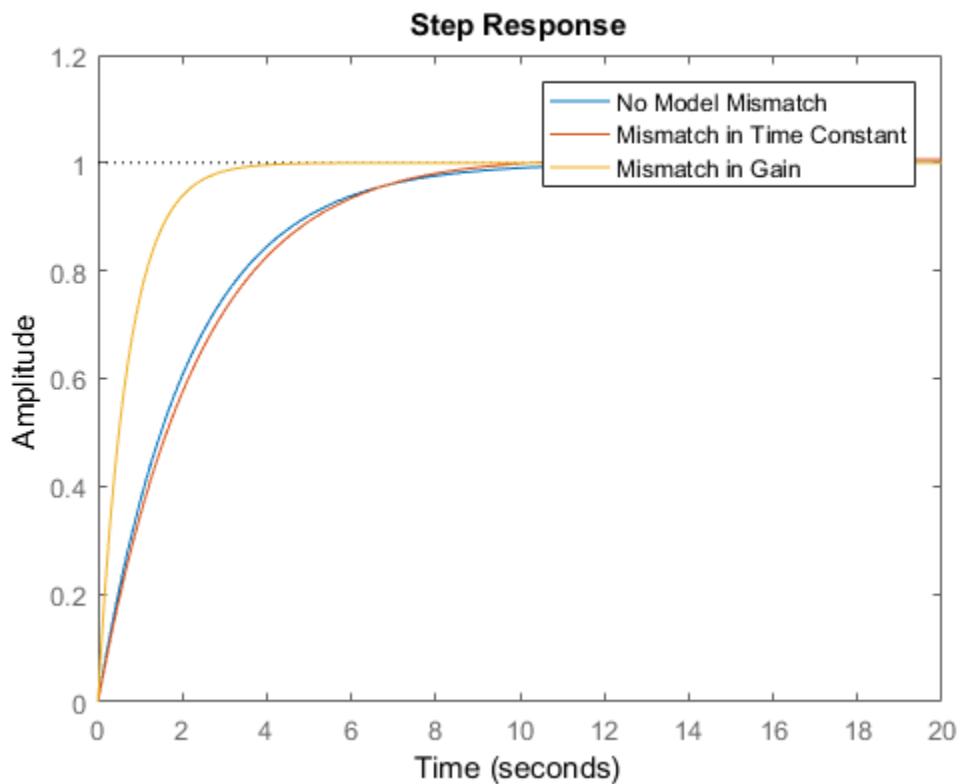
```
G1t = (13.3259*s+3.2239)/(8.7*s+1)^2;
```

- **G1** gain is increased by 3 times:

```
G1g = 3*(13.3259*s+3.2239)/(8.2677*s+1)^2;
```
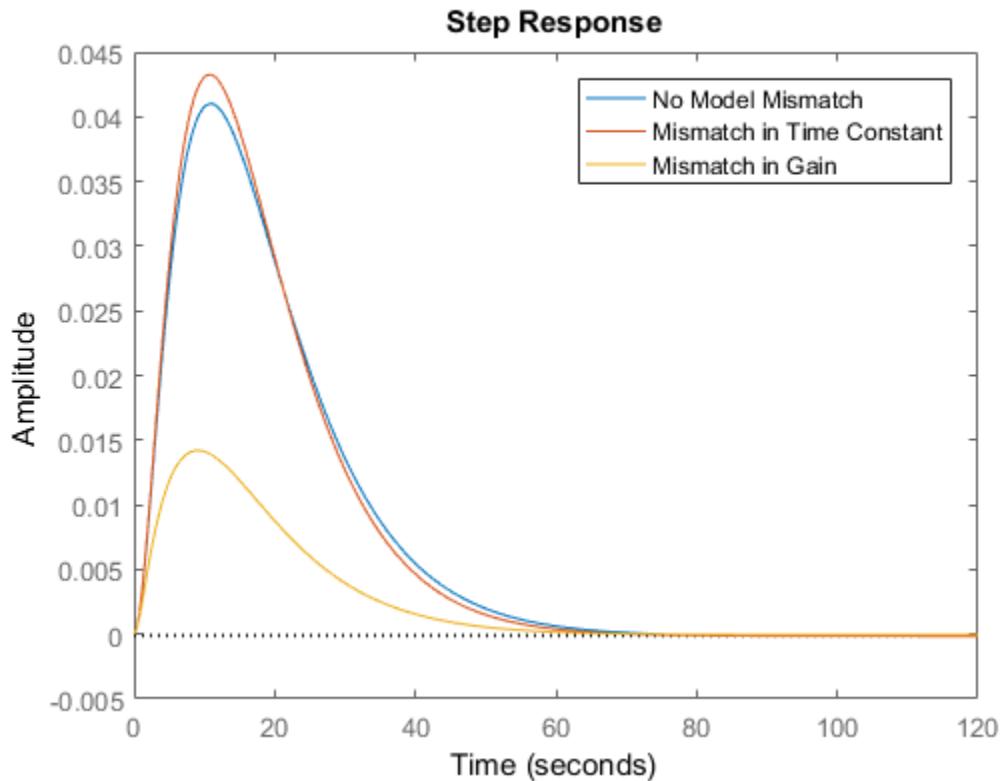
Evaluate the setpoint tracking performance.

```
step(feedback(G1p*C_new,1),feedback(G1t*C_new,1),feedback(G1g*C_new,1))
legend('No Model Mismatch','Mismatch in Time Constant','Mismatch in Gain')
```

Evaluate the disturbance rejection performance.

```
step(Gd*feedback(1,G1p*C_new),Gd*feedback(1,G1t*C_new),Gd*feedback(1,G1g*C_new))
legend('No Model Mismatch','Mismatch in Time Constant','Mismatch in Gain')
```


The controller is fairly robust to uncertainties in the plant parameters.

See Also

Control System Designer

More About

- "Design Compensator Using Automated Tuning Methods" on page 12-96
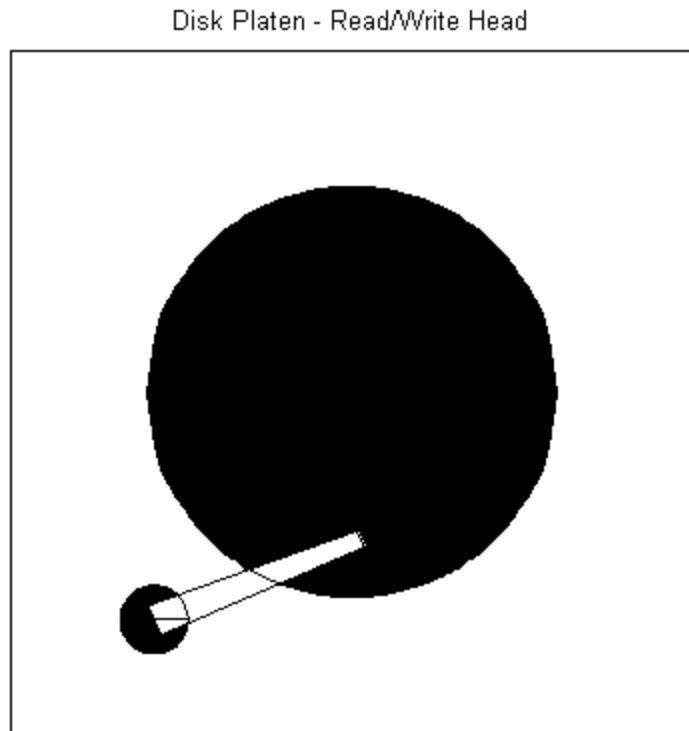
Design LQG Tracker Using Control System Designer

This example shows how to use LQG synthesis to design a feedback controller for a disk drive read/write head using Control System Designer.

For details about the system and model, see Chapter 14 of "Digital Control of Dynamic Systems," by Franklin, Powell, and Workman.

Disk Drive Model

Below is a picture of the system to be modeled.

Disk Platen - Read/Write Head

The model input is the current driving the voice coil motor, and the output is the position error signal (PES, in % of track width). To learn more about the 10th order model, see "Digital Servo Control of a Hard-Disk Drive". The plant includes a small time delay. For the purpose of this example, ignore this delay.

```
load diskdemo
Gr = tf(1e6,[1 12.5 0]);
Gf1 = tf(w1*[a1 b1*w1],[1 2*z1*w1 w1^2]); % first  resonance
Gf2 = tf(w2*[a2 b2*w2],[1 2*z2*w2 w2^2]); % second resonance
Gf3 = tf(w3*[a3 b3*w3],[1 2*z3*w3 w3^2]); % third  resonance
Gf4 = tf(w4*[a4 b4*w4],[1 2*z4*w4 w4^2]); % fourth resonance
G = (ss(Gf1)+Gf2+Gf3+Gf4) * Gr;     % convert to state space for accuracy
```

Design Overview

In this example, design a full-ordered LQG tracker, which places the read/write head at the correct position. Tune the LQG tracker to achieve specific performance requirements and reduce the controller order as much as possible. For example, turn the LQG tracker into a PI controller format.
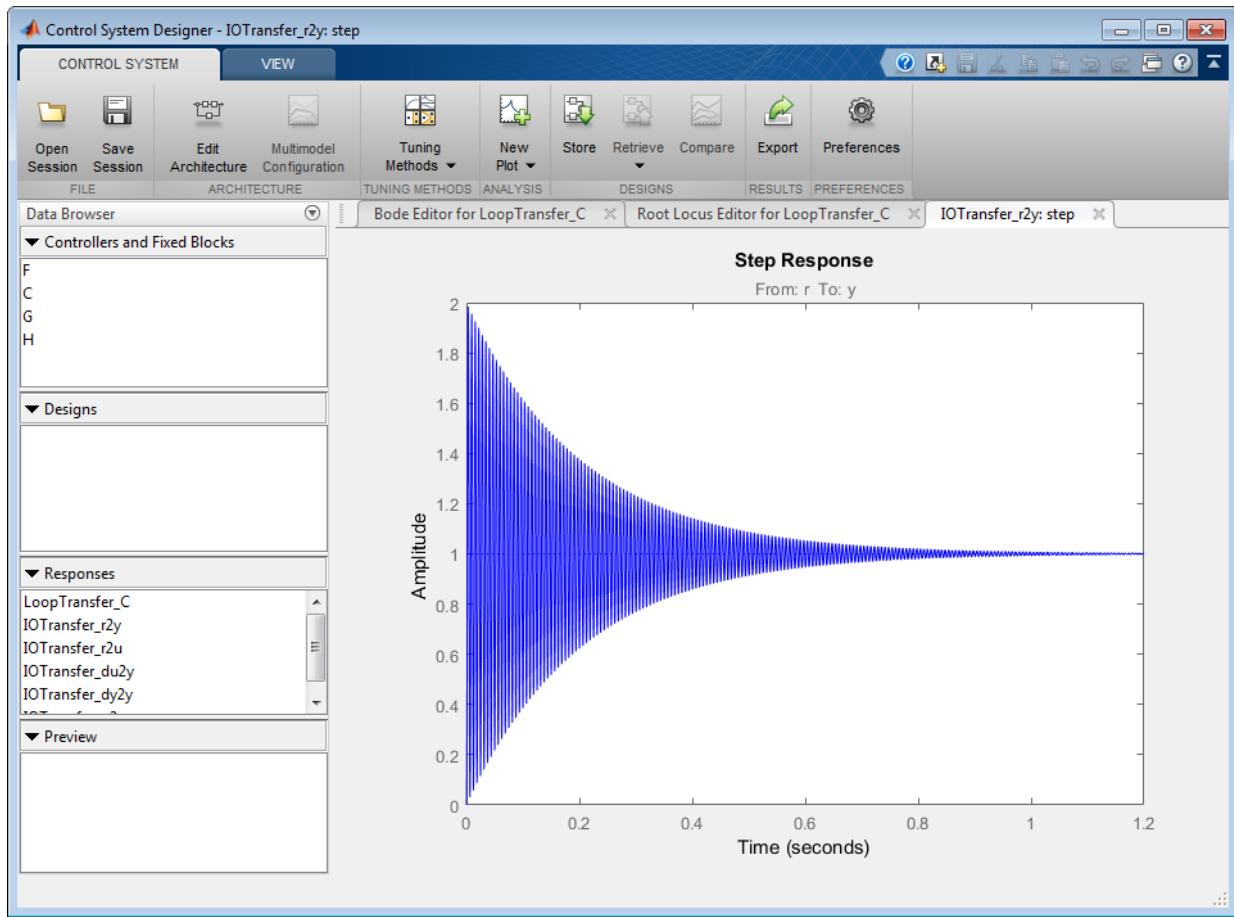
Open Control System Designer

Open Control System Designer, importing the plant model.

```
controlSystemDesigner(G)
```

By default, Control System Designer displays the step response of the closed-loop system along with Bode and root locus graphical editors for the open-loop response.
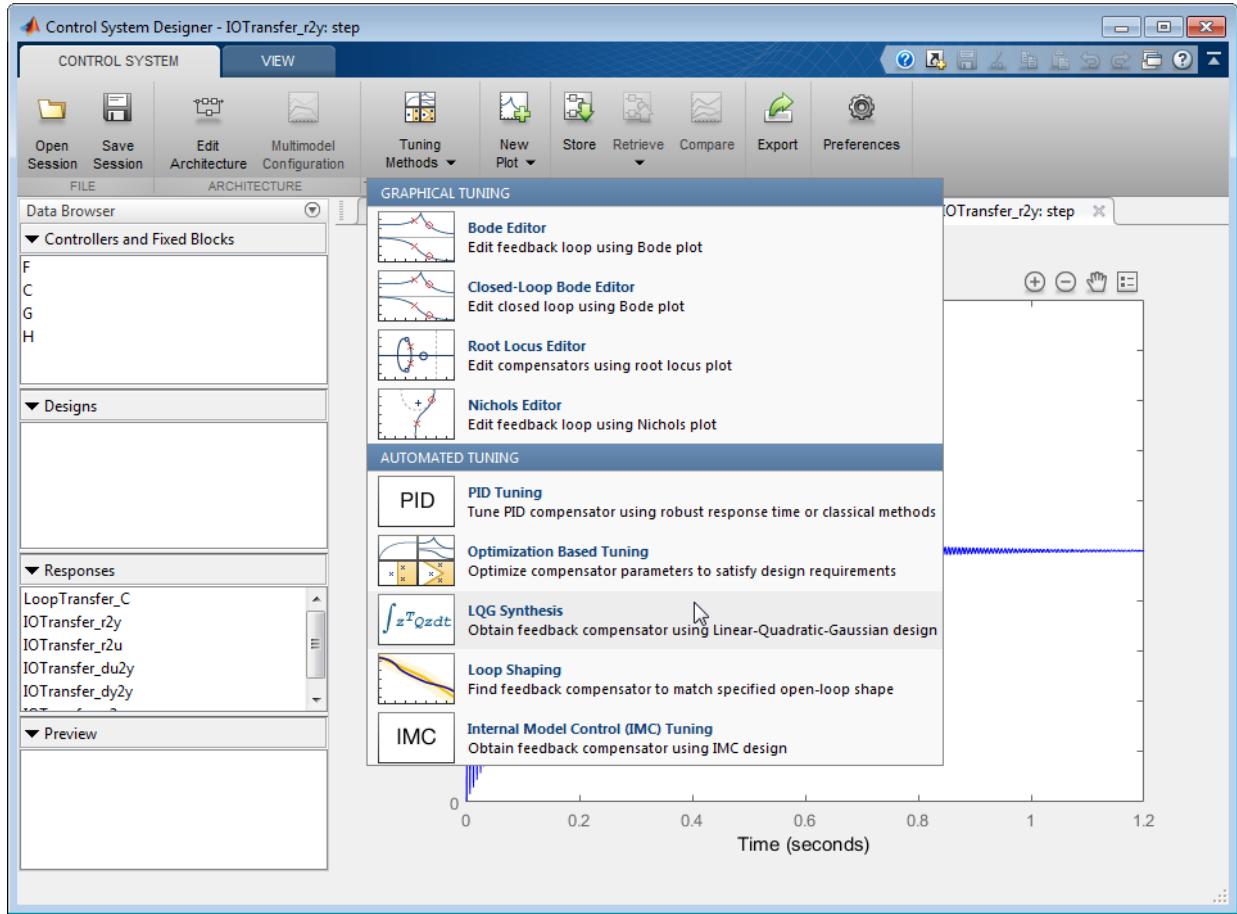
Maximize the step response. Double-click the **IOTransfer_r2y: step** plot tab. Details about how to use the Control System Designer are described in "Getting Started with the Control System Designer".

The default unity gain compensator produces a stable closed-loop system with large oscillations.

Design a Full-Order LQG Tracker

Click **Tuning Methods**, and select **LQG Synthesis**.
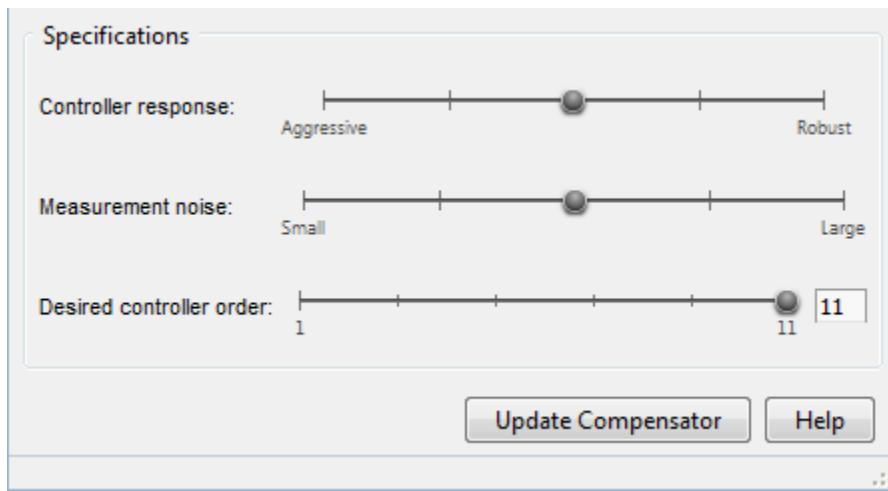
In the LQG Synthesis dialog box, in the **Specifications** section, set requirements on the controller performance:
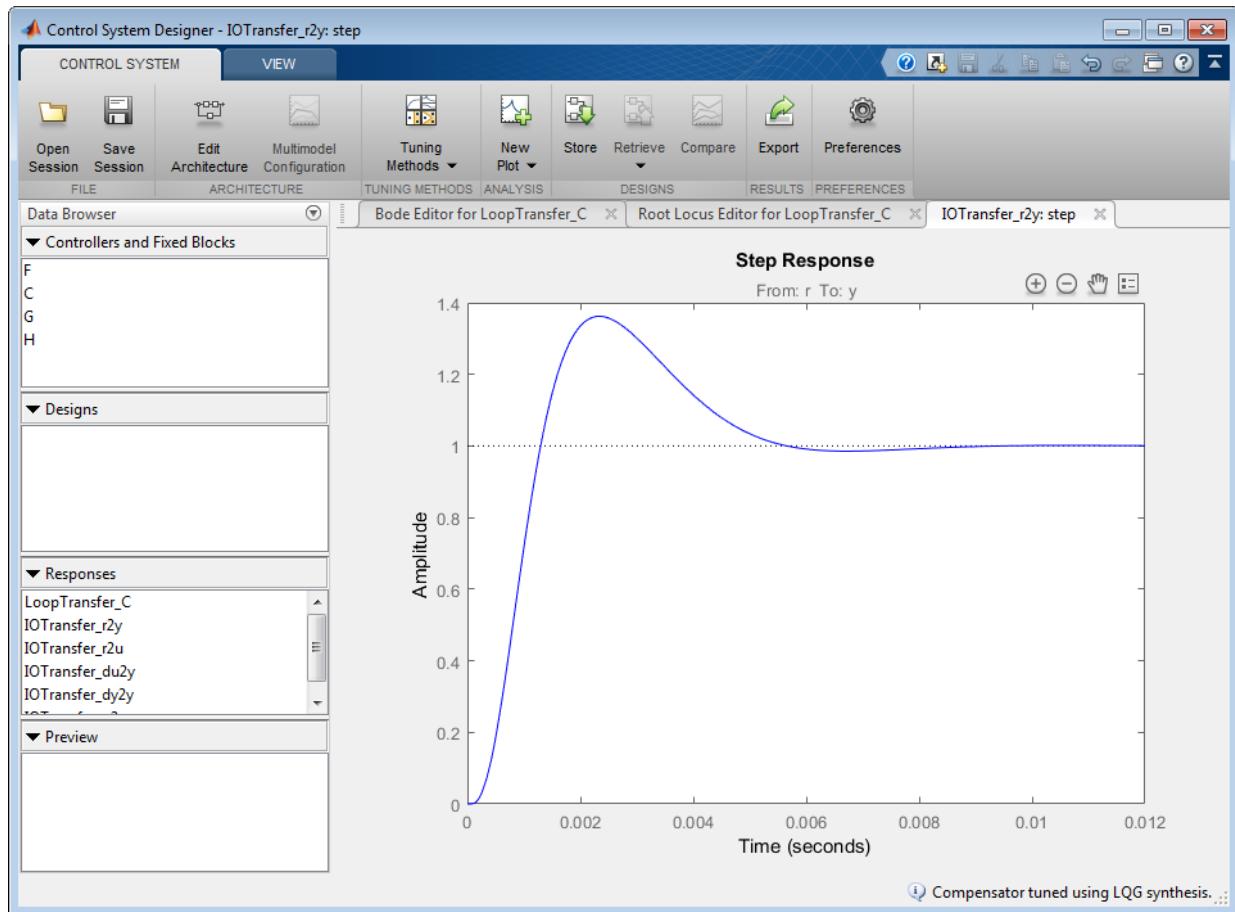
- **Controller response** - Specify the controller transient behavior. You can make the controller more aggressive at disturbance rejection or more robust against plant uncertainty. If you believe your model is accurate and that the manipulated variable has a large enough range, an aggressive controller is preferable.

- **Measurement noise** - Specify an estimate of the level of output measurement noise for your application. To produce a more robust controller, specify a larger noise estimate.

- **Desired controller order** - Specify your controller order preference.
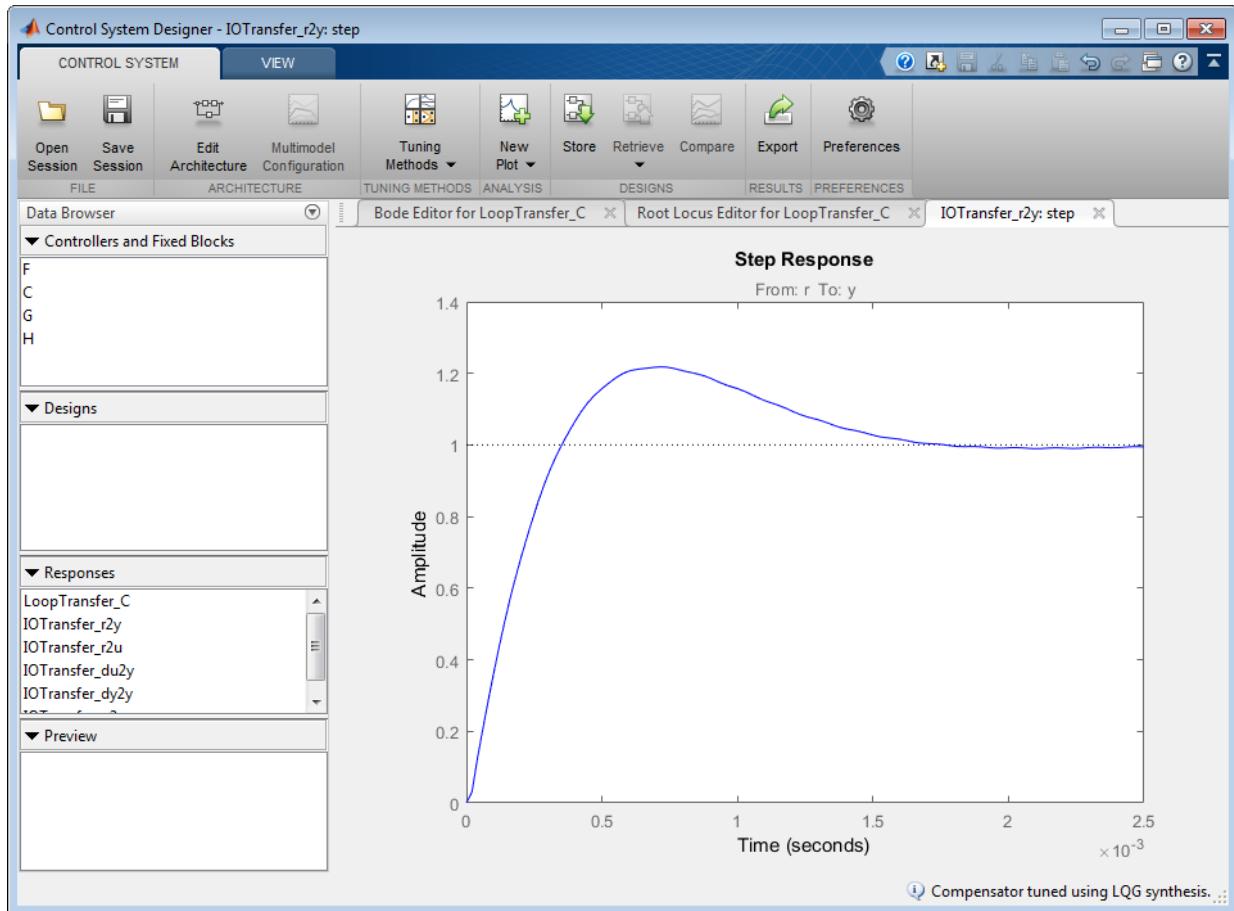
Use default slider settings as the initial controller design.

Click **Update Compensator**. The new **Compensator** is displayed, and the step response updates.
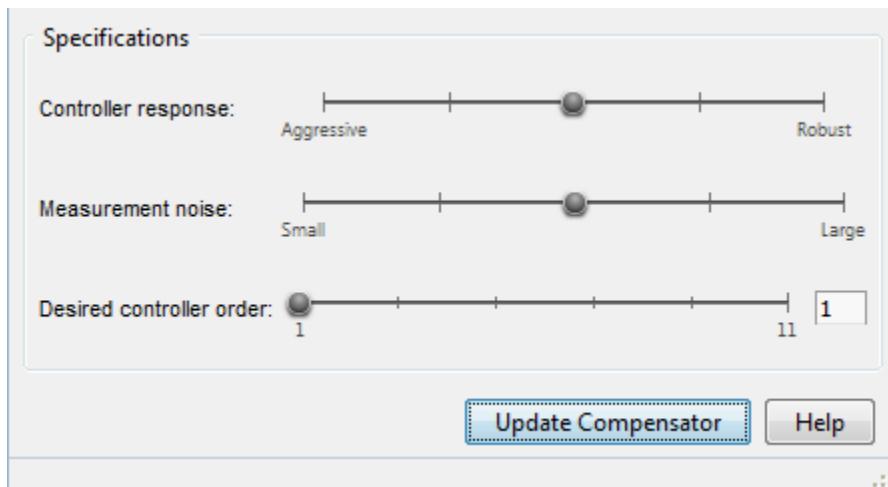
To design a more aggressive controller, move the **Controller response** slider to the far left. The more aggressive controller reduces the overshoot by 50% and reduces the settling time by 70%.
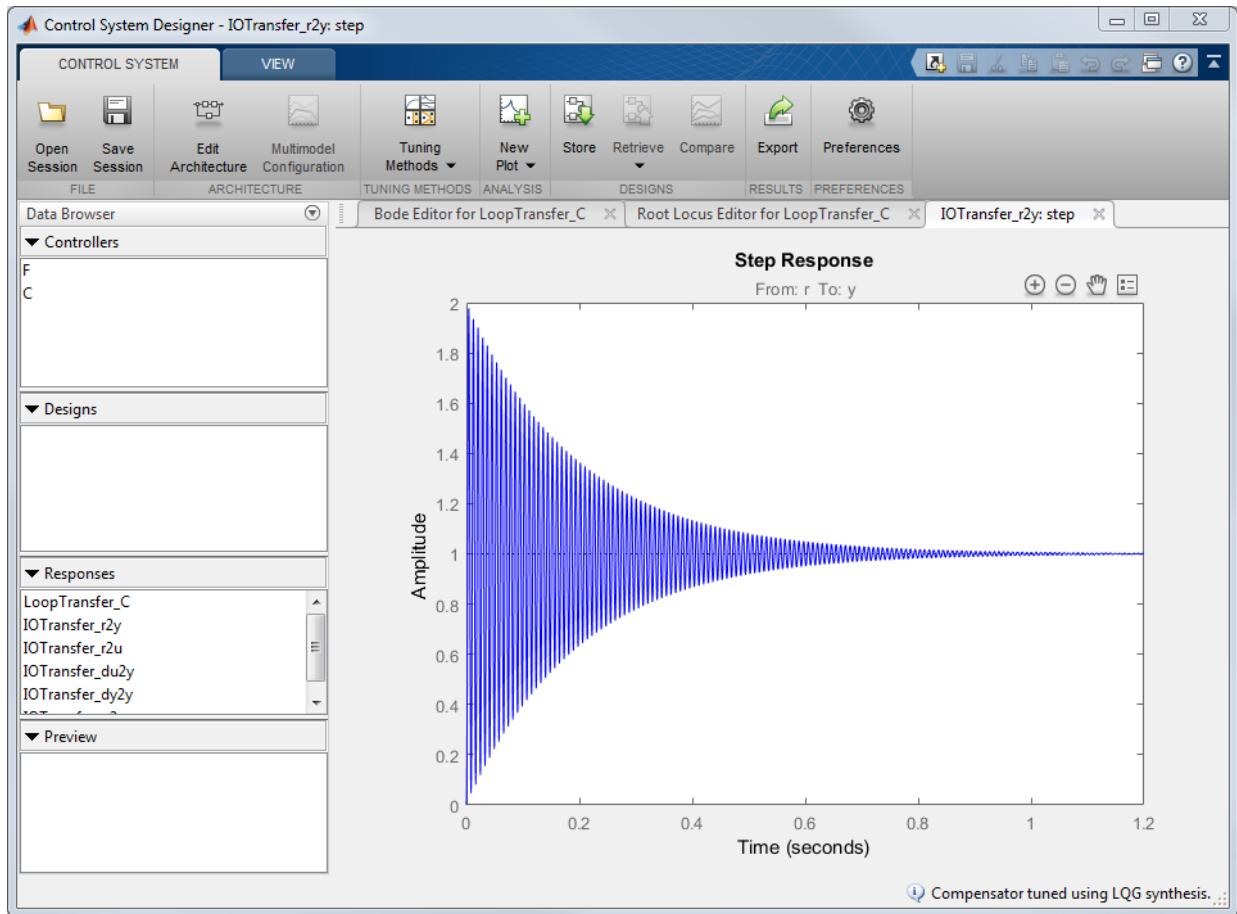
Design a Reduced-Order LQG Tracker
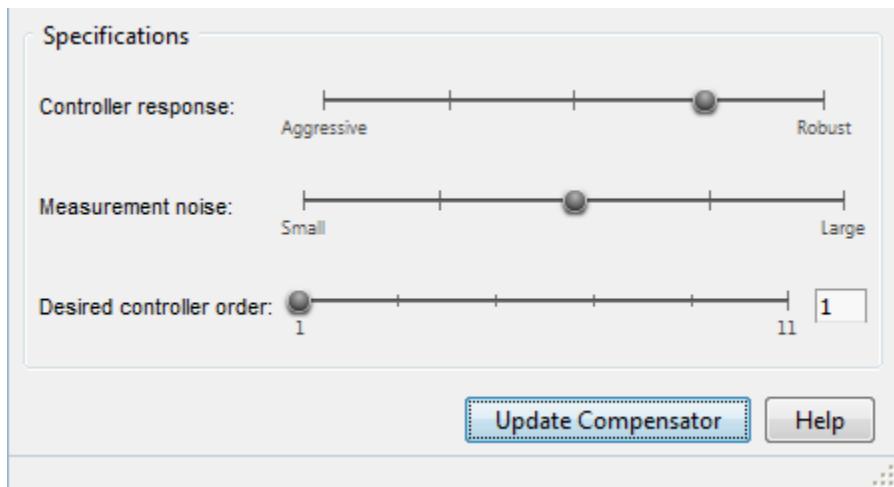
To create a PI controller, reset the **Controller response** slider to the middle default value, and set the **Desired controller order** to 1.
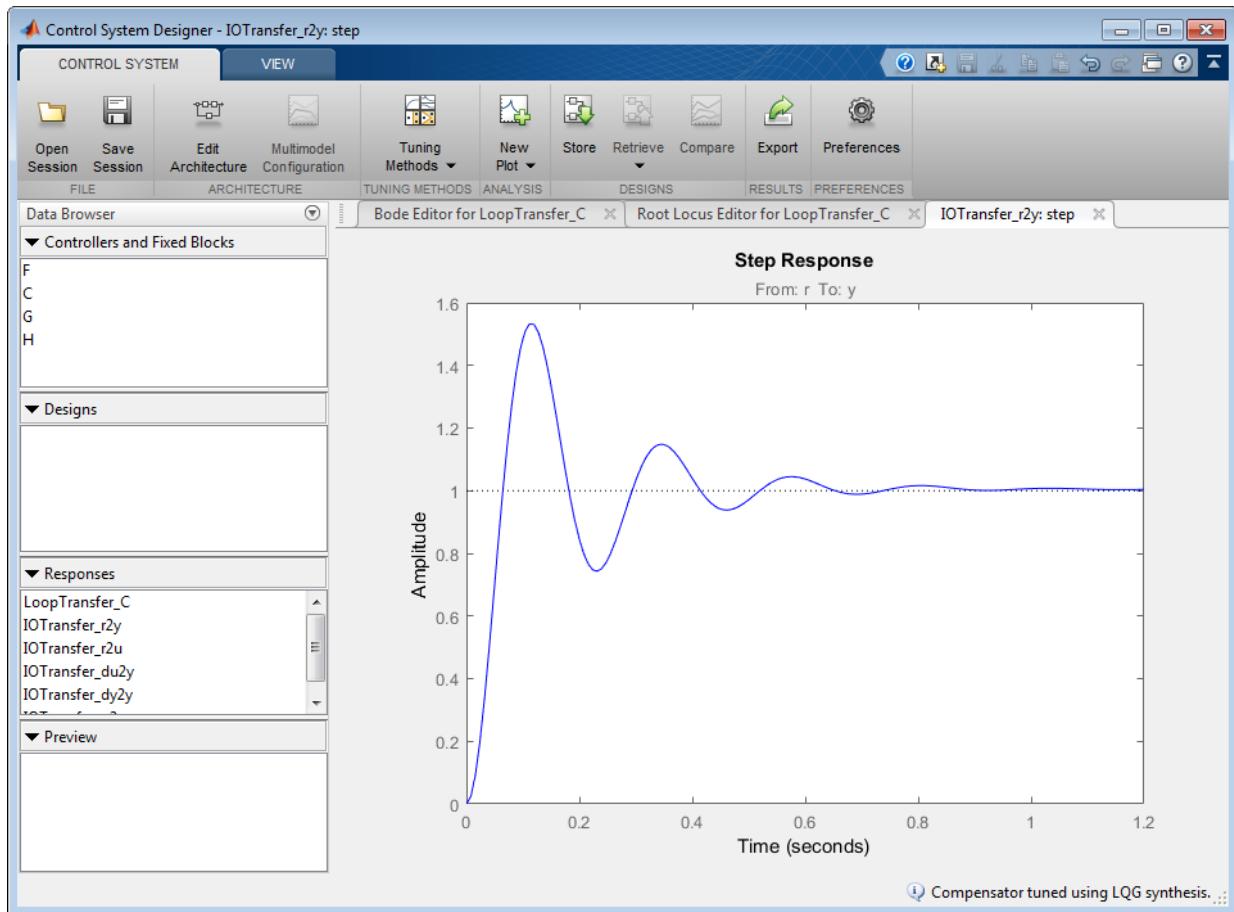
Click **Update Compensator**.

This controller produces a heavily oscillating closed-loop system. To make the controller less aggressive, move the **Controller response** slider to the right.

Click **Update Compensator**.

The step response shows that the PI controller design provides a good starting point for optimization-based design. For more information, see "Getting Started with the Control System Designer".

See Also
Control System Designer

More About
·   "Design Compensator Using Automated Tuning Methods" on page 12-96

Control System Tuning

13

Control System Tuning
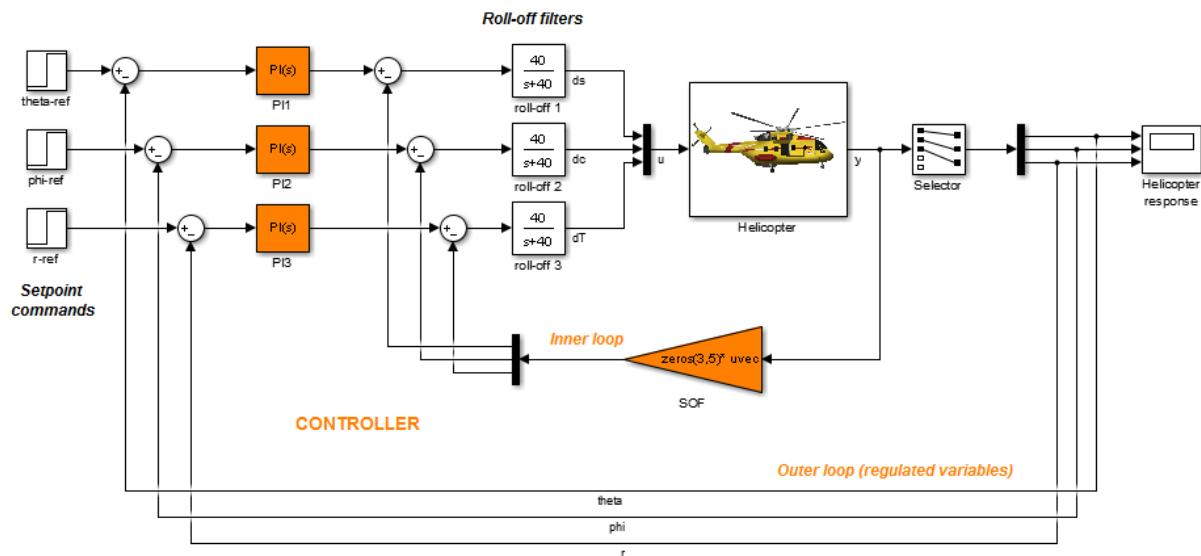
Automated Tuning Overview

The control system tuning tools such as `systune` and Control System Tuner automatically tune control systems from high-level tuning goals you specify, such as reference tracking, disturbance rejection, and stability margins. The software jointly tunes all the free parameters of your control system regardless of control system architecture or the number of feedback loops it contains. For example, the model of the following illustration represents a multiloop control system for a helicopter.

This control system includes a number of fixed elements, such as the helicopter model itself and the roll-off filters. The inner control loop provides static output feedback for decoupling. The outer loop includes PI controllers for setpoint tracking. The tuning tools jointly optimize the gains in the `SOF` and `PI` blocks to meet setpoint tracking, stability margin, and other requirements that you specify. These tools allow you to specify any control structure and designate which blocks in your system are tunable.

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

loop damping, and stability margins. Using these tools, you can perform multi-objective tuning of control systems having any structure.

See Also

Control System Designer | `systune`
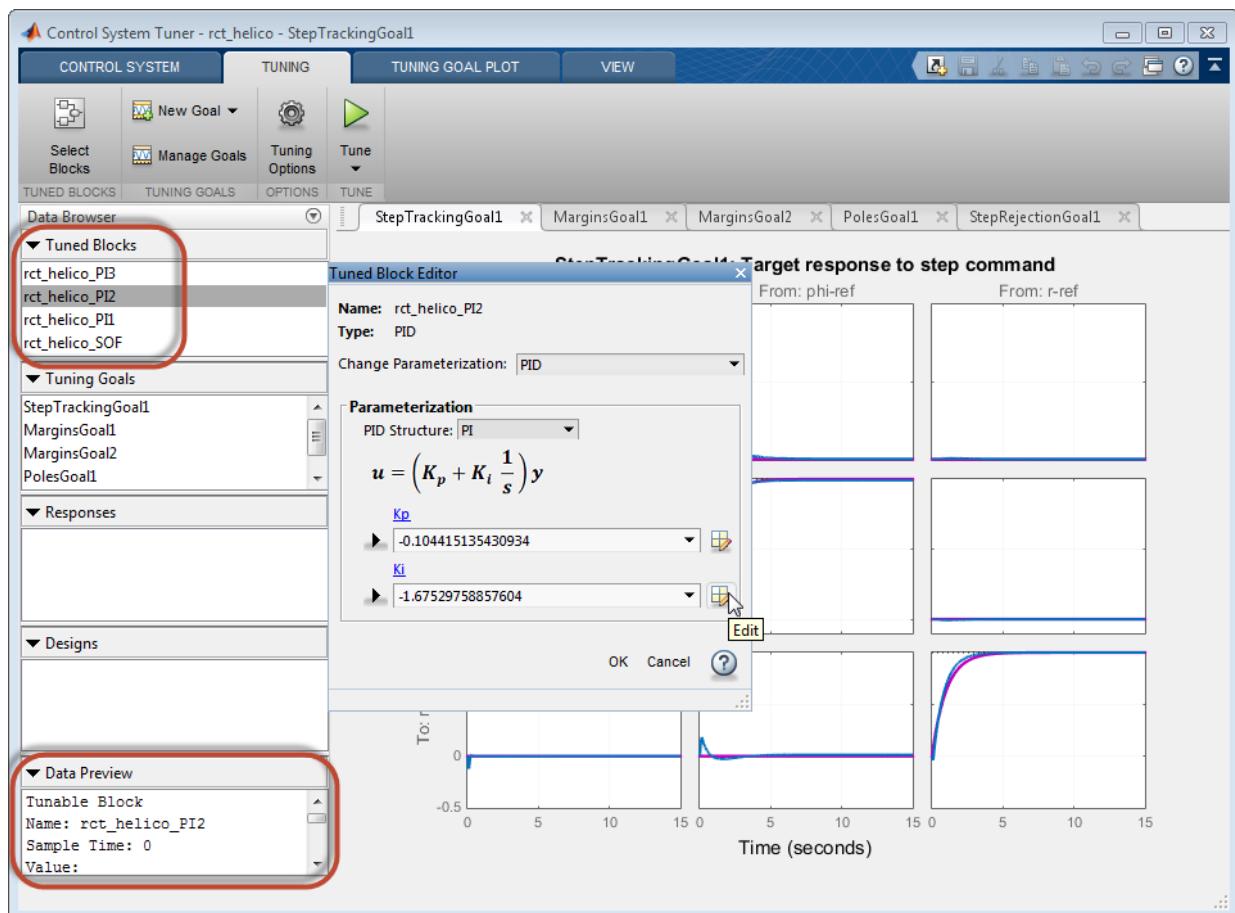
More About

Choosing an Automated Tuning Approach

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

Control System Tuner provides an interactive graphical interface for specifying your tuning goals and validating the performance of the tuned control system.

Use Control System Tuner to tune control systems consisting of any number of feedback loops, with tunable components having any structure (such as PID, gain block, or state-

space). You can represent your control architecture in MATLAB as a tunable generalized state-space (`genss`) model. If you have Simulink Control Design software, you can tune a control system represented by a Simulink model. Use the graphical interface to configure your tuning goals, examine response plots, and validate your controller design.

The `systune` command can perform all the same tuning tasks as Control System Tuner. Tuning at the command line allows you to write scripts for repeated tuning tasks. `systune` also provides advanced techniques such as tuning a controller for multiple plants, or designing gain-scheduled controllers. To use the command-line tuning tools, you can represent your control architecture in MATLAB as a tunable generalized state-space (`genss`) model. If you have Simulink Control Design software, you can tune a control system represented by a Simulink model using an `slTuner` interface. Use the `TuningGoal` requirement objects to configure your tuning goals. Analysis commands such as `getIOTransfer` and `viewSpec` let you examine and validate the performance of your tuned system.

See Also

Control System Designer | `systune`

More About

*   "Automated Tuning Workflow" on page 13-8

Automated Tuning Workflow

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

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

- Create a Simulink model of your control system. (Tuning a Simulink model requires Simulink Control Design software.)

- Use a predefined control architecture available in Control System Tuner.

- At the command line, build a tunable `genss` model of your control system out of numeric LTI models and tunable control design blocks.

For more information, see "Specify Control Architecture in Control System Tuner" on page 13-10.

2 Set up your model for tuning.

- In Control System Tuner, identify which blocks of the model you want to tune. See Model Setup for Control System Tuner.

- If tuning a Simulink model at the command line, create and configure the `slTuner` interface to the model. See Setup for Tuning Simulink Models at the Command Line.

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

- In Control System Tuner, use the graphical interface to specify tuning goals. See "Tuning Goals".

- At the command-line, use the `TuningGoal` requirement objects to specify your tuning goals. See "Tuning Goals".

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

- For tuning in Control System Tuner, see "Tuning, Analysis, and Validation".

- For tuning at the command line, see `systune`.

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

- For validation in Control System Tuner, see "Tuning, Analysis, and Validation".
- For validating at the command line, see "Validate Tuned Control System at the Command Line" on page 13-246.

Specify Control Architecture in Control System Tuner

In this section...
"About Control Architecture" on page 13-10
"Predefined Feedback Architecture" on page 13-11
"Arbitrary Feedback Control Architecture" on page 13-12
"Control System Architecture in Simulink" on page 13-14
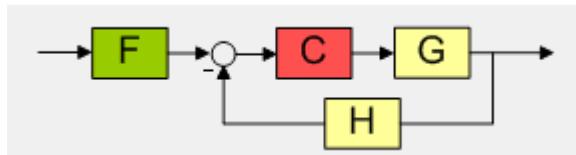
About Control Architecture

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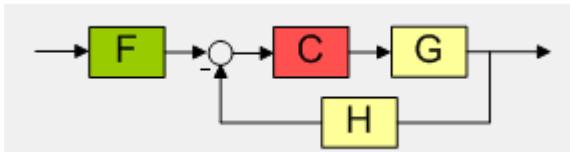
G is the plant model, and H the sensor dynamics. These are usually the fixed components of the control system. The prefilter F and feedback controller C are the tunable elements. Because control systems are so conveniently expressed in this block diagram form, these elements are referred to as fixed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

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

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

Predefined Feedback Architecture

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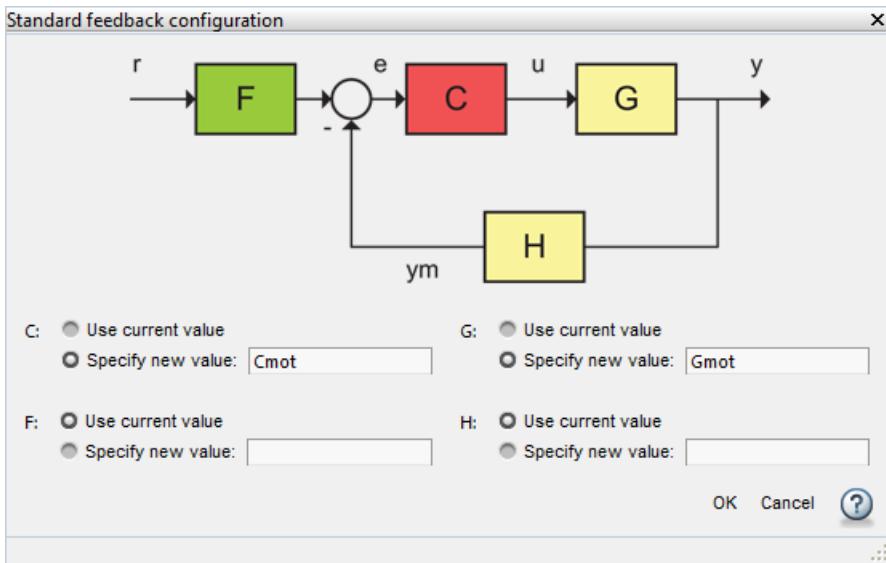
For example, suppose you have a DC motor for which you want to tune a PID controller. The response of the motor is modeled as $G(s) = 1/(s + 1)^2$. Create a fixed LTI model representing the plant, and a tunable PID controller model.

```
Gmot = zpk([],[-1,-1],1);
Cmot = tunablePID('Cmot','PID');
```

Open Control System Tuner.

```
controlSystemTuner
```

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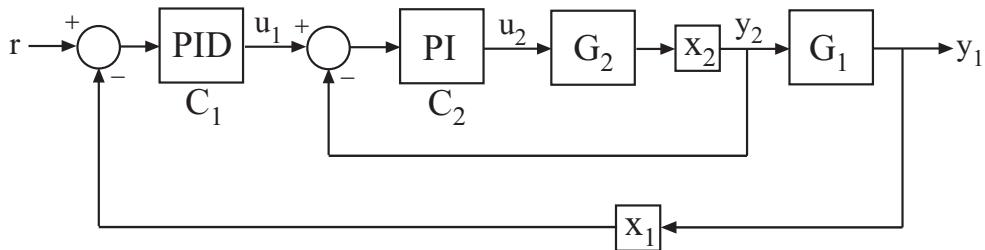
Enter the values for *C* and *G* that you created. Control System Tuner reads these values from the MATLAB workspace. Click **OK**.

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

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

Arbitrary Feedback Control Architecture

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

.

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

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

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

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

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

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

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

Open Control System Tuner to tune this model.

```
controlSystemTuner(CL0)
```

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

Related Examples

- "Building Tunable Models" on page 13-267

- "Specify Blocks to Tune in Control System Tuner" on page 13-24

- "Specify Goals for Interactive Tuning" on page 13-39
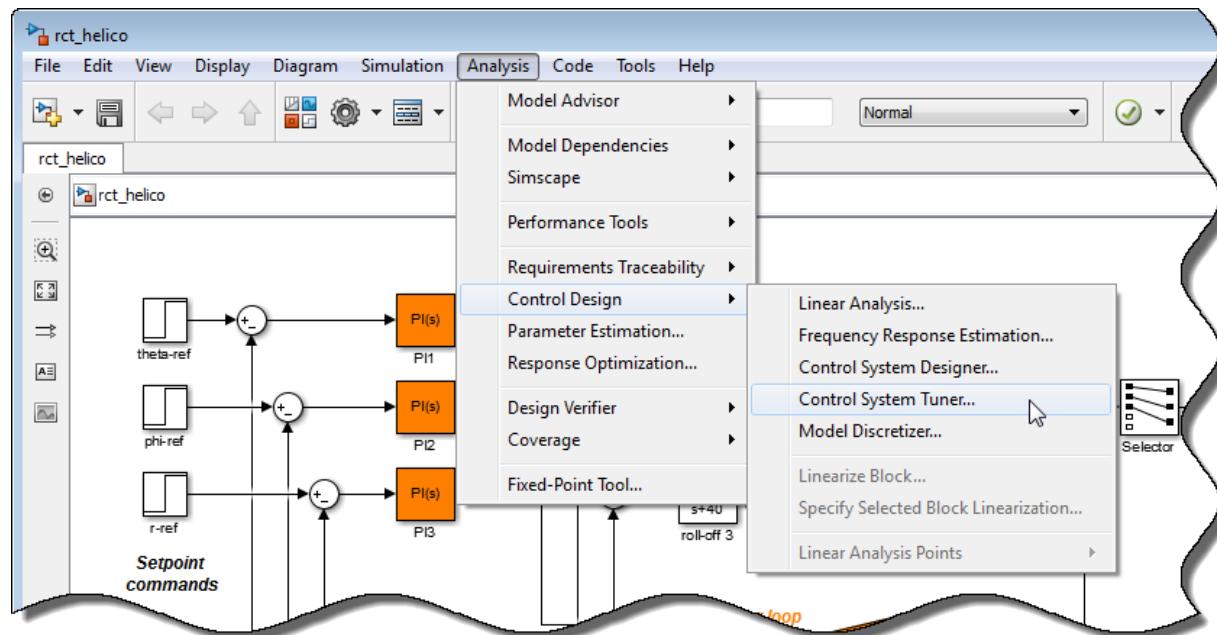
Control System Architecture in Simulink

If you have Simulink Control Design software, you can model an arbitrary control system architecture in a Simulink model and tune the model in Control System Tuner.
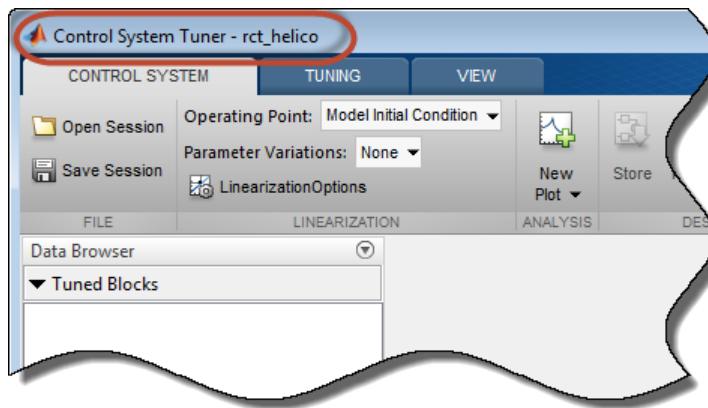
See "Open Control System Tuner for Tuning Simulink Model" on page 13-15.

Open Control System Tuner for Tuning Simulink Model

To open Control System Tuner for tuning a Simulink model, open the model. In the Simulink Editor, select **Analysis** > **Control Design** > **Control System Tuner**.

Each instance of Control System Tuner is linked to the Simulink model from which it is opened. The title bar of the Control System Tuner window reflects the name of the associated Simulink model.

Command-Line Equivalents

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

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

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

```
controlSystemTuner(SLT0)
```

Related Examples

- "Specify Operating Points for Tuning in Control System Tuner" on page 13-17
- "Specify Blocks to Tune in Control System Tuner" on page 13-24

More About

- "Automated Tuning Workflow" on page 13-8

Specify Operating Points for Tuning in Control System Tuner

In this section...
"About Operating Points in Control System Tuner" on page 13-17
"Linearize at Simulation Snapshot Times" on page 13-17
"Compute Operating Points at Simulation Snapshot Times" on page 13-19
"Compute Steady-State Operating Points" on page 13-21

About Operating Points in Control System Tuner

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

By default, Control System Tuner linearizes at the operating point specified in the model, which comprises the initial state values in the model (the model initial conditions). You can specify one or more alternate operating points for tuning the model. Control System Tuner lets you compute two types of alternate operating points:
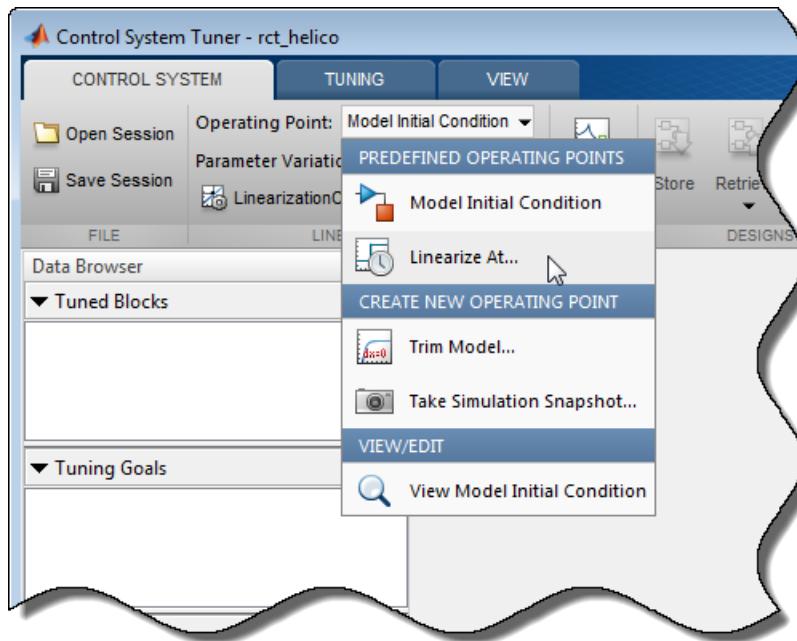
- Simulation snapshot time. Control System Tuner simulates the model for the amount of time you specify, and linearizes using the state values at that time. Simulation snapshot linearization is useful, for instance, when you know your model reaches an equilibrium state after a certain simulation time.

- Steady-state operating point. Control System Tuner finds a steady-state operating point at which some specified condition is met (trimming). For example, if your model represents an automobile motor, you can compute an operating point at which the motor operates steadily at 2000 rpm.

For more information on finding steady-state operating points, see "About Operating Points" and "Computing Steady-State Operating Points" in the Simulink Control Design documentation.
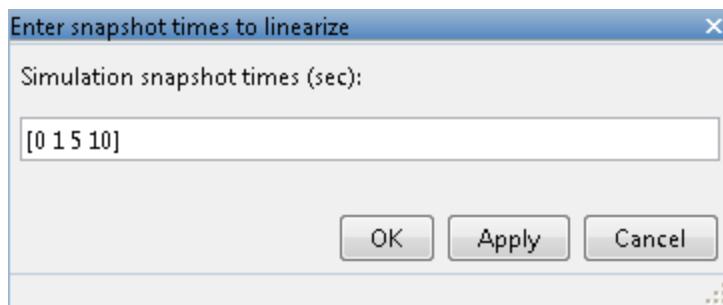
Linearize at Simulation Snapshot Times

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

In the **Control System** tab, in the **Operating Point** menu, select Linearize At.

In the **Enter snapshot times to linearize** dialog box, specify one or more simulation snapshot times. Click **OK**.
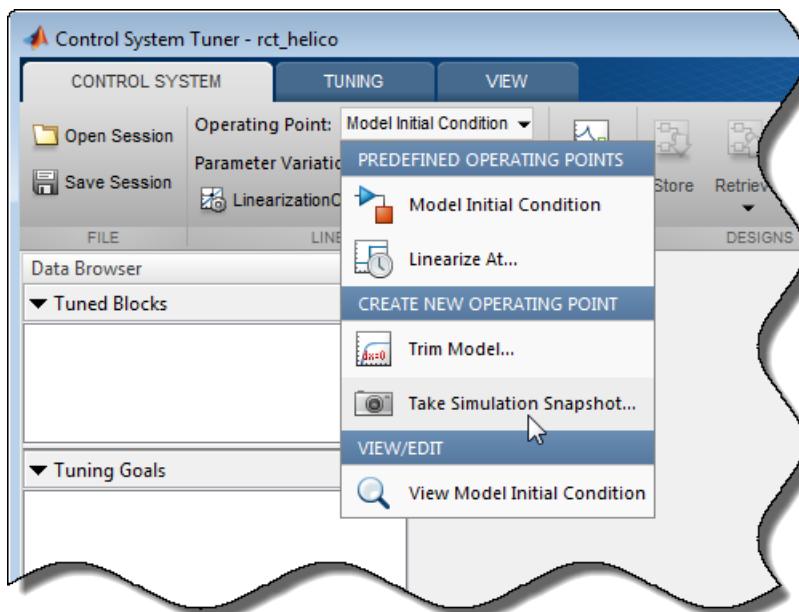
When you are ready to analyze system responses or tune your model, Control System Tuner computes linearizations at the specified snapshot times. If you enter multiple

snapshot times, Control System Tuner computes an array of linearized models, and displays analysis plots that reflect the multiple linearizations in the array. In this case, Control System Tuner also takes into account all linearizations when tuning parameters. This helps to ensure that your tuned controller meets your design requirements at a variety of operating conditions.
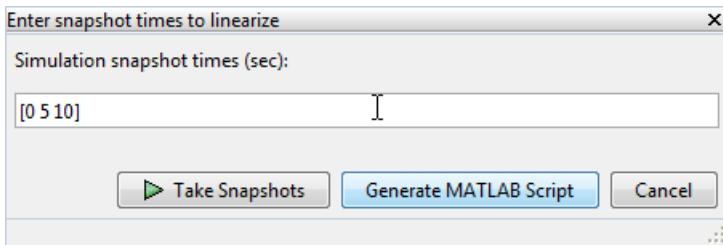
Compute Operating Points at Simulation Snapshot Times

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

In the **Control System** tab, in the **Operating Point** menu, select `Take simulation snapshot`.
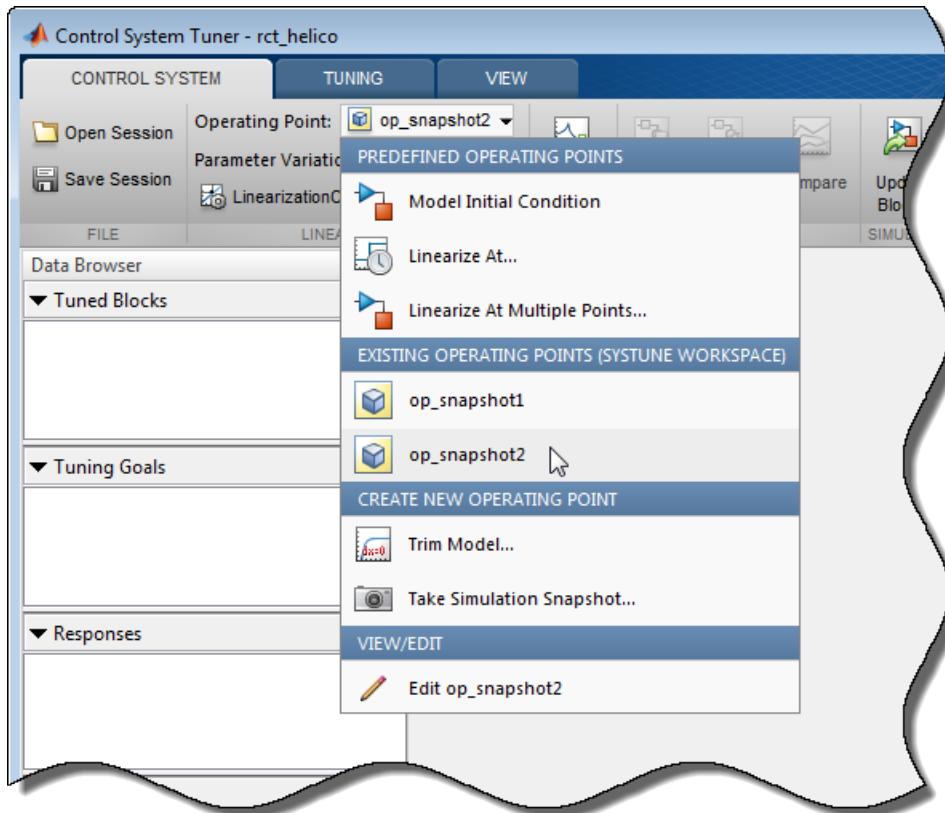
In the **Enter snapshot times to linearize** dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.

Click **Take Snapshots**. Control System Tuner simulates the model and computes the snapshot operating points.

Compute additional snapshot operating points if desired. Enter additional snapshot times and click **Take Snapshots**. Close the dialog box when you are done.

When you are ready to analyze responses or tune your model, select the operating point at which you want to linearize the model. In the **Control System** tab, in the **Operating Point** menu, stored operating points are available.
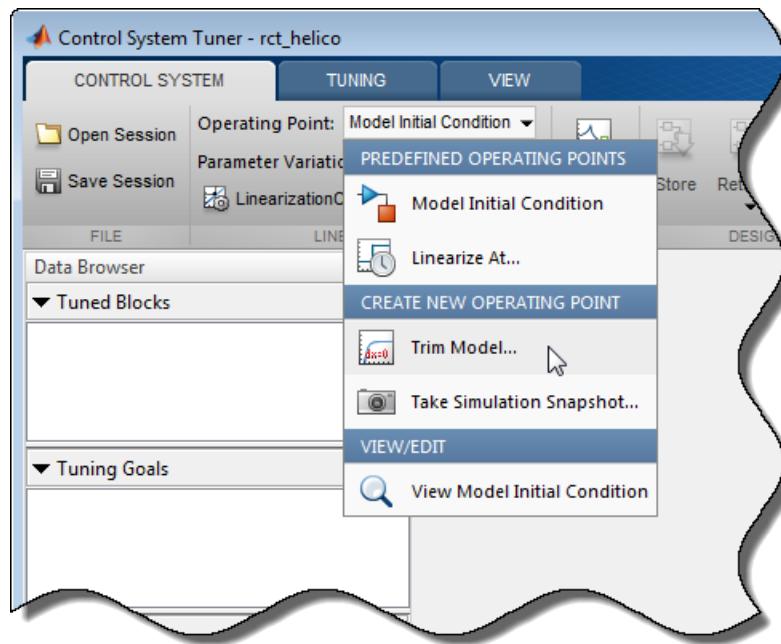
If you entered a vector of snapshot times, all the resulting operating points are stored together in an operating-point vector. You can use this vector to tune a control system at several operating point simultaneously.

Compute Steady-State Operating Points

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

In the **Control System** tab, in the **Operating Point** menu, select Trim model.
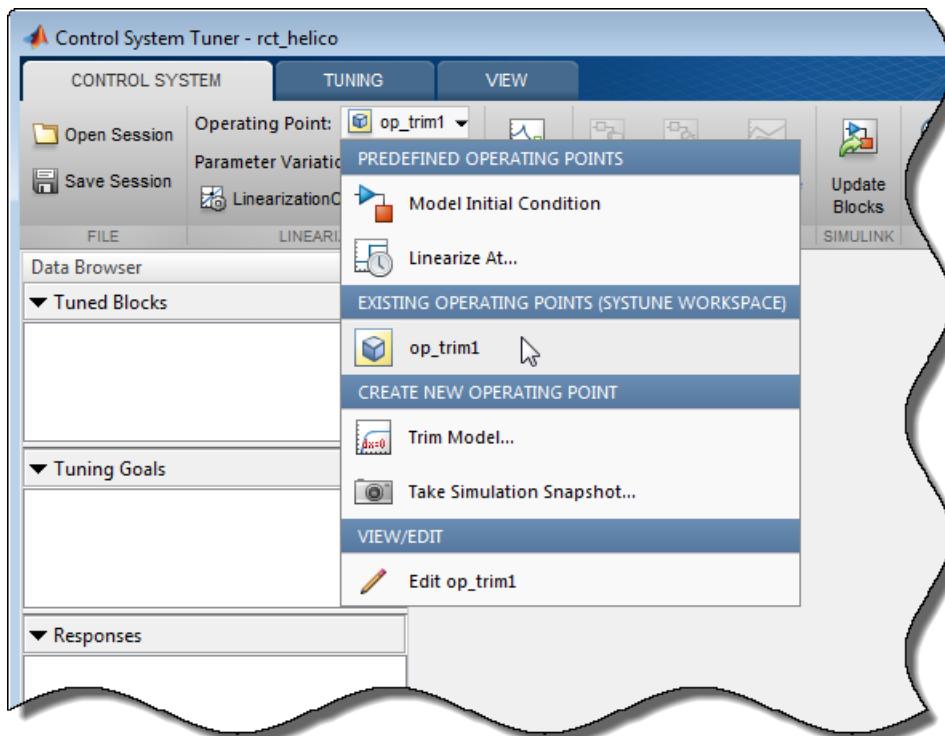
In the **Trim the model** dialog box, enter the specifications for the steady-state state values at which you want to find an operating point.

For examples showing how to use the **Trim the model** dialog box to specify the conditions for a steady-state operating point search, see "Compute Steady-State Operating Points from State Specifications" and "Compute Steady-State Operating Point to Meet Output Specification" in the Simulink Control Design documentation.

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

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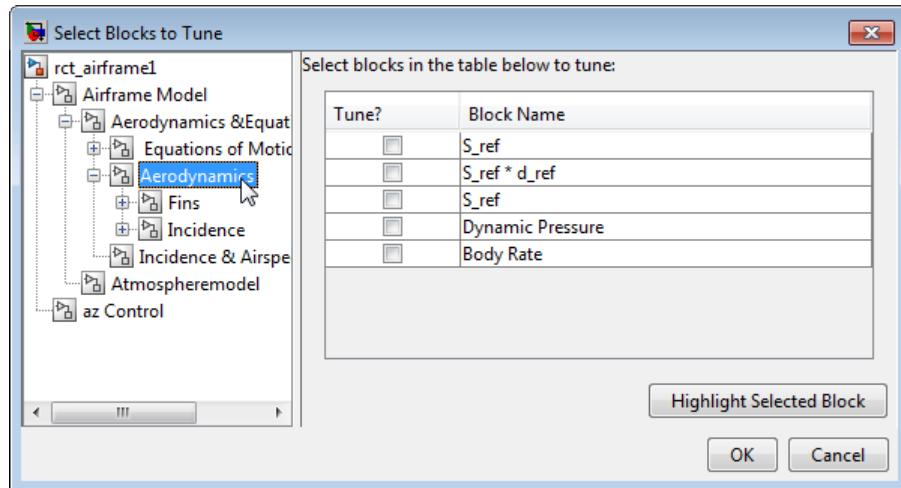
Related Examples

- "Specify Blocks to Tune in Control System Tuner" on page 13-24
- "Robust Tuning Approaches"

Specify Blocks to Tune in Control System Tuner
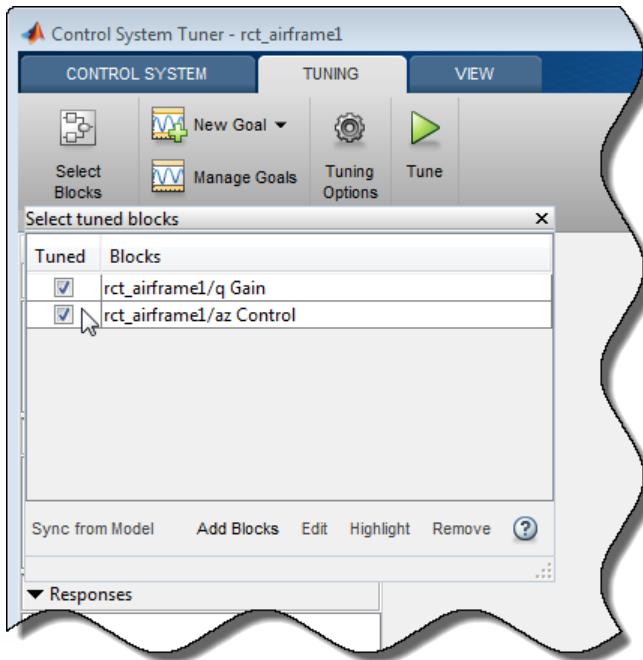
To select which blocks of your Simulink model to tune:

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

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

3   In the **Select Blocks to Tune** dialog box, use the nodes in the left panel to navigate through your model structure to the subsystem that contains blocks you want to tune. Check **Tune?** for the blocks you want to tune. The parameters of blocks you do not check remain constant when you tune the model.

Tip To find a block in your model, select the block in the **Block Name** list and click **Highlight Selected Block**.

4   Click **OK**. The **Select tuned blocks** dialog box now reflects the blocks you added.

To import the current value of a block from your model into the current design in Control System Tuner, select the block in the **Blocks** list and click **Sync from Model**. Doing so is useful when you have tuned a block in Control System Tuner, but wish to restore that block to its original value. To store the current design before restoring a block value, in the **Control System** tab, click ⬚ **Store**.

Related Examples

- "View and Change Block Parameterization in Control System Tuner" on page 13-26

More About

- "How Tuned Simulink Blocks Are Parameterized" on page 13-36

View and Change Block Parameterization in Control System Tuner

Control System Tuner parameterizes every block that you designate for tuning.

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

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

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

- When you tune a predefined control architecture or a MATLAB (generalized state-space) model, you define the parameterization of each tunable block when you create it at the MATLAB command line. For example, you can use `tunablePID` to create a tunable PID block.
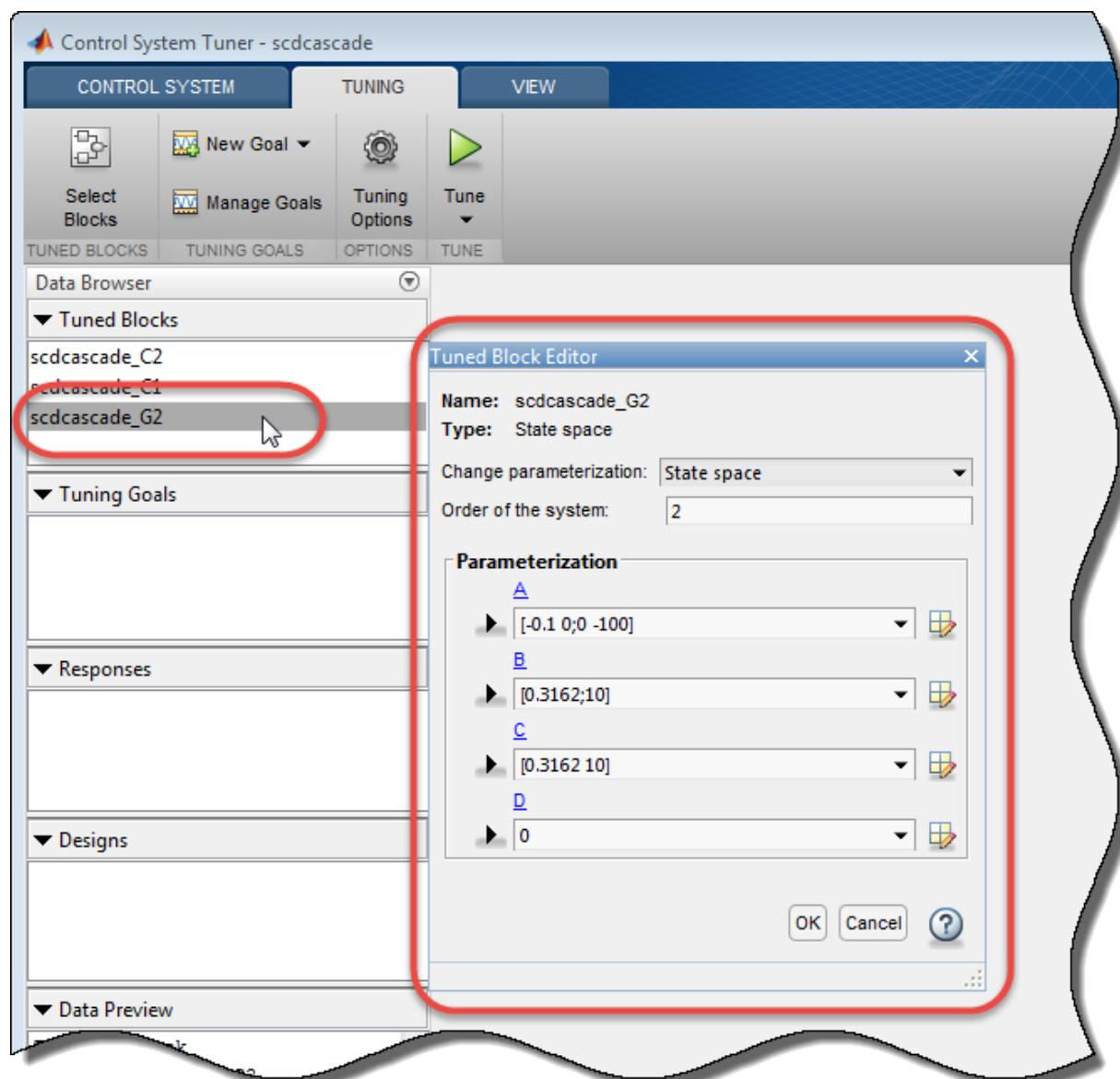
Control System Tuner lets you view and change the parameterization of any block to be tuned. Changing the parameterization can include changing the structure or current parameter values. You can also designate individual block parameters fixed (non-tunable) or limit their tuning range.

In this section...
"View Block Parameterization" on page 13-26
"Fix Parameter Values or Limit Tuning Range" on page 13-28
"Custom Parameterization" on page 13-30
"Block Rate Conversion" on page 13-31

View Block Parameterization

To access the parameterization of a block that you have designated as a tuned block, in the **Data Browser**, in the **Tuned Blocks** area, double-click the name of a block. The Tuned Block Editor dialog box opens, displaying the current block parameterization.

The fields of the **Tuned Block Editor** display the type of parameterization, such as `PID`, `State-Space`, or `Gain`. For more specific information about the fields, click ⓘ.
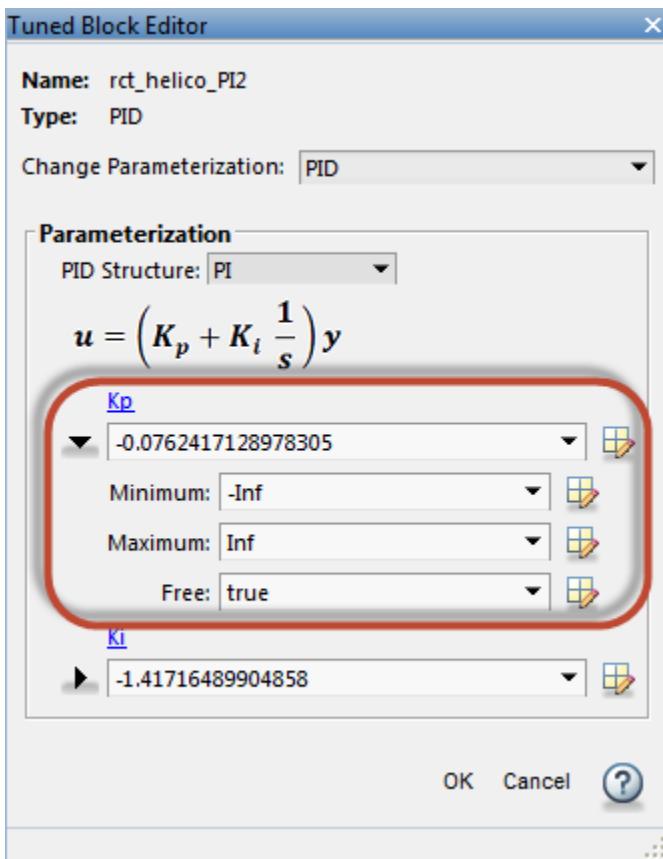
Note: To find a tuned block in the Simulink model, right-click the block name in the **Data Browser** and select **Highlight**.

Fix Parameter Values or Limit Tuning Range

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

To change a current parameter value, type a new value in its text box. Alternatively, click ▦ to use a variable editor to change the current value. If you attempt to enter an invalid value, the parameter returns to its previous value.

Click ▶ to access and edit additional properties of each parameter.

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

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

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

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

Custom Parameterization

When tuning a control system represented by a Simulink model or by a "Predefined Feedback Architecture" on page 13-11, you can specify a custom parameterization for any tuned block using a generalized state-space (`genss`) model. To do so, create and configure a `genss` model in the MATLAB workspace that has the desired parameterization, initial parameter values, and parameter properties. In the **Change parameterization** dialog box, select `Custom`. In the **Parameterization** area, the variable name of the `genss` model.
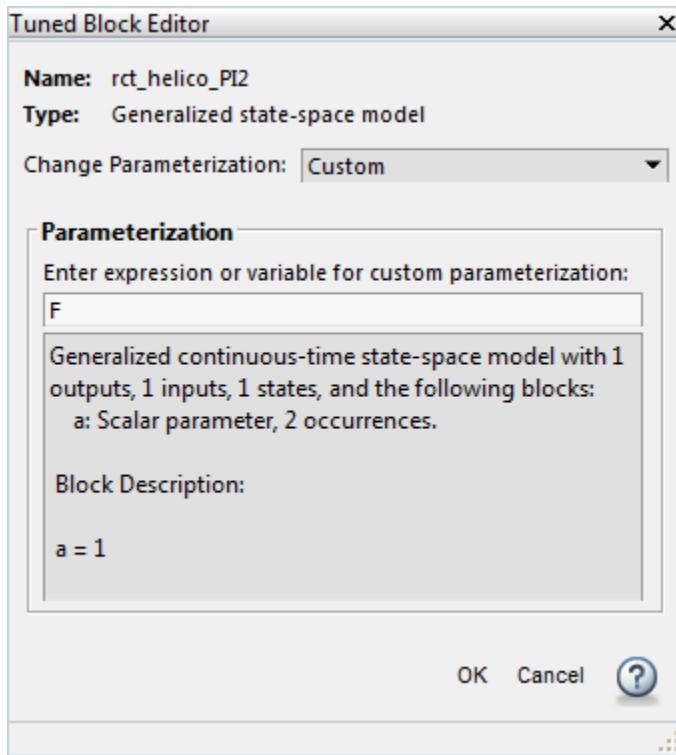
For example, suppose you want to specify a tunable low-pass filter, $F = a/(s + a)$, where a is the tunable parameter. First, at the MATLAB command line, create a tunable `genss` model that represents the low-pass filter structure.

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


F =

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

Type "ss(F)" to see the current value, "get(F)" to see all properties, and
"F.Blocks" to interact with the blocks.
```
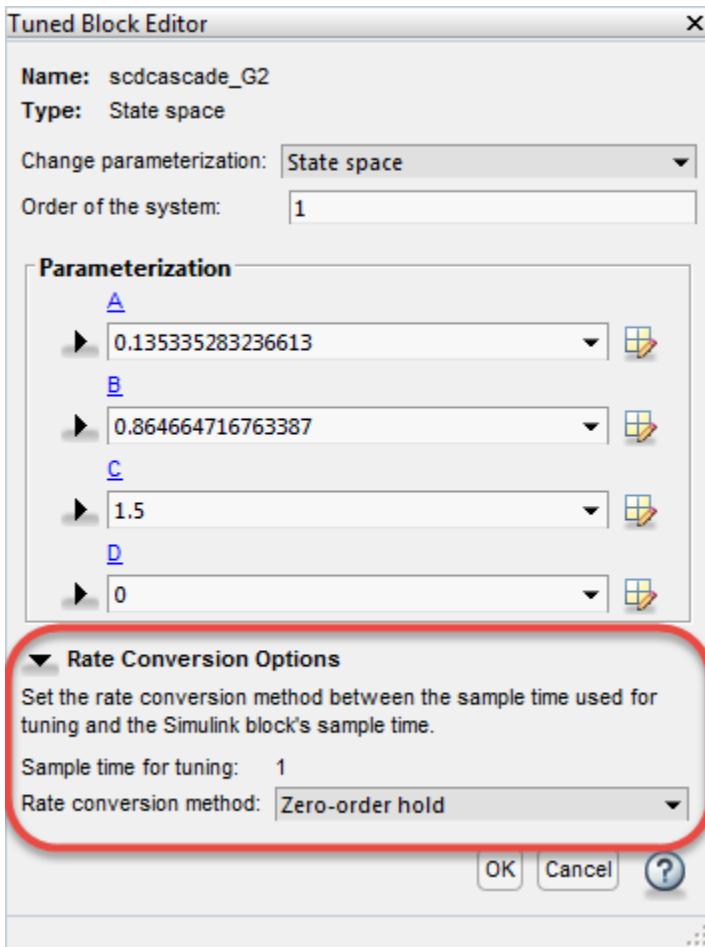
Then, in the Tuned Block Editor, enter `F` in the **Parameterization** area.

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

Block Rate Conversion

When Control System Tuner writes tuned parameters back to the Simulink model, each tuned block value is automatically converted from the sample time used for tuning, to the sample time of the Simulink block. When the two sample times differ, the Tuned Block Editor contains additional rate conversion options that specify how this resampling operation is performed for the corresponding block.

By default, Control System Tuner performs linearization and tuning in continuous time (sample time = 0). You can specify discrete-time linearization and tuning and change the sample time. To do so, on the **Control System** tab, click **Linearization Options**. **Sample time for tuning** reflects the sample time specified in the **Linearization Options** dialog box.

The remaining rate conversion options depend on the parameterized block.

Rate Conversion for Parameterized PID Blocks

For parameterization of continuous-time `PID Controller` and `PID Controller (2-DOF)` blocks, you can independently specify the rate-conversion methods as discretization formulas for the integrator and derivative filter. Each has the following options:

- `Trapezoidal` (default) — Integrator or derivative filter discretized as `(Ts/2)*(z+1)/(z-1)`, where `Ts` is the target sample time.
- `Forward Euler` — `Ts/(z-1)`.
- `Backward Euler` — `Ts*z/(z-1)`.

For more information about PID discretization formulas, see "Discrete-Time Proportional-Integral-Derivative (PID) Controllers" on page 2-25.

For discrete-time `PID Controller` and `PID Controller (2-DOF)` blocks, you set the integrator and derivative filter methods in the block dialog box. You cannot change them in the Tuned Block Editor.

Rate Conversion for Other Parameterized Blocks

For blocks other than `PID Controller` blocks, the following rate-conversion methods are available:

- `Zero-order hold` — Zero-order hold on the inputs. For most dynamic blocks this is the default rate-conversion method.
- `Tustin` — Bilinear (Tustin) approximation.
- `Tustin with prewarping` — Tustin approximation with better matching between the original and rate-converted dynamics at the prewarp frequency. Enter the frequency in the **Prewarping frequency** field.
- `First-order hold` — Linear interpolation of inputs.
- `Matched` (SISO only) — Zero-pole matching equivalents.

For more detailed information about these rate-conversion methods, see "Continuous-Discrete Conversion Methods" on page 5-25.

Blocks with Fixed Rate Conversion Methods

For the following blocks, you cannot set the rate-conversion method in the Tuned Block Editor.

- Discrete-time `PID Controller` and `PID Controller (2-DOF)` block. Set the integrator and derivative filter methods in the block dialog box.
- `Gain` block, because it is static.
- `Transfer Fcn Real Zero` block. This block can only be tuned at the sample time specified in the block.
- Block that has been discretized using the Model Discretizer. Sample time for this block is specified in the Model Discretizer itself.

Related Examples

- "Specify Blocks to Tune in Control System Tuner" on page 13-24

More About

- "How Tuned Simulink Blocks Are Parameterized" on page 13-36

Setup for Tuning Control System Modeled in MATLAB

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

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

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

```
controlSystemTuner(T0)
```

Related Examples

•   "Specify Goals for Interactive Tuning" on page 13-39

How Tuned Simulink Blocks Are Parameterized

Blocks With Predefined Parameterization

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

Blocks that have a predefined parameterization include the following:

Simulink Library	Blocks with Predefined Parameterization
Math Operations	`Gain`
Continuous	• `State-Space` • `Transfer Fcn` • `Zero-Pole` • `PID Controller` • `PID Controller (2 DOF)`
Discrete	• `Discrete State-Space` • `Discrete Transfer Fcn` • `Discrete Zero-Pole` • `Discrete Filter` • Discrete `PID Controller` • Discrete `PID Controller (2 DOF)`
Lookup Tables	• `1-D Lookup Table` • `2-D Lookup Table` • `n-D Lookup Table`
Control System Toolbox	`LTI System`
Discretizing (Model Discretizer Blocks)	• Discretized State-Space • Discretized Transfer Fcn

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

Scalar Expansion

The following tunable blocks support scalar expansion:

- `Discrete Filter`
- `Gain`
- `1-D Lookup Table`, `2-D Lookup Table`, `n-D Lookup Table`
- `PID Controller`, `PID Controller (2DOF)`

Scalar expansion means that the block parameters can be scalar values even when the input and output signals are vectors. For example, you can use a `Gain` block to implement $y = k*u$ with scalar `k` and vector `u` and `y`. To do so, you set the **Multiplication** mode of the block to `Element-wise(K.*u)`, and set the gain value to the scalar `k`.

When a tunable block uses scalar expansion, its default parameterization uses tunable scalars. For example, in the $y = k*u$ `Gain` block, the software parameterizes the scalar `k` as a tunable real scalar (`realp` of size `[1 1]`). If instead you want to tune different gain values for each channel, replace the scalar gain `k` by a `N-by-1` gain vector in the block dialog, where `N` is the number of channels, the length of the vectors `u` and `y`. The software then parameterizes the gain as a `realp` of size `[N 1]`.

Blocks Without Predefined Parameterization

You can specify blocks for tuning that do not have a predefined parameterization. When you do so, the software assigns a state-space parameterization to such blocks based upon the block linearization. For blocks that do not have a predefined parameterization, the software cannot write tuned values back to the block, because there is no clear mapping between the tuned parameters and the block. To validate a tuned control system that contains such blocks, you can specify a block linearization in your model using the value

of the tuned parameterization. (See "Specify Linear System for Block Linearization Using MATLAB Expression" for more information about specifying block linearization.)

View and Change Block Parameterization

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

- In Control System Tuner, see "View and Change Block Parameterization in Control System Tuner" on page 13-26.
- At the command line, use `getBlockParam` to view the current block parameterization. Use `setBlockParam` to change the block parameterization.

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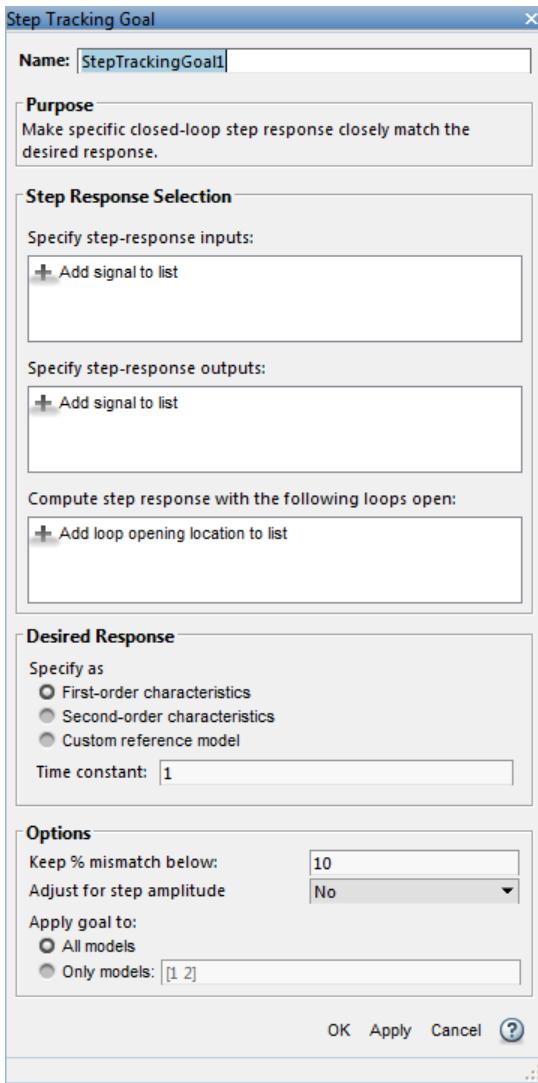
This example shows how to specify your tuning goals for automated tuning in Control System Tuner.

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

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

Choose Tuning Goal Type

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

Step Tracking Goal

Name: StepTrackingGoal1

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

Step Response Selection

Specify step-response inputs:

＋ Add signal to list

Specify step-response outputs:

＋ Add signal to list

Compute step response with the following loops open:

＋ Add loop opening location to list

Desired Response

Specify as
○ First-order characteristics
○ Second-order characteristics
○ Custom reference model
Time constant: 1

Options

Keep % mismatch below: 10
Adjust for step amplitude   No
Apply goal to:
○ All models
○ Only models: [1 2]

OK   Apply   Cancel   ⑦

Choose Signal Locations for Evaluating Tuning Goal

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

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

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

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.
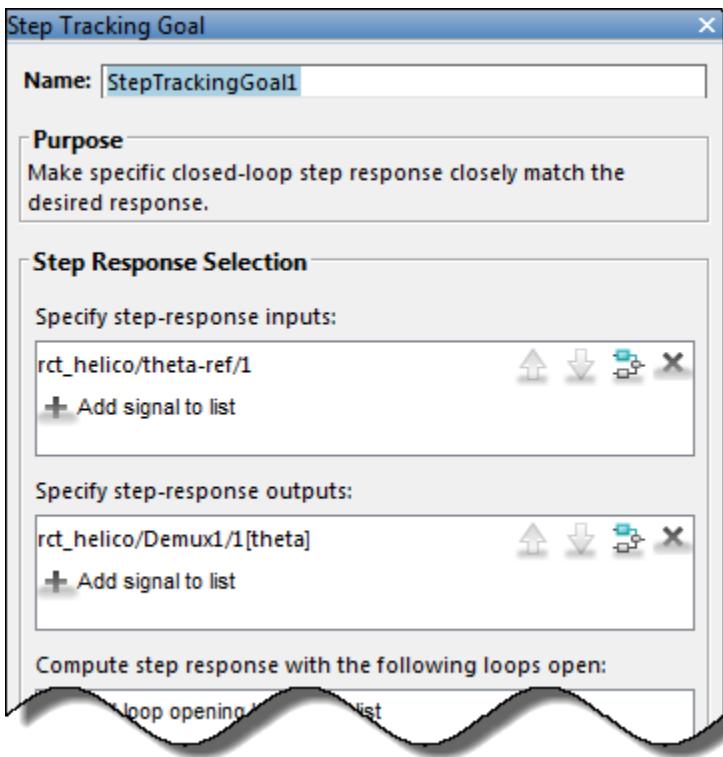
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO tuning goal, and click multiple signals to create a MIMO tuning goal.

Click **Add signal(s)**. The **Select signals** dialog box closes, returning you to the new tuning-goal specification dialog box.

The signals you selected now appear in the list of step-response inputs in the tuning goal dialog box.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration constrains the response to a step input applied at `theta-ref` and measured at `theta` in the Simulink model `rct_helico`.

Tip To highlight any selected signal in the Simulink model, click [icon]. To remove a signal from the input or output list, click [icon]. When you have selected multiple signals, you can reorder them using [icon] and [icon].

Specify Loop Openings

Most tuning goals can be enforced with loops open at one or more locations in the control system. Click [icon]**Add loop opening location to list** to specify such locations for the tuning goal.
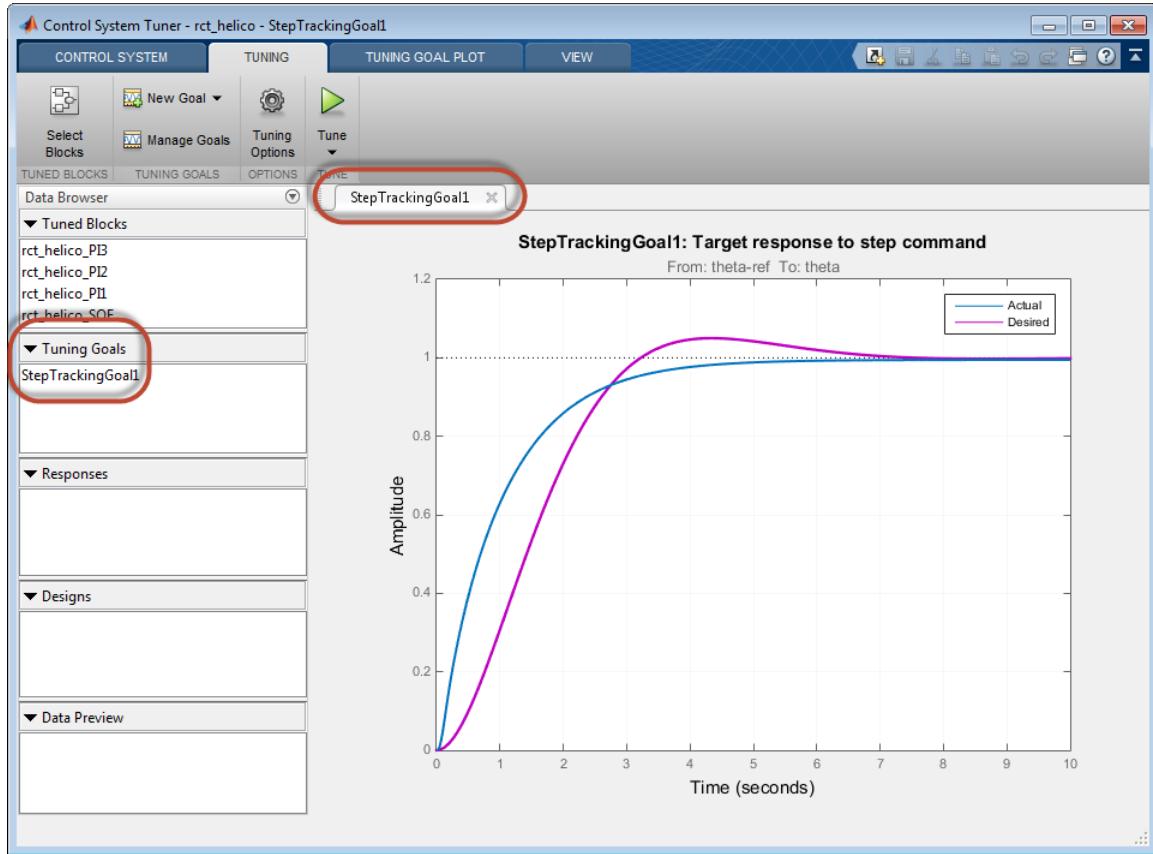
Define Other Specifications of the Tuning Goal

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

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

Store the Tuning Goal for Tuning

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

Tip To edit the specifications of the tuning goal, double-click the tuning goal in the Data Browser.

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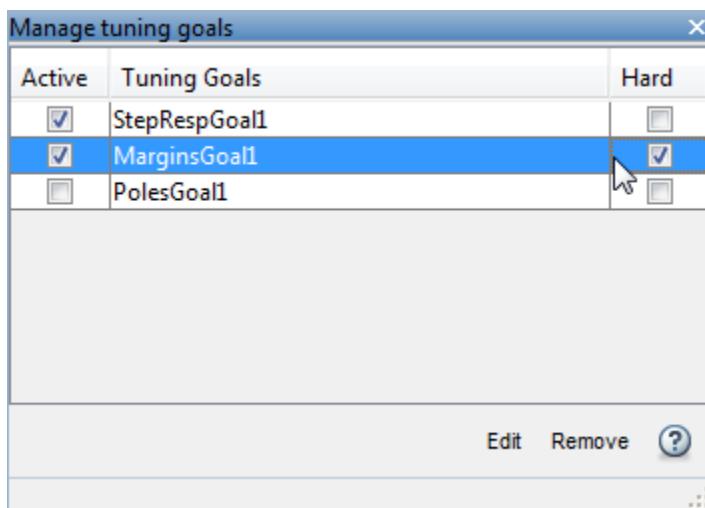
When you have saved your tuning goal, click ![icon] **New Goal** to create additional tuning goals.

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

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

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

For example, if you tune with the following configuration, Control System Tuner optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.

Deactivating tuning goals or designating some goals as soft requirements can be useful when investigating the tradeoffs between different tuning requirements. For example, if you do not obtain satisfactory performance with all your tuning goals active and hard, you might try another design in which less crucial goals are designated as soft or deactivated entirely.

Related Examples

- "Manage Tuning Goals" on page 13-157
- "Quick Loop Tuning of Feedback Loops in Control System Tuner" on page 13-47
- "Create Response Plots in Control System Tuner" on page 13-168

Quick Loop Tuning of Feedback Loops in Control System Tuner

This example shows how to tune a Simulink model of a control system to meet a specified bandwidth and stability margins in Control System Tuner, without explicitly creating tuning goals that capture these requirements. You can use a similar approach for quick loop tuning of control systems modeled in MATLAB.
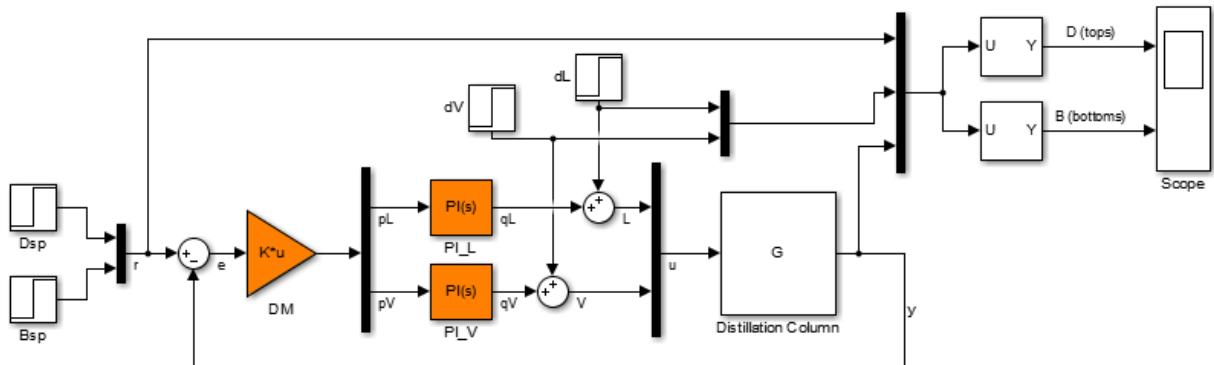
This example demonstrates how the **Quick Loop Tuning** option of Control System Tuner generates tuning goals from your crossover frequency and gain and phase margin specifications. This option lets you quickly set up SISO or MIMO feedback loops for tuning using a loop-shaping approach. The example also shows how to add further tuning requirements to the control system after using the **Quick Loop Tuning** option.

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

Set up the Model for Tuning

Open the Simulink model.

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


This model represents a distillation column, captured in the two-input, two-output plant `G`. The tunable elements are the decoupling gain matrix `DM`, and the two PI controllers, `PI_L` and `PI_V`. (For more information about this model, see "Decoupling Controller for a Distillation Column" on page 14-17.)

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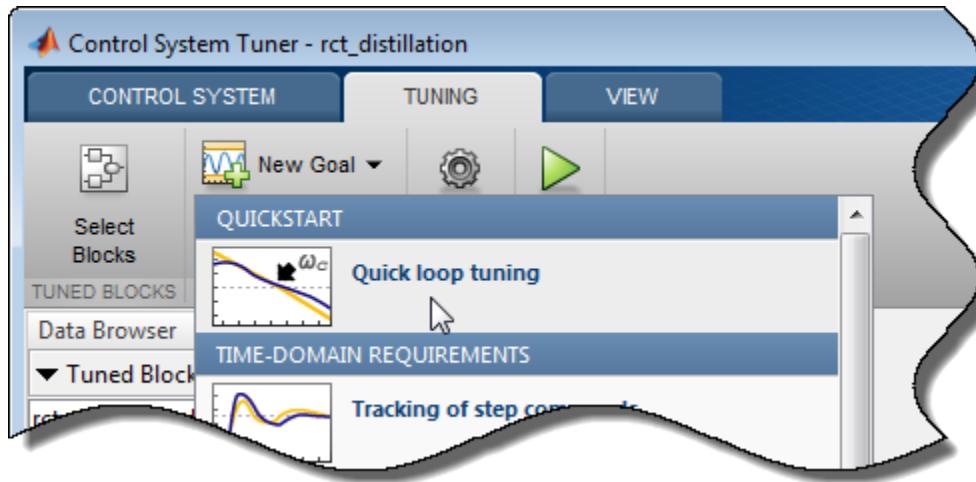
In the Simulink model editor, open Control System Tuner by selecting **Analysis > Control Design > Control System Tuner**.

Designate the blocks you want to tune. In the **Tuning** tab of Control System Tuner, click ⊞ **Select Blocks**. In the **Select tuned blocks** dialog box, click **Add blocks**. Then, select DM, PI_L, and PI_V for tuning. (For more information about selecting tuned blocks, see "Specify Blocks to Tune in Control System Tuner" on page 13-24.)

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

Specify the Goals for Quick Loop Tuning

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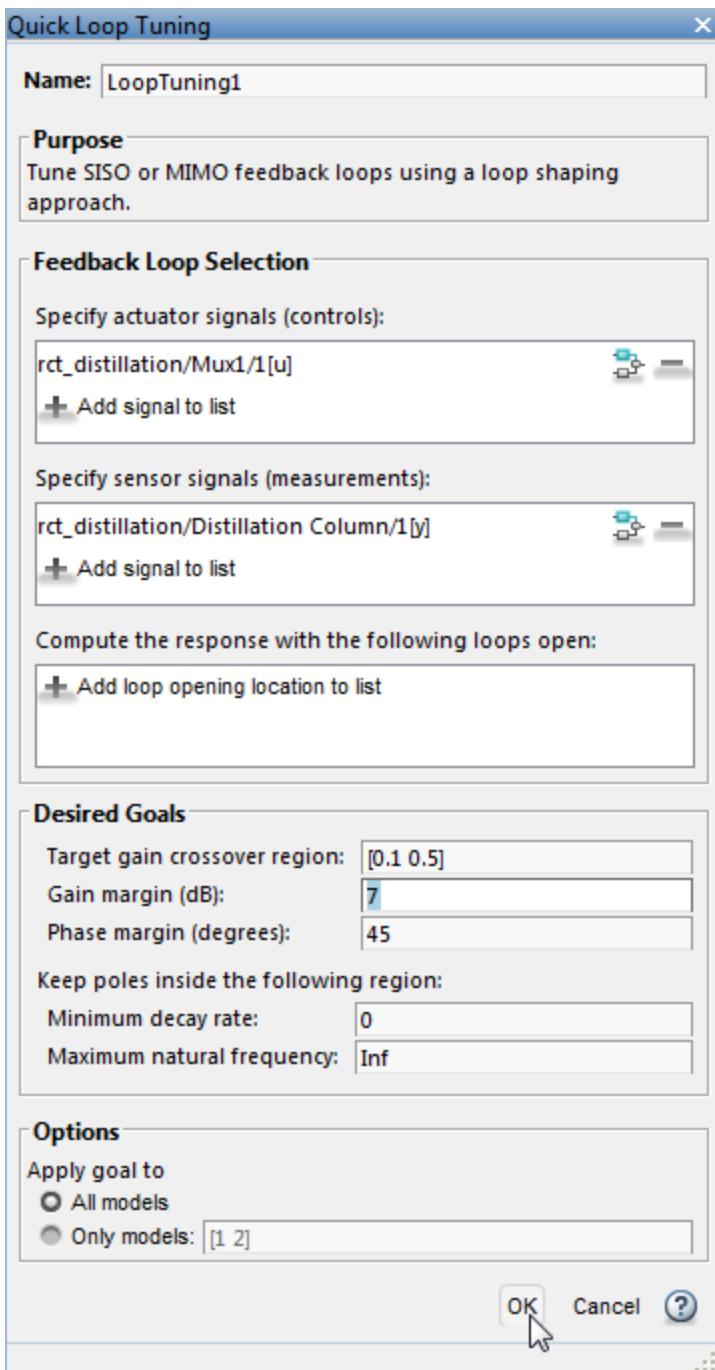
For Quick Loop Tuning, you need to identify the actuator signals and sensor signals that separate the plant portion of the control system from the controller, which for the purpose of Quick Loop Tuning is the rest of the control system. The actuator signals are the controller outputs that drive the plant,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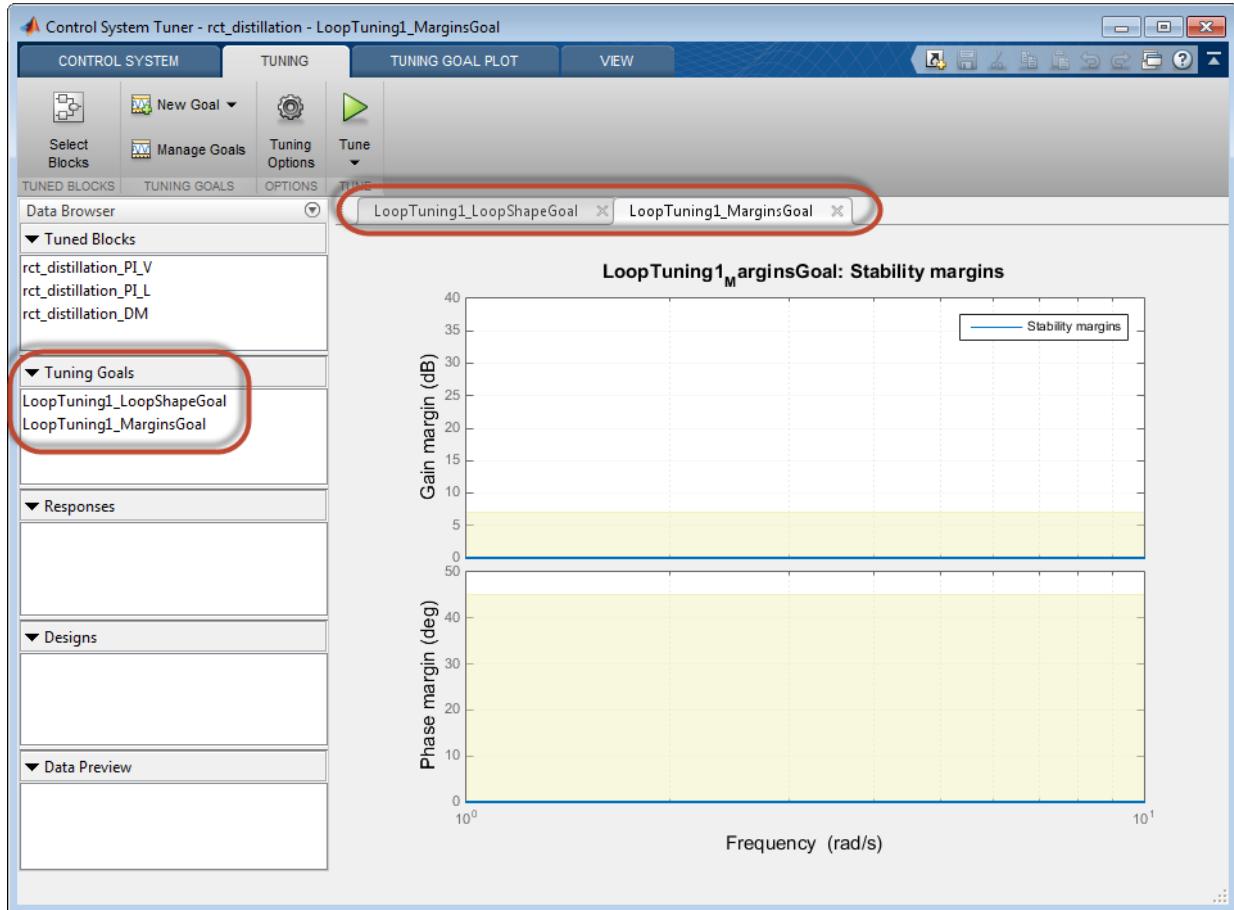
the actuator signals are represented by the vector signal u, and the sensor signals by the vector signal y.

In the **Quick Loop Tuning** dialog box, under **Specify actuator signals (controls)**, add the actuator signal, u. Similarly, under **Specify sensor signals (measurements)**, add the sensor signal, y (For more information about specifying signals for tuning, see "Specify Goals for Interactive Tuning" on page 13-39.)

Under **Desired Goals**, in the **Target gain crossover region** field, enter the target bandwidth range, [0.1 0.5]. Enter the desired gain margin and phase margin in the corresponding fields.
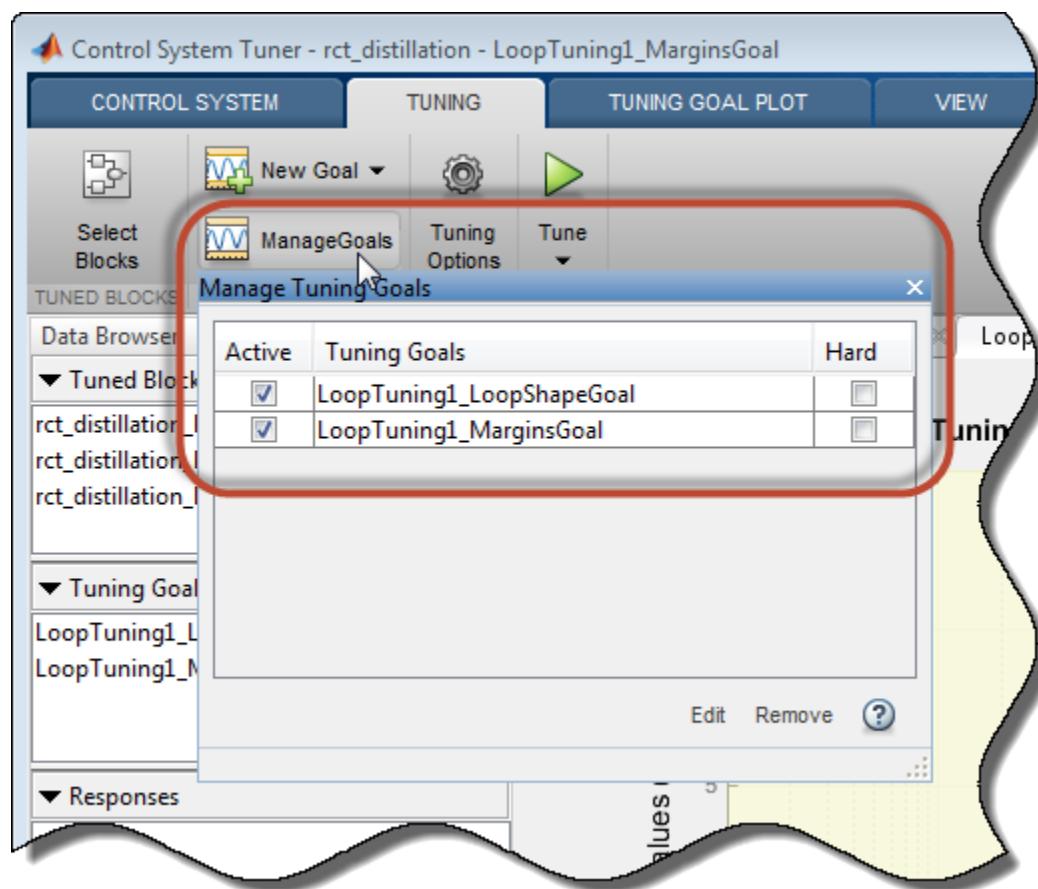
Click **OK**. Control System Tuner automatically generates tuning goals that capture the desired goals you entered in the dialog box.
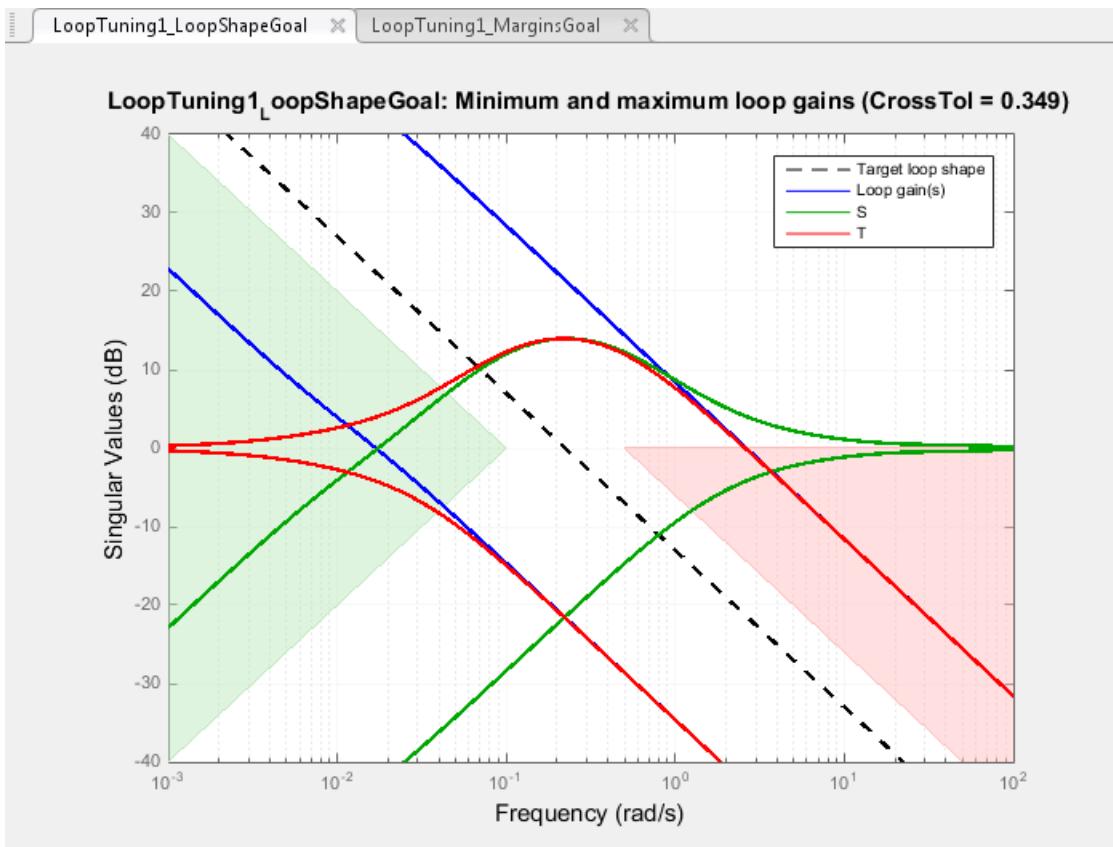
Examine the Automatically-Created Tuning Goals

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

Click ![icon] **Manage Goals** to examine the automatically-created goals. By default, the goals are active and designated as soft tuning goals.

You can double-click the tuning goals to examine their parameters, which are automatically computed and populated. You can also examine the graphical representations of the tuning goals. In the **Tuning** tab, examine the **LoopTuning1_LoopShapeGoal** plot.

The target crossover range is expressed as a Loop Shape goal with an integrator open-loop gain profile. The shaded areas of the graph show that the permitted crossover range is [0.1 0.5] rad/s, as you specified in the **Quick Loop Tuning** dialog box.

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

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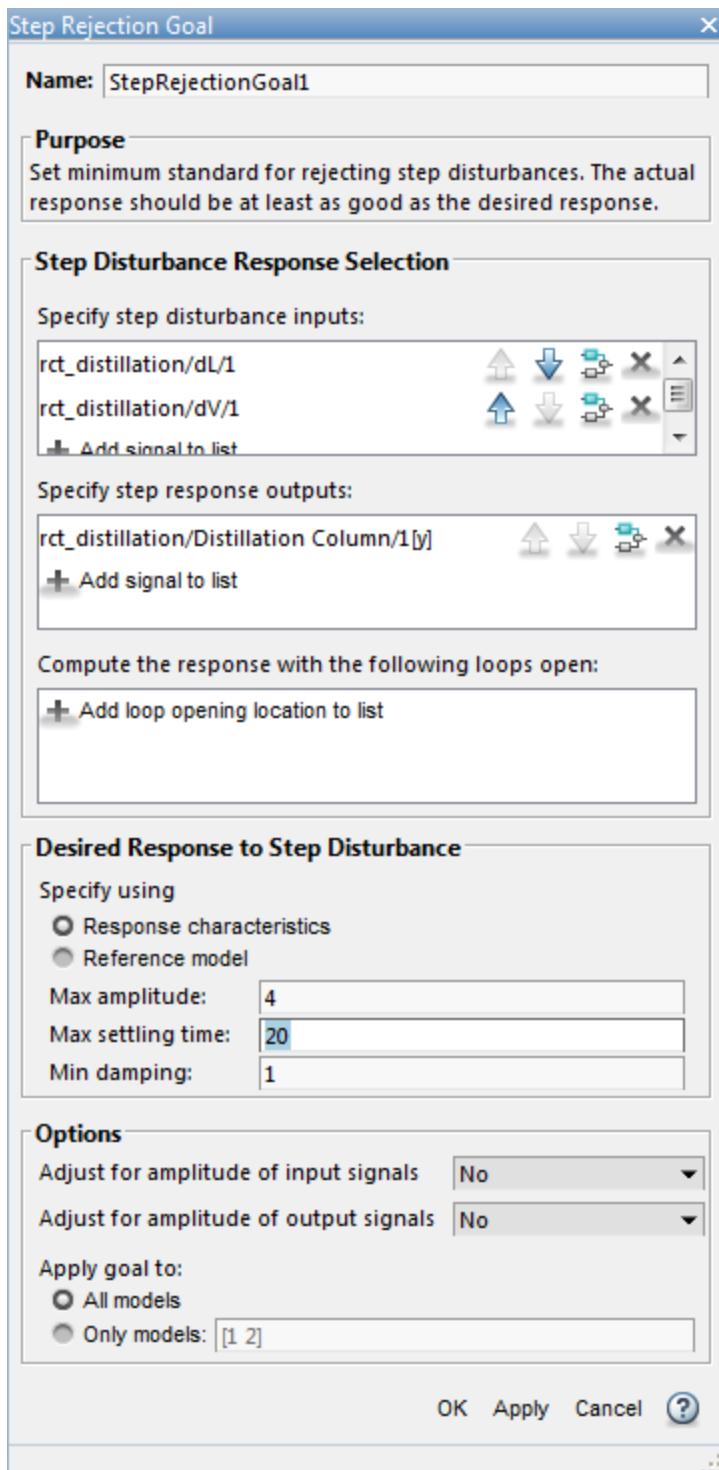
Click ![tune icon] **Tune** to tune the model to meet the automatically-created tuning goals. In the tuning goal plots, you can see that the requirements are satisfied.

To create additional plots for examining other system responses, see "Create Response Plots in Control System Tuner" on page 13-168.

Change Design Requirements

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

For example, add a requirement that limits the response to a disturbance applied at the plant inputs. Limit the response to a step command at dL and dV at the outputs, y, to be well damped, to settle in less than 20 seconds, and not exceed 4 in amplitude. Select **New Goal > Rejection of step disturbances** and enter appropriate values in the Step Rejection Goal dialog box. (For more information about creating tuning goals, see "Specify Goals for Interactive Tuning" on page 13-39.)

Step Rejection Goal ✕

Name: StepRejectionGoal1

Purpose
Set minimum standard for rejecting step disturbances. The actual
response should be at least as good as the desired response.

Step Disturbance Response Selection

Specify step disturbance inputs:

rct_distillation/dL/1	⬆ ⬇ ⮆ ✕
rct_distillation/dV/1	⬆ ⬇ ⮆ ✕
➕ Add signal to list	

Specify step response outputs:

rct_distillation/Distillation Column/1[y]	⬆ ⬇ ⮆ ✕
➕ Add signal to list	

Compute the response with the following loops open:

➕ Add loop opening location to list

Desired Response to Step Disturbance

Specify using
◉ Response characteristics
○ Reference model

Max amplitude:	4
Max settling time:	20
Min damping:	1

Options

Adjust for amplitude of input signals	No ▼
Adjust for amplitude of output signals	No ▼

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

OK   Apply   Cancel   ⑦

13-55

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

See Also

```
looptune (for slTuner)
```

Related Examples

- "Specify Operating Points for Tuning in Control System Tuner" on page 13-17
- "Manage Tuning Goals" on page 13-157
- "Setup for Tuning Control System Modeled in MATLAB" on page 13-35

Quick Loop Tuning

Purpose

Tune SISO or MIMO feedback loops using a loop-shaping approach in Control System Tuner.

Description

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

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

Creation

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

Command-Line Equivalent

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

Feedback Loop Selection

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

- **Specify actuator signals (controls)**

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

- **Specify sensor signals (measurements)**

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

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

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

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

For an example showing in more detail how to specify signal locations, see "Specify Goals for Interactive Tuning" on page 13-39.

Desired Goals

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

- **Target gain crossover region**

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

- **Gain margin (db)**

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

- **Phase margin (degrees)**

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

- **Keep poles inside the following region**

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

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

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

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

Options

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

Related Examples

- "Quick Loop Tuning of Feedback Loops in Control System Tuner" on page 13-47
- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Step Tracking Goal
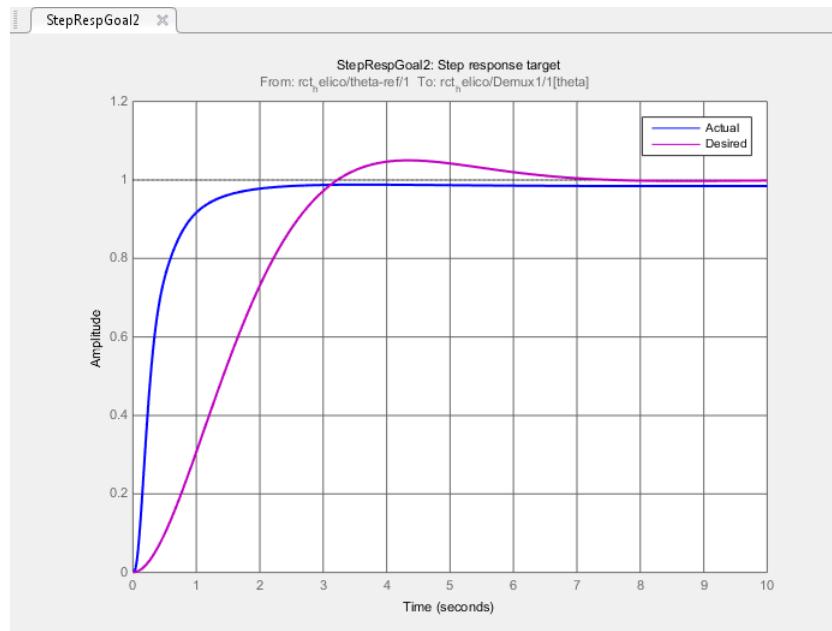
Purpose

Make the step response from specified inputs to specified outputs closely match a target response, when using Control System Tuner.

Description

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

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

Creation

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

Command-Line Equivalent

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

Step Response Selection

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

- **Specify step-response inputs**

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

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

- **Specify step-response outputs**

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

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

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

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

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

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 13-39.

Desired Response

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

- **First-order characteristics**

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

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

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

- **Second-order characteristics**

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

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

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

  The second-order reference system has the form:

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

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

- **Custom reference model**

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

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

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

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

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

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

Options

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

- **Keep % mismatch below**

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

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

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

- **Adjust for step amplitude**

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

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

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

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

· **Apply goal to**

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

For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

$T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $H_{ref}(s)$ is the reference model. e_{rel} is the relative error (see "Options" on page 13-64). $\left\| \cdot \right\|_2$ denotes the H_2 norm (see `norm`).

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Step Rejection Goal

Purpose
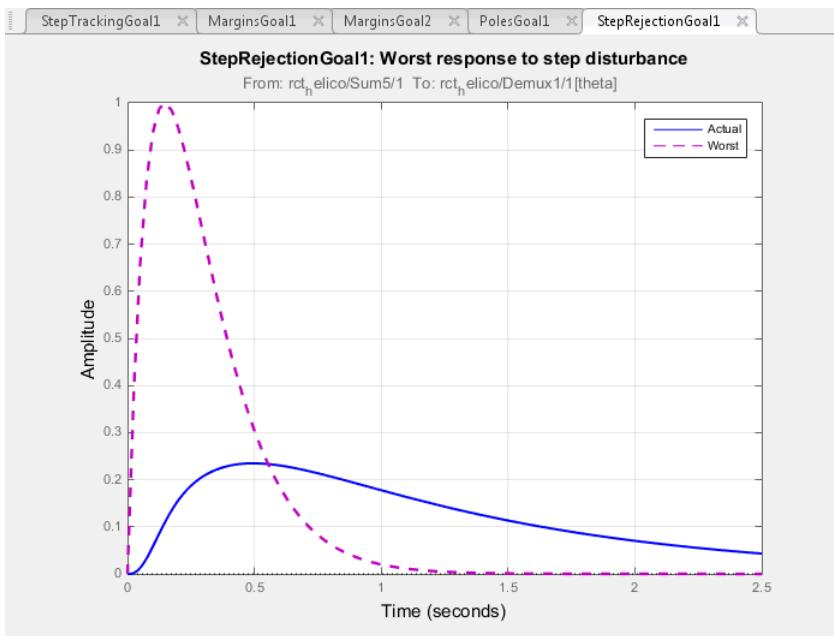
Set a minimum standard for rejecting step disturbances, when using Control System Tuner.

Description

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

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

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

Creation

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

Command-Line Equivalent

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

Step Disturbance Response Selection

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

- **Specify step disturbance inputs**

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

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

- **Specify step response outputs**

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

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

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

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 13-39.

Desired Response to Step Disturbance

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

- **Response Characteristics**

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

- **Reference Model**

  Specify the desired response in terms of a reference model.

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

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

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

Options

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

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

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

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

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

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

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

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

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Transient Goal
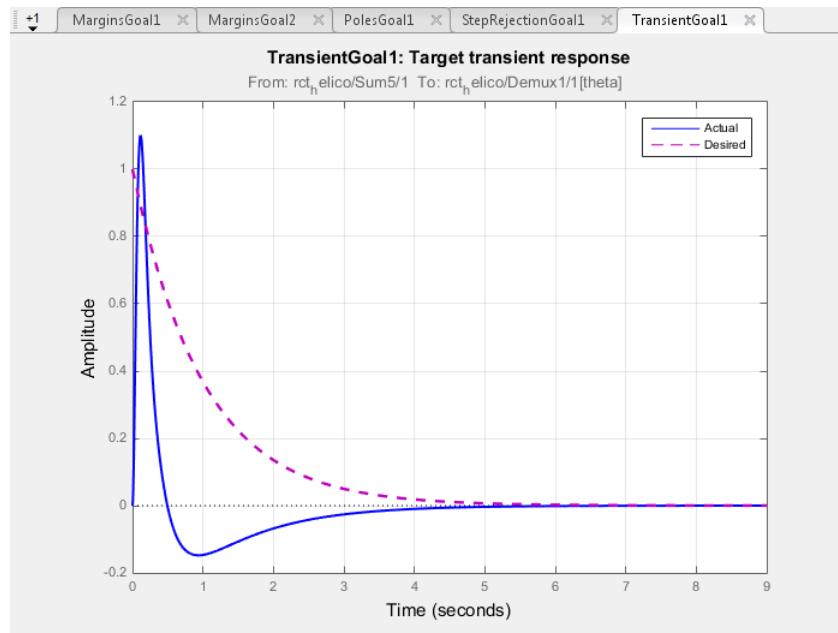
Purpose

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

Description

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

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

Creation

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

Command-Line Equivalent

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

Response Selection

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

- **Specify response inputs**

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

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

- **Specify response outputs**

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

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

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

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

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

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 13-39.

Initial Signal Selection

Select the input signal shape for the transient response you want to constrain in Control System Tuner.

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

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

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

Desired Transient Response

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

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

Options

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

- **Keep % mismatch below**

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

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

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

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

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

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

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

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

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 13-77), is infinite for continuous-time systems with nonzero feedthrough.

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

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

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 13-26.

Algorithms

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

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

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

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

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

LQR/LQG Goal

Purpose

Minimize or limit Linear-Quadratic-Gaussian (LQG) cost in response to white-noise inputs, when using Control System Tuner.

Description

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

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

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

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

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

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

Creation

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

Command-Line Equivalent

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

Signal Selection

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

- **Specify noise inputs (w)**

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

- **Specify performance outputs (z)**

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

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

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

LQG Objective

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

- **Performance weight Qz**

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

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

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

- **Noise Covariance Qw**

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

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

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

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

Options

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

- **Apply goal to**

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

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

For more information about tuning for multiple models, see "Robust Tuning Approaches".

Tips

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

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

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

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 13-26.

Algorithms

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

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

When you use the LQG requirement as a hard goal, the software tries to drive the cost function $J < 1$. When you use it as a soft goal, the cost function J is minimized

subject to any hard goals and its value is contributed to the overall objective function. Therefore, select `Qz` values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Gain Goal
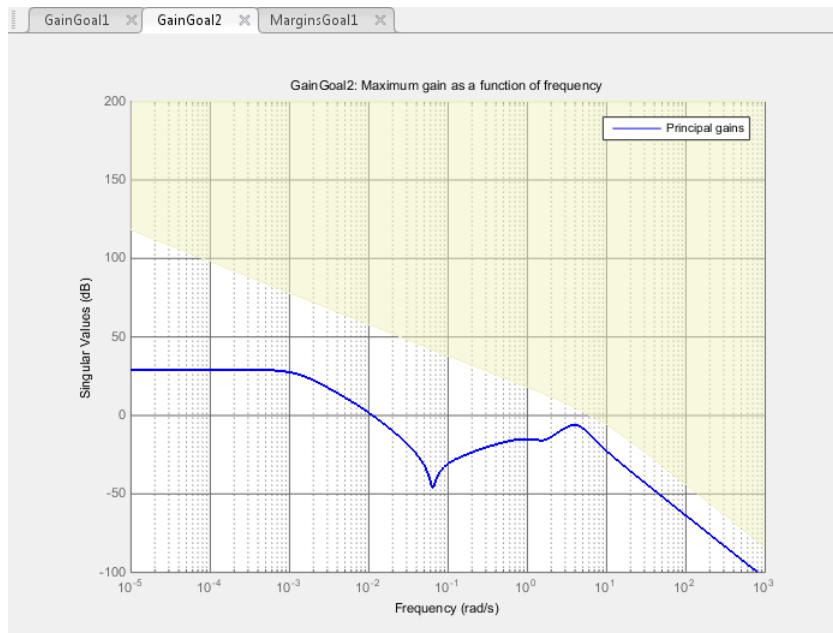
Purpose

Limit gain of a specified input/output transfer function, when using Control System Tuner.

Description

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

Use Gain Goal, for example, to enforce a custom roll-off rate in a particular frequency band. To do so, specify a maximum gain profile in that band. You can also use Gain Goal to enforce disturbance rejection across a particular input/output pair by constraining the gain to be less than 1.

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

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

Creation

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

Command-Line Equivalent

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

I/O Transfer Selection

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

· **Specify input signals**

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

· **Specify output signals**

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

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

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

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 13-39.

Options

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

- **Limit gain to**

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

- **Stabilize I/O transfer**

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

- **Enforce goal in frequency range**

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

- **Adjust for signal amplitude**

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

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

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

$T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. MaxGain is the maximum gain profile you provide for the gain goal. D_o and D_i are the scaling matrices described in "Options" on page 13-85. $\left\| \cdot \right\|_\infty$ denotes the H_∞ norm (see norm).

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Variance Goal

Purpose

Limit white-noise impact on specified output signals, when using Control System Tuner.

Description

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

For stochastic inputs with a nonuniform spectrum (colored noise), use "Weighted Variance Goal" on page 13-115 instead.

Creation

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

Command-Line Equivalent

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

I/O Transfer Selection

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

- **Specify stochastic inputs**

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

- **Specify stochastic outputs**

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

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

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

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

Options

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

- **Attenuate input variance by a factor**

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

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

- **Adjust for signal amplitude**

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

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

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

· **Apply goal to**

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

For more information about tuning for multiple models, see "Robust Tuning Approaches".

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 13-91), is infinite for continuous-time systems with nonzero feedthrough.

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

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

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 13-26.

Algorithms

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

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

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

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

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

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

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

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Reference Tracking Goal

Purpose

Make specified outputs track reference inputs with prescribed performance and fidelity, when using Control System Tuner. Limit cross-coupling in MIMO systems.
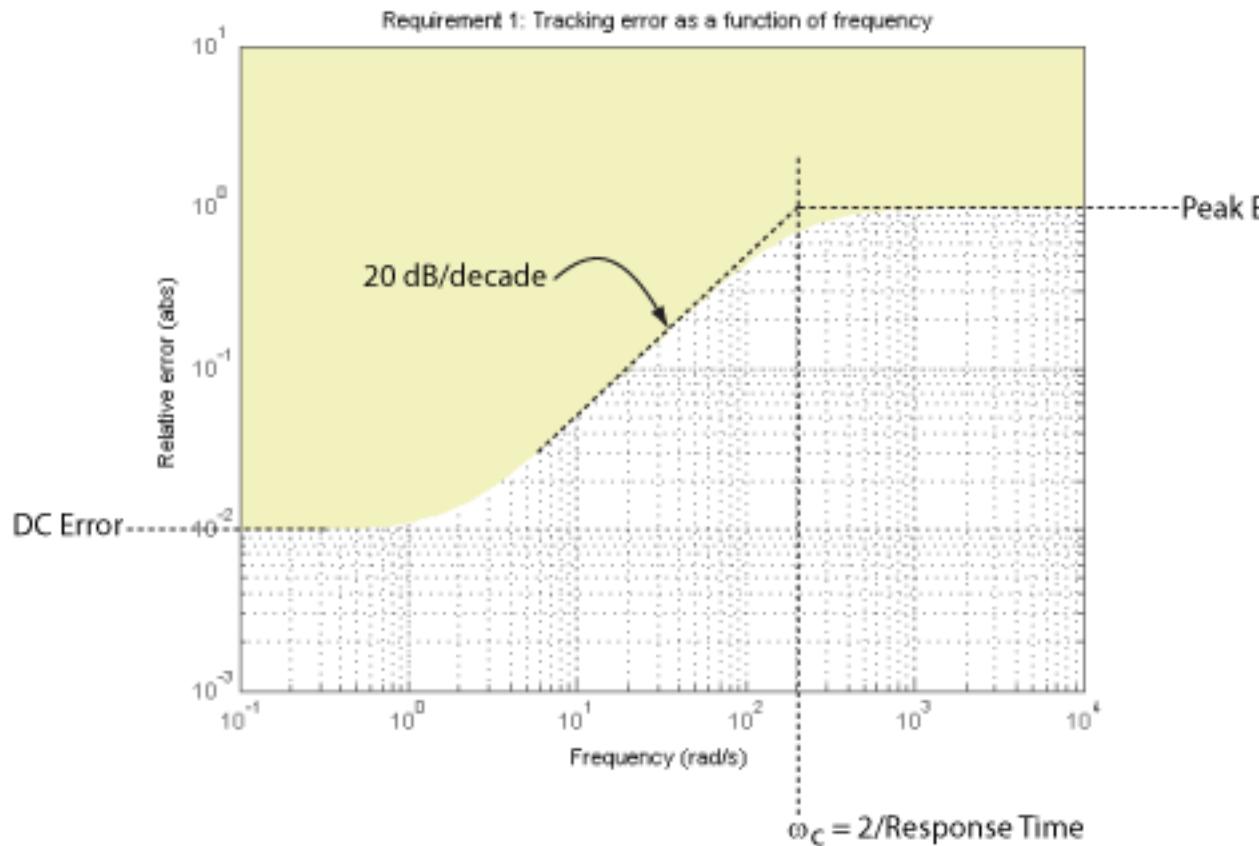
Description

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

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

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

Here, ω_c is 2/(response time). The following plot illustrates these relationships for an example set of values.

Requirement 1: Tracking error as a function of frequency

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

Creation

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

Command-Line Equivalent

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

Response Selection

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

- **Specify reference inputs**

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

- **Specify reference-tracking outputs**

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

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

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

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 13-39.

Tracking Performance

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

Response time, DC error, and peak error

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

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

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

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

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

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

Maximum error as a function of frequency

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

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

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

Options

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

- **Enforce goal in frequency range**

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

- **Adjust for step amplitude**

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

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

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

- **Apply goal to**

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

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

For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

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

Related Examples

· "Specify Goals for Interactive Tuning" on page 13-39
· "Manage Tuning Goals" on page 13-157

Overshoot Goal
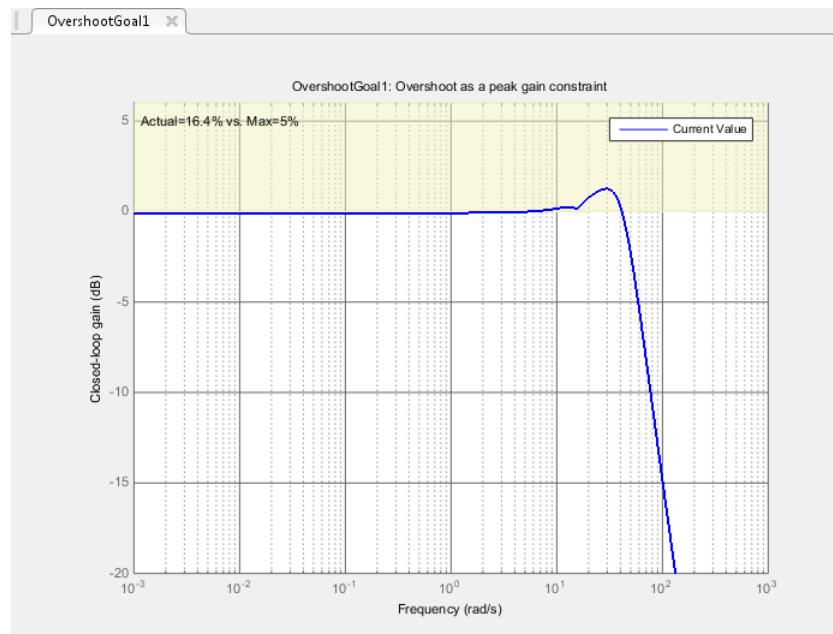
Purpose

Limit overshoot in the step response from specified inputs to specified outputs, when using Control System Tuner.

Description

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

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

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

Creation

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

Command-Line Equivalent

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

Response Selection

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

*   **Specify step-response inputs**

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

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

*   **Specify step-response outputs**

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

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

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

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

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

13-99

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 13-39.

Options

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

- **Limit % overshoot to**

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

- **Adjust for step amplitude**

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

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

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

- **Apply goal to**

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

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

For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

Related Examples
- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Disturbance Rejection Goal

Purpose
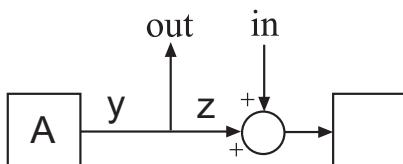
Attenuate disturbances at particular locations and in particular frequency bands, when using Control System Tuner.

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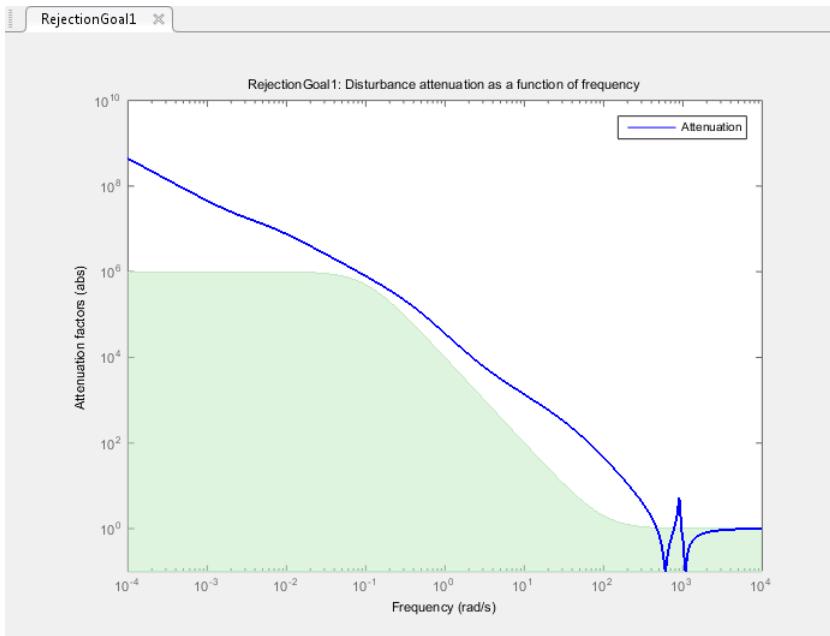
Disturbance Rejection Goal specifies the minimum attenuation of a disturbance injected at a specified location in a control system.

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

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

To specify a Disturbance Rejection Goal, you specify one or more locations at which to attenuate disturbance. You also provide the frequency-dependent minimum attenuation factor as a numeric LTI model. You can achieve disturbance attenuation only inside the control bandwidth. The loop gain must be larger than one for the disturbance to be attenuated (attenuation factor > 1).

RejectionGoal1: Disturbance attenuation as a function of frequency

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

If you prefer to specify sensitivity to disturbance at a location, rather than disturbance attenuation, you can use "Sensitivity Goal" on page 13-107.

Creation

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

Command-Line Equivalent

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

Disturbance Scenario

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

- **Inject disturbances at the following locations**

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

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

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

Rejection Performance

Specify the minimum disturbance attenuation as a function of frequency.

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

Options

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

- **Enforce goal in frequency range**

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

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

- **Equalize cross-channel effects**

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

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

Related Examples
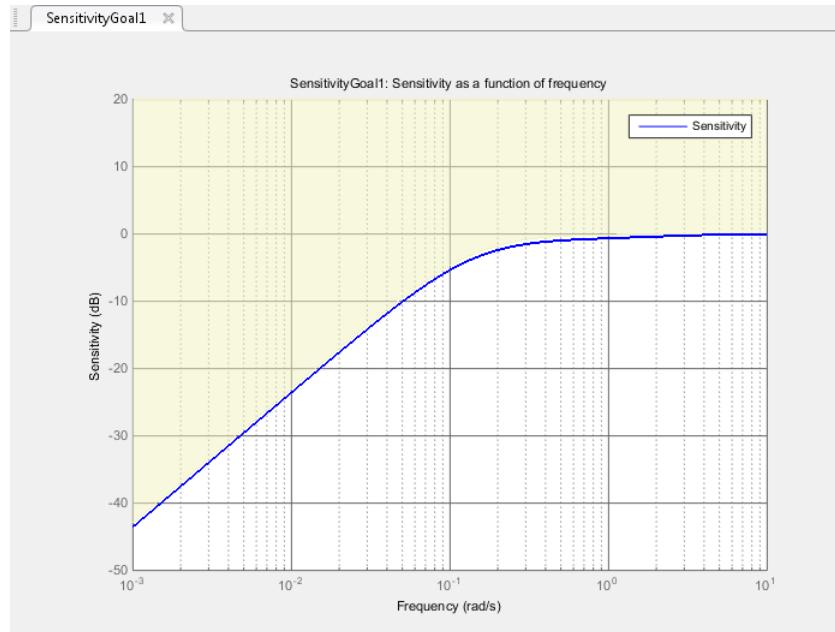
Sensitivity Goal

Purpose

Limit sensitivity of feedback loops to disturbances, when using Control System Tuner.

Description

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

To specify a Sensitivity Goal, you specify one or more locations at which to limit sensitivity. You also provide the frequency-dependent maximum sensitivity as a numeric LTI model whose magnitude represents the desired sensitivity as a function of frequency.

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

If you prefer to specify disturbance attenuation at a particular location, rather than sensitivity to disturbance, you can use "Disturbance Rejection Goal" on page 13-102.

Creation

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

Command-Line Equivalent

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

Sensitivity Evaluation

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

- **Measure sensitivity at the following locations**

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

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

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

Sensitivity Bound

Specify the maximum sensitivity as a function of frequency.

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

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

Options

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

- **Enforce goal in frequency range**

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

- **Equalize cross-channel effects**

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

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

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Weighted Gain Goal

Purpose

Frequency-weighted gain limit for tuning with Control System Tuner.

Description

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

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

Creation

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

Command-Line Equivalent

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

I/O Transfer Selection

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

- **Specify input signals**

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

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

- **Specify output signals**

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

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

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

For an example showing in more detail how to specify signal locations for a tuning goal, see "Specify Goals for Interactive Tuning" on page 13-39.

Weights

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

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

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

Options

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

- **Stabilize I/O transfer**

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

- **Enforce goal in frequency range**

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

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

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

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

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Weighted Variance Goal

Purpose

Frequency-weighted limit on noise impact on specified output signals for tuning with Control System Tuner.

Description

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

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

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

Creation

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

Command-Line Equivalent

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

I/O Transfer Selection

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

- **Specify stochastic inputs**

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

- **Specify stochastic outputs**

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

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

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

Weights

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

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

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

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

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

Options

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Tips

When you use this requirement to tune a control system, Control System Tuner attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 13-118), is infinite for continuous-time systems with nonzero feedthrough.

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

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough

might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 13-26.

Algorithms

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

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

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

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

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

$$f(x) = \frac{1}{\sqrt{T_s}} \left\| WL(z)\, H(z,x)\, WR(z) \right\|_2.$$

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

Related Examples
- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Minimum Loop Gain Goal

Purpose

Boost gain of feedback loops at low frequency when using Control System Tuner.
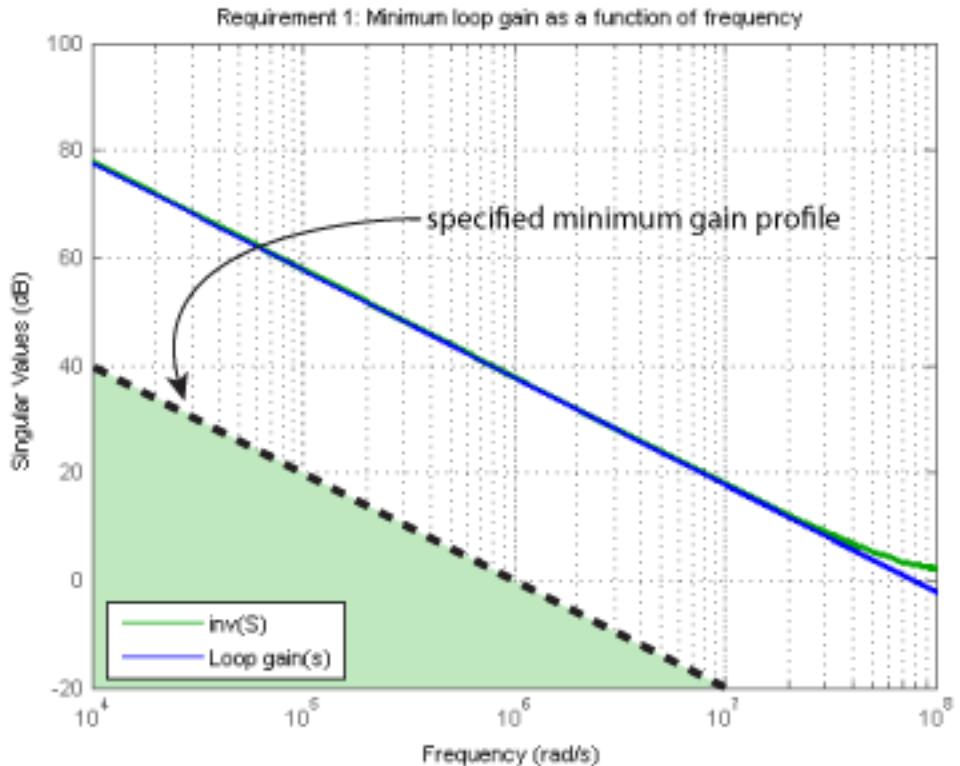
Description

Minimum Loop Gain Goal enforces a minimum loop gain in a particular frequency band. This tuning goal is useful, for example, for improving disturbance rejection at a particular location.

Minimum Loop Gain Goal imposes a minimum gain on the open-loop frequency response (L) at a specified location in your control system. You specify the minimum open-loop gain as a function of frequency (a minimum *gain profile*). For MIMO feedback loops, the specified gain profile is interpreted as a lower bound on the smallest singular value of L.

When you tune a control system, the minimum gain profile is converted to a minimum gain constraint on the inverse of the sensitivity function, inv(S) = ($I + L$).

The following figure shows a typical specified minimum gain profile (dashed line) and a resulting tuned loop gain, L (blue line). The green region represents gain profile values that are forbidden by this requirement. The figure shows that when L is much larger than 1, imposing a minimum gain on inv(S) is a good proxy for a minimum open-loop gain.

Requirement 1: Minimum loop gain as a function of frequency

Minimum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the **Open-Loop Response Selection** section of the dialog box.

Minimum Loop Gain Goal and Maximum Loop Gain Goal specify only low-gain or high-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use "Loop Shape Goal" on page 13-129 to specify that target loop shape.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Minimum gain for open-loop response** to create a Minimum Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use TuningGoal.MinLoopGain to specify a minimum loop gain goal.

Open-Loop Response Selection

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

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location.

  For example, to constrain the open-loop gain at a location named 'y', click ✚ **Add signal to list** and select 'y'. To constrain a MIMO response, select multiple signals or a vector-valued signal.

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

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

Desired Loop Gain

Use this section of the dialog box to specify the target minimum loop gain.

- **Pure integrator K/s**

  Check to specify a pure integrator shape for the target minimum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a

target minimum gain and frequency. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain above** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the minimum loop gain is 1 at 10 rad/s.

- **Other gain profile**

  Check to specify the minimum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum loop gain. For example, to specify minimum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

Options

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

- **Enforce goal in frequency range**

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

- **Stabilize closed loop system**

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

- **Equalize loop interactions**

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

$$f(x) = \left\| W_S \left(D^{-1} S D \right) \right\|_\infty .$$

W_S is the minimum loop gain profile. D is a diagonal scaling (for MIMO loops). S is the sensitivity function at the specified location.

Although S is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing a lower bound on the open-loop transfer function, L, in a frequency band where the gain of L is greater than 1. To see why, note that $S = 1/(1 + L)$. For SISO loops, when $|L| \gg 1$, $|S| \approx 1/|L|$. Therefore, enforcing the open-loop minimum gain requirement, $|L| > |W_S|$, is roughly equivalent to enforcing $|W_s S| < 1$. For MIMO loops, similar reasoning applies, with $||S|| \approx 1/\sigma_{\min}(L)$, where $\sigma_{\min}$ is the smallest singular value.

Related Examples

Maximum Loop Gain Goal

Purpose

Suppress gain of feedback loops at high frequency when using Control System Tuner.
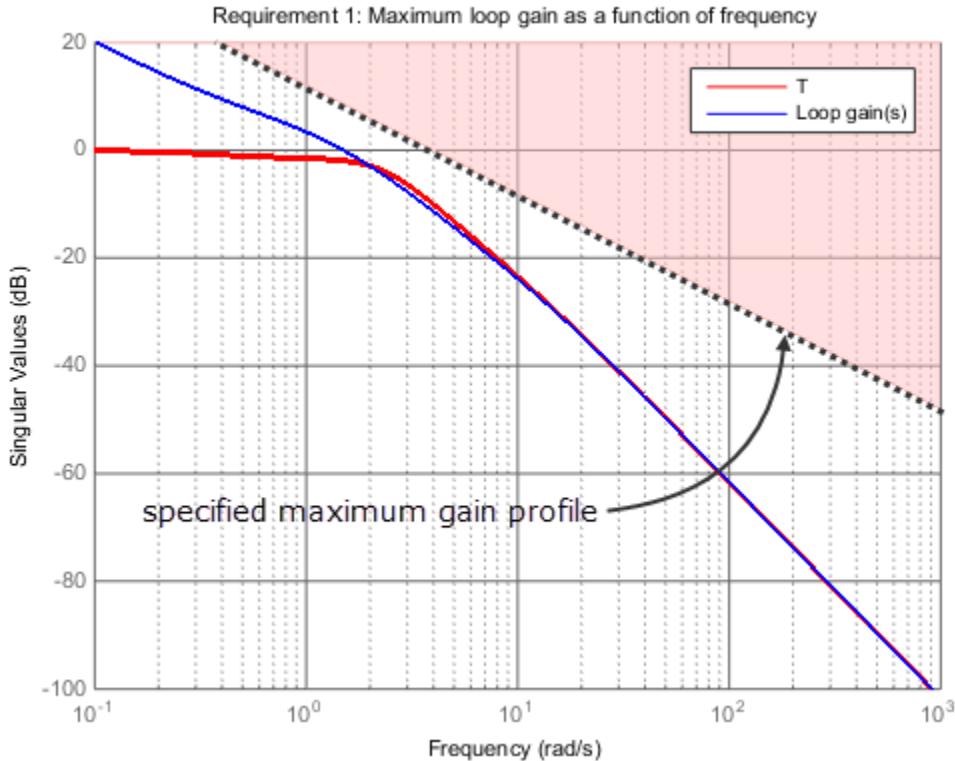
Description

Maximum Loop Gain Goal enforces a maximum loop gain in a particular frequency band. This tuning goal is useful, for example, for increasing system robustness to unmodeled dynamics.

Maximum Loop Gain Goal imposes a maximum gain on the open-loop frequency response (*L*) at a specified location in your control system. You specify the maximum open-loop gain as a function of frequency (a maximum *gain profile*). For MIMO feedback loops, the specified gain profile is interpreted as an upper bound on the largest singular value of *L*.

When you tune a control system, the maximum gain profile is converted to a maximum gain constraint on the complementary sensitivity function, $T) = L/(I + L)$.

The following figure shows a typical specified maximum gain profile (dashed line) and a resulting tuned loop gain, *L* (blue line). The pink region represents gain profile values that are forbidden by this requirement. The figure shows that when *L* is much smaller than 1, imposing a maximum gain on *T* is a good proxy for a maximum open-loop gain.

Requirement 1: Maximum loop gain as a function of frequency

Maximum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the **Open-Loop Response Selection** section of the dialog box.

Maximum Loop Gain Goal and Minimum Loop Gain Goal specify only high-gain or low-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use "Loop Shape Goal" on page 13-129 to specify that target loop shape.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Maximum gain for open-loop response** to create a Maximum Gain Goal.

Command-Line Equivalent

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

Open-Loop Response Selection

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

· **Shape open-loop response at the following locations**

Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location.

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

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

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

Desired Loop Gain

Use this section of the dialog box to specify the target maximum loop gain.

· **Pure integrator K/s**

Check to specify a pure integrator shape for the target maximum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a

target maximum gain and frequency. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain below** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the maximum loop gain is 1 at 10 rad/s.

- **Other gain profile**

  Check to specify the maximum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired maximum loop gain. For example, to specify maximum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

Options

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

- **Enforce goal in frequency range**

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

- **Stabilize closed loop system**

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

- **Equalize loop interactions**

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

$$f\left(x\right) = \left\|W_T\left(D^{-1}TD\right)\right\|_\infty .$$

W_T is the reciprocal of the maximum loop gain profile. D is a diagonal scaling (for MIMO loops). T is the complementary sensitivity function at the specified location.

Although T is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing an upper bound on the open-loop transfer, L, in a frequency band where the gain of L is less than one. To see why, note that $T = L/(I + L)$. For SISO loops, when $|L| << 1$, $|T| \approx |L|$. Therefore, enforcing the open-loop maximum gain requirement, $|L| < 1/|W_T|$, is roughly equivalent to enforcing $|W_T T| < 1$. For MIMO loops, similar reasoning applies, with $||T|| \approx \sigma_{\max}(L)$, where $\sigma_{\max}$ is the largest singular value.

Related Examples
- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

Loop Shape Goal
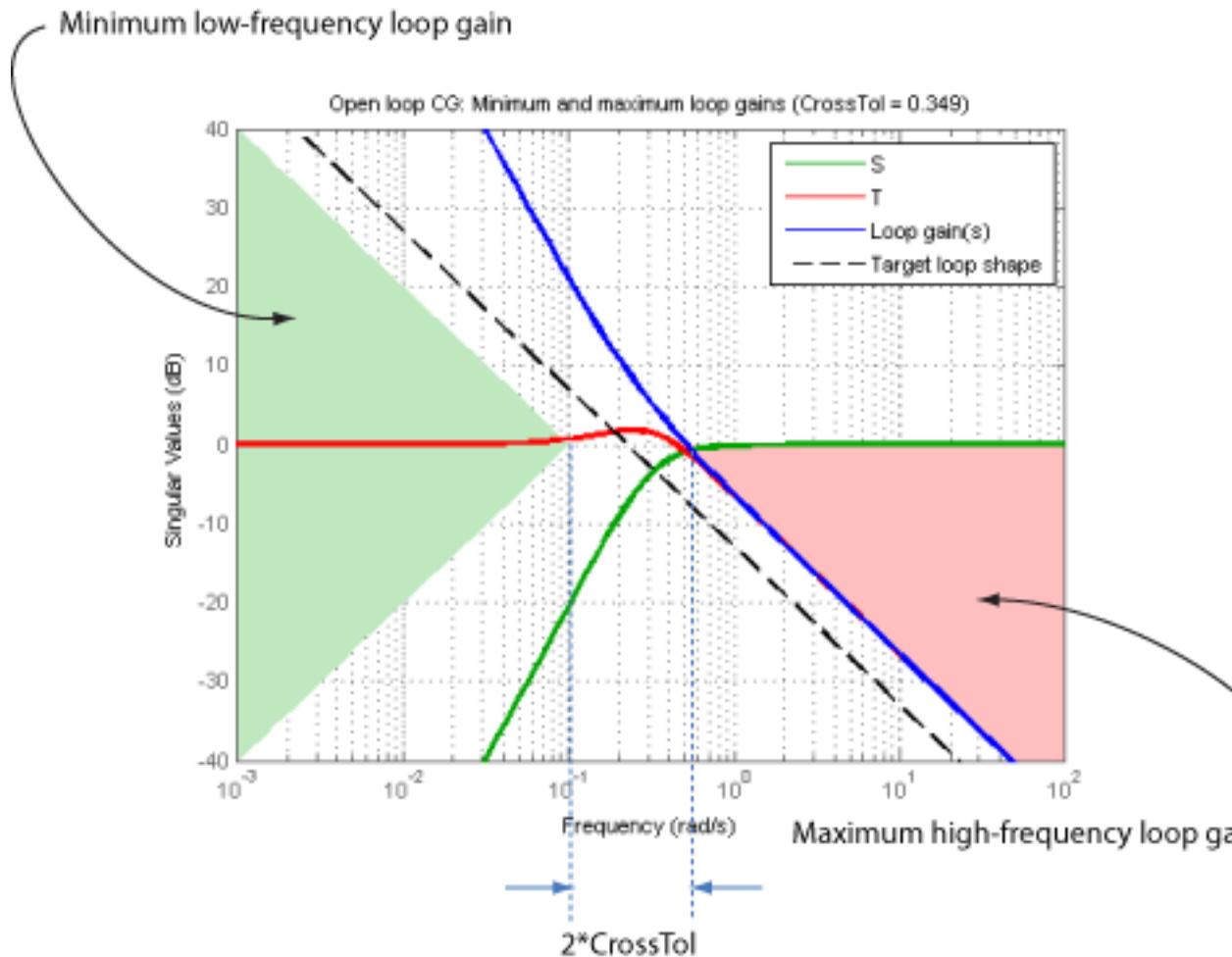
Purpose

Shape open-loop response of feedback loops when using Control System Tuner.

Description

Loop Shape Goal specifies a target *gain profile* (gain as a function of frequency) of an open-loop response. Loop Shape Goal constrains the open-loop, point-to-point response (*L*) at a specified location in your control system.

When you tune a control system, the target open-loop gain profile is converted into constraints on the inverse sensitivity function $\text{inv}(S) = (I + L)$ and the complementary sensitivity function $T = 1 - S$. These constraints are illustrated for a representative tuned system in the following figure.

Where L is much greater than 1, a minimum gain constraint on inv(S) (green shaded region) is equivalent to a minimum gain constraint on L. Similarly, where L is much smaller than 1, a maximum gain constraint on T (red shaded region) is equivalent to a maximum gain constraint on L. The gap between these two constraints is twice the crossover tolerance, which specifies the frequency band where the loop gain can cross 0 dB.

For multi-input, multi-output (MIMO) control systems, values in the gain profile greater than 1 are interpreted as minimum performance requirements. Such values are lower

bounds on the smallest singular value of the open-loop response. Gain profile values less than one are interpreted as minimum roll-off requirements, which are upper bounds on the largest singular value of the open-loop response. For more information about singular values, see `sigma`.

Use Loop Shape Goal when the loop shape near crossover is simple or well understood (such as integral action). To specify only high gain or low gain constraints in certain frequency bands, use "Minimum Loop Gain Goal" on page 13-119 or "Maximum Loop Gain Goal" on page 13-124. When you do so, the software determines the best loop shape near crossover.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Target shape for open-loop response** to create a Loop Shape Goal.

Command-Line Equivalent

When tuning control systems at the command line, use TuningGoal.LoopShape to specify a loop-shape goal.

Open-Loop Response Selection

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

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location.

  For example, to constrain the open-loop gain at a location named `'y'`, click ➕ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

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

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

Desired Loop Shape

Use this section of the dialog box to specify the target loop shape.

- **Pure integrator wc/s**

  Check to specify a pure integrator and crossover frequency for the target loop shape. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 10 in the **Crossover frequency wc** text box.

- **Other gain profile**

  Check to specify the target loop shape as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop shape using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired loop shape. For example, to specify a target loop shape of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

Options

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

- **Enforce loop shape within**

  Specify the tolerance in the location of the crossover frequency, in decades. For example, to allow gain crossovers within half a decade on either side of the target crossover frequency, enter 0.5. Increase the crossover tolerance to increase the ability of the tuning algorithm to enforce the target loop shape for all loops in a MIMO control system.

- **Enforce goal in frequency range**

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

- **Stabilize closed loop system**

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

- **Equalize loop interactions**

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

$$f(x) = \left\| \begin{matrix} W_S S \\ W_T T \end{matrix} \right\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor. (If **Equalize loop interactions** is set to Off, then $D = I$.)

$T = S - I$ is the complementary sensitivity function.

W_S and W_T are weighting functions derived from the specified loop shape.

Related Examples
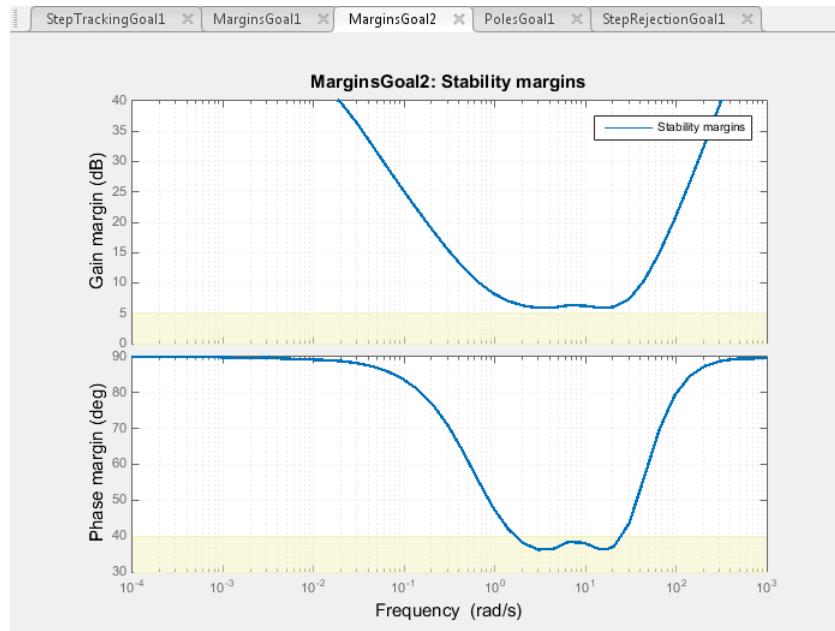
Margins Goal

Purpose

Enforce specified gain and phase margins when using Control System Tuner.

Description

Margins Goal enforces specified gain and phase margins on a SISO or MIMO feedback loop. For MIMO feedback loops, the gain and phase margins are based on the notion of disk margins, which guarantee stability for concurrent gain and phase variations in all feedback channels. See `loopmargin` for more information about disk margins.

In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the margins goal is not met. For more information about interpreting this plot, see "Interpreting Stability Margins in Control System Tuning" on page 13-237.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Minimum stability margins** to create a Margins Goal.

Command-Line Equivalent

When tuning control systems at the command line, use TuningGoal.Margins to specify a stability margin goal.

Feedback Loop Selection

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

- **Measure stability margins at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the stability margins. To constrain a SISO loop, select a single-valued location. For example, to constrain the stability margins at a location named 'y', click ✚ **Add signal to list** and select 'y'. To constrain a MIMO loop, select multiple signals or a vector-valued signal.

- **Measure stability margins with the following loops open**

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

Desired Margins

Use this section of the dialog box to specify the minimum gain and phase margins for the feedback loop.

- **Gain margin (dB)**

  Enter the required minimum gain margin for the feedback loop as a scalar value expressed in dB.

- **Phase margin (degrees)**

  Enter the required minimum phase margin for the feedback loop as a scalar value expressed in degrees.

For MIMO feedback loops, the gain and phase margins are based on the notion of disk margins, which guarantee stability for concurrent gain and phase variations in all feedback channels. See `loopmargin` for more information about disk margins.

Options

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

- **Enforce goal in frequency range**

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

  For best results with stability margin requirements, pick a frequency band extending about one decade on each side of the gain crossover frequencies.

- **D scaling order**

  This value controls the order (number of states) of the scalings involved in computing MIMO stability margins. Static scalings (scaling order 0) are used by default. Increasing the order may improve results at the expense of increased computations. If the stability margin plot shows a large gap between the optimized and actual margins, consider increasing the scaling order. See "Interpreting Stability Margins in Control System Tuning" on page 13-237.

- **Apply goal to**

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

13-137

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

For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

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

$$f(x) = \|2\alpha S - \alpha I\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor.

α is a scalar parameter computed from the specified gain and phase margin.

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157

More About

- "Interpreting Stability Margins in Control System Tuning" on page 13-237

Passivity Goal

Purpose

Enforce passivity of specific input/output map when using Control System Tuner.

Description

Passivity Goal enforces passivity of the response of the transfer function between the specified signal locations. A system is *passive* if all its I/O trajectories ($u(t),y(t)$) satisfy:
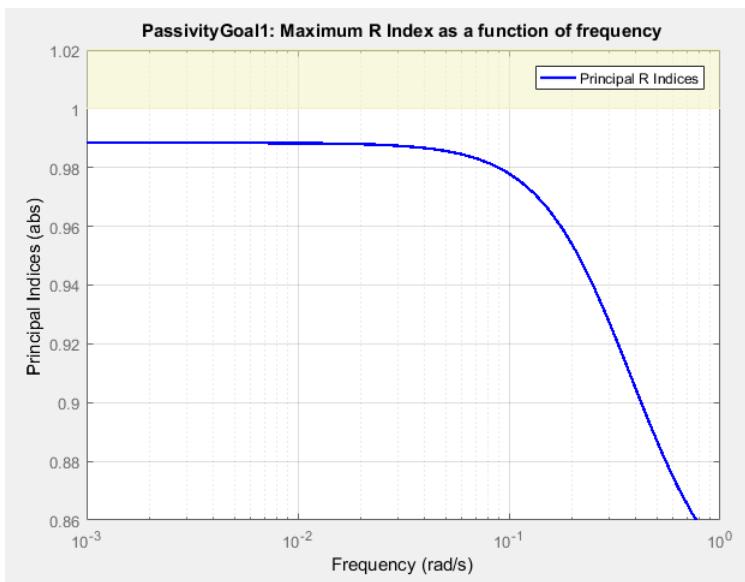
$$\int_0^T y(t)^{\mathrm{T}} u(t)\,dt > 0,$$

for all $T > 0$. Equivalently, a system is passive if its frequency response is positive real, which means that for all $\omega > 0$,

$$G(j\omega) + G(j\omega)^H > 0$$

Passivity Goal creates a constraint that enforces:

$$\int_0^T y(t)^{\mathrm{T}} u(t)\,dt > \nu \int_0^T u(t)^{\mathrm{T}} u(t)\,dt + \rho \int_0^T y(t)^{\mathrm{T}} y(t)\,dt,$$

for all $T > 0$. To enforce the overall passivity condition, set the minimum input passivity index (ν) and the minimum output passivity index (ρ) to zero. To enforce an excess of passivity at the inputs or outputs, set ν or ρ to a positive value. To permit a shortage of passivity, set ν or ρ to a negative value. See "About Passivity and Passivity Indices" on page 10-2 for more information about these indices.

PassivityGoal1: Maximum R Index as a function of frequency

In Control System Tuner, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the index described in "Algorithms" on page 13-142.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal > Passivity Goal**.

Command-Line Equivalent

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

I/O Transfer Selection

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

- **Specify input signals**

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

- **Specify output signals**

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

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

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

Options

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

- **Minimum input passivity index**

   Enter the target value of ν in the text box. To enforce an excess of passivity at the specified inputs, set $\nu > 0$. To permit a shortage of passivity, set $\nu < 0$. By default, the passivity goal enforces $\nu = 0$, passive at the inputs with no required excess of passivity.

- **Minimum output passivity index**

   Enter the target value of ρ in the text box. To enforce an excess of passivity at the specified outputs, set $\rho > 0$. By default, the passivity goal enforces $\rho = 0$, passive at the outputs with no required excess of passivity.

- **Enforce goal in frequency range**

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

- **Apply goal to**

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

   For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

For **Passivity Goal**, for a closed-loop transfer function $G(s,x)$ from the specified inputs to the specified outputs, $f(x)$ is given by:

$$f\left(x\right) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

R is the relative sector index (see `getSectorIndex`) of $[G(s,x); I]$, for the sector represented by:

$$Q = \begin{pmatrix} 2\rho & -I \\ -I & 2\nu \end{pmatrix},$$

where ρ is the minimum output passivity index and ν is the minimum input passivity index specified in the dialog box. $R_{\max}$ is fixed at 10_6, included to avoid numerical errors for very large R.

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157
- "Passive Control of Water Tank Level" on page 13-448

More About

- "About Passivity and Passivity Indices" on page 10-2

Weighted Passivity Goal

Purpose

Enforce passivity of a frequency-weighted transfer function when tuning in Control System Tuner.

Description
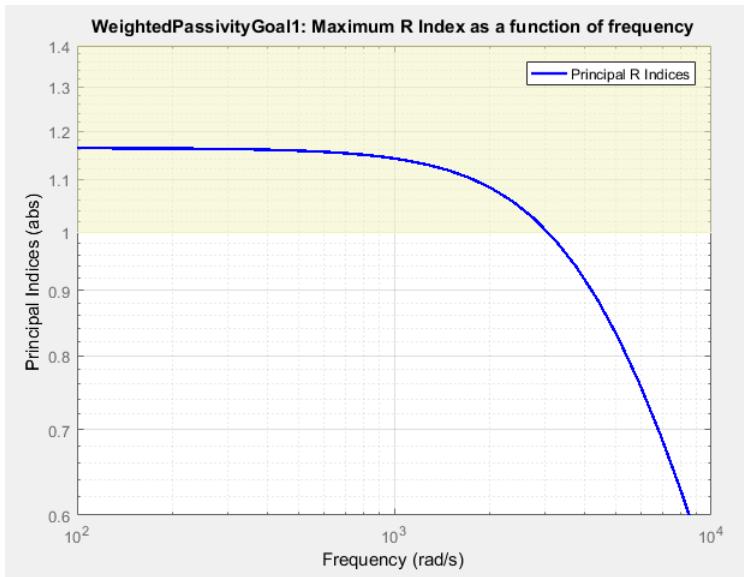
Weighted Passivity Goal enforces the passivity of $H(s) = W_L(s)T(s)W_R(s)$, where $T(s)$ is the transfer function from specified inputs to outputs. $W_L(s)$ and $W_R(s)$ are frequency weights used to emphasize particular frequency bands. A system is *passive* if all its I/O trajectories $(u(t),y(t))$ satisfy:

$$\int_0^T y(t)^{\mathrm{T}} u(t) \, dt > 0,$$

for all $T > 0$. Weighted Passivity Goal creates a constraint that enforces:

$$\int_0^T y(t)^{\mathrm{T}} u(t) \, dt > v \int_0^T u(t)^{\mathrm{T}} u(t) \, dt + \rho \int_0^T y(t)^{\mathrm{T}} y(t) \, dt,$$

for the trajectories of the weighted transfer function $H(s)$, for all $T > 0$. To enforce the overall passivity condition, set the minimum input passivity index (v) and the minimum output passivity index (ρ) to zero. To enforce an excess of passivity at the inputs or outputs of the weighted transfer function, set v or ρ to a positive value. To permit a shortage of passivity, set v or ρ to a negative value. See `getPassiveIndex` for more information about these indices.

In Control System Tuner, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the index described in "Algorithms" on page 13-148.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Weighted Passivity Goal**.

Command-Line Equivalent

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

I/O Transfer Selection

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

- **Specify input signals**

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

· **Specify output signals**

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

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

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

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal. $H(s) = W_L(s)T(s)W_R(s)$, where $T(s)$ is the transfer function from specified inputs to outputs.

W_L provides the weighting for the output channels of $H(s)$, and W_R provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting function. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify different weights for each channel by specifying matrices or MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of W_L and W_R. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

Options

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

- **Minimum input passivity index**

  Enter the target value of v in the text box. To enforce an excess of passivity at the specified inputs, set $v > 0$. To permit a shortage of passivity, set $v < 0$. By default, the passivity goal enforces $v = 0$, passive at the inputs with no required excess of passivity.

- **Minimum output passivity index**

  Enter the target value of ρ in the text box. To enforce an excess of passivity at the specified outputs, set $\rho > 0$. By default, the passivity goal enforces $\rho = 0$, passive at the outputs with no required excess of passivity.

- **Enforce goal in frequency range**

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

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

For **Weighted Passivity Goal**, for a closed-loop transfer function $T(s,x)$ from the specified inputs to the specified outputs, and the weighted transfer function $H(s,x) = W_L(s)T(s,x)W_R(s)$, $f(x)$ is given by:

$$f(x) = \frac{R}{1 + R/R_{max}}, \quad R_{max} = 10^6.$$

R is the relative sector index (see `getSectorIndex`) of $[H(s,x); I]$, for the sector represented by:

$$Q = \begin{pmatrix} 2\rho & -I \\ -I & 2\nu \end{pmatrix},$$

where ρ is the minimum output passivity index and ν is the minimum input passivity index specified in the dialog box. R_{max} is fixed at 10_6, included to avoid numerical errors for very large R.

Related Examples

More About
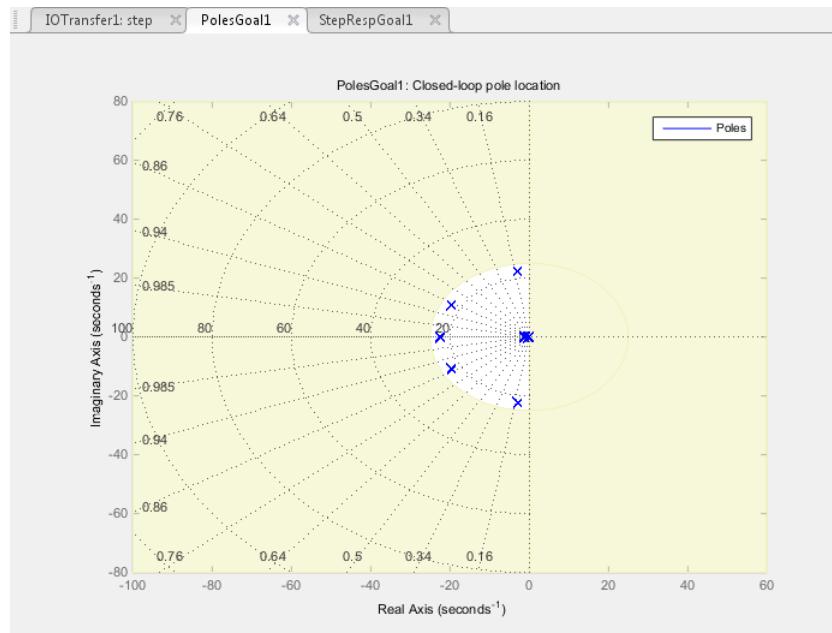
Poles Goal

Purpose

Constrain the dynamics of the closed-loop system, specified feedback loops, or specified open-loop configurations, when using Control System Tuner.

Description

Poles Goal constrains the dynamics of your entire control system or of specified feedback loops of your control system. Constraining the dynamics of a feedback loop means constraining the dynamics of the sensitivity function measured at a specified location in the control system.

Using Poles Goal, you can specify finite minimum decay rate or minimum damping for the poles in the control system or specified loop. You can specify a maximum natural frequency for these poles, to eliminate fast dynamics in the tuned control system.

In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met.

To constrain dynamics or ensure stability of a single tunable component of the control system, use "Controller Poles Goal" on page 13-154.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Constraint on closed-loop dynamics** to create a Poles Goal.

Command-Line Equivalent

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

Feedback Configuration

Use this section of the dialog box to specify the portion of the control system for which you want to constrain dynamics. You can also specify loop-opening locations for evaluating the tuning goal.

- **Entire system**

  Select this option to constrain the locations of closed-loop poles of the control system.

- **Specific feedback loop(s)**

  Select this option to specify one or more feedback loops to constrain. Specify a feedback loop by selecting a signal location in your control system. Poles Goal constrains the dynamics of the sensitivity function measured at that location. (See getSensitivity for information about sensitivity functions.)

  To constrain the dynamics of a SISO loop, select a single-valued location. For example, to constrain the dynamics of the sensitivity function measured at a location named 'y', click ➕**Add signal to list** and select 'y'. To constrain the dynamics of a MIMO loop, select multiple signals or a vector-valued signal.

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

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against

the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, ✚ **Add signal to list** and select `'x'`.

Pole Location

Use this section of the dialog box to specify the limits on pole locations.

- **Minimum decay rate**

    Enter the target minimum decay rate for the system poles. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDecay` for continuous-time systems, or `log(|z|) < -MinDecay*Ts` for discrete-time systems with sample time `Ts`. This constraint helps ensure stable dynamics in the tuned system.

    Enter 0 to impose no constraint on the decay rate.

- **Minimum damping**

    Enter the target minimum damping of closed-loop poles of tuned system, as a value between 0 and 1. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDamping*|s|`. In discrete time, the damping ratio is computed using `s=log(z)/Ts`.

    Enter 0 to impose no constraint on the damping ratio.

- **Maximum natural frequency**

    Enter the target maximum natural frequency of poles of tuned system, in the units of the control system model you are tuning. When you tune the control system using this requirement, closed-loop system poles that depend on the tunable parameters are constrained to satisfy `|s| < MaxFrequency` for continuous-time systems, or `|log(z)| < MaxFrequency*Ts` for discrete-time systems with sample time `Ts`. This constraint prevents fast dynamics in the control system.

    Enter `Inf` to impose no constraint on the natural frequency.

Options

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

- **Enforce goal in frequency range**

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

  The Poles Goal applies only to poles with natural frequency within the range you specify.

- **Apply goal to**

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

  For more information about tuning for multiple models, see "Robust Tuning Approaches".

Algorithms

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

For **Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Poles Goal constrains the closed-loop poles of a feedback loop to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the smallest damping among the constrained poles is $\zeta = 0.5$ exactly.
- $f(x) = 1.1$ means the smallest damping $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the smallest damping $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

Related Examples
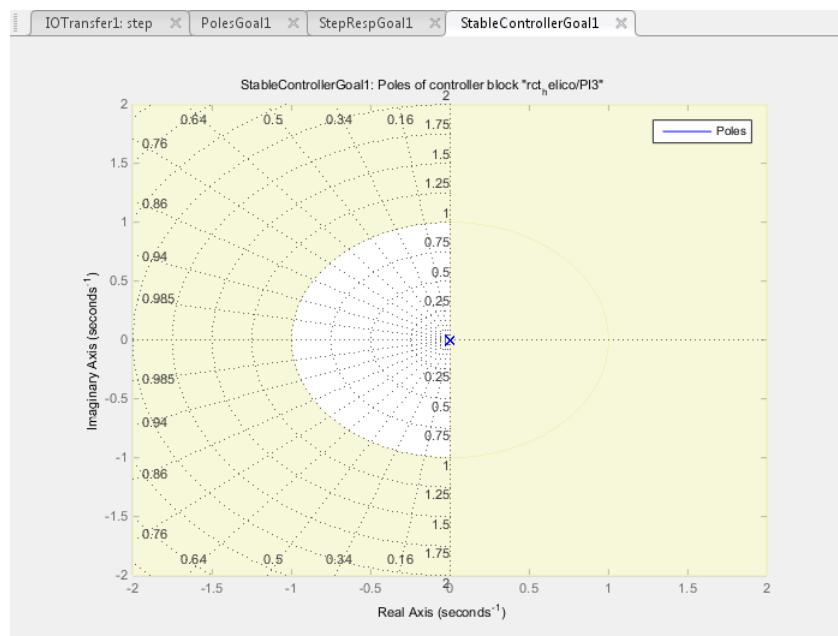
Controller Poles Goal

Purpose

Constrain the dynamics of a specified tunable block in the tuned control system, when using Control System Tuner.

Description

Controller Poles Goal constrains the dynamics of a tunable block in your control system model. Controller Poles Goal can impose a stability constraint on the specified block. You can also specify a finite minimum decay rate, a minimum damping rate, or a maximum natural frequency for the poles of the block. These constraints allow you to eliminate fast dynamics and control ringing in the response of the tunable block.

In Control System Tuner, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met. The constraint applies to all poles in the block except fixed integrators, such as the I term of a PID controller.

To constrain dynamics or ensure stability of an entire control system or a feedback loop in the control system, use "Poles Goal" on page 13-149.

Creation

In the **Tuning** tab of Control System Tuner, select **New Goal** > **Constraint on controller dynamics** to create a Controller Poles Goal.

Command-Line Equivalent

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

Constrain Dynamics of Tuned Block

From the drop-down menu, select the tuned block in your control system to which to apply the Controller Poles Goal.

If the block you want to constrain is not in the list, add it to the Tuned Blocks list. In Control System Tuner, in the **Tuning** tab, click 🔲**Select Blocks**. For more information about adding tuned blocks, see "Specify Blocks to Tune in Control System Tuner" on page 13-24.

Keep Poles Inside the Following Region

Use this section of the dialog box to specify the limits on pole locations.

- **Minimum decay rate**

  Enter the desired minimum decay rate for the poles of the tunable block. Poles of the block are constrained to satisfy $Re(s) < -MinDecay$ for continuous-time blocks, or $log(|z|) < -MinDecay*Ts$ for discrete-time blocks with sample time Ts.

  Specify a nonnegative value to ensure that the block is stable. If you specify a negative value, the tuned block can include unstable poles.

- **Minimum damping**

  Enter the desired minimum damping ratio of poles of the tunable block, as a value between 0 and 1. Poles of the block that depend on the tunable parameters are constrained to satisfy $Re(s) < -MinDamping*|s|$. In discrete time, the damping ratio is computed using $s=log(z)/Ts$.

- **Maximum natural frequency**

  Enter the target maximum natural frequency of poles of the tunable block, in the units of the control system model you are tuning. Poles of the block are constrained to satisfy `|s| < MaxFrequency` for continuous-time blocks, or `|log(z)| < MaxFrequency*Ts` for discrete-time blocks with sample time `Ts`. This constraint prevents fast dynamics in the tunable block.

Algorithms

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

For **Controller Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Controller Poles Goal constrains the pole of a tuned block to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the damping of the pole is $\zeta = 0.5$ exactly.
- $f(x) = 1.1$ means the damping is $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the damping is $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

Related Examples

- "Specify Goals for Interactive Tuning" on page 13-39
- "Manage Tuning Goals" on page 13-157
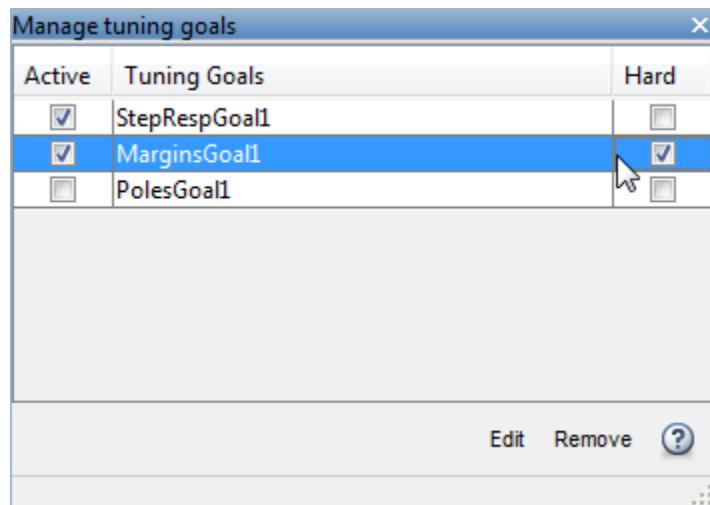
Manage Tuning Goals

Control System Tuner lets you designate one or more tuning goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have goals. Control System Tuner attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

By default, new goals are designated soft goals. In the **Tuning** tab, click **Manage Goals** to open the **Manage tuning goals** dialog box. Check **Hard** for any goal to designate it a hard goal.

You can also designate any tuning goal as inactive for tuning. In this case the software ignores the tuning goal entirely. Use this dialog box to select which tuning goals are active when you tune the control system. **Active** is checked by default for any new goals. Uncheck **Active** for any design goal that you do not want enforced.

For example, if you tune with the following configuration, Control System Tuner optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.

All tuning goals you have created in the Control System Tuner session are listed in the dialog box. To edit an existing tuning goal, select it in the list and click **Edit**. To delete a tuning goal from the list, select it and click **Remove**.
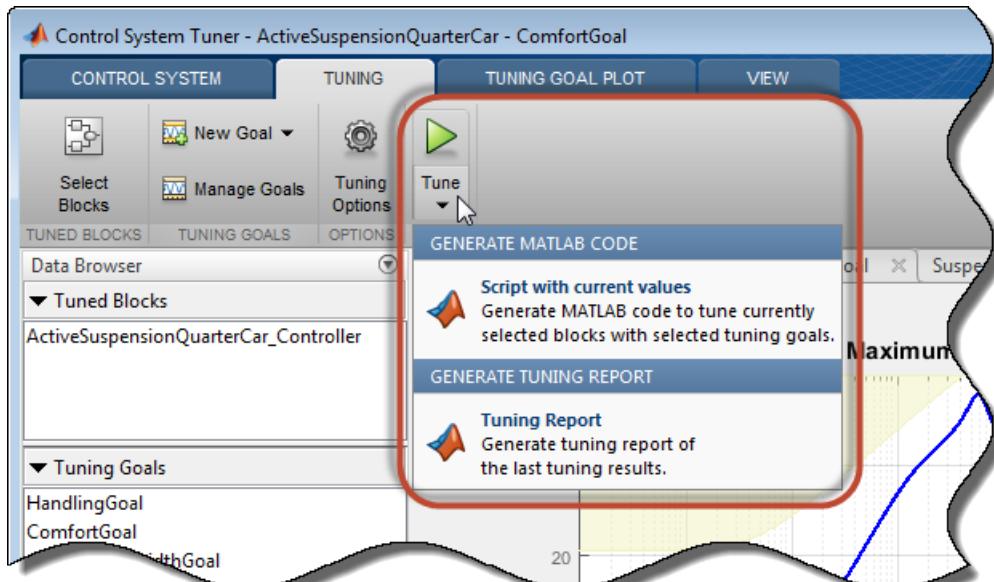
To add more tuning goals to the list, in Control System Tuner, in the **Tuning** tab, click **New Goal**. For more information about creating tuning goals, see "Specify Goals for Interactive Tuning" on page 13-39.

Generate MATLAB Code from Control System Tuner for Command-Line Tuning

This example shows how to generate a MATLAB script in Control System Tuner for tuning a control system at the command line. Generated scripts are useful when you want to programmatically reproduce a result you obtained interactively. A generated MATLAB script also enables you to programmatically perform multiple tuning operations with variations in tuning goals, system parameters, or model conditions such as operating point.

Tip You can also save a Control System Tuner session to reproduce within Control System Tuner. To do so, in the **Control System** tab, click 🖫 **Save Session**.

To generate a MATLAB script in Control System Tuner, in the **Tuning** tab, click **Tune**▾. Select **Script with current values**.

The MATLAB Editor displays the generated script, which script reproduces programmatically the current tuning configuration of Control System Tuner.

For example, suppose you generate a MATLAB script after completing all steps in the example "Control of a Linear Electric Actuator Using Control System Tuner" on page 13-309. The generated script computes the operating point used for tuning, designates the blocks to tune, creates the tuning goals, and performs other operations to reproduce the result at the command line.

The first section of the script creates the `slTuner` interface to the Simulinkmodel (`rct_linact` in this example). The `slTuner` interface stores a linearization of the model and parameterizations of the blocks to tune.

```matlab
%% Create system data with slTuner interface
TunedBlocks = {'rct_linact/Current Controller/Current PID'; ...
               'rct_linact/Speed Controller/Speed PID'};
AnalysisPoints = {'rct_linact/Speed Demand (rpm)/1'; ...
                  'rct_linact/Current Sensor/1'; ...
                  'rct_linact/Hall Effect Sensor/1'; ...
                  'rct_linact/Speed Controller/Speed PID/1'; ...
                  'rct_linact/Current Controller/Current PID/1'};
OperatingPoints = 0.5;
% Create the slTuner object
CLO = slTuner('rct_linact',TunedBlocks,AnalysisPoints,OperatingPoints);
```

The `slTuner` interface also specifies the operating point at which the model is linearized, and marks as analysis points all the signal locations required to specify the tuning goals for the example. (See "Create and Configure slTuner Interface to Simulink Model" on page 13-183.)

If you are tuning a control system modeled in MATLAB instead of Simulink, the first section of the script constructs a `genss` model that has equivalent dynamics and parameterization to the `genss` model of the control system that you specified Control System Tuner.

Next, the script creates the three tuning goals specified in the example. The script uses `TuningGoal` objects to capture these tuning goals. For instance, the script uses `TuningGoal.Tracking` to capture the Tracking Goal of the example.

```matlab
%% Create tuning goal to follow reference commands with prescribed performance
% Inputs and outputs
Inputs = {'rct_linact/Speed Demand (rpm)/1'};
Outputs = {'rct_linact/Hall Effect Sensor/1[rpm]'};
```

```
% Tuning goal specifications
ResponseTime = 0.1; % Approximately reciprocal of tracking bandwidth
DCError = 0.001; % Maximum steady-state error
PeakError = 1; % Peak error across frequency
% Create tuning goal for tracking
TR = TuningGoal.Tracking(Inputs,Outputs,ResponseTime,DCError,PeakError);
TR.Name = 'TR'; % Tuning goal name
```

After creating the tuning goals, the script sets any algorithm options you had set in Control System Tuner. The script also designates tuning goals as soft or hard goals, according to the configuration of tuning goals in Control System Tuner. (See "Manage Tuning Goals" on page 13-157.)

```
%% Create option set for systune command
Options = systuneOptions();

%% Set soft and hard goals
SoftGoals = [ TR ; ...
              MG1 ; ...
              MG2 ];
HardGoals = [];
```

In this example, all the goals are designated as soft goals when the script is generated. Therefore, `HardGoals` is empty.

Finally, the script tunes the control system by calling `systune` on the `slTuner` interface using the tuning goals and options.

```
%% Tune the parameters with soft and hard goals
[CL1,fSoft,gHard,Info] = systune(CL0,SoftGoals,HardGoals,Options);
```

The script also includes an optional call to `viewSpec`, which displays graphical representations of the tuning goals to aid you in interpreting and validating the tuning results. Uncomment this line of code to generate the plots.

```
%% View tuning results
% viewSpec([SoftGoals;HardGoals],CL1,Info);
```

You can add calls to functions such `getIOTransfer` to make the script generate additional analysis plots.

Related Examples

- "Create and Configure slTuner Interface to Simulink Model" on page 13-183

Interpreting Tuning Results

When you tune a control system with `systune` or Control System Tuner, the software reports on the tuning progress and provides results that help you determine how well the tuned control system meets your design requirements.

In this section...

Optimization Problem

The automated tuning software converts each soft and hard tuning goal into normalized values $f_i(x)$ and $g_j(x)$, respectively. Here, x is the vector of tunable parameters in the control system to tune.

The software then solves the minimization problem:

Minimize $\max_i f_i(x)$ subject to $\max_j g_j(x) < 1$, for $x_{\min} < x < x_{\max}$.

x_{min} and x_{max} are the minimum and maximum values of the free parameters of the control system.

When you use both soft and hard tuning goals, the software solves the optimization as a sequence of subproblems of the form:

$$\min_x \ \max\big(\alpha f(x), g(x)\big).$$

The software adjusts the multiplier α so that the solution of the subproblems converges to the solution of the original constrained optimization problem.
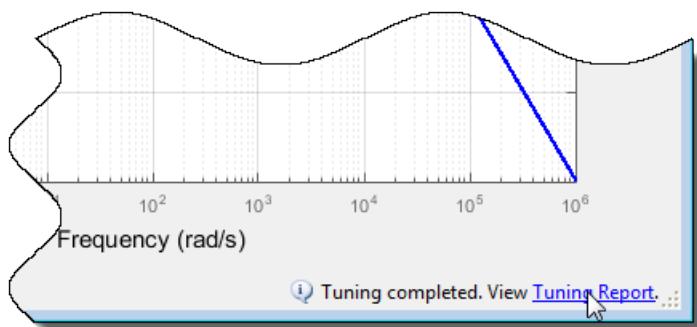
For information about the functions $f_i(x)$ and $g_j(x)$ for each type of constraint, see the reference pages for each tuning goal.

When you tune a control system, the form of the output depends on the work environment.
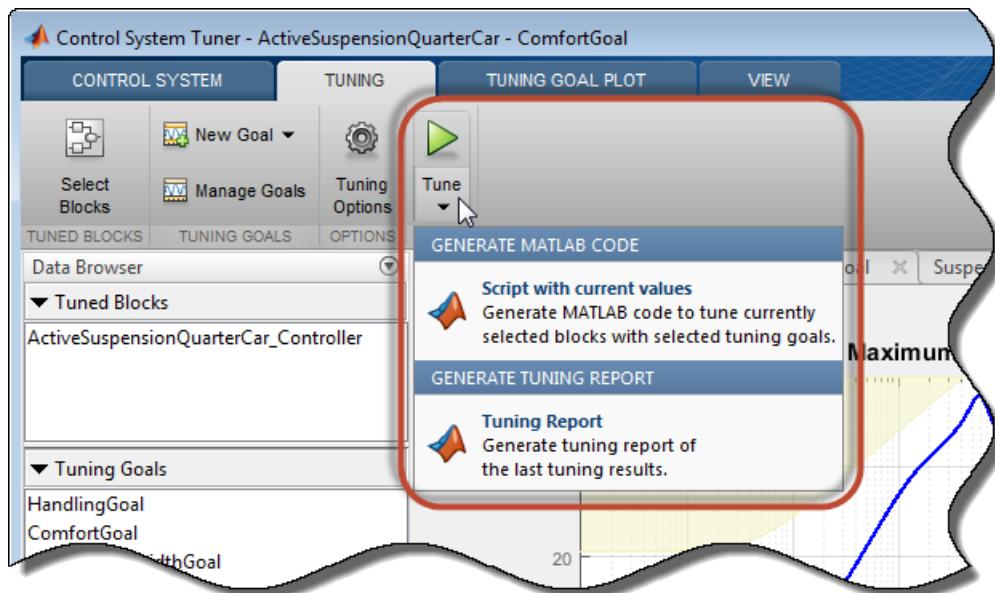
Results of Tuning in Control System Tuner

When you click ▶ in the **Tuning** tab, Control System Tuner updates the current design to use the tuned parameters that best satisfy the minimization problem. By default, the current design is reflected in all tuning goal plots and response plots you have active in Control System Tuner.
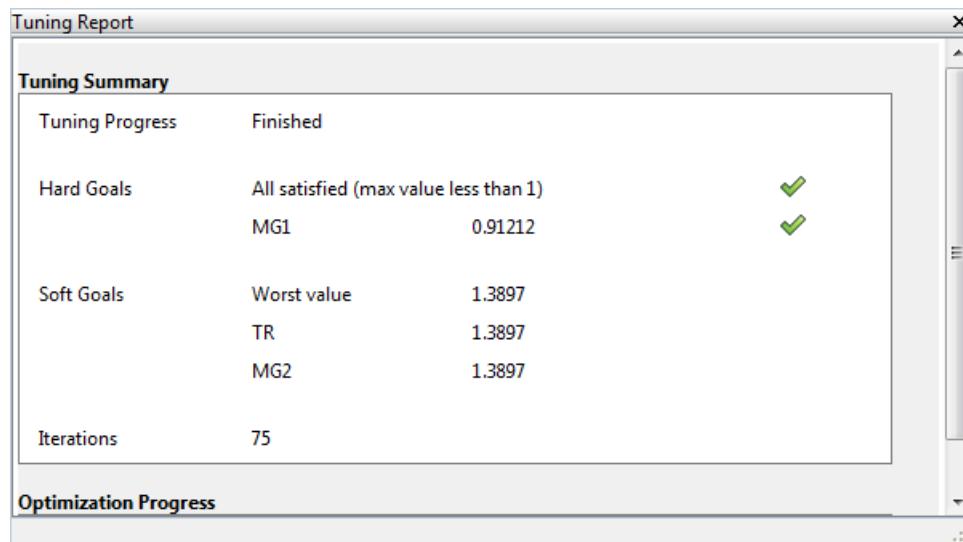
Control System Tuner also compiles a Tuning Report summarizing the best achieved values of $f_i(x)$ and $g_j(x)$. To view the tuning report immediately after tuning a control system, click **Tuning Report** at the bottom-right corner of Control System Tuner.

Tip You can view a report from the most recent tuning run at any time. In the **Tuning** tab, click **Tune▼**, and select `Tuning Report`.

The tuning report provides a measure of how close the optimization came to satisfying the hard goals and soft goals. The values displayed are the $f_i(x)$ and $g_j(x)$ obtained by the optimization.

The **Hard Goals** area shows the minimized $g_i(x)$ values and indicates which are satisfied. The **Soft Goals** area highlights the largest of the minimized $f_i(x)$ values as **Worst Value**, and lists the values for all the requirements. For each requirement, the closer the value is to 1, the closer that requirement is to being satisfied.

Results of Command-Line Tuning

The `systune` command returns the control system model or `slTuner` interface with the tuned parameter values. `systune` also returns the best achieved values of each $f_i(x)$ and $g_j(x)$ as the vector-valued output arguments `fSoft` and `gHard`, respectively. See the `systune` reference page for more information.

Algorithms

The software uses the nonsmooth optimization algorithms described in [1].

The software computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] Apkarian, P. and D. Noll, "Nonsmooth H-infinity Synthesis." *IEEE Transactions on Automatic Control*, Vol. 51, No. 1, (2006), pp. 71–86.

[2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, Vol. 14, No, 4 (1990), pp. 287–293.

Related Examples

Create Response Plots in Control System Tuner
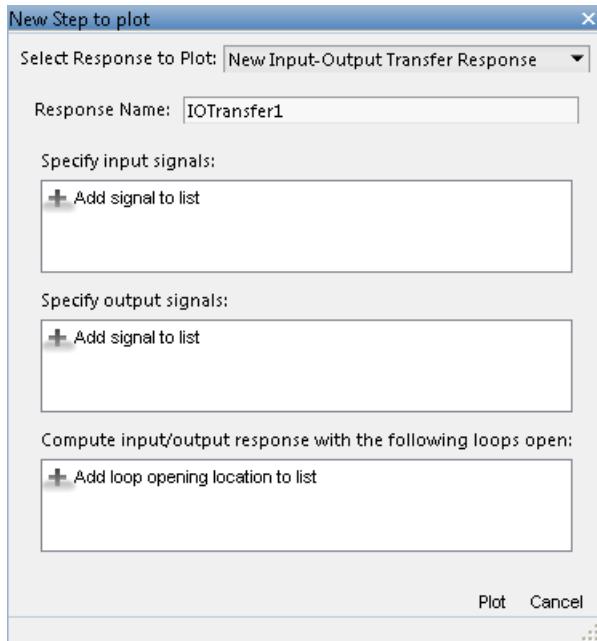
This example shows how to create response plots for analyzing system performance in Control System Tuner. Control System Tuner can generate many types of response plots in the time and frequency domains. You can view responses of SISO or MIMO transfer functions between inputs and outputs at any location in your model. Use response plots to validate the performance of your tuned control system.

This example creates response plots for analyzing the sample model `rct_helico`.

Choose Response Plot Type
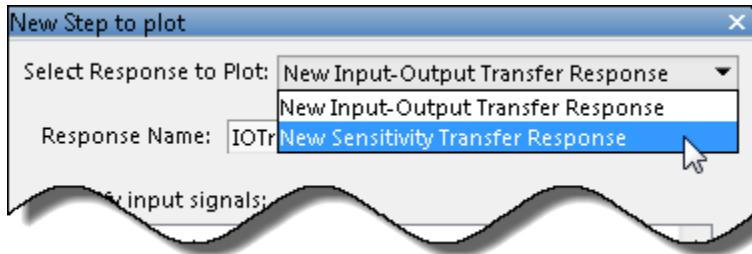
In Control System Tuner, in the **Control System** tab, click ⬚ **New Plot**. Select the type of plot you want to create.

A new plot dialog box opens in which you specify the inputs and outputs of the portion of your control system whose response you want to plot. For example, select **New step** to create a step response plot from specified inputs to specified outputs of your system.

Specify Transfer Function

Choose which transfer function associated with the specified inputs and outputs you want to analyze.

For most response plots types, the **Select Response to Plot** menu lets you choose one of the following transfer functions:

- `New Input-Output Transfer Response` — Transfer function between specified inputs and outputs, computed with loops open at any additionally specified loop-opening locations.
- `New Sensitivity Transfer Response` — Sensitivity function computed at the specified location and with loops open at any specified loop-opening locations.
- `New Open-Loop Response` — Open loop point-to-point transfer function computed at the specified location and with loops open at any additionally specified loop-opening locations.
- `Entire System Response` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations for the entire closed-loop control system.
- `Response of Tuned Block` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations of tuned blocks.

Name the Response

Type a name for the response in the **Response Name** text box. Once you have specified signal locations defining the response, Control System Tuner stores the response under this name. When you create additional new response plots, the response appears by this name in **Select Response to Plot** menu.
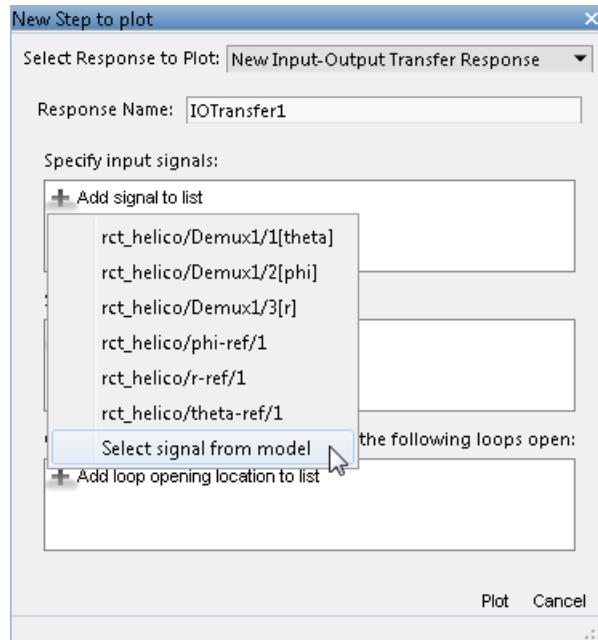
Choose Signal Locations for Evaluating System Response

Specify the signal locations in your control system at which to evaluate the selected response. For example, the step response plot displays the response of the system at one

or more output locations to a unit step applied at one or more input locations. Use the **Specify input signals** and **Specify output signals** sections of the dialog box to specify these locations. (Other tuning goal types, such as loop-shape or stability margins, require you to specify only one location for evaluation. The procedure for specifying the location is the same as illustrated here.)
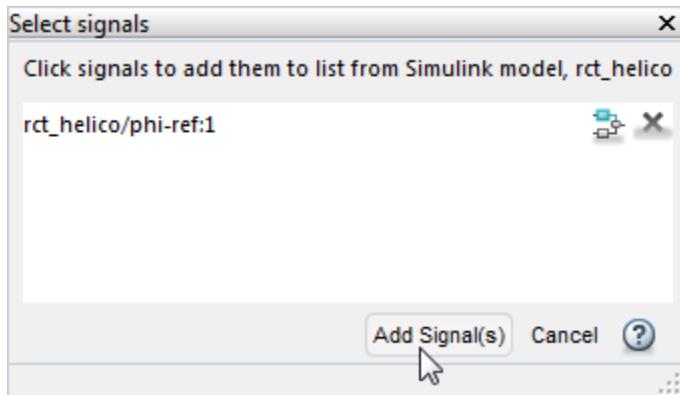
Under **Specify input signals**, click ✚ **Add signal to list**. A list of available input locations appears.

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.
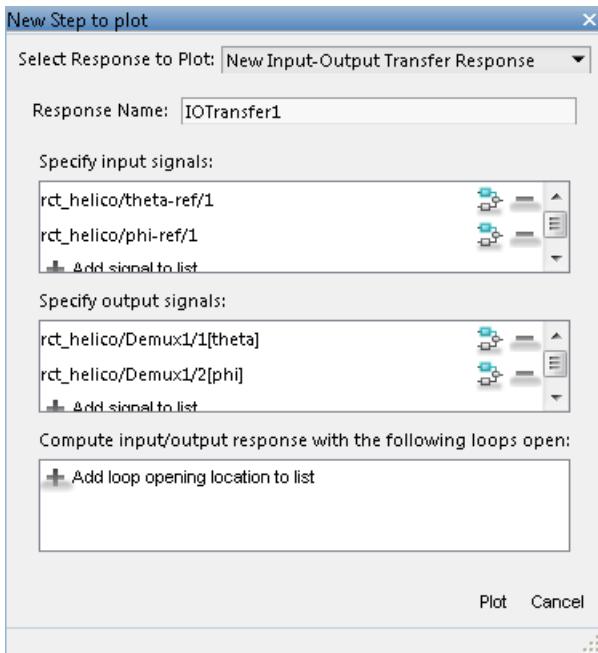
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO response, and click multiple signals to create a MIMO response.

Click **Add signal(s)**. The **Select signals** dialog box closes.

The signal or signals you selected now appear in the list of step-response inputs in the new-plot dialog box.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration plots the MIMO response to a step input applied at `theta-ref` and `phi-ref` and measured at `theta` and `phi` in the Simulink model `rct_helico`.

Tip To highlight any selected signal in the Simulink model, click ⧉. To remove a signal from the input or output list, click ▬ . When you have selected multiple signals, you can reorder them using ⬆ and ⬇.
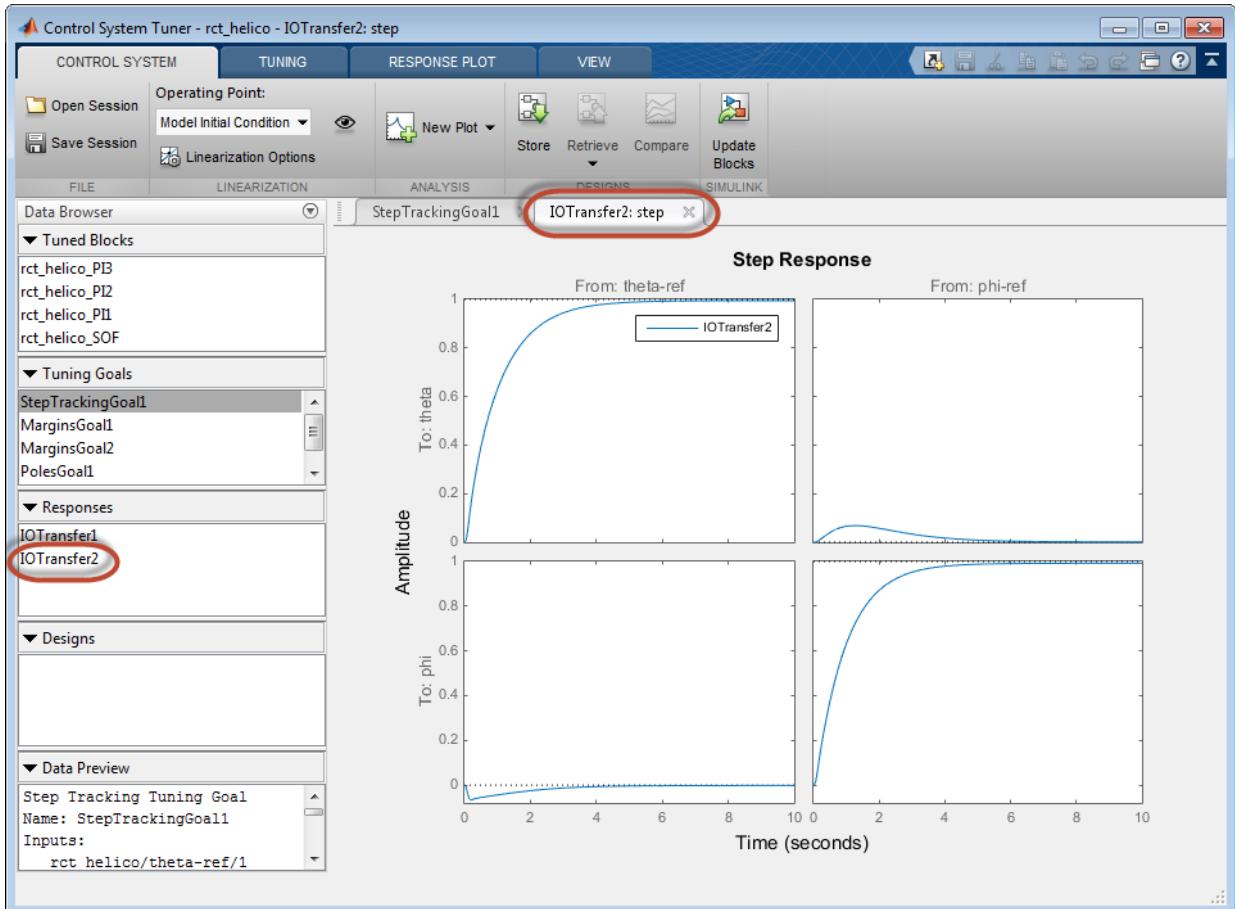
Specify Loop Openings

You can evaluate most system responses with loops open at one or more locations in the control system. Click ➕**Add loop opening location to list** to specify such locations for the response.

Store and Plot the Response
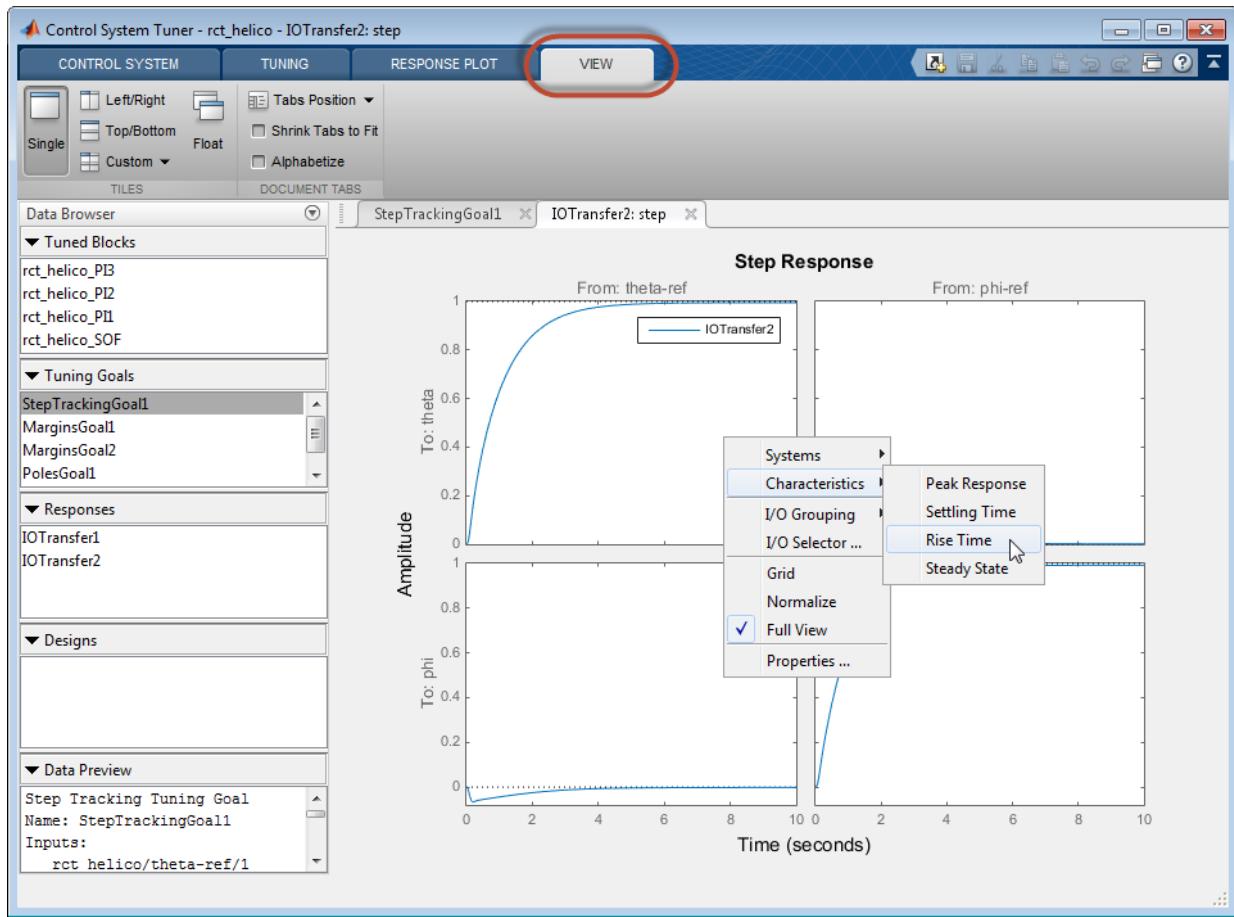
When you have finished specifying the response, click **Plot** in the new plot dialog box. The new response appears in the **Responses** section of the Data Browser. A new figure

opens displaying the response plot. When you tune your control system, you can refer to this figure to evaluate the performance of the tuned system.

Tip To edit the specifications of the response, double-click the response in the Data Browser. Any plots using that response update to reflect the edited response.

View response characteristics such as rise-times or peak values by right-clicking on the plot. Other options for managing and organizing multiple plots are available in the **View** tab.
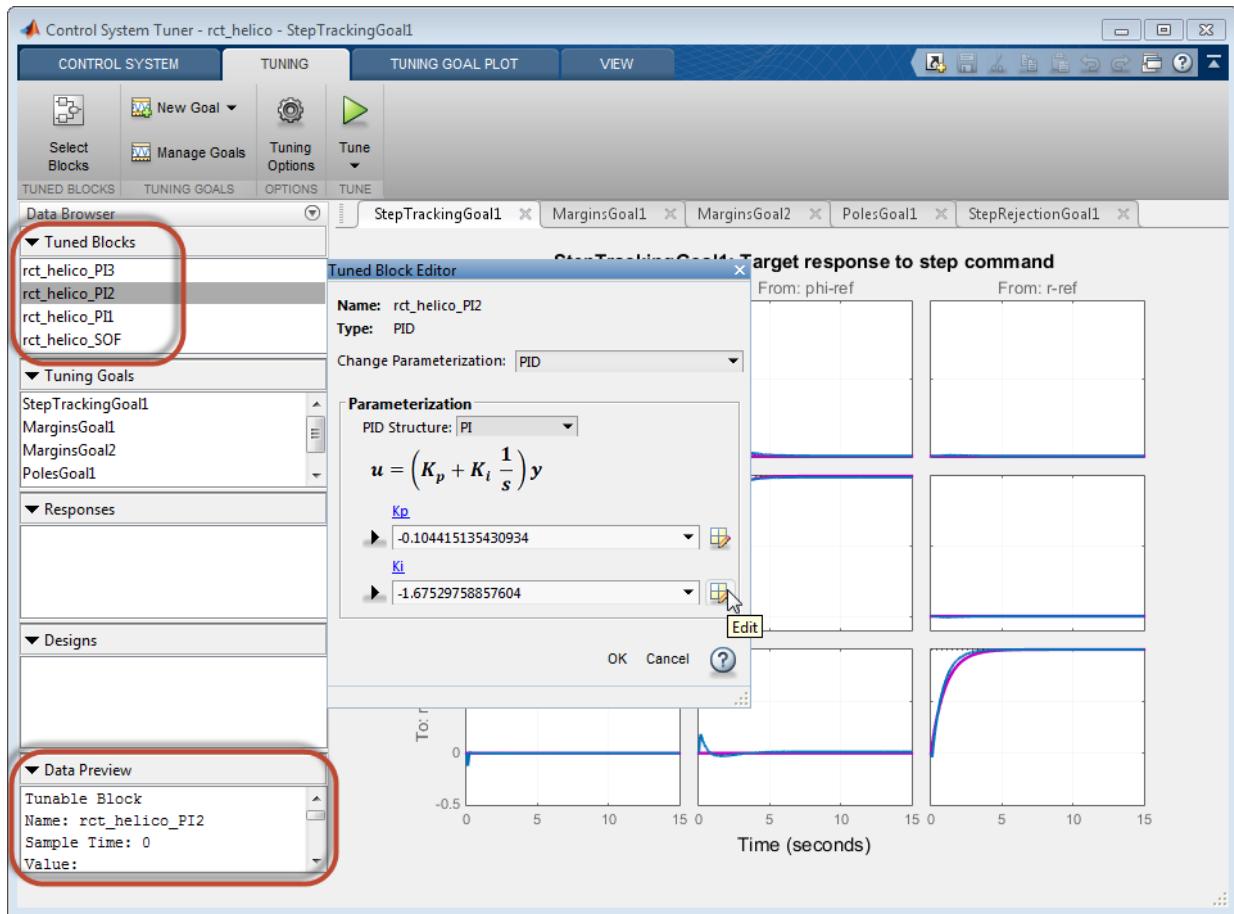
Related Examples

- "Compare Performance of Multiple Tuned Controllers" on page 13-177
- "Examine Tuned Controller Parameters in Control System Tuner" on page 13-175
- "Validate Tuned Controller in Simulink" on page 13-182

Examine Tuned Controller Parameters in Control System Tuner

After you tune your control system, Control System Tuner gives you two ways to view the current values of the tuned block parameters:

- In the Data Browser, in the **Tuned Blocks** area, select the block whose parameters you want to view. A text summary of the block and its current parameter values appears in the Data Browser in the **Data Preview** area.

- In the Data Browser, in the **Tuned Blocks** area, double-click the block whose parameters you want to view. The Tuned Block Editor opens, displaying the current values of the parameters. For array-valued parameters, click to open a variable editor displaying values in the array.

Note: To find a tuned block in the Simulink model, right-click the block name in the **Data Browser** and select **Highlight**.

Related Examples

- "View and Change Block Parameterization in Control System Tuner" on page 13-26

Compare Performance of Multiple Tuned Controllers
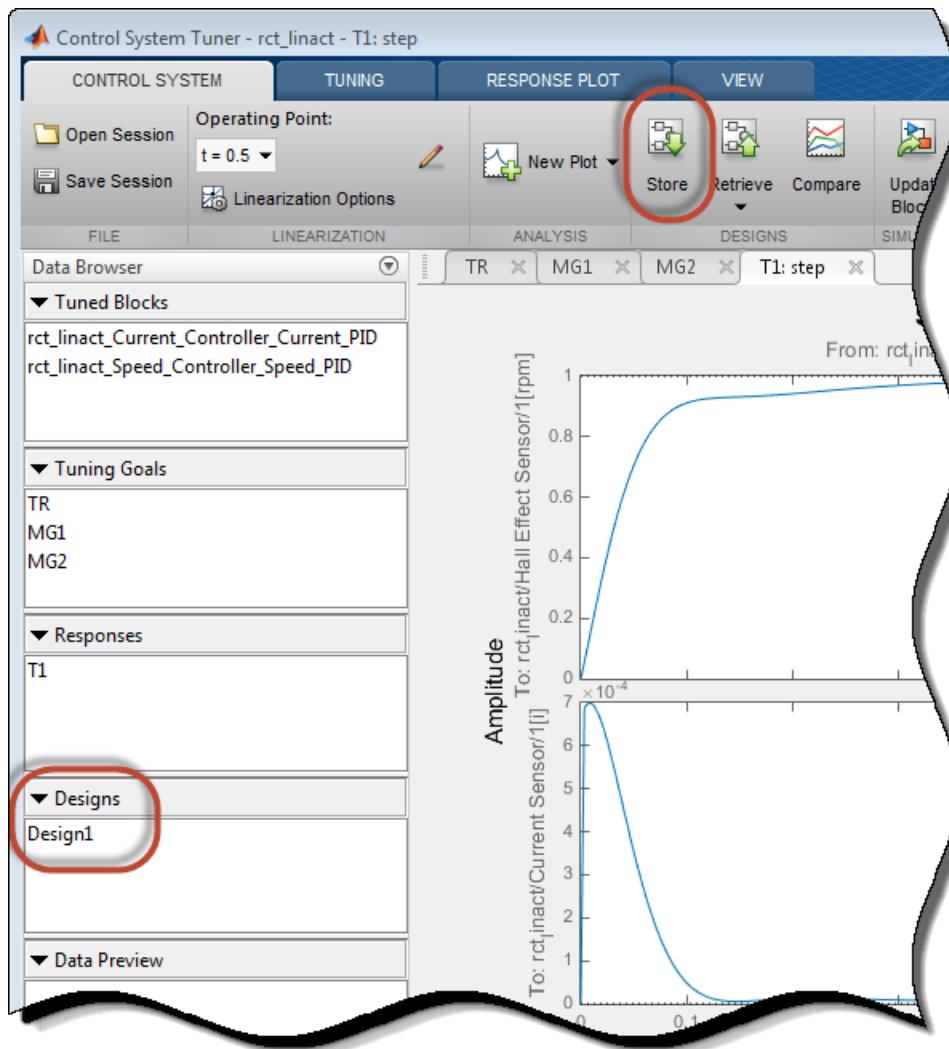
This example shows how to compare the performance of a control system tuned with two different sets of tuning goals. Such comparison is useful, for example, to see the effect on performance of changing a tuning goal from hard goal to soft goal. Comparing performance is also useful to see the effect of adding an additional tuning goal when an initial design fails to satisfy all your performance requirements either in the linearized system or when validated against a full nonlinear model.

This example compares tuning results for the sample model `rct_linact`.

Store First Design

After tuning a control system with a first set of design requirements, store the design in Control System Tuner.

In the **Control System** tab, click 🔃 **Store**. The stored design appears in the Data Browser in the **Designs** area.

Change the name of the stored design, if desired, by right-clicking on the data browser entry.

Compute New Design

In the **Tuning** tab, make any desired changes to the tuning goals for the second design. For example, add new tuning goals or edit existing tuning goals to change specifications.

Or, in ![Manage Goals icon] **Manage Goals**, change which tuning goals are active and which are designated hard constraints or soft requirements.
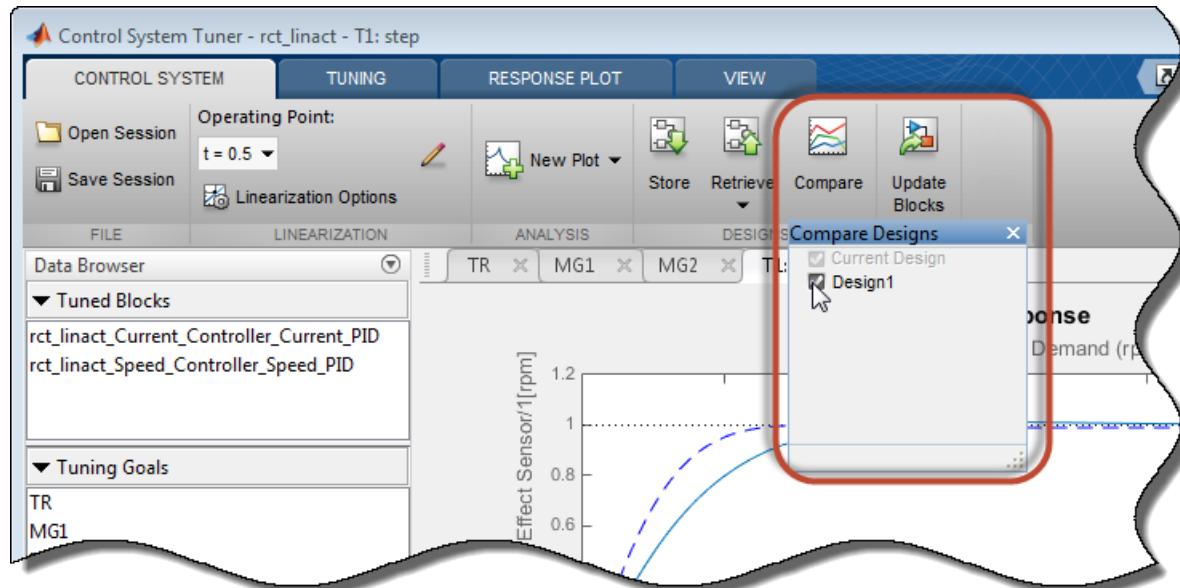
When you are ready, retune the control system with the new set of tuning goals. Click

![Tune icon] **Tune**. Control System Tuner updates the current design (the current set of controller parameters) with the new tuned design. All existing plots, which by default show the current design, are updated to reflect the new current design.
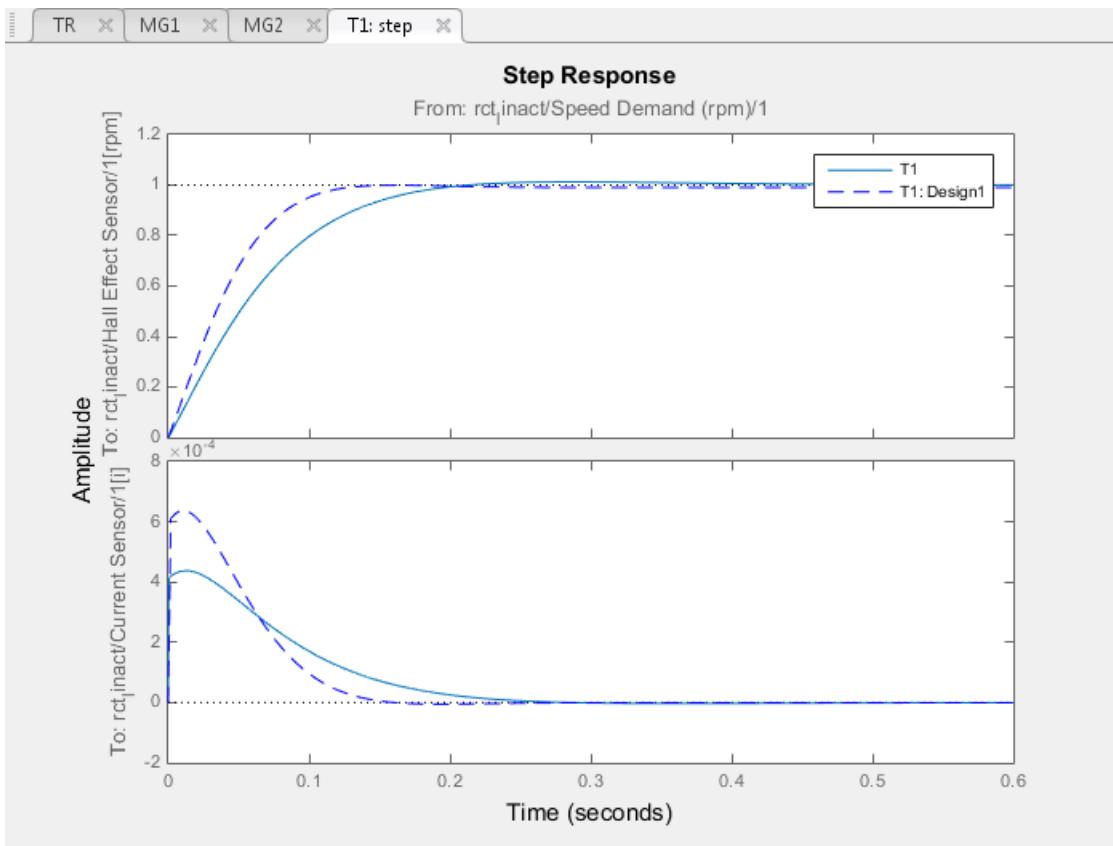
Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design. In the **Control System** tab, click ![Compare icon] **Compare**. The **Compare Designs** dialog box opens.

In the **Compare Designs** dialog box, the current design is checked by default. Check the box for the design you want to compare to the current design. All response plots and

tuning goal plots update to reflect the checked designs. The solid trace corresponds to the current design. Other designs are identified by name in the plot legend.
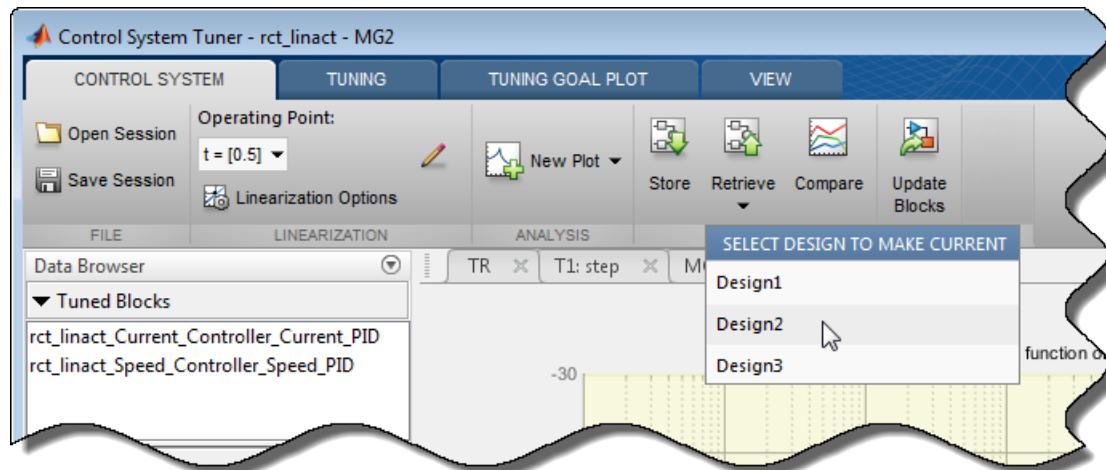
Use the same procedure save and compare as many designs as you need.

Restore Previously Saved Design

Under some conditions, it is useful to restore the tuned parameter values from a previously saved design as the current design. For example, clicking ⧉ **Update Blocks** writes the current parameter values to the Simulink model. If you decide to test a stored controller design in your full nonlinear model, you must first restore those stored values as the current design.

To do so, click **Retrieve**. Select the stored design that you want to make the current design.
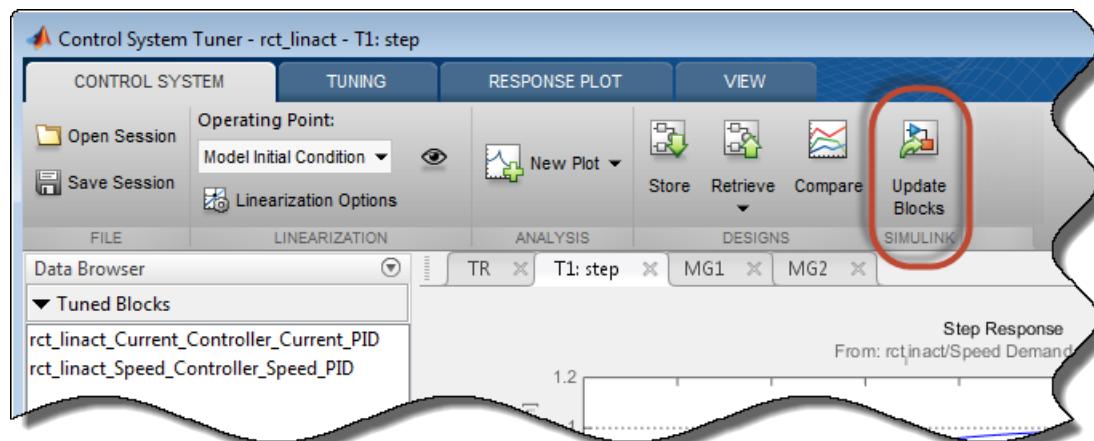
Related Examples

- "Create Response Plots in Control System Tuner" on page 13-168

Validate Tuned Controller in Simulink

Because Control System Tuner designs for a linearization of your Simulink model, tuned block parameters must be validated by simulating the full nonlinear model, even if the tuned system meets all your tuning goals in Control System Tuner.

To write tuned block parameters to a Simulink model, in the **Control System** tab, click ![icon]**Update Blocks**.

Control System Tuner transfers the current values of the tuned block parameters to the corresponding blocks in the Simulink model. Simulate the model to evaluate model performance using the tuned values.

Tip If you tune the Simulink model at an operating point other than the model initial condition, you might want to initialize the model at the same operating point before simulating. See "Simulate Simulink Model at Specific Operating Point" in the Simulink Control Design documentation.
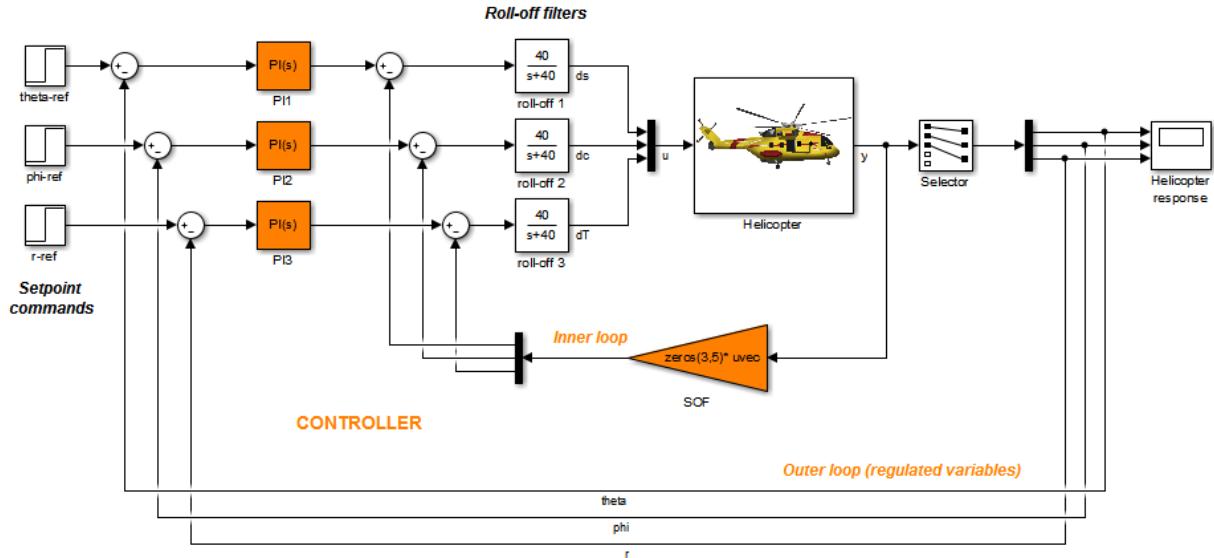
To update Simulink model with parameter values from a previous design stored in Control System Tuner, click ![icon]**Retrieve** and select the stored design that you want to make the current design. Then click ![icon]**Update Blocks**.

Create and Configure slTuner Interface to Simulink Model

This example shows how to create and configure an `slTuner` interface for a Simulink® model. The `slTuner` interface parameterizes blocks in your model that you designate as tunable and and allows you to tune them using `systune`. The `slTuner` interface generates a linearization of your Simulink model, and also allows you to extract linearized system responses for analysis and validation of the tuned control system.

For this example, create and configure an `slTuner` interface for tuning the Simulink model `rct_helico`, a multiloop controller for a rotorcraft. Open the model.

```
open_system('rct_helico');
```


The control system consists of two feedback loops. The inner loop (static output feedback) provides stability augmentation and decoupling. The outer loop (PI controllers) provides the desired setpoint tracking performance.

Suppose that you want to tune this model to meet the following control objectives:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, specified rise times, minimal overshoot, and minimal cross-coupling.

13-183

- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise.
- Provide strong multivariable gain and phase margins (robustness to simultaneous gain/phase variations at the plant inputs and outputs).

The `systune` command can jointly tune the controller blocks `SOF` and the PI controllers) to meet these design requirements. The `slTuner` interface sets up this tuning task.

Create the `slTuner` interface.

```
ST0 = slTuner('rct_helico',{'PI1','PI2','PI3','SOF'});
```

This command initializes the `slTuner` interface with the three PI controllers and the `SOF` block designated as tunable. Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model.

To configure the `slTuner` interface, designate as analysis points any signal locations of relevance to your design requirements. First, add the outputs and reference inputs for the tracking requirements.

```
addPoint(ST0,{'theta-ref','theta','phi-ref','phi','r-ref','r'});
```

When you create a `TuningGoal.Tracking` object that captures the tracking requirement, this object references the same signals.

Configure the `slTuner` interface for the stability margin requirements. Designate as analysis points the plant inputs and outputs (control and measurement signals) where the stability margins are measured.

```
addPoint(ST0,{'u','y'});
```

Display a summary of the `slTuner` interface configuration in the command window.

```
ST0
```

```
slTuner tuning interface for "rct_helico":

4 Tuned blocks: (Read-only TunedBlocks property)
-------------------------
Block 1: rct_helico/PI1
Block 2: rct_helico/PI2
```

```
Block 3: rct_helico/PI3
Block 4: rct_helico/SOF

8 Analysis points:
-------------------------
Point 1: Port 1 of rct_helico/theta-ref
Point 2: Signal "theta", located at port 1 of rct_helico/Demux1
Point 3: Port 1 of rct_helico/phi-ref
Point 4: Signal "phi", located at port 2 of rct_helico/Demux1
Point 5: Port 1 of rct_helico/r-ref
Point 6: Signal "r", located at port 3 of rct_helico/Demux1
Point 7: Signal "u", located at port 1 of rct_helico/Mux3
Point 8: Signal "y", located at port 1 of rct_helico/Helicopter

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
     Parameters         : []
     OperatingPoints    : [] (model initial condition will be used.)
     BlockSubstitutions : []
     Options            : [1x1 linearize.SlTunerOptions]
     Ts                 : 0
```
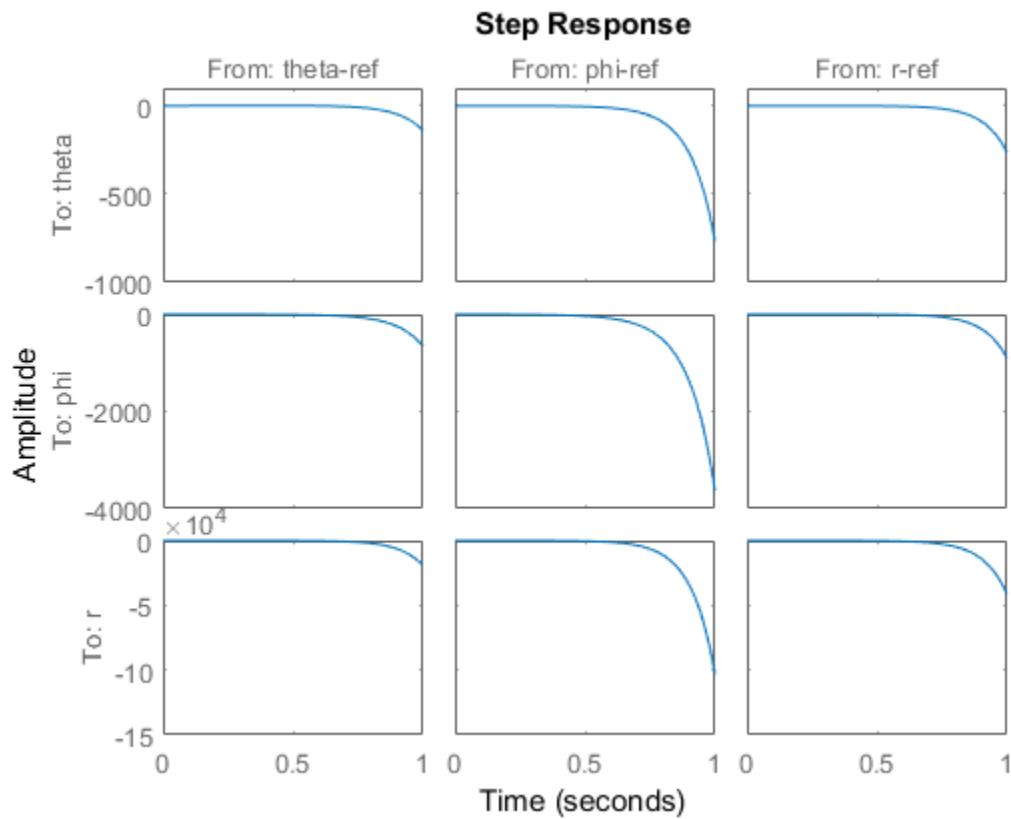
In the command window, click on any highlighted signal to see its location in the Simulink model.

In addition to specifying design requirements, you can use analysis points for extracting system responses. For example, extract and plot the step responses between the reference signals and `'theta'`, `'phi'`, and `'r'`.
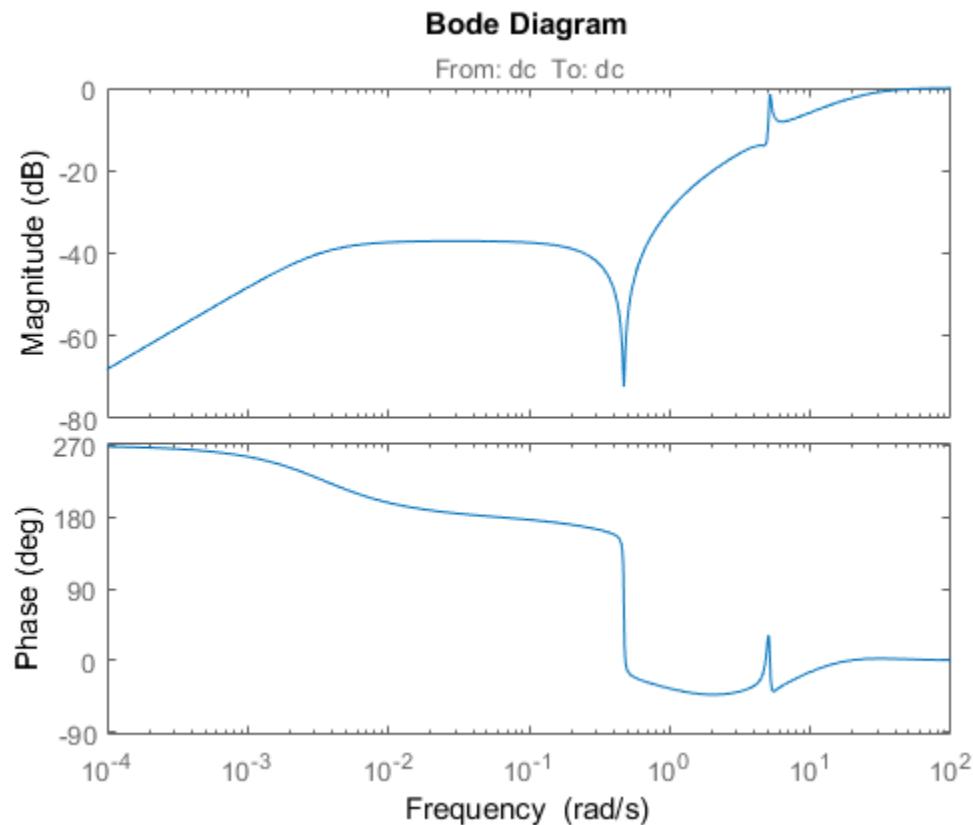
```
TO = getIOTransfer(STO,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
stepplot(TO,1)
```

Step Response

All the step responses are unstable, including the cross-couplings, because this model has not yet been tuned.

After you tune the model, you can similarly use the designated analysis points to extract system responses for validating the tuned system. If you want to examine system responses at locations that are not needed to specify design requirements, add these locations to the `slTuner` interface as well. For example, plot the sensitivity function measured at the output of the block `roll-off 2`.

```
addPoint(ST0,'dc')
dcSO = getSensitivity(ST0,'dc');
bodeplot(dcSO)
```

Suppose you want to change the parameterization of tunable blocks in the `slTuner` interface. For example, suppose that after tuning the model, you want to test whether changing from PI to PID controllers yields improved results. Change the parameterization of the three PI controllers to PID controllers.

```
PID0 = pid(0,0.001,0.001,.01);  % initial value for PID controllers
PID1 = tunablePID('C1',PID0);
PID2 = tunablePID('C2',PID0);
PID3 = tunablePID('C3',PID0);

setBlockParam(ST0,'PI1',PID1);
setBlockParam(ST0,'PI2',PID2);
setBlockParam(ST0,'PI3',PID3);
```

After you configure the `slTuner` interface to your Simulink model, you can create tuning goals and tune the model using `systune` or `looptune`.

See Also

`addBlock` | `addPoint` | `getIOTransfer` | `getSensitivity` | `setBlockParam` | `slTuner`

Related Examples

- "Multi-Loop Control of a Helicopter" on page 13-416
- "Control of a Linear Electric Actuator" on page 13-341

More About

- "Marking Signals of Interest for Control System Analysis and Design"
- "Tuning Goals"
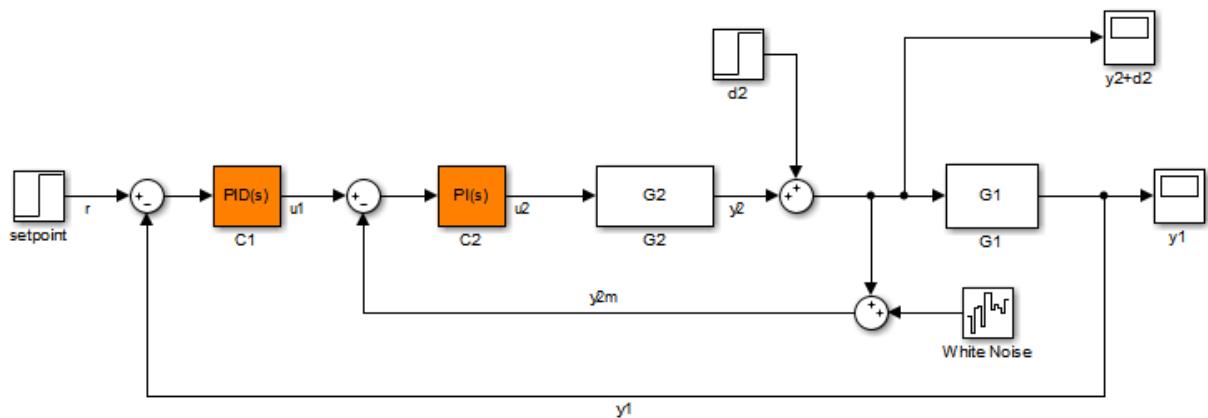
Tuning Multi-Loop Control Systems

This example shows how to jointly tune the inner and outer loops of a cascade architecture with the `systune` command.

Cascaded PID Loops

Cascade control is often used to achieve smooth tracking with fast disturbance rejection. The simplest cascade architecture involves two control loops (inner and outer) as shown in the block diagram below. The inner loop is typically faster than the outer loop to reject disturbances before they propagate to the outer loop.

```
open_system('rct_cascade')
```


Plant Models and Bandwidth Requirements

In this example, the inner loop plant G2 is

$$G_2(s) = \frac{3}{s+2}$$

and the outer loop plant G1 is

$$G_1(s) = \frac{10}{(s+1)^3}$$

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

We use a PI controller in the inner loop and a PID controller in the outer loop. The outer loop must have a bandwidth of at least 0.2 rad/s and the inner loop bandwidth should be ten times larger for adequate disturbance rejection.

Tuning the PID Controllers with SYSTUNE

When the control system is modeled in Simulink, use the `slTuner` interface in Simulink Control Design™ to set up the tuning task. List the tunable blocks, mark the signals `r` and `d2` as inputs of interest, and mark the signals `y1` and `y2` as locations where to measure open-loop transfers and specify loop shapes.

```
ST0 = slTuner('rct_cascade',{'C1','C2'});
addPoint(ST0,{'r','d2','y1','y2'})
```

You can query the current values of `C1` and `C2` in the Simulink model using `showTunable`. The control system is unstable for these initial values as confirmed by simulating the Simulink model.

```
showTunable(ST0)


Block 1: rct_cascade/C1 =

            1
  Kp + Ki * ---
            s

  with Kp = 0.1, Ki = 0.1

Name: C1
Continuous-time PI controller in parallel form.

---------------------------------

Block 2: rct_cascade/C2 =

            1
  Kp + Ki * ---
            s

  with Kp = 0.1, Ki = 0.1
```

```
Name: C2
Continuous-time PI controller in parallel form.
```

Next use "LoopShape" requirements to specify the desired bandwidths for the inner and outer loops. Use $0.2/s$ as target loop shape for the outer loop to enforce integral action with a gain crossover frequency at 0.2 rad/s:

```
% Outer loop bandwidth = 0.2
s = tf('s');
Req1 = TuningGoal.LoopShape('y1',0.2/s); % loop transfer measured at y1
Req1.Name = 'Outer Loop';
```

Use $2/s$ for the inner loop to make it ten times faster (higher bandwidth) than the outer loop. To constrain the inner loop transfer, make sure to open the outer loop by specifying y1 as a loop opening:

```
% Inner loop bandwidth = 2
Req2 = TuningGoal.LoopShape('y2',2/s); % loop transfer measured at y2
Req2.Openings = 'y1';                  % with outer loop opened at y1
Req2.Name = 'Inner Loop';
```

You can now tune the PID gains in C1 and C2 with `systune`:

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

```
Final: Soft = 0.859, Hard = -Inf, Iterations = 80
```

Use `showTunable` to see the tuned PID gains.

```
showTunable(ST)
```

```
Block 1: rct_cascade/C1 =

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

  with Kp = 0.0521, Ki = 0.0186, Kd = 0.0471, Tf = 0.00963

Name: C1
```

```
Continuous-time PIDF controller in parallel form.

---------------------------------

Block 2: rct_cascade/C2 =

              1
  Kp + Ki * ---
              s

  with Kp = 0.721, Ki = 1.23

Name: C2
Continuous-time PI controller in parallel form.
```
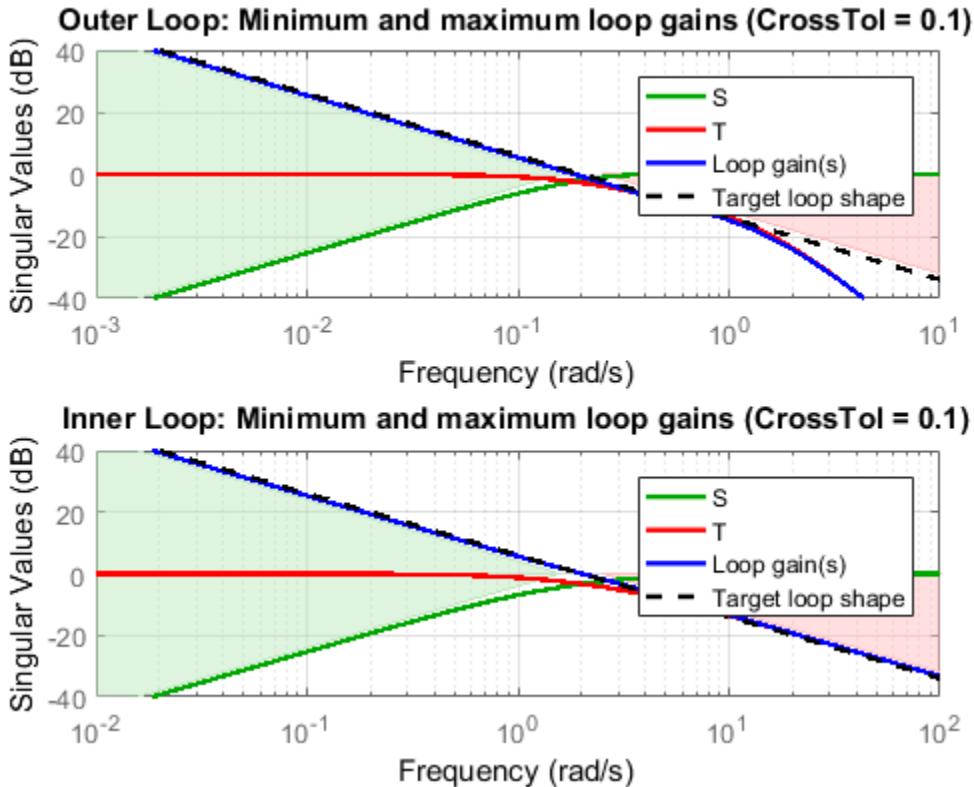
Validating the Design
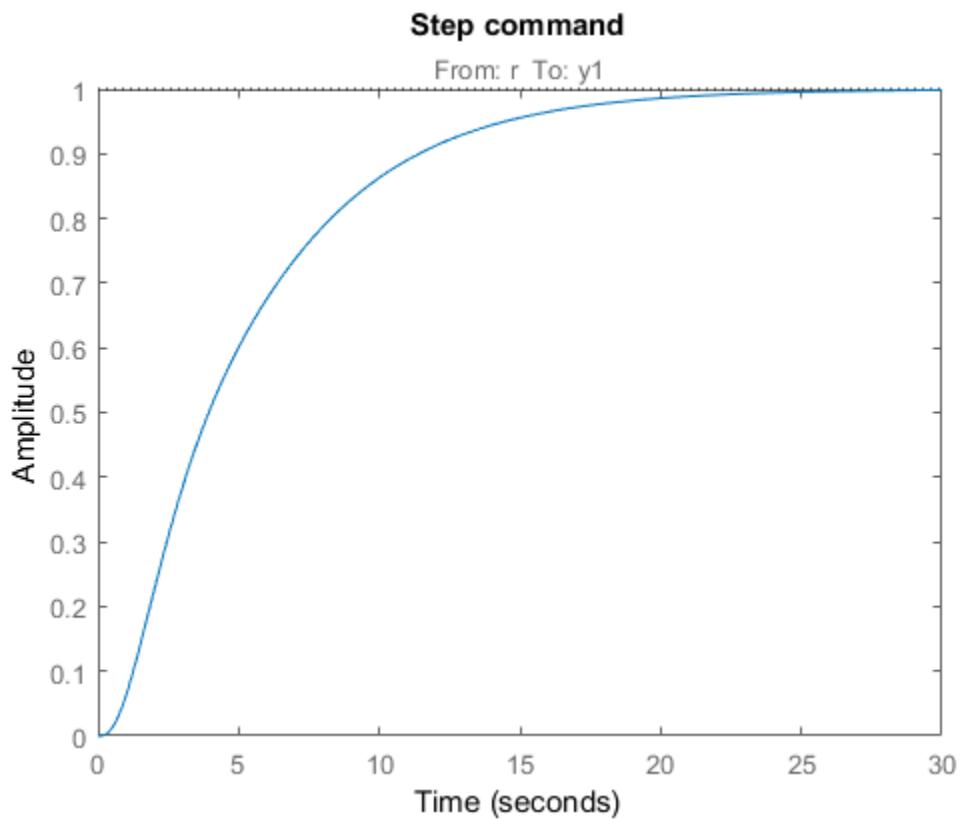
The final value is less than 1 which means that `systune` successfully met both loop shape requirements. Confirm this by inspecting the tuned control system `ST` with `viewSpec`

```
viewSpec([Req1,Req2],ST)
```

Outer Loop: Minimum and maximum loop gains (CrossTol = 0.1)

Inner Loop: Minimum and maximum loop gains (CrossTol = 0.1)
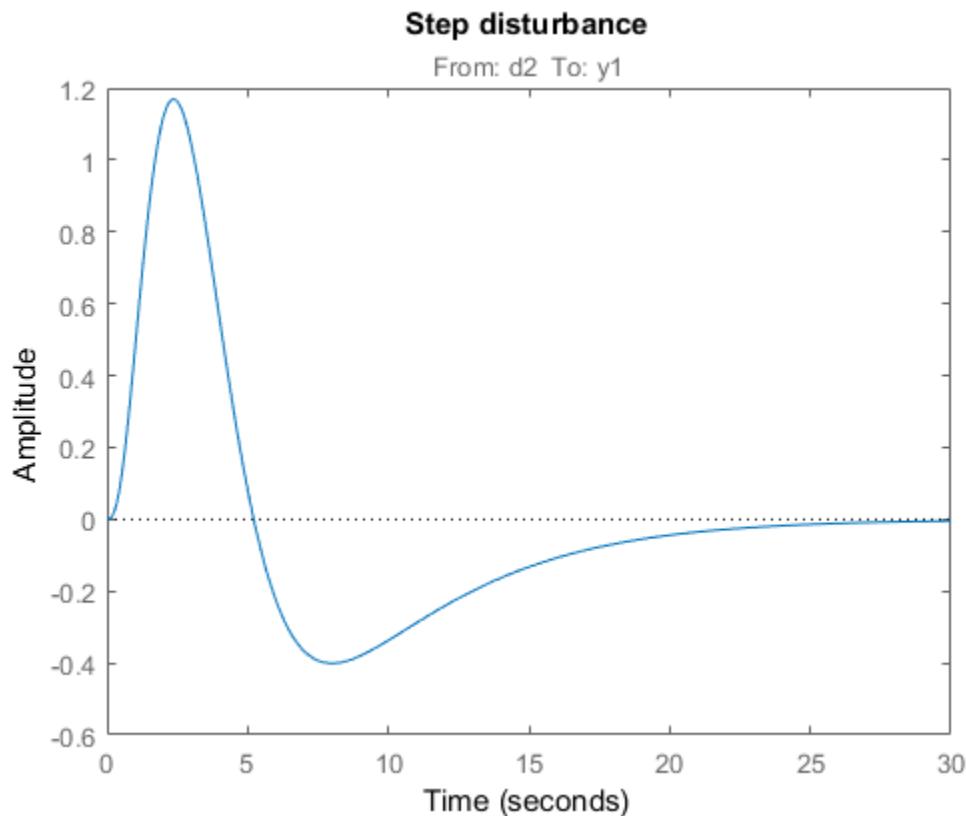
Note that the inner and outer loops have the desired gain crossover frequencies. To further validate the design, plot the tuned responses to a step command r and step disturbance d2:

```
% Response to a step command
H = getIOTransfer(ST,'r','y1');
clf, step(H,30), title('Step command')
```

Step command

From: r To: y1


```
% Response to a step disturbance
H = getIOTransfer(ST,'d2','y1');
step(H,30), title('Step disturbance')
```

Once you are satisfied with the linear analysis results, use `writeBlockValue` to write the tuned PID gains back to the Simulink blocks. You can then conduct a more thorough validation in Simulink.

```
writeBlockValue(ST)
```

Equivalent Workflow in MATLAB

If you do not have a Simulink model of the control system, you can perform the same steps using LTI models of the plant and Control Design blocks to model the tunable elements.

loop switches
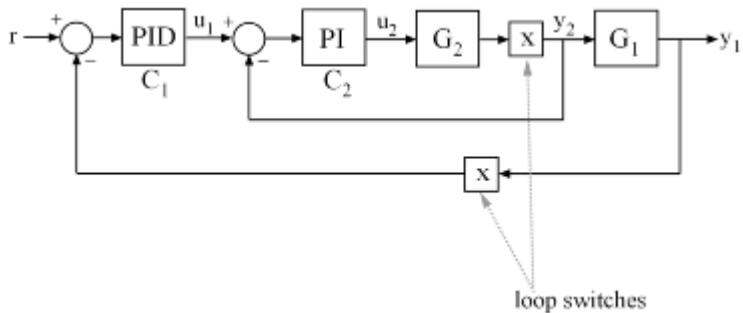
Figure 1: Cascade Architecture

First create parametric models of the tunable PI and PID controllers.

```
C1 = tunablePID('C1','pid');
C2 = tunablePID('C2','pi');
```

Then use "analysis point" blocks to mark the loop opening locations y1 and y2.

```
LS1 = AnalysisPoint('y1');
LS2 = AnalysisPoint('y2');
```

Finally, create a closed-loop model T0 of the overall control system by closing each feedback loop. The result is a generalized state-space model depending on the tunable elements C1 and C2.

```
InnerCL = feedback(LS2*G2*C2,1);
T0 = feedback(G1*InnerCL*C1,LS1);
T0.InputName = 'r';
T0.OutputName = 'y1';
```
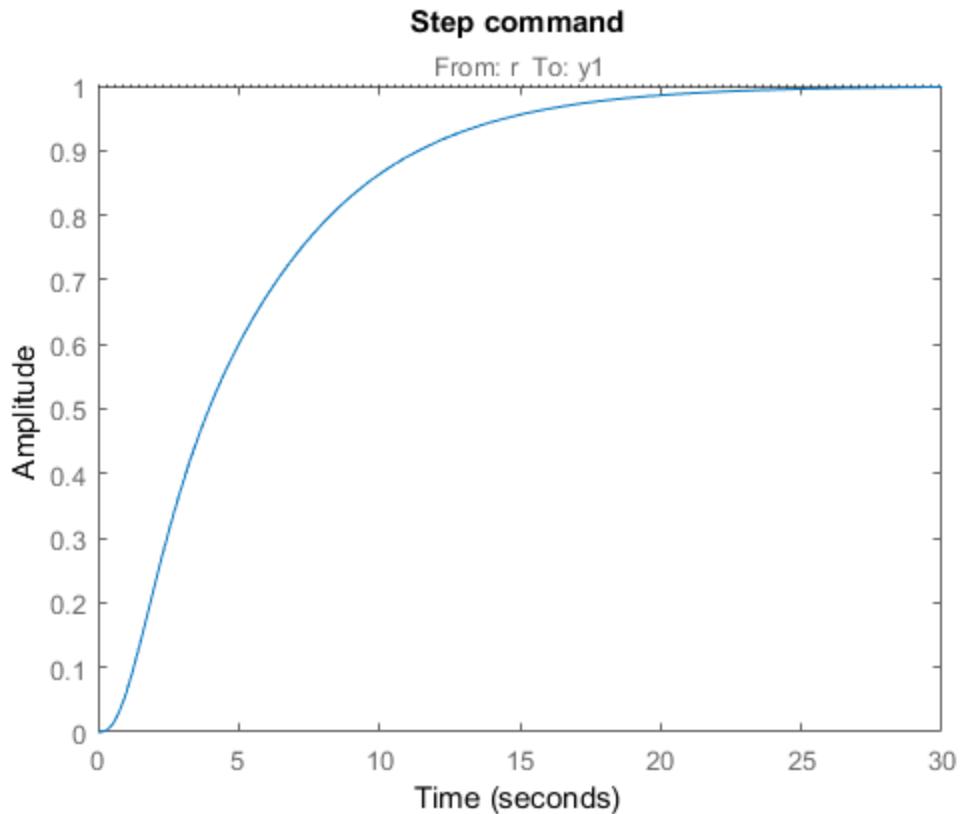
You can now tune the PID gains in C1 and C2 with systune.

```
T = systune(T0,[Req1,Req2]);
```

```
Final: Soft = 0.859, Hard = -Inf, Iterations = 80
```
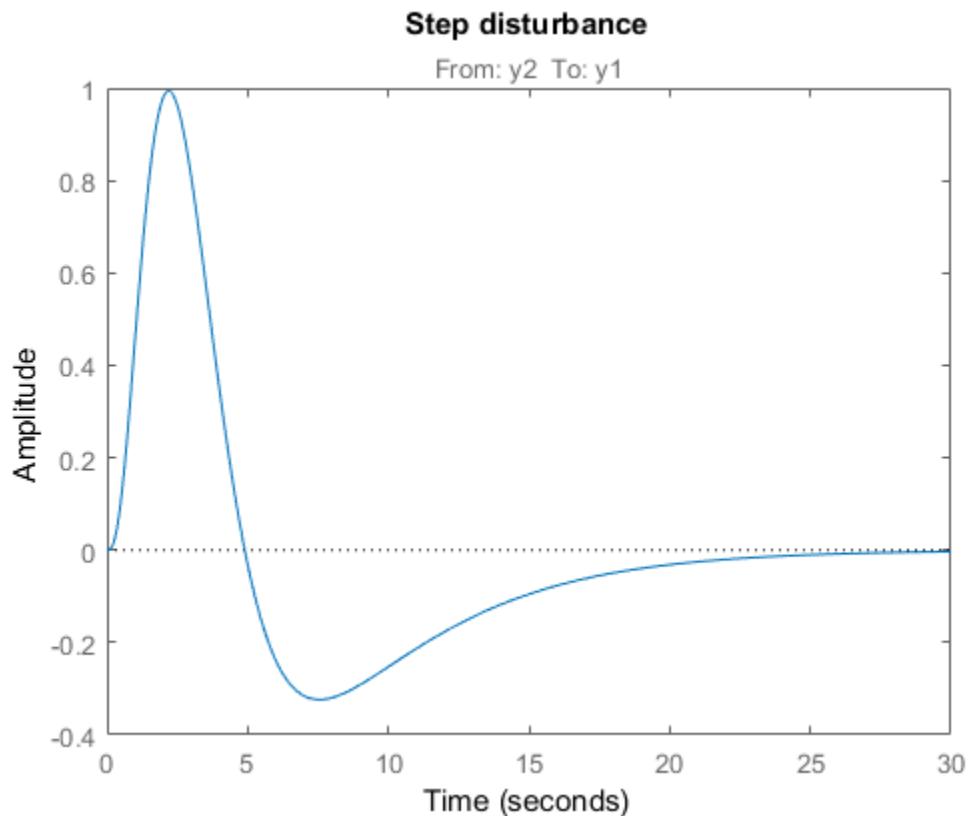
As before, use getIOTransfer to compute and plot the tuned responses to a step command r and step disturbance entering at the location y2:

```
% Response to a step command
```

```
H = getIOTransfer(T,'r','y1');
clf, step(H,30), title('Step command')
```

Step command
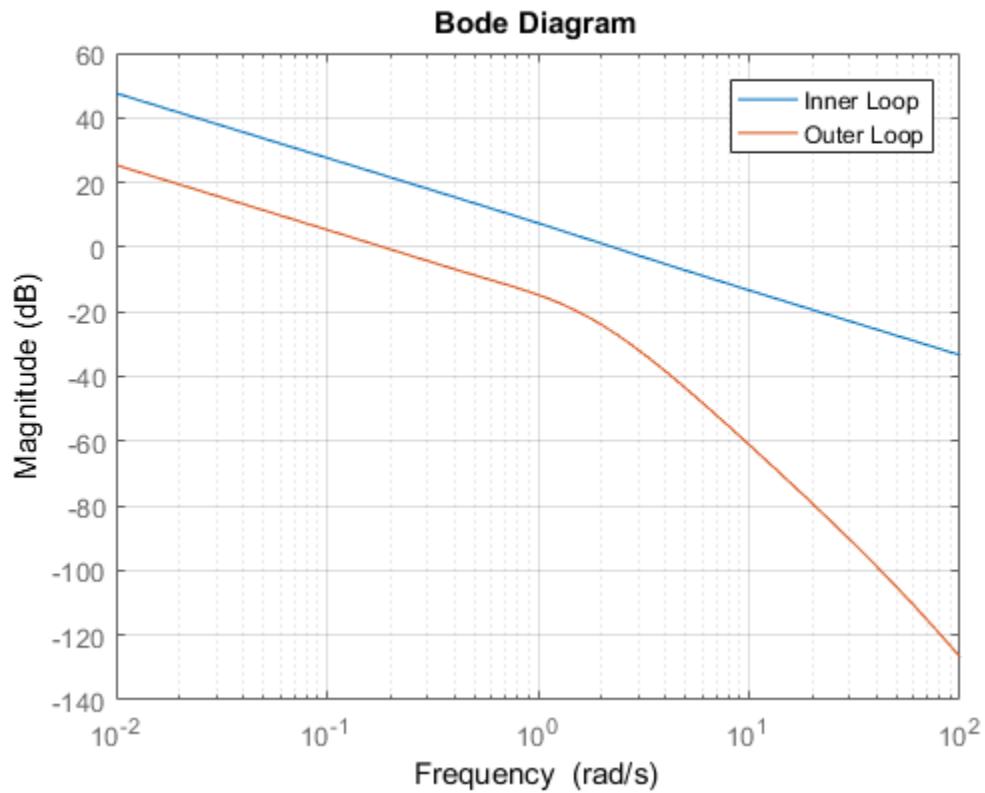

```
% Response to a step disturbance
H = getIOTransfer(T,'y2','y1');
step(H,30), title('Step disturbance')
```

Step disturbance

You can also plot the open-loop gains for the inner and outer loops to validate the bandwitdth requirements. Note the -1 sign to compute the negative-feedback open-loop transfer:

```
L1 = getLoopTransfer(T,'y1',-1);        % crossover should be at .2
L2 = getLoopTransfer(T,'y2',-1,'y1');   % crossover should be at 2
bodemag(L2,L1,{1e-2,1e2}), grid
legend('Inner Loop','Outer Loop')
```

PID Tuning for Setpoint Tracking vs. Disturbance Rejection

This example uses `systune` to explore trade-offs between setpoint tracking and disturbance rejection when tuning PID controllers.

PID Tuning Trade-Offs

When tuning 1-DOF PID controllers, it is often impossible to achieve good tracking and fast disturbance rejection at the same time. Assuming the control bandwidth is fixed, faster disturbance rejection requires more gain inside the bandwidth, which can only be achieved by increasing the slope near the crossover frequency. Because a larger slope means a smaller phase margin, this typically comes at the expense of more overshoot in the response to setpoint changes.

Figure 1: Trade-off in 1-DOF PID Tuning.

This example uses `systune` to explore this trade-off and find the right compromise for your application. See also `pidtool` for a more direct way to make such trade-off (see "Design Focus" under Controller Options).

Tuning Setup

Consider the PID loop of Figure 2 with a load disturbance at the plant input.

Figure 2: PID Control Loop.

For this example we use the plant model

$$G(s) = \frac{10(s+5)}{(s+1)(s+2)(s+10)}.$$

The target control bandwidth is 10 rad/s. Create a tunable PID controller and fix its derivative filter time constant to $T_f = 0.01$ (10 times the bandwidth) so that there are only three gains to tune (proportional, integral, and derivative gains).

```
G = zpk(-5,[-1 -2 -10],10);
C = tunablePID('C','pid');
C.Tf.Value = 0.01;  C.Tf.Free = false;  % fix Tf=0.01
```

Construct a tunable model TO of the closed-loop transfer from r to y. Use an "analysis point" block to mark the location u where the disturbance enters.

```
LS = AnalysisPoint('u');
TO = feedback(G*LS*C,1);
TO.u = 'r'; TO.y = 'y';
```

The gain of the open-loop response $L = GC$ is a key indicator of the feedback loop behavior. The open-loop gain should be high (greater than one) inside the control

bandwidth to ensure good disturbance rejection, and should be low (less than one) outside the control bandwidth to be insensitive to measurement noise and unmodeled plant dynamics. Accordingly, use three requirements to express the control objectives:

- "Tracking" requirement to specify a response time of about 0.2 seconds to step changes in r
- "MaxLoopGain" requirement to force a roll-off of -20 dB/decade past the crossover frequency 10 rad/s
- "MinLoopGain" requirement to adjust the integral gain at frequencies below 0.1 rad/s.

```
s = tf('s');
wc = 10; % target crossover frequency

% Tracking
R1 = TuningGoal.Tracking('r','y',2/wc);

% Bandwidth and roll-off
R2 = TuningGoal.MaxLoopGain('u',wc/s);

% Disturbance rejection
R3 = TuningGoal.MinLoopGain('u',wc/s);
R3.Focus = [0 0.1];
```

Tuning of 1-DOF PID Controller

Use `systune` to tune the PID gains to meet these requirements. Treat the bandwidth and disturbance rejection goals as hard constraints and optimize tracking subject to these constraints.

```
T1 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.12, Hard = 0.99976, Iterations = 144
```

Verify that all three requirements are nearly met. The blue curves are the achieved values and the yellow patches highlight regions where the requirements are violated.

```
figure('Position',[100,100,560,580])
viewSpec([R1 R2 R3],T1)
```

Tracking vs. Rejection

To gain insight into the trade-off between tracking and disturbance rejection, increase the minimum loop gain in the frequency band [0,0.1] rad/s by a factor α. Re-tune the PID gains for the values $\alpha = 2, 4$.

```
% Increase loop gain by factor 2
alpha = 2;
R3.MinGain = alpha*wc/s;
T2 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.17, Hard = 0.99973, Iterations = 117

% Increase loop gain by factor 4
alpha = 4;
R3.MinGain = alpha*wc/s;
T3 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.27, Hard = 0.9994, Iterations = 161
```

Compare the responses to a step command r and to a step disturbance d entering at the plant input u.

```
figure, step(T1,T2,T3,3)
title('Setpoint tracking')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```

Setpoint tracking

From: r To: y


```
% Compute closed-loop transfer from u to y
D1 = getIOTransfer(T1,'u','y');
D2 = getIOTransfer(T2,'u','y');
D3 = getIOTransfer(T3,'u','y');
step(D1,D2,D3,10)
title('Disturbance rejection')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```

Note how disturbance rejection improves as `alpha` increases, but at the expense of increased overshoot in setpoint tracking. Plot the open-loop responses for the three designs, and note how the slope before crossover (0dB) increases with `alpha`.

```matlab
L1 = getLoopTransfer(T1,'u');
L2 = getLoopTransfer(T2,'u');
L3 = getLoopTransfer(T3,'u');
bodemag(L1,L2,L3,{1e-2,1e2}), grid
title('Open-loop response')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```

Which design is most suitable depends on the primary purpose of the feedback loop you are tuning.

Tuning of 2-DOF PID Controller

If you cannot compromise tracking to improve disturbance rejection, consider using a 2-DOF architecture instead. A 2-DOF PID controller is capable of fast disturbance rejection without significant increase of overshoot in setpoint tracking.

Figure 3: 2-DOF PID Control Loop.

Use the `tunablePID2` object to parameterize the 2-DOF PID controller and construct a tunable model `T0` of the closed-loop system in Figure 3.

```
C = tunablePID2('C','pid');
C.Tf.Value = 0.01;  C.Tf.Free = false;  % fix Tf=0.01

T0 = feedback(G*LS*C,1,2,1,+1);
T0 = T0(:,1);
T0.u = 'r';  T0.y = 'y';
```

Next tune the 2-DOF PI controller for the largest loop gain tried earlier ($\alpha = 4$).

```
% Minimum loop gain inside bandwidth (for disturbance rejection)
alpha = 4;
R3.MinGain = alpha*wc/s;

% Tune 2-DOF PI controller
T4 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.09, Hard = 0.99942, Iterations = 157
```

Compare the setpoint tracking and disturbance rejection properties of the 1-DOF and 2-DOF designs for $\alpha = 4$.

```
clf, step(T3,'b',T4,'g--',4)
title('Setpoint tracking')
legend('1-DOF','2-DOF')
```

```
D4 = getIOTransfer(T4,'u','y');
step(D3,'b',D4,'g--',4)
title('Disturbance rejection')
legend('1-DOF','2-DOF')
```

The responses to a step disturbance are similar but the 2-DOF controller eliminates the overshoot in the response to a setpoint change. You can use `showTunable` to compare the tuned gains in the 1-DOF and 2-DOF controllers.

```
showTunable(T3)   % 1-DOF PI

C =

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

  with Kp = 9.64, Ki = 16.1, Kd = 0.885, Tf = 0.01
```

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

```
showTunable(T4)    % 2-DOF PI
```

```
C =

                       1                    s
   u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                       s                Tf*s+1

   with Kp = 6.75, Ki = 19.7, Kd = 0.971, Tf = 0.01, b = 0.74, c = 1.13

Name: C
Continuous-time 2-DOF PIDF controller in parallel form.
```

Time-Domain Specifications

This example gives a tour of available time-domain requirements for control system tuning with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

Step Command Following

The `TuningGoal.StepTracking` requirement specifies how the tuned closed-loop system should respond to a step input. You can specify the desired response either in terms of first- or second-order characteristics, or as an explicit reference model. This requirement is satisfied when the relative gap between the actual and desired responses is small enough in the least-squares sense. For example,
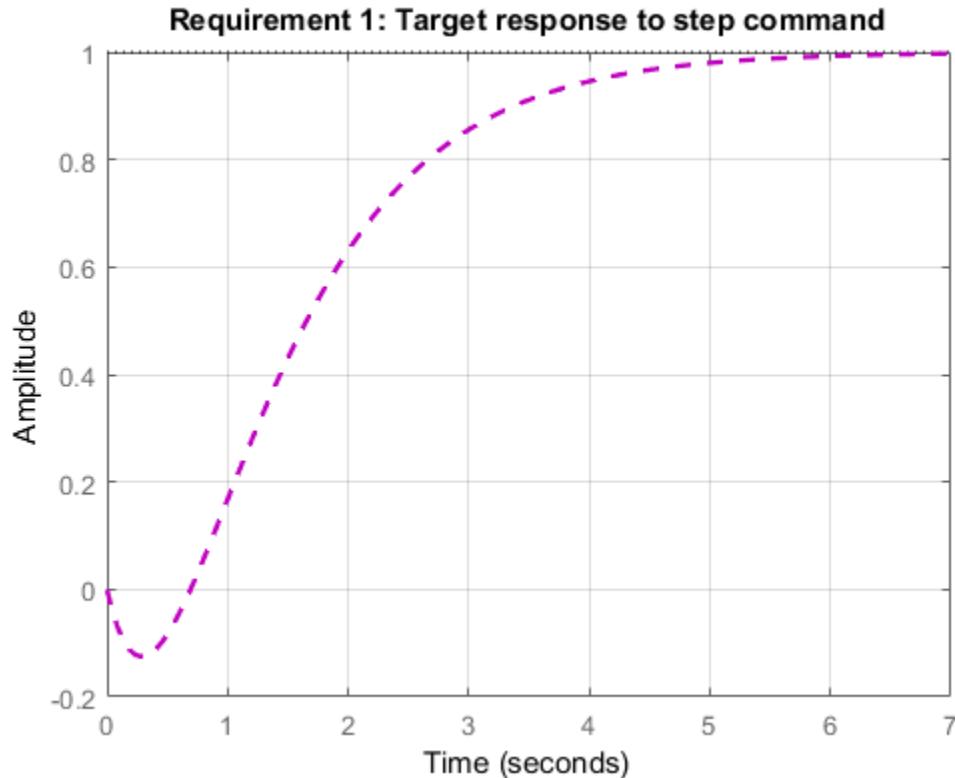
```
R1 = TuningGoal.StepTracking('r','y',0.5);
```

stipulates that the closed-loop response from `r` to `y` should behave like a first-order system with time constant 0.5, while

```
R2 = TuningGoal.StepTracking('r','y',zpk(2,[-1 -2],-1));
```

specifies a second-order, non-minimum-phase behavior. Use `viewSpec` to visualize the desired response.

```
viewSpec(R2)
```

Requirement 1: Target response to step command

This requirement can be used to tune both SISO and MIMO step responses. In the MIMO case, the requirement ensures that each output tracks the corresponding input with minimum cross-couplings.
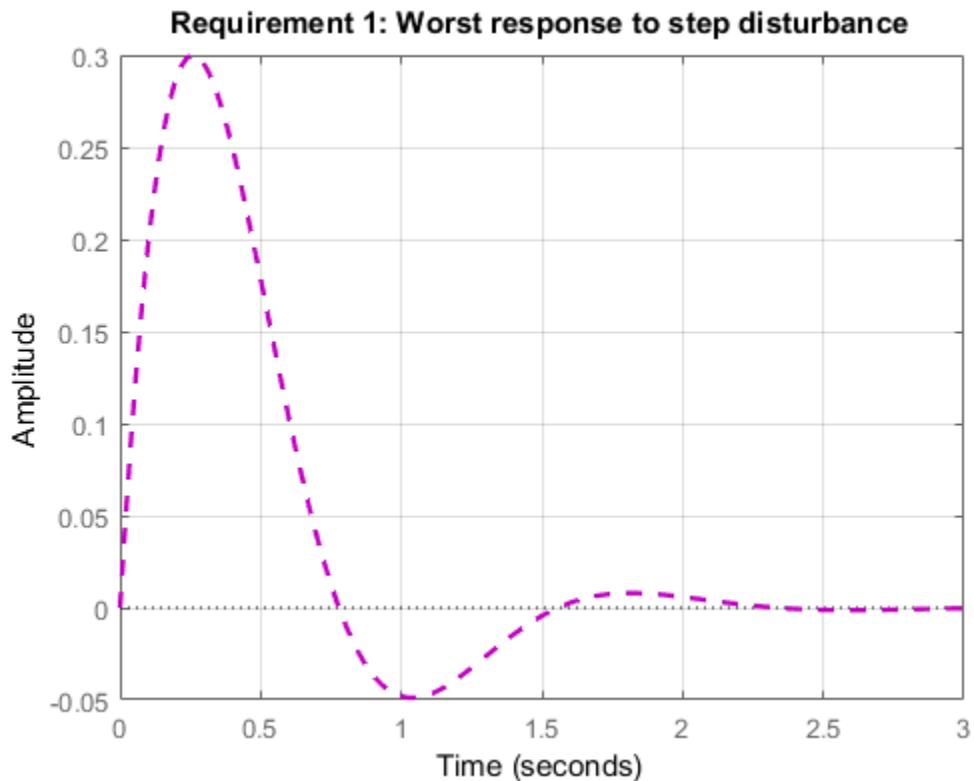
Step Disturbance Rejection

The `TuningGoal.StepRejection` requirement specifies how the tuned closed-loop system should respond to a step disturbance. You can specify worst-case values for the response amplitude, settling time, and damping of oscillations. For example,

```
R1 = TuningGoal.StepRejection('d','y',0.3,2,0.5);
```

limits the amplitude of $y(t)$ to 0.3, the settling time to 2 time units, and the damping ratio to a minimum of 0.5. Use `viewSpec` to see the corresponding time response.

```
viewSpec(R1)
```


Requirement 1: Worst response to step disturbance

You can also use a "reference model" to specify the desired response. Note that the actual and specified responses may differ substantially when better disturbance rejection is possible. Use the `TuningGoal.Transient` requirement when a close match is desired. For best results, adjust the gain of the reference model so that the actual and specified responses have similar peak amplitudes (see `TuningGoal.StepRejection` documentation for details).
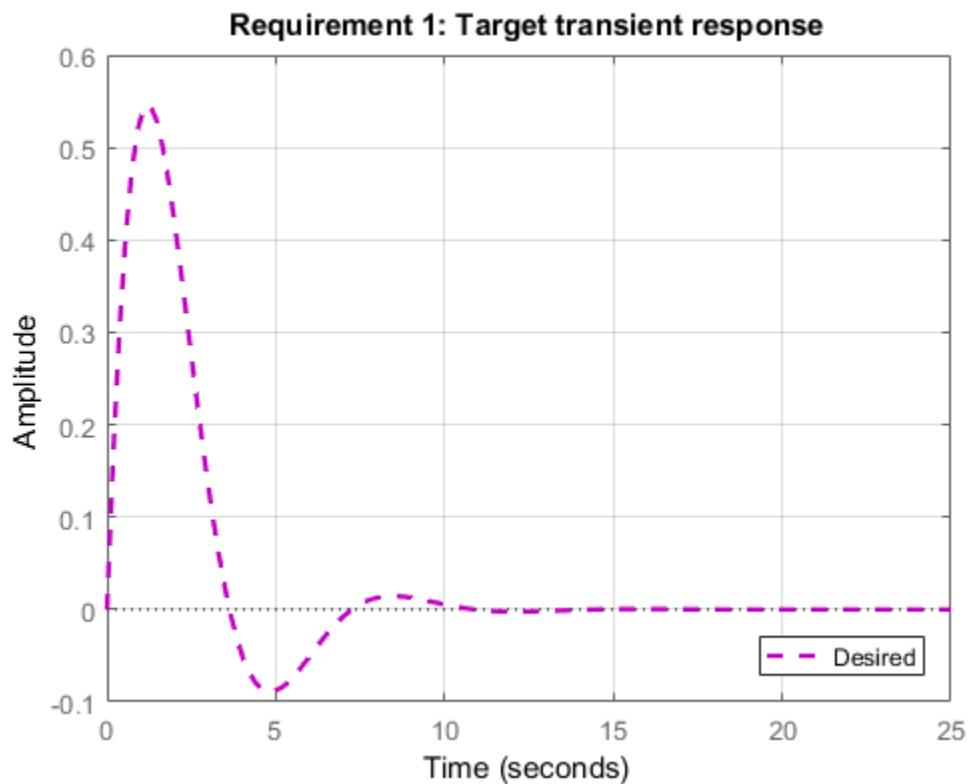
Transient Response Matching

The `TuningGoal.Transient` requirement specifies the transient response for a specific input signal. This is a generalization of the `TuningGoal.StepTracking` requirement. For example,

```
R1 = TuningGoal.Transient('r','y',tf(1,[1 1 1]),'impulse');
```

requires that the tuned response from r to y look like the impulse response of the reference model $1/(s^2 + s + 1)$.
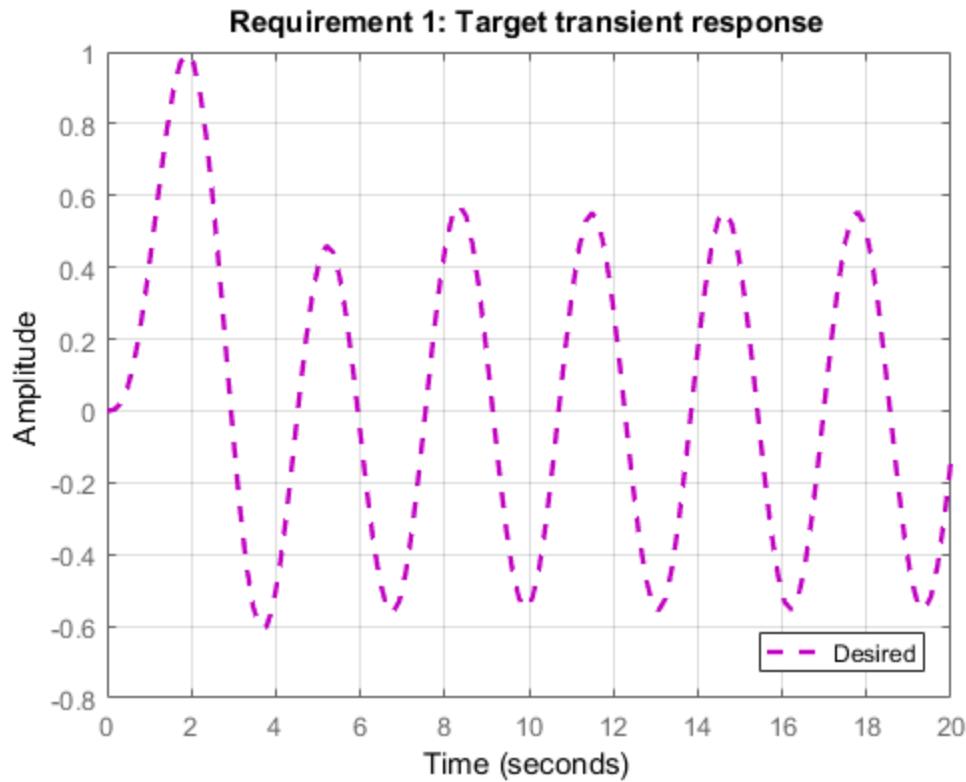
```
viewSpec(R1)
```


The input signal can be an impulse, a step, a ramp, or a more general signal modeled as the impulse response of some input shaping filter. For example, a sine wave with frequency ω_0 can be modeled as the impulse response of $\omega_0^2/(s^2 + \omega_0^2)$.

```
w0 = 2;
```

13-215

```
F = tf(wO^2,[1 O wO^2]);   % input shaping filter
R2 = TuningGoal.Transient('r','y',tf(1,[1 1 1]),F);
viewSpec(R2)
```


LQG Design

Use the `TuningGoal.LQG` requirement to create a linear-quadratic-Gaussian objective for tuning the control system parameters. This objective is applicable to any control structure, not just the classical observer structure of LQG control. For example, consider the simple PID loop of Figure 2 where d and n are unit-variance disturbance and noise inputs, and S_d and S_n are lowpass and highpass filters that model the disturbance and noise spectral contents.
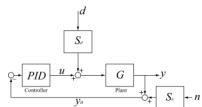
Figure 2: Regulation loop.

To regulate y around zero, you can use the following LQG criterion:

$$J = lim_{T \to \infty} E \left(\frac{1}{T} \int_0^T (y^2(t) + 0.05u^2) dt \right)$$

The first term in the integral penalizes the deviation of $y(t)$ from zero, and the second term penalizes the control effort. Using `systune`, you can tune the PID controller to minimize the cost J. To do this, use the LQG requirement

```
Qyu = diag([1 0.05]);  % weighting of y^2 and u^2
R4 = TuningGoal.LQG({'d','n'},{'y','u'},1,Qyu);
```

Frequency-Domain Specifications

This example gives a tour of available frequency-domain requirements for control system tuning with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.
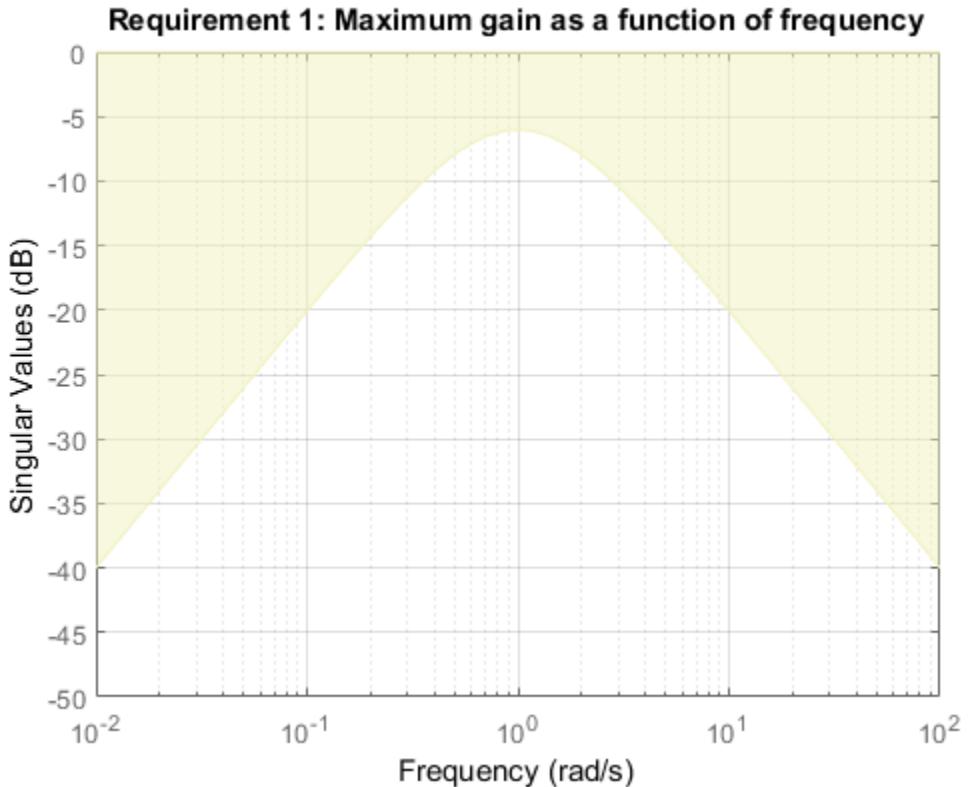
Gain Limit

The `TuningGoal.Gain` requirement enforces gain limits on SISO or MIMO closed-loop transfer functions. This requirement is useful to enforce adequate disturbance rejection and roll off, limit sensitivity and control effort, and prevent saturation. For MIMO transfer functions, "gain" refers to the largest singular value of the frequency response matrix. The gain limit can be frequency dependent. For example

```
s = tf('s');
R1 = TuningGoal.Gain('d','y',s/(s+1)^2);
```

specifies that the gain from d to y should not exceed the magnitude of the transfer function $s/(s+1)^2$.

```
viewSpec(R1)
```

Requirement 1: Maximum gain as a function of frequency
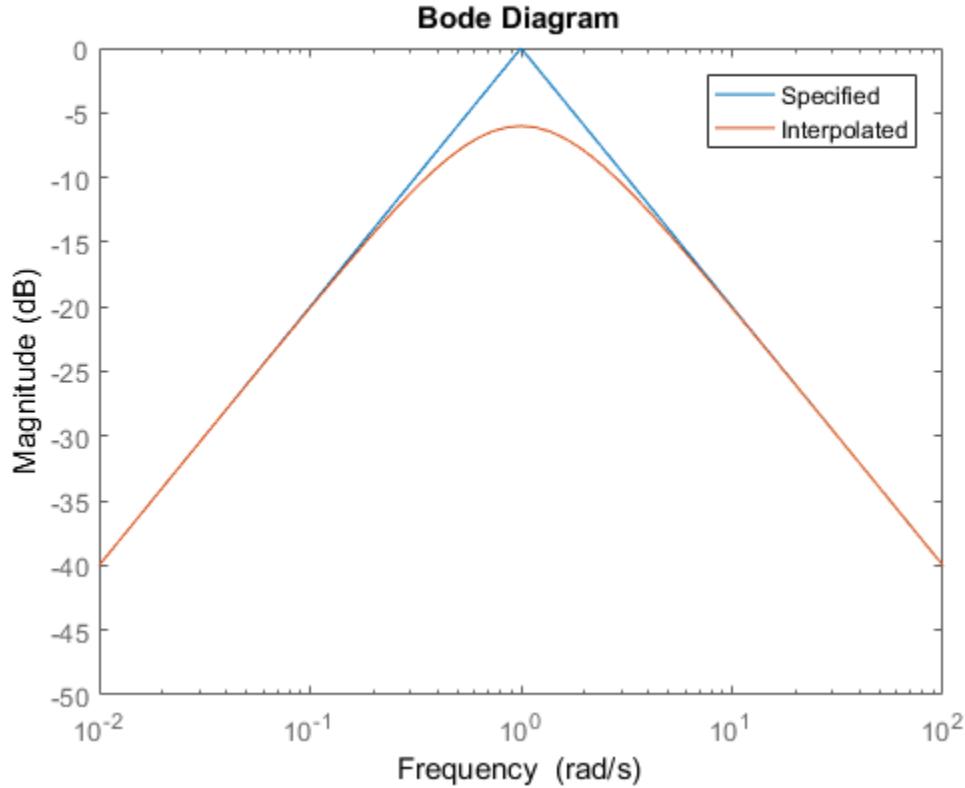
It is often convenient to just sketch the asymptotes of the desired gain profile. For example, instead of the transfer function $s/(s+1)^2$, we could just specify gain values of 0.01,1,0.01 at the frequencies 0.01,1,100, the point (1,1) being the breakpoint of the two asymptotes s and $1/s$.

```
Asymptotes = frd([0.01,1,0.01],[0.01,1,100]);
R2 = TuningGoal.Gain('d','y',Asymptotes);
```

The requirement object automatically turns this discrete gain profile into a gain limit defined at all frequencies.

```
bodemag(Asymptotes,R2.MaxGain)
legend('Specified','Interpolated')
```

Variance Amplification

The `TuningGoal.Variance` requirement limits the noise variance amplification from specified inputs to specified outputs. In technical terms, this requirement constrains the H_2 norm of a closed-loop transfer function. This requirement is preferable to `TuningGoal.Gain` when the input signals are random processes and the average gain matters more than the peak gain. For example,

```
R = TuningGoal.Variance('n','y',0.1);
```

limits the output variance of y to 0.1^2 for a unit-variance white-noise input n.

Reference Tracking and Overshoot Reduction

The `TuningGoal.Tracking` requirement enforces reference tracking and loop decoupling objectives in the frequency domain. For example
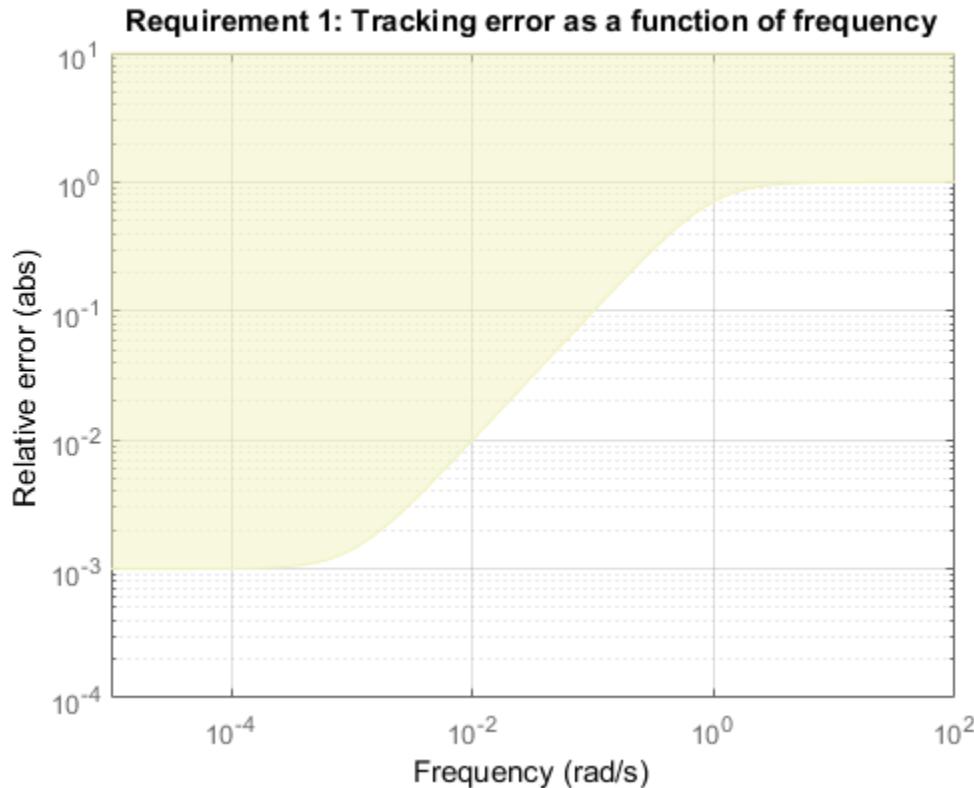
```
R1 = TuningGoal.Tracking('r','y',2);
```

specifies that the output y should track the reference r with a two-second response time. Similarly

```
R2 = TuningGoal.Tracking({'Vsp','wsp'},{'V','w'},2);
```

specifies that V should track Vsp and w should track wsp with minimum cross-coupling between the two responses. Tracking requirements are converted into frequency-domain constraints on the tracking error as a function of frequency. For the first requirement R1, for example, the gain from r to the tracking error e = r-y should be small at low frequency and approach 1 (100%) at frequencies greater than 1 rad/s (bandwidth for a two-second response time). You can use viewSpec to visualize this frequency-domain constraint. Note that the yellow region indicates where the requirement is violated.

```
viewSpec(R1)
```

If the response has excessive overshoot, use the `TuningGoal.Overshoot` requirement in conjunction with the `TuningGoal.Tracking` requirement. For example, you can limit the overshoot from r to y to 10% using

```
R3 = TuningGoal.Overshoot('r','y',10);
```

Disturbance Rejection

In feedback loops such as the one shown in Figure 1, the open- and closed-loop responses from disturbance d to output y are related by

$$G_{CL}(s) = \frac{G_{OL}(s)}{1 + L(s)}$$

where $L(s)$ is the loop transfer function measured at the disturbance entry point. The gain of $1 + L$ is the disturbance attenuation factor, the ratio between the open- and closed-loop sensitivities to the disturbance. Its reciprocal $S = 1/(1 + L)$ is the sensitivity at the disturbance input.
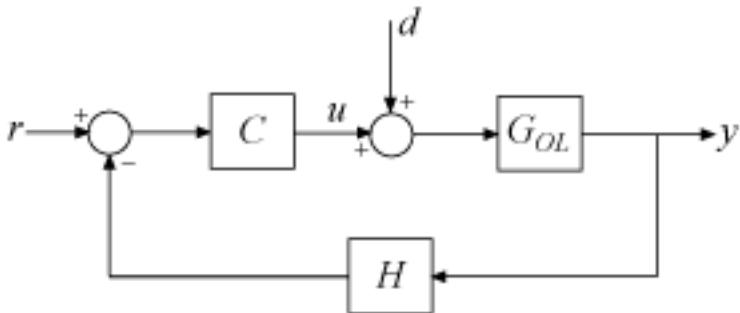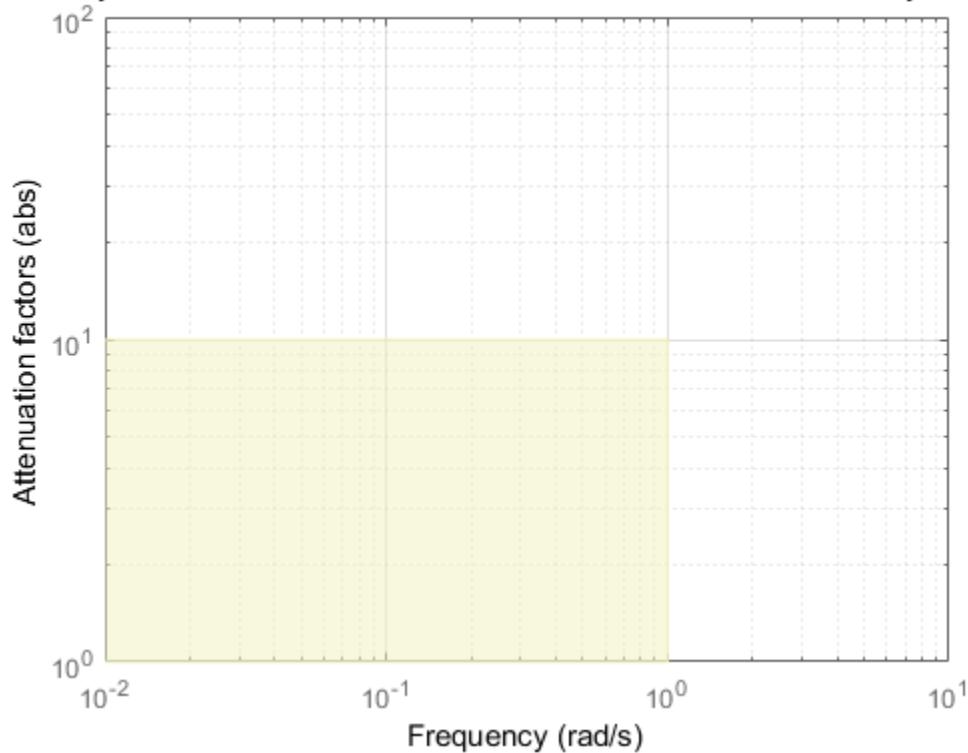
Figure 1: Sample feedback loop.

The `TuningGoal.Rejection` requirement specifies the disturbance attenuation as a function of frequency. The attenuation factor is greater than one inside the control bandwidth since feedback control reduces the impact of disturbances. As a rule of thumb, a 10-times-larger attenuation requires a 10-times-larger loop gain. For example

```
R1 = TuningGoal.Rejection('u',10);
R1.Focus = [0 1];
```

specifies that a disturbance entering at the plant input "u" should be attenuated by a factor 10 in the frequency band from 0 to 1 rad/s.

```
viewSpec(R1)
```
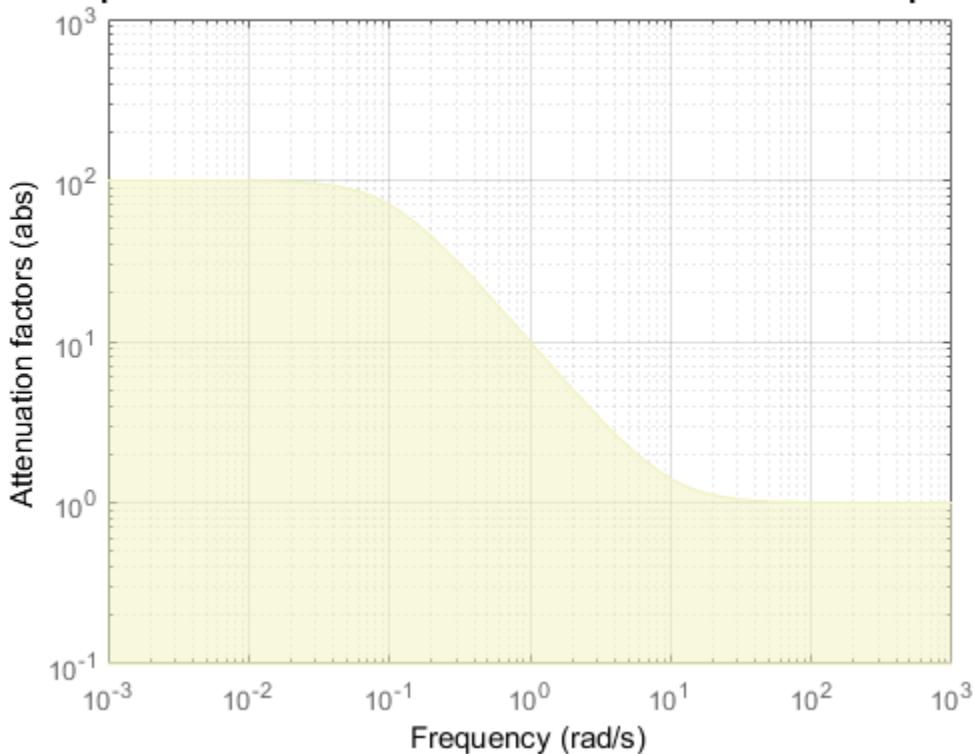
Requirement 1: Disturbance attenuation as a function of frequency

More generally, you can specify a frequency-dependent attenuation profile, for example

```
s = tf('s');
R2 = TuningGoal.Rejection('u',(s+10)/(s+0.1));
```

specifies an attenuation factor of 100 below 0.1 rad/s gradually decreasing to 1 (no attenuation) after 10 rad/s.

```
viewSpec(R2)
```
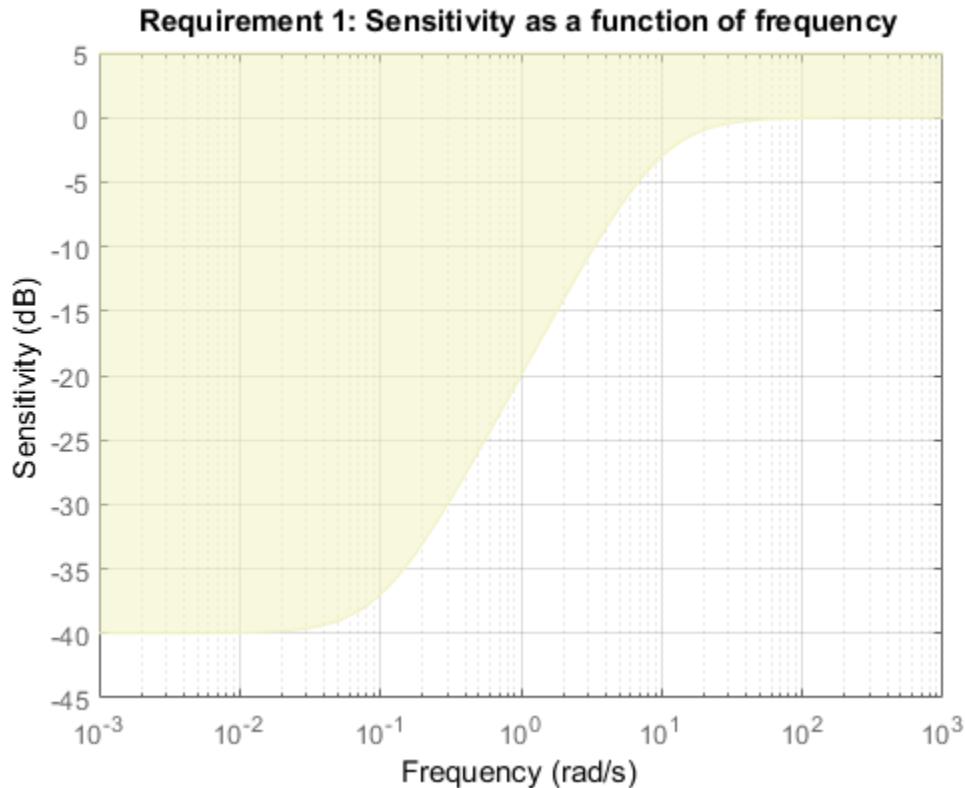
Requirement 1: Disturbance attenuation as a function of frequency

Instead of specifying the minimum attenuation, you can use the `TuningGoal.Sensitivity` requirement to specify the maximum sensitivity, that is, the maximum gain of $S = 1/(1 + L)$. For example,

```
R3 = TuningGoal.Sensitivity('u',(s+0.1)/(s+10));
```

is equivalent to the rejection requirement R2 above. The sensitivity increases from 0.01 (1%) below 0.1 rad/s to 1 (100%) above 10 rad/s.

```
viewSpec(R3)
```

Requirement 1: Sensitivity as a function of frequency

Frequency-Weighted Gain and Variance

The `TuningGoal.WeightedGain` and `TuningGoal.WeightedVariance` requirements are generalizations of the `TuningGoal.Gain` and `TuningGoal.Variance` requirements. These requirements constrain the H_∞ or H_2 norm of a frequency-weighted closed-loop transfer function $W_L(s)T(s)W_R(s)$, where $W_L(s)$ and $W_R(s)$ are user-defined weighting functions. For example

```
WL = blkdiag(1/(s+0.001),s/(0.001*s+1));
WR = [];
R = TuningGoal.WeightedGain('r',{'e','y'},WL,[]);
```
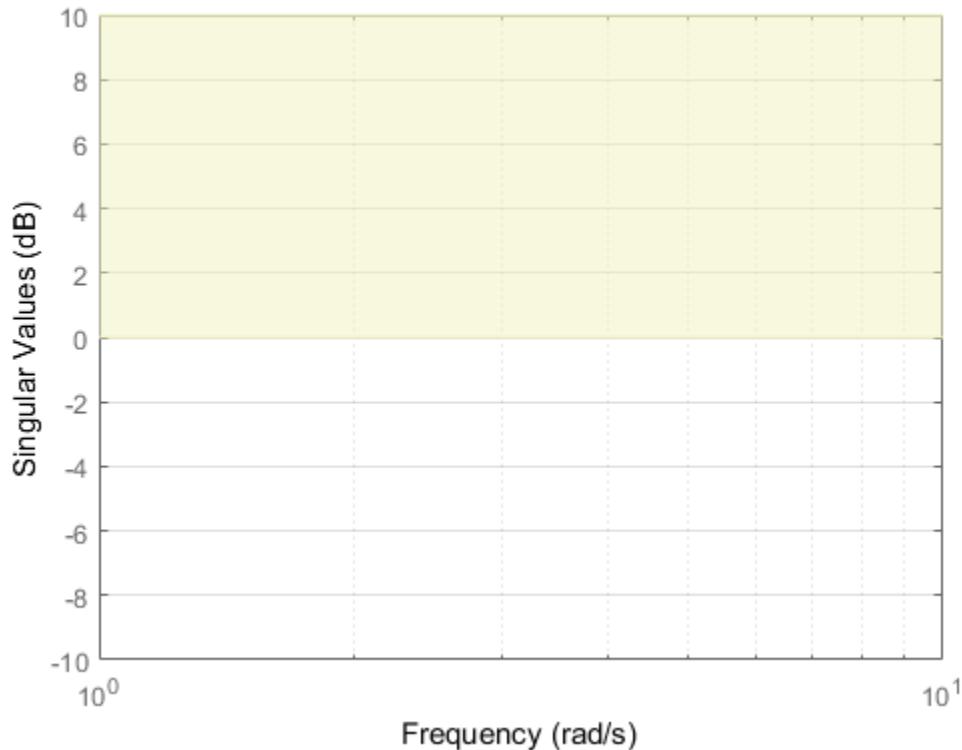
specifies the constraint

$$\left\| \begin{pmatrix} \frac{1}{s+0.001} T_{re} \\ \frac{s}{0.001s+1} T_{ry} \end{pmatrix} \right\|_{\infty} < 1$$

Note that this is a normalized gain constraint (unit bound across frequency).

```
viewSpec(R)
```

Requirement 1: Gain of W$_L$ H W$_R$ as a function of frequency

Loop Shape and Stability Margin Specifications

This example shows how to specify loop shapes and stability margins when tuning control systems with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

Loop Shape

The `TuningGoal.LoopShape` requirement is used to shape the open-loop response gain(s), a design approach known as *loop shaping*. For example,
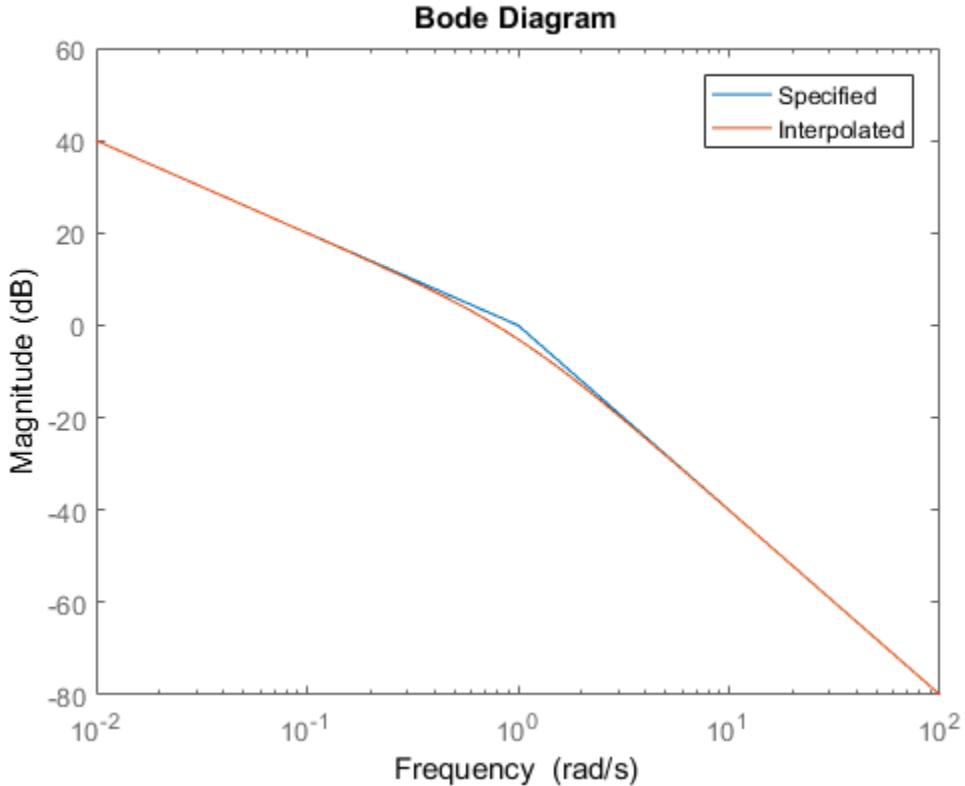
```
s = tf('s');
R1 = TuningGoal.LoopShape('u',1/s);
```

specifies that the open-loop response measured at the location "u" should look like a pure integrator (as far as its gain is concerned). In MATLAB, use an `AnalysisPoint` block to mark the location "u", see the *"Building Tunable Models"* example for details. In Simulink, use the `addPoint` method of the `slTuner` interface to mark "u" as a point of interest.

As with other gain specifications, you can just specify the asymptotes of the desired loop shape using a few frequency points. For example, to specify a loop shape with gain crossover at 1 rad/s, -20 dB/decade slope before 1 rad/s, and -40 dB/decade slope after 1 rad/s, just specify that the gain at the frequencies 0.1,1,10 should be 10,1,0.01, respectively.
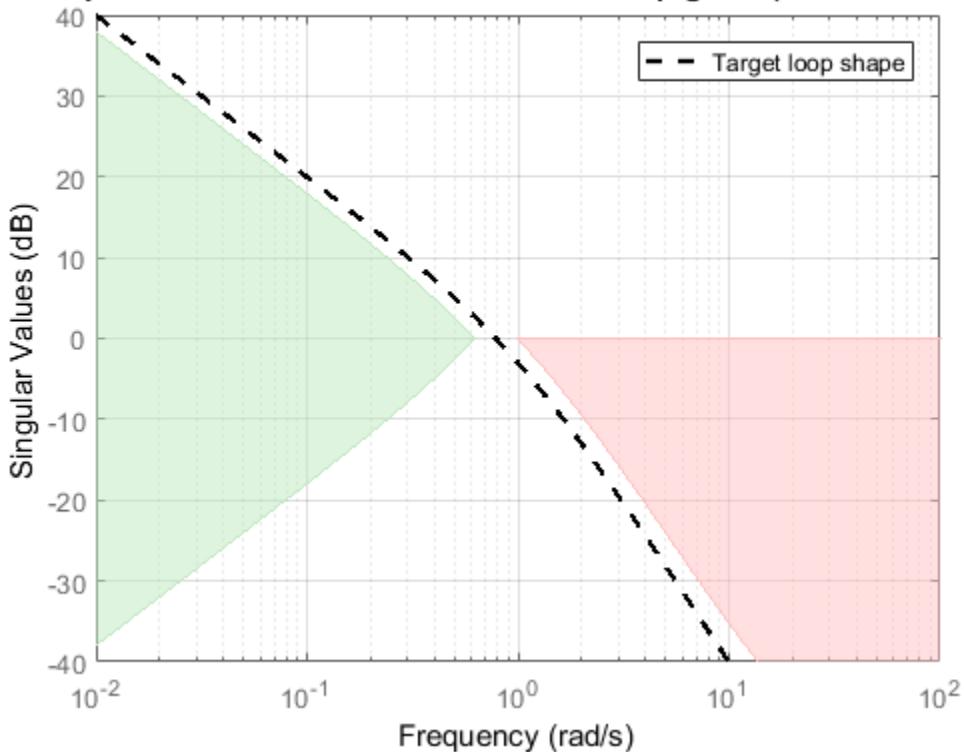
```
LS = frd([10,1,0.01],[0.1,1,10]);
R2 = TuningGoal.LoopShape('u',LS);

bodemag(LS,R2.LoopGain)
legend('Specified','Interpolated')
```

Loop shape requirements are constraints on the open-loop response L. For tuning purposes, they are converted into closed-loop gain constraints on the sensitivity function $S = 1/(1 + L)$ and complementary sensitivity function $T = L/(1 + L)$. Use `viewSpec` to visualize the target loop shape and corresponding gain bounds on S (green) and T (red).

```
viewSpec(R2)
```

Requirement 1: Minimum and maximum loop gains (CrossTol = 0.1)

Minimum and Maximum Loop Gain

Instead of `TuningGoal.LoopShape`, you can use `TuningGoal.MinLoopGain` and `TuningGoal.MaxLoopGain` to specify minimum or maximum values for the loop gain in a particular frequency band. This is useful when the actual loop shape near crossover is best left to the tuning algorithm to figure out. For example, the following requirements specify the minimum loop gain inside the bandwidth and the roll-off characteristics outside the bandwidth, but do not specify the actual crossover frequency nor the loop shape near crossover.
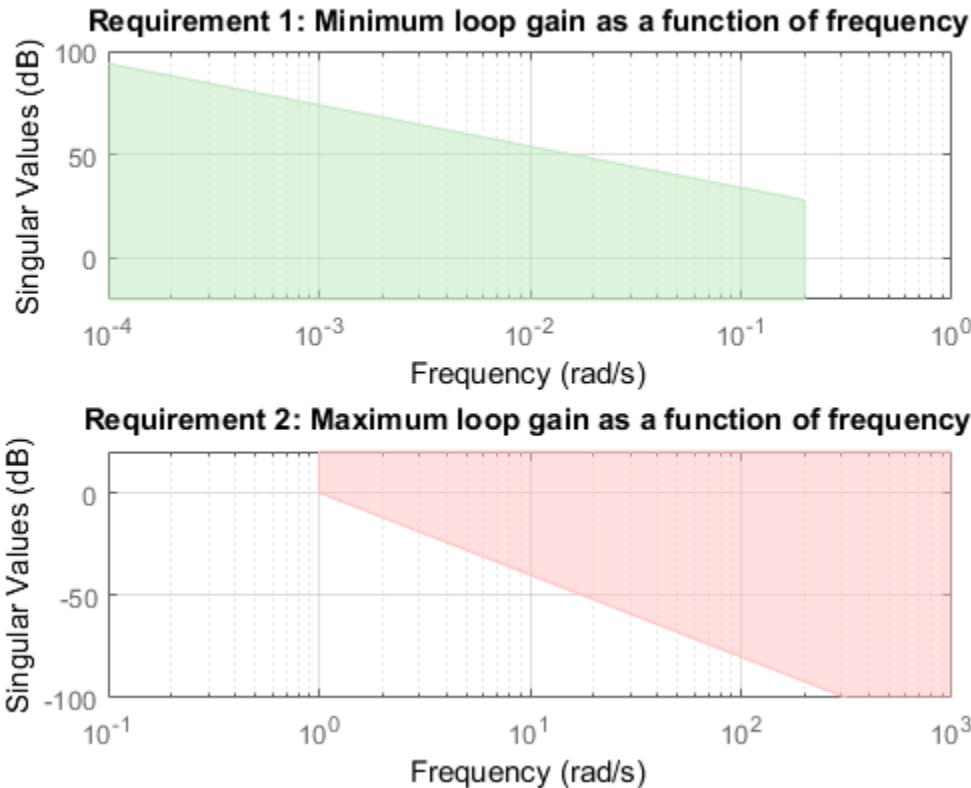
```
MinLG = TuningGoal.MinLoopGain('u',5/s);  % integral action
MinLG.Focus = [0 0.2];
```

```
MaxLG = TuningGoal.MaxLoopGain('u',1/s^2);  % -40dB/decade roll off
MaxLG.Focus = [1 Inf];

viewSpec([MinLG MaxLG])
```

Requirement 1: Minimum loop gain as a function of frequency

Requirement 2: Maximum loop gain as a function of frequency

The `TuningGoal.MaxLoopGain` requirement rests on the fact that the open- and closed-loop gains are comparable when the loop gain is small ($|L| \ll 1$). As a result, it can be ineffective at keeping the loop gain below some value close to 1. For example, suppose that flexible modes cause gain spikes beyond the crossover frequency and that you need to keep these spikes below 0.5 (-6 dB). Instead of using `TuningGoal.MaxLoopGain`, you can directly constrain the gain of L using `TuningGoal.Gain` with a loop opening at "u".

```
MaxLG = TuningGoal.Gain('u','u',0.5);
```

```
MaxLG.Opening = 'u';
```

If the open-loop response is unstable, make sure to further disable the implicit stability constraint associated with this requirement.

```
MaxLG.Stabilize = false;
```

Figure 1 shows this requirement evaluated for an open-loop response with flexible modes.

Figure 1: Gain constraint on L.

Stability Margins
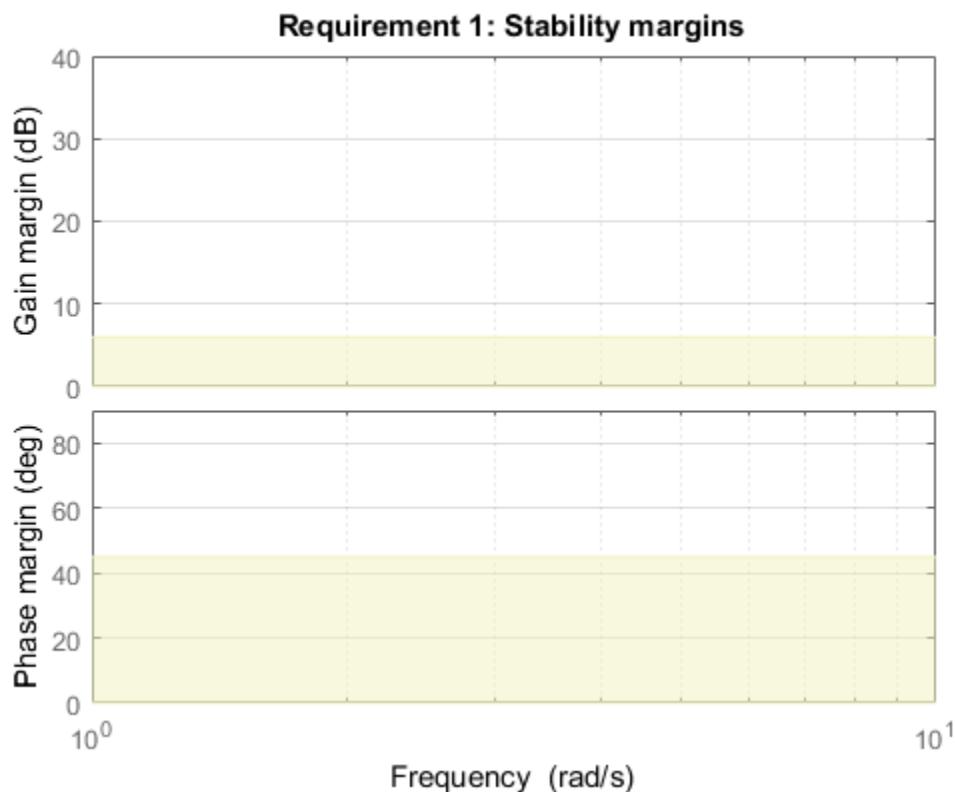
The `TuningGoal.Margins` requirement enforces minimum amounts of gain and phase margins at the specified loop opening site(s). For MIMO feedback loops, this requirement uses the notion of *disk margins*, which guarantee stability for concurrent gain and phase variations of the specified amount in all feedback channels (see `loopmargin` for details). For example,

```
R = TuningGoal.Margins('u',6,45);
```

enforces ± 6 dB of gain margin and 45 degrees of phase margin at the location "u". In MATLAB, use a `AnalysisPoint` block to mark the location "u", see the *"Building Tunable Models"* example for details. In Simulink, use the `addPoint` method of the `slTuner` interface to mark "u" as a point of interest. Stability margins are typically measured at the plant inputs or plant outputs or both.

The target gain and phase margin values are converted into a normalized gain constraint on some appropriate closed-loop transfer function. The desired margins are achieved at frequencies where the gain is less than 1.

```
viewSpec(R)
```

System Dynamics Specifications

This example shows how to constrain the poles of a control system tuned with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

Closed-Loop Poles

The `TuningGoal.Poles` goal constrains the location of the closed-loop poles. You can enforce some minimum decay rate

$$Re(s) < -\alpha,$$

impose some minimum damping ratio

$$Re(s) < -\zeta|s|,$$

or constrain the pole magnitude to
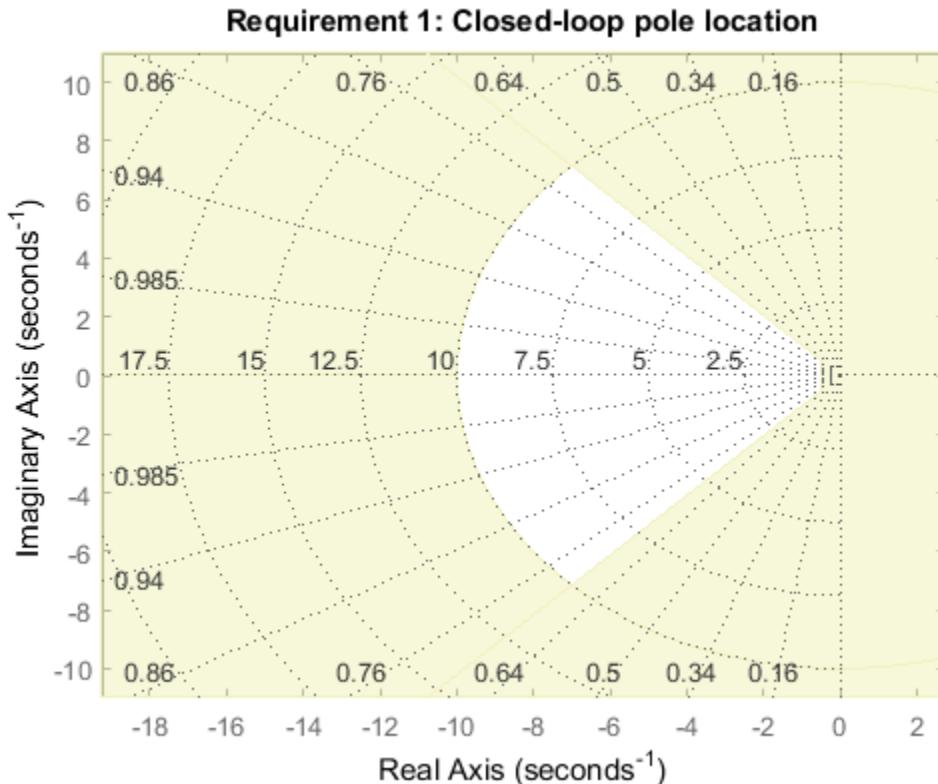
$$|s| < \omega_{\max}.$$

For example

```
MinDecay = 0.5;
MinDamping = 0.7;
MaxFrequency = 10;
R = TuningGoal.Poles(MinDecay,MinDamping,MaxFrequency);
```

constrains the closed-loop poles to lie in the white region below.

```
viewSpec(R)
```

Requirement 1: Closed-loop pole location

Increasing the `MinDecay` value results in faster transients. Increasing the `MinDamping` value results in better damped transients. Decreasing the `MaxFrequency` value prevents fast dynamics.

Controller Poles

The `TuningGoal.ControllerPoles` goal constrains the pole locations for tuned elements such as filters and compensators. The tuning algorithm may produce unstable compensators for unstable plants. To prevent this, use the `TuningGoal.ControllerPoles` goal to keep the compensator poles in the left-half plane. For example, if your compensator is parameterized as a second-order transfer function,

```
C = tunableTF('C',1,2);
```

you can force it to have stable dynamics by adding the requirement

```
MinDecay = 0;
R = TuningGoal.ControllerPoles('C',MinDecay);
```
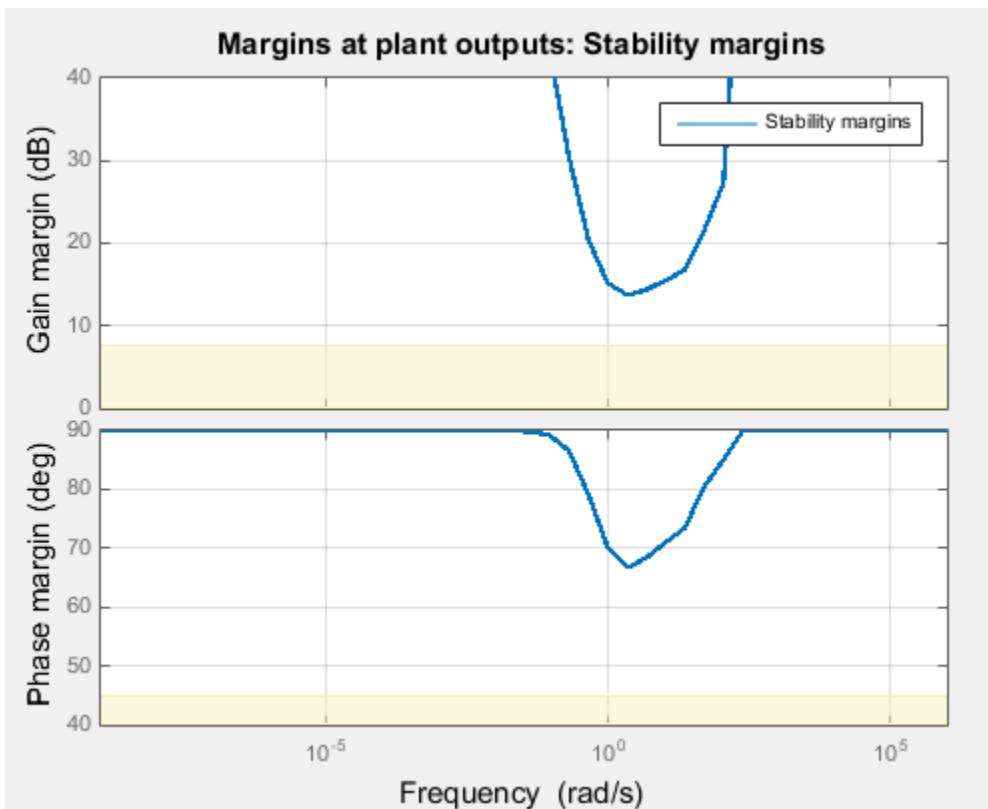
Interpreting Stability Margins in Control System Tuning

Stability Margins Plot

This topic explains how to interpret a stability-margin plot for control system tuning, such as the following plot.

. You obtain this plot in one of the following ways:

- Tuning in Control System Tuner using a "Margins Goal" on page 13-135 or "Quick Loop Tuning" on page 13-57.

- Tuning at the command line using `systune` with TuningGoal.Margins. If `S` is the control system model or `slTuner` interface, and `Req` is a `TuningGoal.Margins` goal, obtain the stability-margin plot by entering:

```
viewSpec(S,Req)
```

Gain and Phase Margins

For SISO systems, the gain and phase margins at a frequency ω indicate how much the gain or phase of the open-loop response $L(j\omega)$ can change without loss of stability.

For example, a gain margin of 5dB at 2 rad/s indicates that closed-loop stability is maintained when the loop gain increases or decreases by as much as 5dB at this frequency. Gain and phase margins typically vary across frequencies.
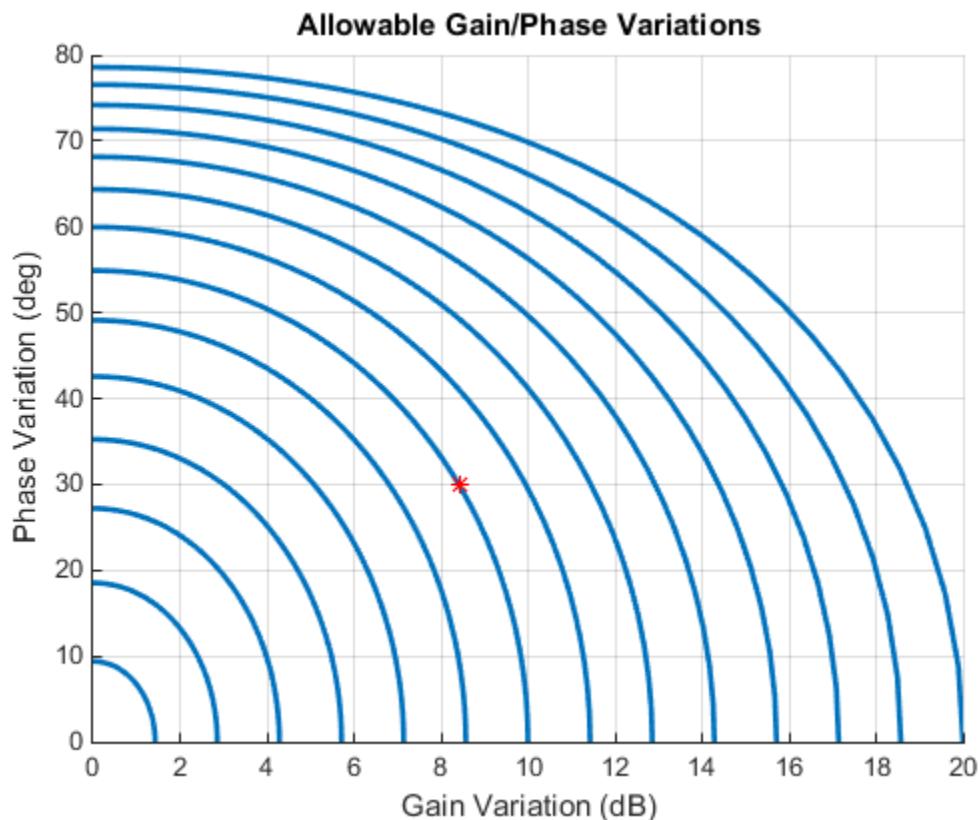
For MIMO systems, gain and phase margins are interpreted as follows:

- Gain margin: Stability is preserved when the gain increases or decreases by up to the gain margin value in each channel of the feedback loop.
- Phase margin: Stability is preserved when the phase increases or decreases by up to the phase margin value in each channel of the feedback loop.

In MIMO systems, the gain or phase can change in all channels at once, and by a different amount in each channel. The Margins Goal and `TuningGoal.Margins` rely on the notion of disk margin for MIMO systems. (See "Algorithm" on page 13-242.) Like SISO stability margins, gain and phase margins in MIMO systems typically vary across frequency.

Combined Gain and Phase Variations

To assess robustness to changes in both gain and phase, use the following chart.
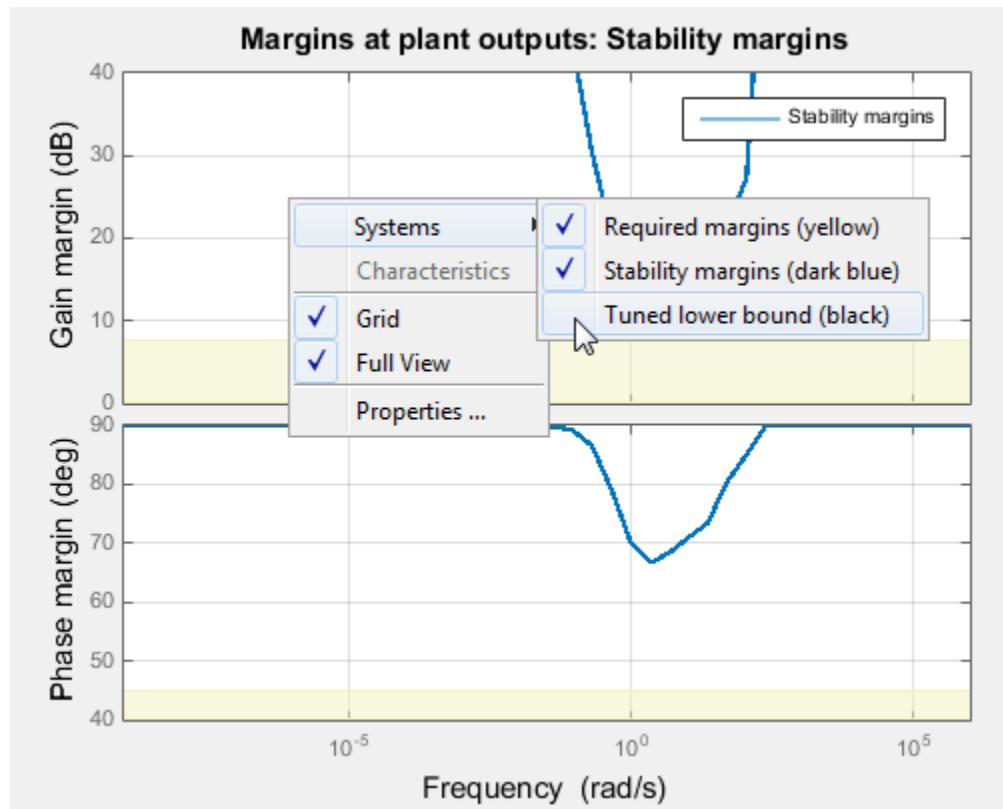
For example, if the gain margin plot in Control System Tuner indicates a 10 dB margin at a particular frequency, then trace the contour starting at (Gain,Phase) = (10,0) to see how a given amount of phase variation reduces the allowable gain variation at that frequency. For example, if the phase can vary by 30 degrees than the gain can only vary by about 8.4 dB (red mark).

Interpreting the Gain and Phase Margin Plot

The stability-margin plot for Margins Goal or TuningGoal.Margins shows in shaded yellow the region where the target margins are not met. The plot displays the current gain and phase margins (computed using the current values of the tunable parameters in the control system) as a blue trace.

These gain and phase margin curves are obtained using an exact calculation involving μ-analysis. For computational efficiency, however, the tuning algorithm uses an approximate calculation that can yield smaller margins in parts of the frequency range. To see the lower bound used by the tuner, right-click on the plot, and select **Systems > Tuned Lower Bound**.

If there is a significant gap between the true margins and the tuner approximation, try increasing the D-scaling order. The default order is zero (static scaling). For tuning in Control System Tuner, set the D-scaling order in the Margins Goal dialog box. For command-line tuning, set this value using the `ScalingOrder` property of TuningGoal.Margins.

Algorithm

The gain and phase margin values are both derived from the disk margin (see `loopmargin`). The disk margin measures the radius of a circular exclusion region centered near the critical point. This radius is a decreasing function of the scaled norm:

$$\min_{D \text{ diagonal}} \left\| D^{-1} \left(I - L(j\omega) \right) \left(I + L(j\omega) \right)^{-1} D \right\|_2 .$$

Unlike the traditional gain and phase margins, the disk margins and associated gain and phase margins guarantee that the open-loop response $L(j\omega)$ stays at a safe distance from the critical point at all frequencies.

See Also

TuningGoal.Margins | `loopmargin`

More About

*   "Margins Goal" on page 13-135

Tune Control System at the Command Line

After specifying your tuning goals using `TuningGoal` objects (see "Tuning Goals"), use `systune` to tune the parameters of your model.

The `systune` command lets you designate one or more design goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have tuning goals. `systune` attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

Organize your `TuningGoal` objects into a vector of soft requirements and a vector of hard requirements. For example, suppose you have created a tracking requirement, a rejection requirement, and stability margin requirements at the plant inputs and outputs. The following commands tune the control system represented by `T0`, treating the stability margins as hard goals, the tracking and rejection requirements as soft goals. (`T0` is either a `genss` model or an `slTuner` interface previously configured for tuning.)

```
SoftReqs = [Rtrack,Rreject];
HardReqs = [RmargIn,RmargOut];
[T,fSoft,gHard] = systune(T0,SoftReqs,HardReqs);
```

`systune` converts each tuning requirement into a normalized scalar value, f for the soft constraints and g for the hard constraints. The command adjusts the tunable parameters of `T0` to minimize the f values, subject to the constraint that each $g < 1$. `systune` returns the vectors `fSoft` and `gHard` that contain the final normalized values for each tuning goal in `SoftReqs` and `HardReqs`.

Use `systuneOptions` to configure additional options for the `systune` algorithm, such as the number of independent optimization runs, convergence tolerance, and output display options.

See Also

`systune` | `systune (for slTuner)` | `systuneOptions`

More About

- "Interpreting Tuning Results" on page 13-163

Speed Up Tuning with Parallel Computing Toolbox Software

If you have the Parallel Computing Toolbox™ software installed, you can speed up the tuning of fixed-structure control systems. When you run multiple randomized optimization starts with systune, looptune, or hinfstruct, parallel computing speeds up tuning by distributing the optimization runs among workers.

To distribute randomized optimization runs among workers:

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences, manually start a parallel pool using parpool. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a systuneOptions, looptuneOptions, or hinfstructOptions set that specifies multiple random starts. For example, the following options set specifies 20 random restarts to run in parallel for tuning with looptune:

```
options = systuneOptions('RandomStart',20,'UseParallel',true);
```

Setting UseParallel to true enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the options set when you call the tuning command. For example, if you have already created a tunable control system model, CL0, and tunable controller, and tuning requirement vectors SoftReqs and HardReqs, the following command uses parallel computing to tune the control system of CL0 with systune.

```
[CL,fSoft,gHard,info] = systune(CL0,SoftReq,Hardreq,options);
```

To learn more about configuring a parallel pool, see the Parallel Computing Toolbox documentation.

See Also
parpool

Related Examples
• "Using Parallel Computing to Accelerate Tuning" on page 13-304

More About

- "Parallel Preferences"

Validate Tuned Control System at the Command Line

In this section...
"Extract and Plot System Responses" on page 13-246
"View Tuning Goals" on page 13-246
"Write Tuned Parameters to Simulink Model" on page 13-247
"Improve Tuning Results" on page 13-247

To validate your tuned control system, use the following tools and techniques.

Extract and Plot System Responses

Evaluate the performance of your tuned control system by extracting and plotting system responses. For instance, evaluate reference tracking or overshoot performance by plotting the step response of transfer function from the reference input to the controlled output. Or, evaluate stability margins by extracting an open-loop transfer function and using the `margin` command. You can extract any transfer function you need for analysis from the tuned model of your control system.

- To extract responses from a tuned generalized state-space (`genss`) model, use analysis functions such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`.
- For a Simulink tuned through an `slTuner` interface, extract responses from the interface using analysis functions such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`.

In either case, the extracted responses are represented by state-space (`ss`) models. You can analyze these models using commands such as `step`, `bode`, `sigma`, or `margin`.

View Tuning Goals

Visualize your tuning goals using the `viewSpec` command. For each type of tuning goal, `viewSpec` plots the target requirement and the achieved response of your tuned system. This visualization allows you to examine how far your control system is from ideal performance. It can also help you determine where you can achieve better tuning results by limiting the frequency range of a tuning goal, relaxing a tuning goal from hard to soft, increasing the tolerance of a tuning goal, or similar adjustments.

For example, suppose you have tuned your control system with a tracking requirement `Rtrack` and a rejection requirement `Rreject`. The following commands display plots that let you evaluate how closely the tuned system meets those requirements. (`T` is the tuned output of `systune`, either a `genss` model or an `slTuner` interface.)

```
viewSpec(Rtrack,T)
viewSpec(Rreject,T)
```

Write Tuned Parameters to Simulink Model

When you tune a Simulink model, the software evaluates tuning goals for a linearization of the model stored in the `slTuner` interface. Therefore, you must validate the tuned controllers by simulating the full nonlinear model, even if the tuned linear system meets all your tuning goals.

To write tuned block values from a tuned `slTuner` interface to the corresponding Simulink model, use the `writeBlockValue` command. For example, suppose `SLT` is a tuned `slTuner` interface returned by `systune`. The following command writes the tuned parameters from `SLT` to the associated Simulink model.

```
writeBlockValue(SLT)
```

Simulate the Simulink model to evaluate the tuned system performance.

Improve Tuning Results

If `systune` does not find a set of controller parameters that meet your design requirements, make adjustments to your set of tuning goals to improve the results. For example:

- Designate tuning goals that are must-have requirements as hard goals. Or, relax tuning goals that are not absolute requirements by designating them as soft goals.
- Limit the frequency range in which frequency-domain goals are enforced. Use the `Focus` property of the `TuningGoal` object to do this.
- Increase the tolerance of tuning goals for which a tolerance is applicable.
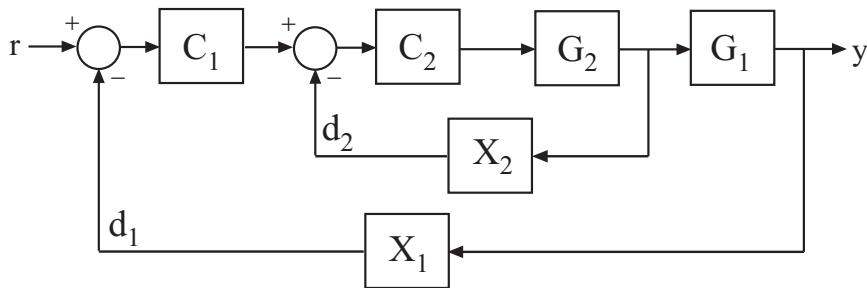
Related Examples

- "Extract Responses from Tuned MATLAB Model at the Command Line" on page 13-248

Extract Responses from Tuned MATLAB Model at the Command Line

This example shows how to analyze responses of a tuned control system by using `getIOTransfer` to compute responses between various inputs and outputs of a closed-loop model of the system.

Consider the following control system.

Suppose you have used `systune` to tune a `genss` model of this control system. The result is a `genss` model, T, which contains tunable blocks representing the controller elements C1 and C2. The tuned model also contains `AnalysisPoint` blocks that represent the analysis-point locations, X1 and X2.

Analyze the tuned system performance by examining various system responses extracted from T. For example, examine the response at the output, *y*, to a disturbance injected at the point d_1.

```
H1 = getIOTransfer(T,'X1','y');
```

H1 represents the closed-loop response of the control system to a disturbance injected at the implicit input associated with the `AnalysisPoint` block X1, which is the location of d_1:

H1 is a `genss` model that includes the tunable blocks of `T`. `H1` allows you to validate the disturbance response of your tuned system. For example, you can use analysis commands such as `bodeplot` or `stepplot` to analyze `H1`. You can also use `getValue` to obtain the current value of `H1`, in which all the tunable blocks are evaluated to their current numeric values.

Similarly, examine the response at the output to a disturbance injected at the point d_2.

```
H2 = getIOTransfer(T,'X2','y');
```

You can also generate a two-input, one-output model representing the response of the control system to simultaneous disturbances at both d_1 and d_2. To do so, provide `getIOTransfer` with a cell array that specifies the multiple input locations.

```
H = getIOTransfer(T,{'X1','X2'},'y');
```
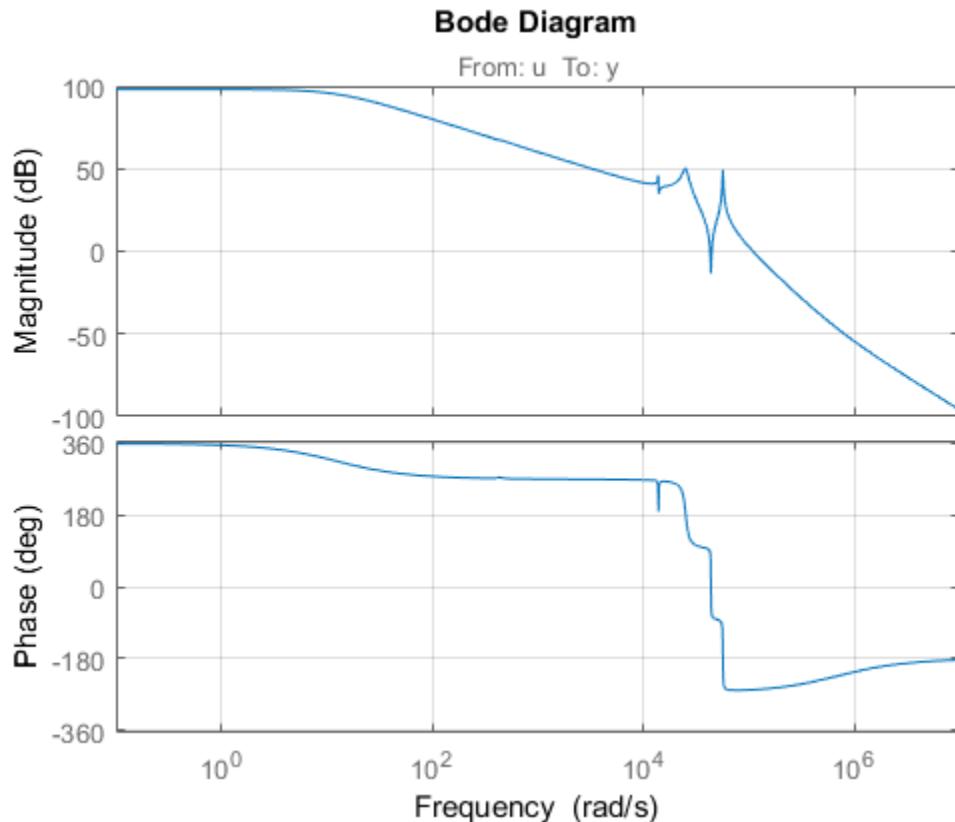
Tuning Control Systems with SYSTUNE

The `systune` command can jointly tune the gains of your control system regardless of its architecture and number of feedback loops. This example outlines the `systune` workflow on a simple application.

Head-Disk Assembly Control

This example uses a 9th-order model of the head-disk assembly (HDA) in a hard-disk drive. This model captures the first few flexible modes in the HDA.

```
load rctExamples G
bode(G), grid
```


Bode Diagram

We use the feedback loop shown below to position the head on the correct track. This control structure consists of a PI controller and a low-pass filter in the return path. The head position y should track a step change r with a response time of about one millisecond, little or no overshoot, and no steady-state error.
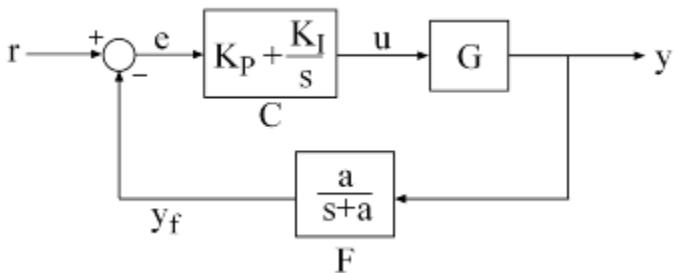
Figure 1: Control Structure

You can use systune to directly tune the PI gains and filter coefficient a subject to a variety of time- and frequency-domain requirements.

Specifying the Tunable Elements

There are two tunable elements in the control structure of Figure 1: the PI controller $C(s)$ and the low-pass filter

$$F(s) = \frac{a}{s + a}.$$

You can use the tunablePID object to parameterize the PI block:

```
C0 = tunablePID('C','pi');  % tunable PI
```

To parameterize the lowpass filter $F(s)$, create a tunable real parameter a and construct a first-order transfer function with numerator a and denominator $s + a$:

```
a = realp('a',1);    % filter coefficient
F0 = tf(a,[1 a]);    % filter parameterized by a
```

See the *"Building Tunable Models"* example for an overview of available tunable elements.

Building a Tunable Closed-Loop Model

Next build a closed-loop model of the feedback loop in Figure 1. To facilitate open-loop analysis and specify open-loop requirements such as desired stability margins, add an analysis point at the plant input u:

```
AP = AnalysisPoint('u');
```
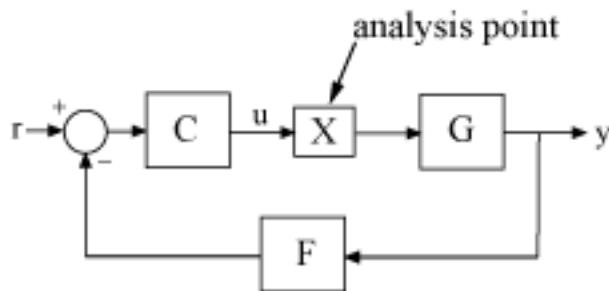

Figure 2: Analysis Point Block

Use `feedback` to build a model of the closed-loop transfer from reference r to head position y:

```
TO = feedback(G*AP*CO,FO);  % closed-loop transfer from r to y
TO.InputName = 'r';
TO.OutputName = 'y';
```

The result TO is a generalized state-space model (`genss`) that depends on the tunable elements C and F.

Specifying the Design Requirements

The `TuningGoal` package contains a variety of control design requirements for specifying the desired behavior of the control system. These include requirements on the response time, deterministic and stochastic gains, loop shape, stability margins, and pole locations. Here we use two requirements to capture the control objectives:

- **Tracking requirement** : The position y should track the reference r with a 1 millisecond response time

- **Stability margin requirement** : The feedback loop should have 6dB of gain margin and 45 degrees of phase margin

Use the `TuningGoal.Tracking` and `TuningGoal.Margins` objects to capture these requirements. Note that the margins requirement applies to the open-loop response measured at the plant input u (location marked by the analysis point AP).

```
Req1 = TuningGoal.Tracking('r','y',0.001);
Req2 = TuningGoal.Margins('u',6,45);
```

Tuning the Controller Parameters

You can now use `systune` to tune the PI gain and filter coefficient a. This function takes the tunable closed-loop model T0 and the requirements `Req1,Req2`. Use a few randomized starting points to improve the chances of getting a globally optimal design.

```
rng('default')
Options = systuneOptions('RandomStart',3);
[T,fSoft] = systune(T0,[Req1,Req2],Options);

Final: Soft = 2.61, Hard = -Inf, Iterations = 125
Final: Soft = 1.35, Hard = -Inf, Iterations = 136
Final: Soft = 2.78e+03, Hard = -Inf, Iterations = 189
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Soft = 1.35, Hard = -Inf, Iterations = 61
```

All requirements are normalized so a requirement is satisfied when its value is less than 1. Here the final value is slightly greater than 1, indicating that the requirements are nearly satisfied. Use the output `fSoft` to see the tuned value of each requirement. Here we see that the first requirement (tracking) is slightly violated while the second requirement (margins) is satisfied.

```
fSoft


fSoft =

    1.3461    0.6326
```

The first output T of `systune` is the "tuned" closed-loop model. Use `showTunable` or `getBlockValue` to access the tuned values of the PI gains and filter coefficient:
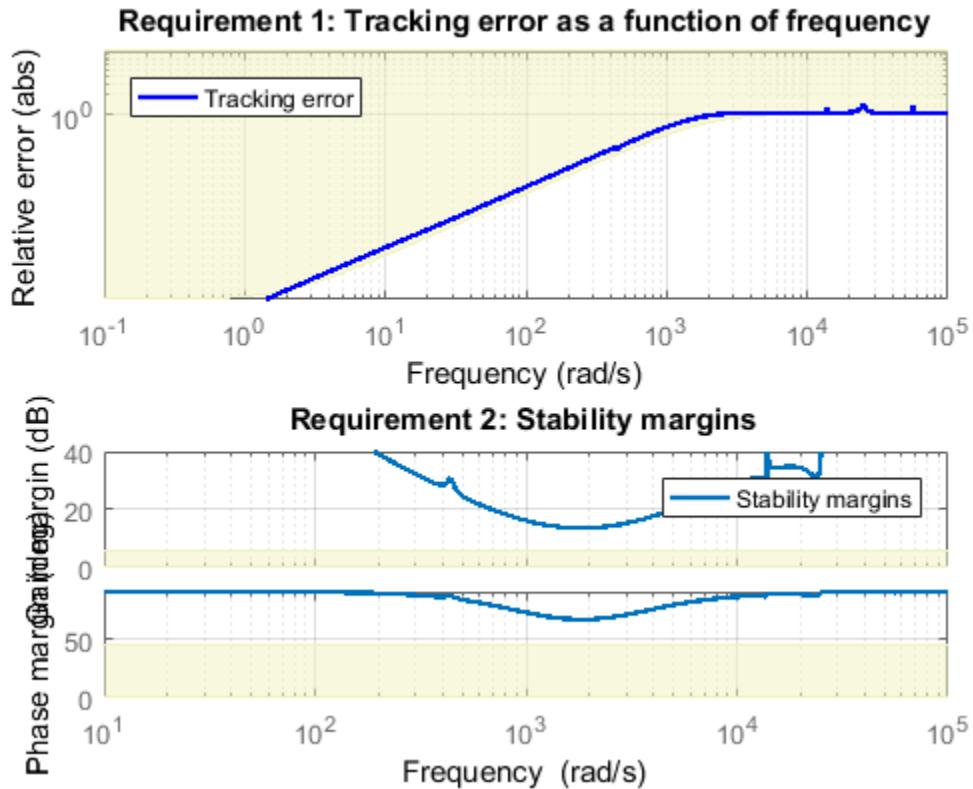
```
getBlockValue(T,'C')  % tuned value of PI controller
```

```
ans =

              1
  Kp + Ki * ---
              s

  with Kp = 0.00104, Ki = 0.0122

Name: C
Continuous-time PI controller in parallel form.


showTunable(T)  % tuned values of all tunable elements

C =

              1
  Kp + Ki * ---
              s

  with Kp = 0.00104, Ki = 0.0122

Name: C
Continuous-time PI controller in parallel form.
---------------------------------
a = 3.19e+03
```

Validating Results
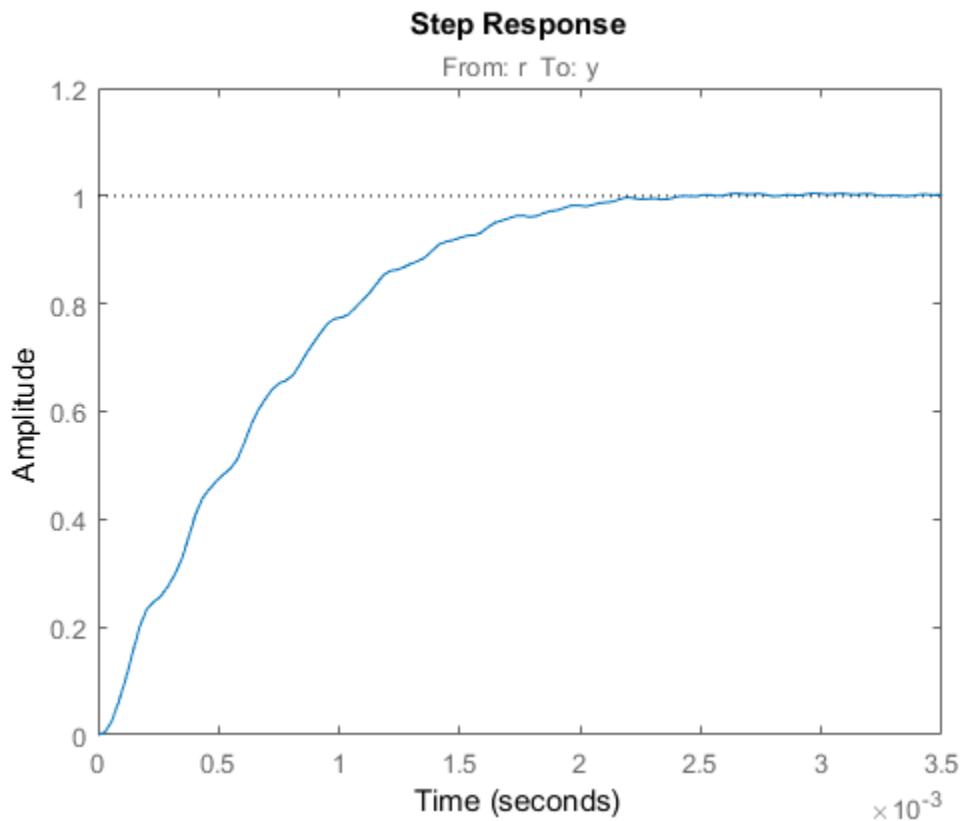
First use `viewSpec` to inspect how the tuned system does against each requirement.
The first plot shows the tracking error as a function of frequency, and the second plot
shows the normalized disk margins as a function of frequency (see `loopmargin`). See the
"Creating Design Requirements" example for details.

```
clf, viewSpec([Req1 Req2],T)
```
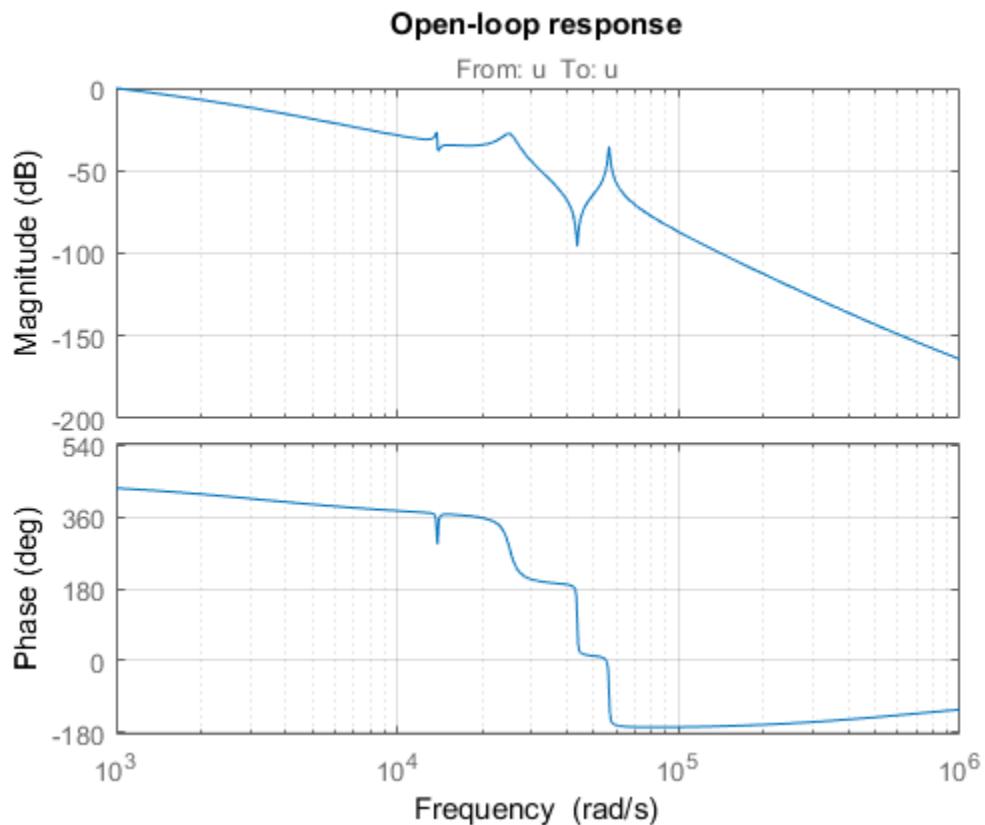
Next plot the closed-loop step response from reference r to head position y. The response has no overshoot but wobbles a little.

```
clf, step(T)
```

Step Response

From: r To: y

To investigate further, use `getLoopTransfer` to get the open-loop response at the plant input.

```
L = getLoopTransfer(T,'u');
bode(L,{1e3,1e6}), grid
title('Open-loop response')
```

The wobble is due to the first resonance after the gain crossover. To eliminate it, you could add a notch filter to the feedback loop and tune its coefficients along with the lowpass coefficient and PI gains using `systune`.
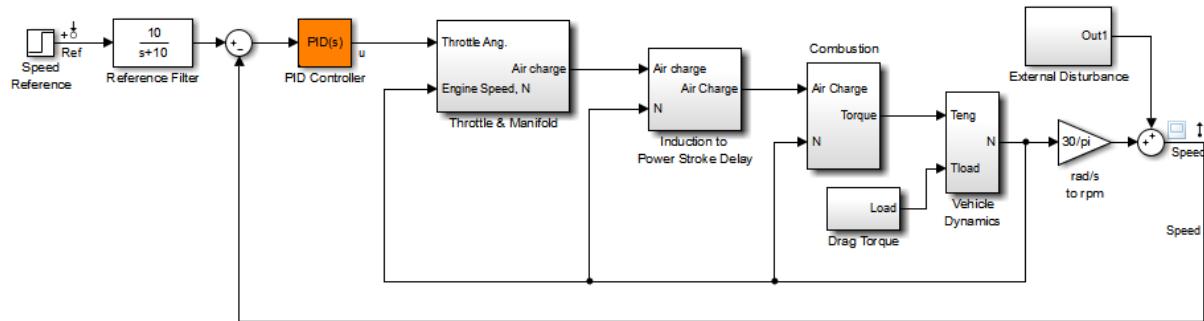
Tune Control Systems in Simulink

This example shows how to use `systune` or `looptune` to automatically tune control systems modeled in Simulink.

Engine Speed Control

For this example we use the following model of an engine speed control system:

```
open_system('rct_engine_speed')
```


Copyright 2004-2010 The MathWorks, Inc.

The control system consists of a single PID loop and the PID controller gains must be tuned to adequately respond to step changes in the desired speed. Specifically, we want the response to settle in less than 5 seconds with little or no overshoot. While `pidtune` is a faster alternative for tuning a single PID controller, this simple example is well suited for an introduction to the `systune` and `looptune` workflows in Simulink.

Controller Tuning with SYSTUNE

The `slTuner` interface provides a convenient gateway to `systune` for control systems modeled in Simulink. This interface lets you specify which blocks in the Simulink model are tunable and what signals are of interest for open- or closed-loop validation. Create an `slTuner` instance for the `rct_engine_speed` model and list the "PID Controller" block (highlighted in orange) as tunable. Note that all Linear Analysis points in the model (signals "Ref" and "Speed" here) are automatically available as points of interest for tuning.

```
ST0 = slTuner('rct_engine_speed','PID Controller');
```

The PID block is initialized with its value in the Simulink model, which you can access using `getBlockValue`. Note that the proportional and derivative gains are initialized to zero.

```
getBlockValue(ST0,'PID Controller')
```

```
ans =

         1
  Ki * ---
         s

  with Ki = 0.01

Name: PID_Controller
Continuous-time I-only controller.
```

Next create a reference tracking requirement to capture the target settling time. Use the signal names in the Simulink model to refer to the reference and output signals, and use a two-second response time target to ensure settling in less than 5 seconds.

```
TrackReq = TuningGoal.Tracking('Ref','Speed',2);
```

You can now tune the control system `ST0` subject to the requirement `TrackReq`.

```
ST1 = systune(ST0,TrackReq);
```

```
Final: Soft = 1.07, Hard = -Inf, Iterations = 60
```

The final value is close to 1 indicating that the tracking requirement is met. `systune` returns a "tuned" version `ST1` of the control system described by `ST0`. Again use `getBlockValue` to access the tuned values of the PID gains:

```
getBlockValue(ST1,'PID Controller')
```
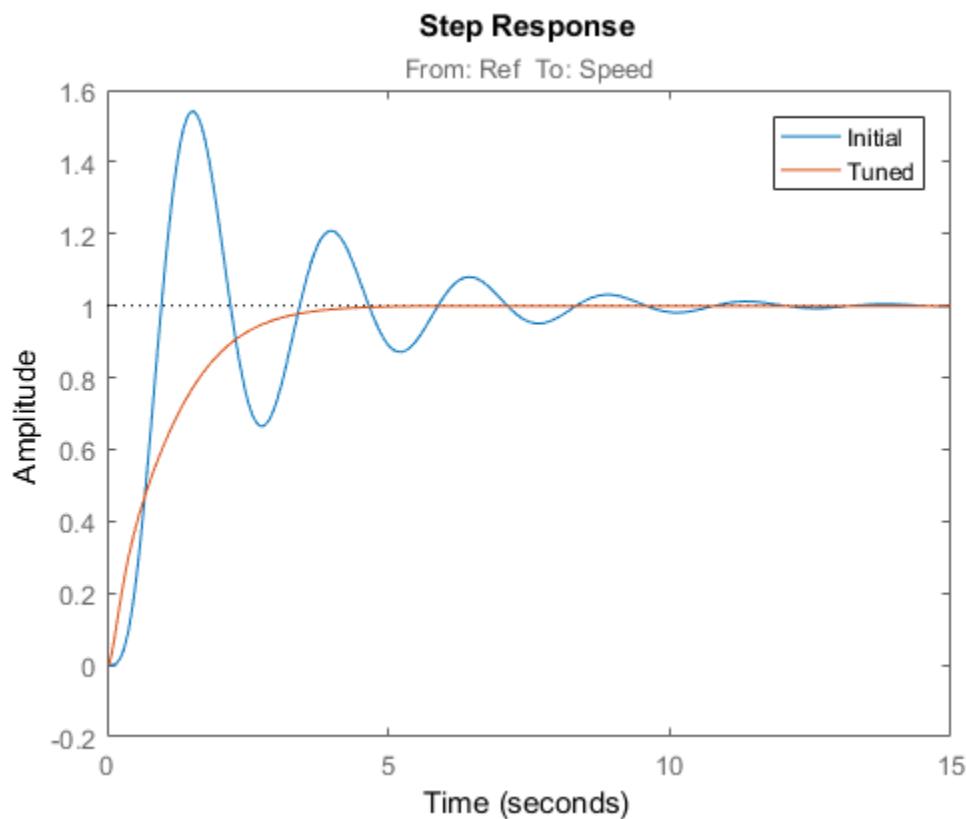
```
ans =

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

with Kp = 0.00188, Ki = 0.00339, Kd = 0.00047, Tf = 4.08e-05

Name: PID_Controller
Continuous-time PIDF controller in parallel form.

To simulate the closed-loop response to a step command in speed, get the initial and tuned transfer functions from speed command "Ref" to "Speed" output and plot their step responses:

```
T0 = getIOTransfer(ST0,'Ref','Speed');
T1 = getIOTransfer(ST1,'Ref','Speed');
step(T0,T1)
legend('Initial','Tuned')
```

Controller Tuning with LOOPTUNE

You can also use `looptune` to tune control systems modeled in Simulink. The `looptune` workflow is very similar to the `systune` workflow. One difference is that `looptune` needs to know the boundary between the plant and controller, which is specified in terms of *controls* and *measurements* signals. For a single loop the performance is essentially captured by the response time, or equivalently by the open-loop crossover frequency. Based on first-order characteristics the crossover frequency should exceed 1 rad/s for the closed-loop response to settle in less than 5 seconds. You can therefore tune the PID loop using 1 rad/s as target 0-dB crossover frequency.
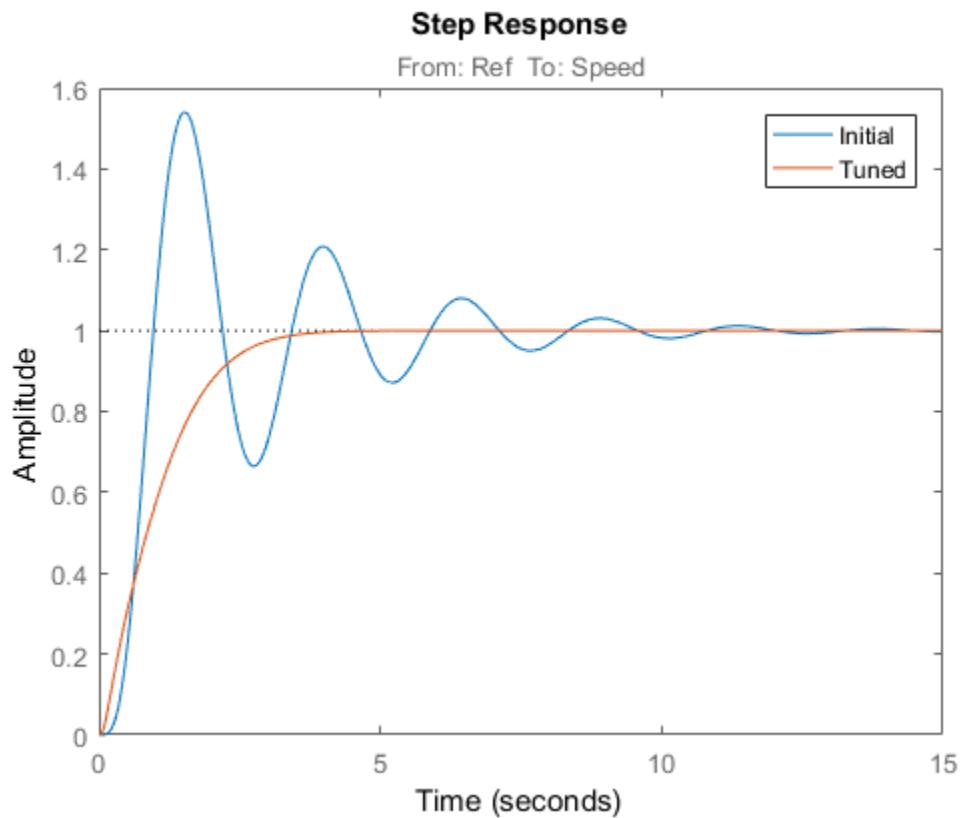
```
% Mark the signal "u" as a point of interest
addPoint(STO,'u')

% Tune the controller parameters
Control = 'u';
Measurement = 'Speed';
wc = 1;
ST1 = looptune(STO,Control,Measurement,wc);

Final: Peak gain = 0.942, Iterations = 10
Achieved target gain value TargetGain=1.
```

Again the final value is close to 1, indicating that the target control bandwidth was achieved. As with `systune`, use `getIOTransfer` to compute and plot the closed-loop response from speed command to actual speed. The result is very similar to that obtained with `systune`.

```
TO = getIOTransfer(STO,'Ref','Speed');
T1 = getIOTransfer(ST1,'Ref','Speed');
step(TO,T1)
legend('Initial','Tuned')
```

Step Response

From: Ref To: Speed

You can also perform open-loop analysis, for example, compute the gain and phase margins at the plant input u.

```
% Note: -1 because |margin| expects the negative-feedback loop transfer
L = getLoopTransfer(ST1,'u',-1);

margin(L), grid
```

Bode Diagram

Gm = 36.6 dB (at 67.5 rad/s) , Pm = 83.3 deg (at 0.998 rad/s)
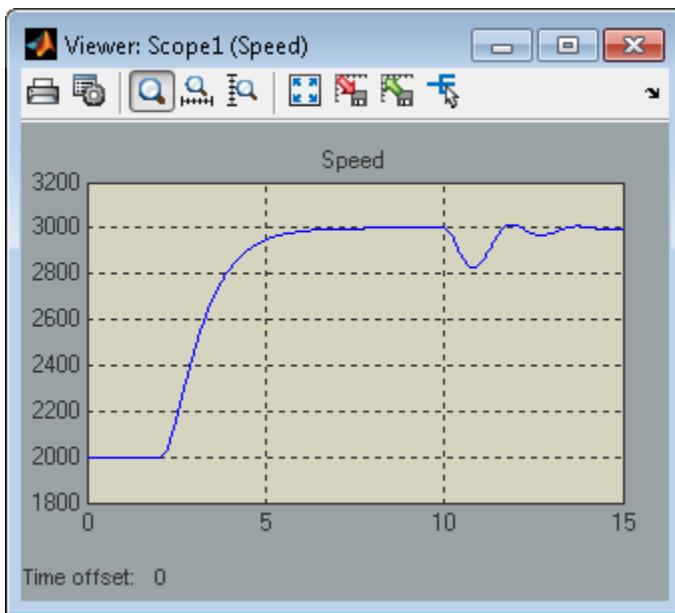
Validation in Simulink

Once you are satisfied with the `systune` or `looptune` results, you can upload the tuned controller parameters to Simulink for further validation with the nonlinear model.

```
writeBlockValue(ST1)
```

You can now simulate the engine response with the tuned PID controller.

The nonlinear simulation results closely match the linear responses obtained in MATLAB.

Comparison of PI and PID Controllers

Closer inspection of the tuned PID gains suggests that the derivative term contributes little because of the large value of the Tf coefficient.

```
showTunable(ST1)
```

```
Block 1: rct_engine_speed/PID Controller =

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

  with Kp = 0.0014, Ki = 0.00327, Kd = 0.000305, Tf = 0.01

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

This suggests using a simpler PI controller instead. To do this, you need to override the default parameterization for the "PID Controller" block:

```
setBlockParam(ST0,'PID Controller',tunablePID('C','pi'))
```
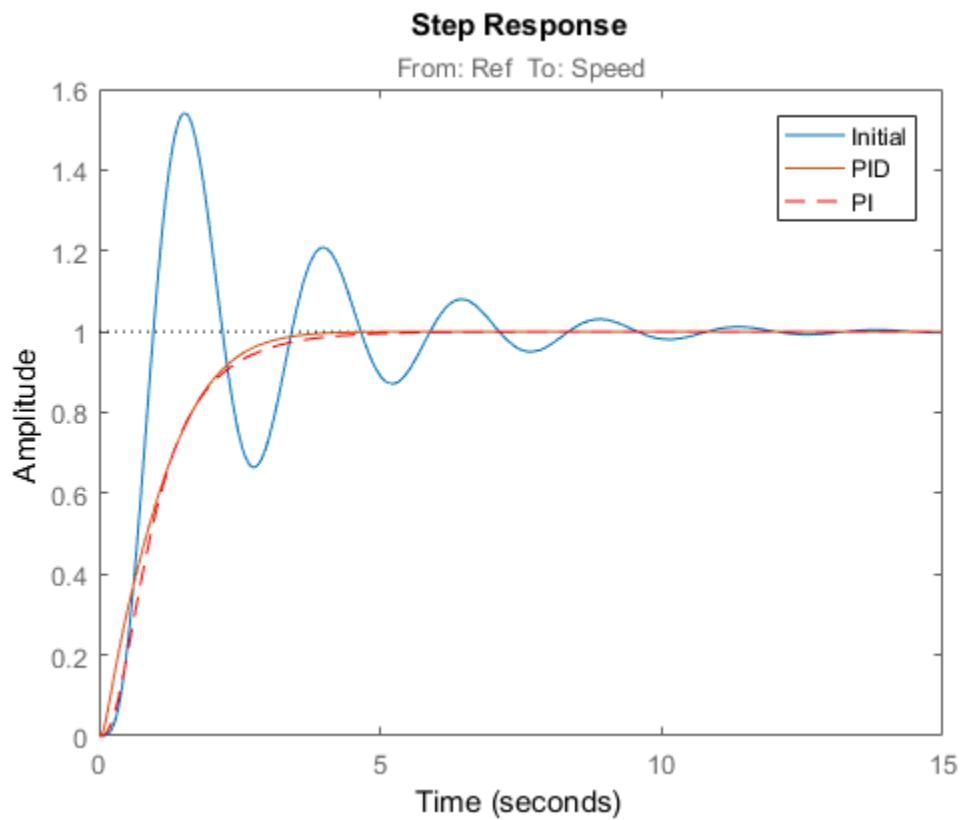
This specifies that the "PID Controller" block should now be parameterized as a mere PI controller. Next re-tune the control system for this simpler controller:

```
ST2 = looptune(ST0,Control,Measurement,wc);

Final: Peak gain = 0.915, Iterations = 5
Achieved target gain value TargetGain=1.
```

Again the final value is less than one indicating success. Compare the closed-loop response with the previous ones:

```
T2 = getIOTransfer(ST2,'Ref','Speed');
step(T0,T1,T2,'r--')
legend('Initial','PID','PI')
```

Step Response

From: Ref To: Speed

Clearly a PI controller is sufficient for this application.

Building Tunable Models

This example shows how to create tunable models of control systems for use with `systune` or `looptune`.

Background

You can tune the gains and parameters of your control system with `systune` or `looptune`. To use these commands, you need to construct a tunable model of the control system that identifies and parameterizes its tunable elements. This is done by combining numeric LTI models of the fixed elements with parametric models of the tunable elements.

Using Pre-Defined Tunable Elements

You can use one of the following "parametric" blocks to model commonly encountered tunable elements:

- **tunableGain**: Tunable gain
- **tunablePID**: Tunable PID controller
- **tunablePID2**: Tunable two-degree-of-freedom PID controller
- **tunableTF**: Tunable transfer function
- **tunableSS**: Tunable state-space model.

For example, create a tunable model of the feedforward/feedback configuration of Figure 1 where C is a tunable PID controller and F is a tunable first-order transfer function.
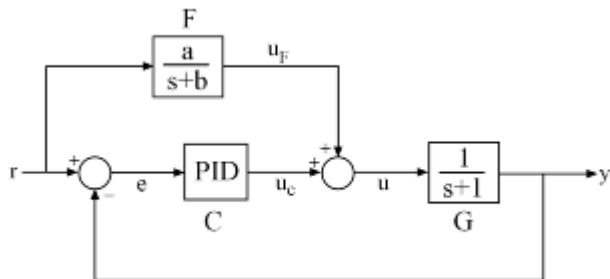

Figure 1: Control System with Feedforward and Feedback Paths

First model each block in the block diagram, using suitable parametric blocks for C and F.

```
G = tf(1,[1 1]);
C = tunablePID('C','pid');  % tunable PID block
F = tunableTF('F',0,1);     % tunable first-order transfer function
```

Then use `connect` to build a model of the overall block diagram. To specify how the blocks are connected, label the inputs and outputs of each block and model the summing junctions using `sumblk`.

```
G.u = 'u';  G.y = 'y';
C.u = 'e';  C.y = 'uC';
F.u = 'r';  F.y = 'uF';

% Summing junctions
S1 = sumblk('e = r-y');
S2 = sumblk('u = uF + uC');

T = connect(G,C,F,S1,S2,'r','y')


T =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, and
    C: Parametric PID controller, 1 occurrences.
    F: Parametric SISO transfer function, 0 zeros, 1 poles, 1 occurrences.

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

This creates a generalized state-space model `T` of the closed-loop transfer function from `r` to `y`. This model depends on the tunable blocks `C` and `F`. You can use `systune` to automatically tune the PID gains and the feedforward coefficients `a,b` subject to your performance requirements. Use `showTunable` to see the current value of the tunable blocks.

```
showTunable(T)

C =

         1
  Ki * ---
         s
```

```
  with Ki = 0.001

Name: C
Continuous-time I-only controller.
---------------------------------
F =

     10
   ------
   s + 10

Name: F
Continuous-time transfer function.
```

Interacting with the Tunable Parameters

You can adjust the parameterization of the tunable elements C and F by interacting with the objects C and F. Use get to see their list of properties.

```
get(C)

             Kp: [1x1 param.Continuous]
             Ki: [1x1 param.Continuous]
             Kd: [1x1 param.Continuous]
             Tf: [1x1 param.Continuous]
       IFormula: ''
       DFormula: ''
             Ts: 0
       TimeUnit: 'seconds'
      InputName: {'e'}
      InputUnit: {''}
     InputGroup: [1x1 struct]
     OutputName: {'uC'}
     OutputUnit: {''}
    OutputGroup: [1x1 struct]
           Name: 'C'
          Notes: {}
       UserData: []
```

A PID controller has four tunable parameters Kp,Ki,Kd,Tf. The tunable block C contains a description of each of these parameters. Parameter attributes include current value, minimum and maximum values, and whether the parameter is free or fixed.

```
C.Kp


ans =

         Name: 'Kp'
        Value: 0
      Minimum: -Inf
      Maximum: Inf
         Free: 1
        Scale: 1
         Info: [1x1 struct]


1x1 param.Continuous
```

Set the corresponding attributes to override defaults. For example, you can fix the time constant Tf to the value 0.1 by

```
C.Tf.Value = 0.1;
C.Tf.Free = false;
```

Creating Custom Tunable Elements

For tunable elements not covered by the pre-defined blocks listed above, you can create your own parameterization in terms of elementary real parameters (realp). Consider the low-pass filter

$$F(s) = \frac{a}{s+a}$$

where the coefficient a is tunable. To model this tunable element, create a real parameter a and define F as a transfer function whose numerator and denominator are functions of a. This creates a generalized state-space model F of the low-pass filter parameterized by the tunable scalar a.

```
a = realp('a',1);    % real tunable parameter, initial value 1
F = tf(a,[1 a])


F =
```

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

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

Similarly, you can use real parameters to model the notch filter

$$N(s) = \frac{s^2 + 2\zeta_1\omega_n s + \omega_n^2}{s^2 + 2\zeta_2\omega_n s + \omega_n^2}$$

with tunable coefficients $\omega_n, \zeta_1, \zeta_2$.

```
wn = realp('wn',100);
zeta1 = realp('zeta1',1);   zeta1.Maximum = 1;    % zeta1 <= 1
zeta2 = realp('zeta2',1);   zeta2.Maximum = 1;    % zeta2 <= 1
N = tf([1 2*zeta1*wn wn^2],[1 2*zeta2*wn wn^2]);  % tunable notch filter
```

You can also create tunable elements with matrix-valued parameters. For example, model the observer-based controller $C(s)$ with equations

$$\frac{dx}{dt} = Ax + Bu + L(y - Cx), \quad u = -Kx$$

and tunable gain matrices K and L.

```
% Plant with 6 states, 2 controls, 3 measurements
[A,B,C] = ssdata(rss(6,3,2));

K = realp('K',zeros(2,6));
L = realp('L',zeros(6,3));

C = ss(A-B*K-L*C,L,-K,0)


C =

  Generalized continuous-time state-space model with 2 outputs, 3 inputs, 6 states, and
     K: Parametric 2x6 matrix, 2 occurrences.
     L: Parametric 6x3 matrix, 2 occurrences.
```

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

Enabling Open-Loop Requirements

The `systune` command takes a closed-loop model of the overall control system, like the tunable model `T` built at the beginning of this example. Such models do not readily support open-loop analysis or open-loop specifications such as loop shapes and stability margins. To gain access to open-loop responses, insert an `AnalysisPoint` block as shown in Figure 2.
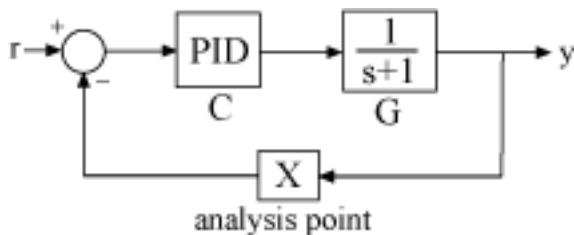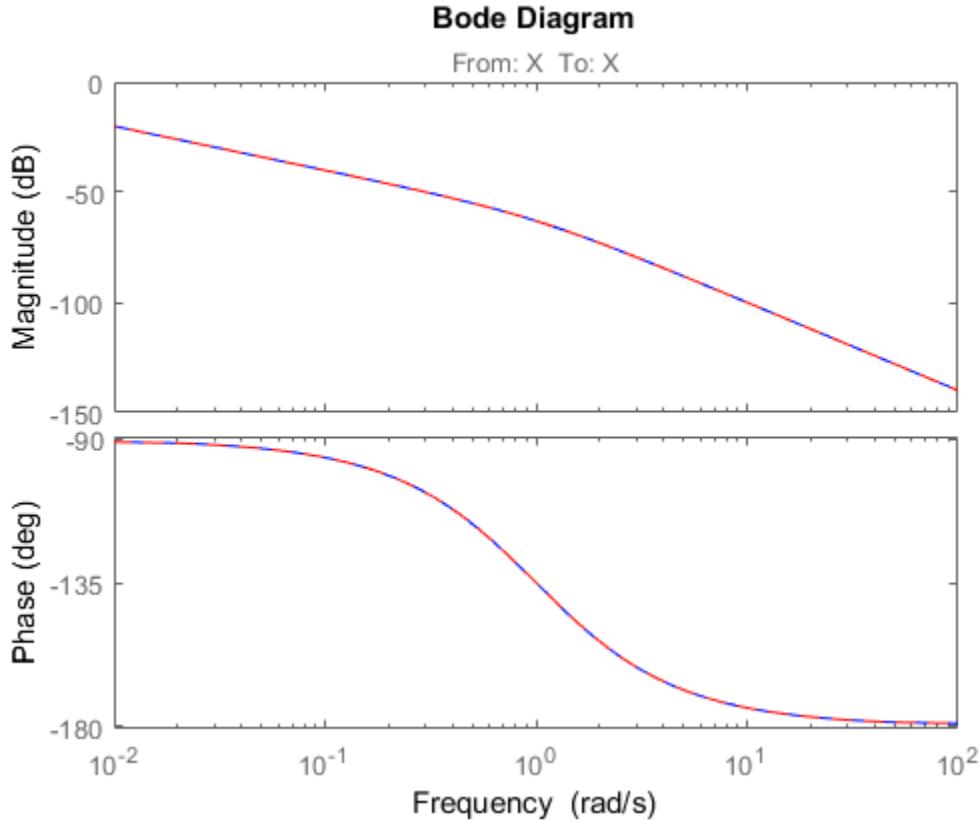

Figure 2: Analysis Point Block

The `AnalysisPoint` block can be used to mark internal signals of interest as well as locations where to open feedback loops and measure open-loop responses. This block evaluates to a unit gain and has no impact on the model responses. For example, construct a closed-loop model `T` of the feedback loop of Figure 2 where C is a tunable PID.

```
G = tf(1,[1 1]);
C = tunablePID('C','pid');
AP = AnalysisPoint('X');
T = feedback(G*C,AP);
```

You can now use `getLoopTransfer` to compute the (negative-feedback) loop transfer function measured at the location "X". Note that this loop transfer function is $L = GC$ for the feedback loop of Figure 2.

```
L = getLoopTransfer(T,'X',-1);   % loop transfer at "X"
clf, bode(L,'b',G*C,'r--')
```

You can also refer to the location "X" when specifying target loop shapes or stability margins for `systune`. The requirement then applies to the loop transfer measured at this location.

```
% Target loop shape for loop transfer at "X"
Req1 = TuningGoal.LoopShape('X',tf(5,[1 0]));

% Target stability margins for loop transfer at "X"
Req2 = TuningGoal.Margins('X',6,40);
```

In general, loop opening locations are specified in the `Location` property of `AnalysisPoint` blocks. For single-channel analysis points, the block name is used as default location name. For multi-channel analysis points, indices are appended to the block name to form the default location names.

13-273

```
AP = AnalysisPoint('Y',2);  % two-channel analysis point
AP.Location
```

```
ans =

    'Y(1)'
    'Y(2)'
```

You can override the default location names and use more descriptive names by modifying the Location property.

```
% Rename loop opening locations to "InnerLoop" and "OuterLoop".
AP.Location = {'InnerLoop' ; 'OuterLoop'};
AP.Location
```

```
ans =

    'InnerLoop'
    'OuterLoop'
```

# Tune a Control System Using Control System Tuner
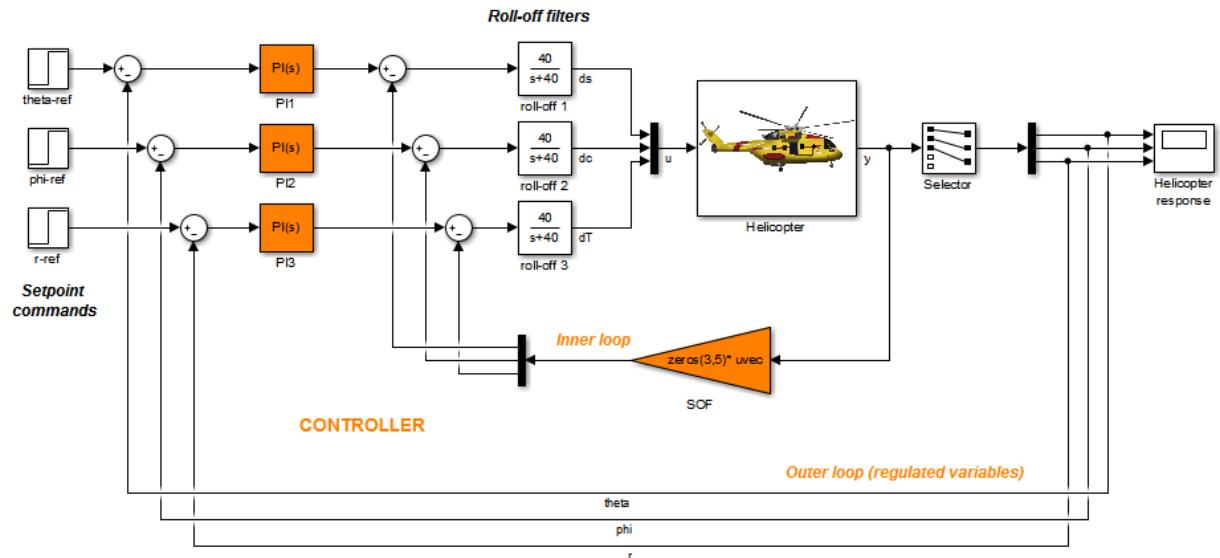
This example shows how to use the Control System Tuner app to tune a MIMO, multiloop control system modeled in Simulink.

Control System Tuner lets you model any control architecture and specify the structure of controller components, such as PID controllers, gains, and other elements. You specify which blocks in the model are tunable. Control System Tuner parameterizes those blocks and tunes the free parameters system to meet design requirements that you specify, such as setpoint tracking, disturbance rejection, and stability margins.

Control System Model

This example uses the Simulink model `rct_helico`. Open the model.

```
open_system('rct_helico')
```


The plant, `Helicopter`, is an 8-state helicopter model trimmed to a steady-state hovering condition. The state vector `x = [u,w,q,theta,v,p,phi,r]` consists of:

· Longitudinal velocity `u` (m/s)

- Normal velocity `w` (m/s)
- Pitch rate `q` (deg/s)
- Pitch angle `theta` (deg)
- Lateral velocity `v` (m/s)
- Roll rate `p` (deg/s)
- Roll angle `phi` (deg)
- Yaw rate `r` (deg/s)

The control system of the model has two feedback loops. The inner loop provides static output feedback for stability augmentation and decoupling, represented in the model by the gain block `SOF`. The outer loop has a PI controller for each of the three attitude angles. The controller generates commands `ds,dc,dT` in degrees for the longitudinal cyclic, lateral cyclic, and tail rotor collective using measurements of `theta`, `phi`, `p`, `q`, and `r`. This loop provides the desired setpoint tracking for the three angles.
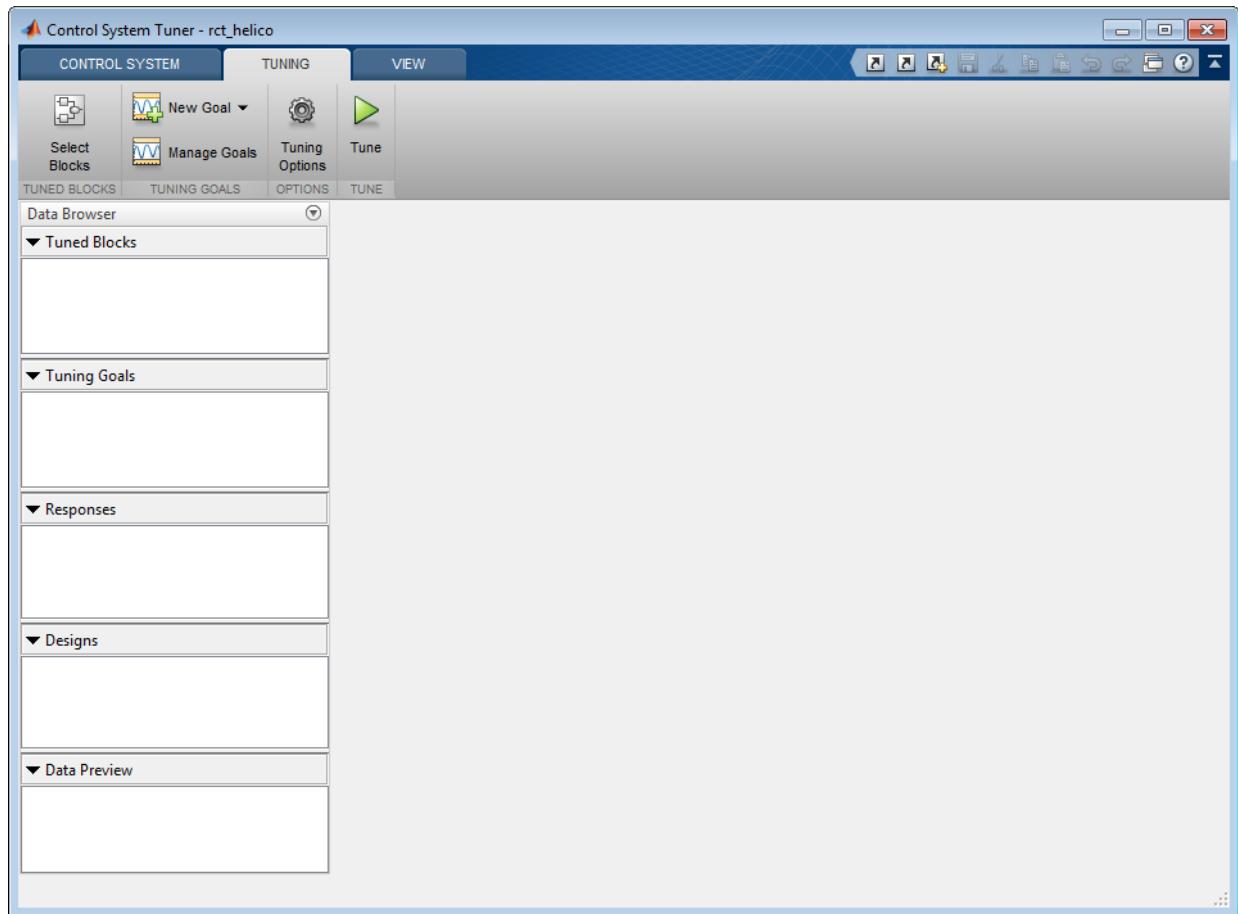
This example uses these control objectives:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, rise times of about 2 seconds, minimal overshoot, and minimal cross-coupling.
- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise. (The model contains low-pass filters that partially enforce this objective.)
- Provide strong multivariable gain and phase margins. (Multivariable margins measure robustness to simultaneous gain or phase variations at the plant inputs and outputs. See the `loopmargin` reference page for details.)
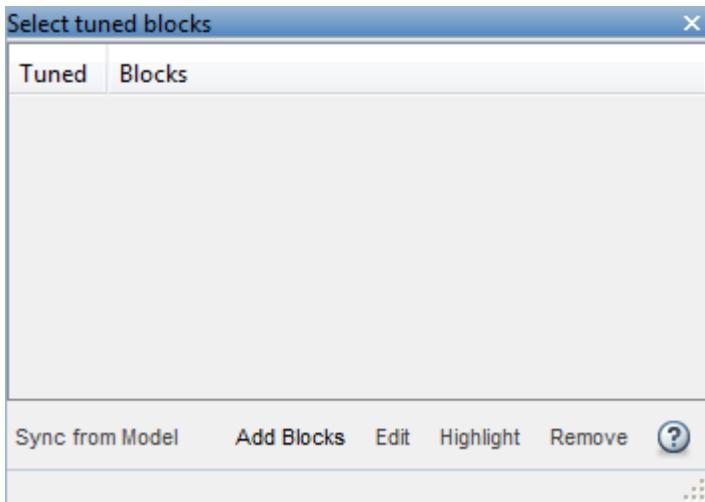
Set Up the Model for Tuning

Using Control System Tuner, you can jointly tune the inner and outer loops to meet all the design requirements. To set up the model for tuning, open the app and specify which blocks of the Simulink model you want to tune.
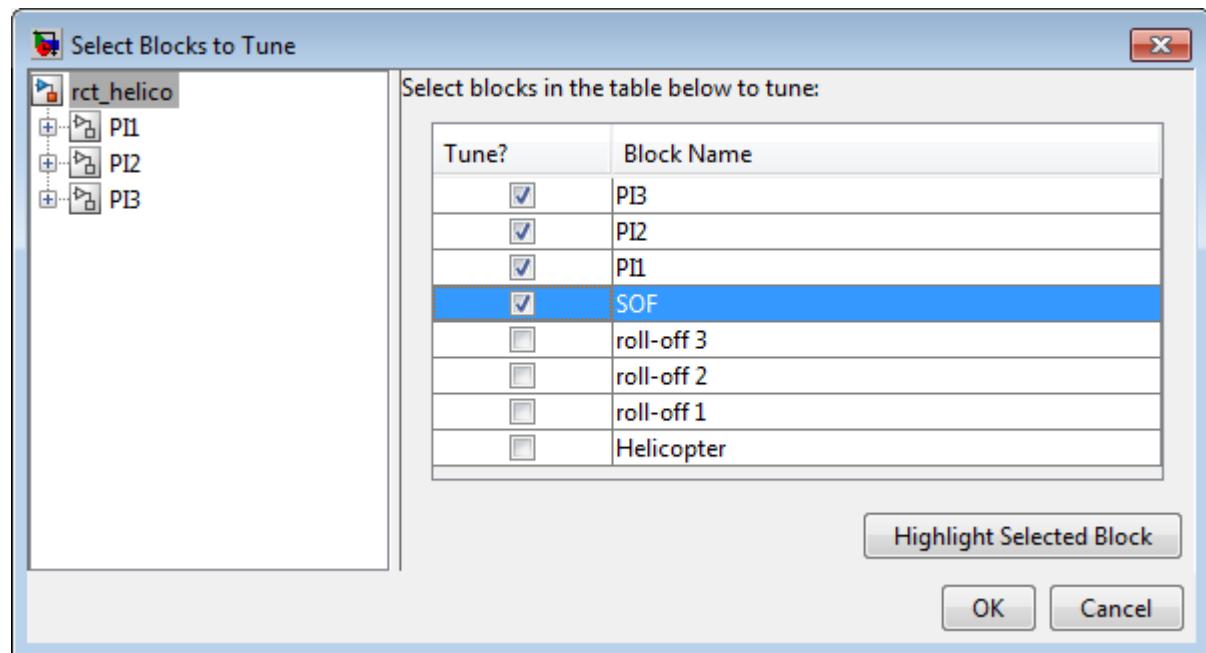
In the Simulink model window, in the Analysis menu, select **Control Design > Control System Tuner**.
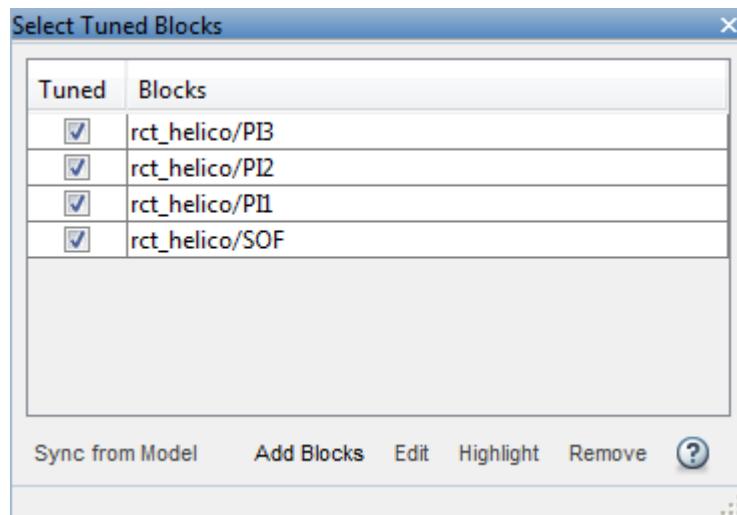
In Control System Tuner, on the **Tuning** tab, click **Select Blocks**. Use the Select tuned blocks dialog box to specify the blocks to tune.
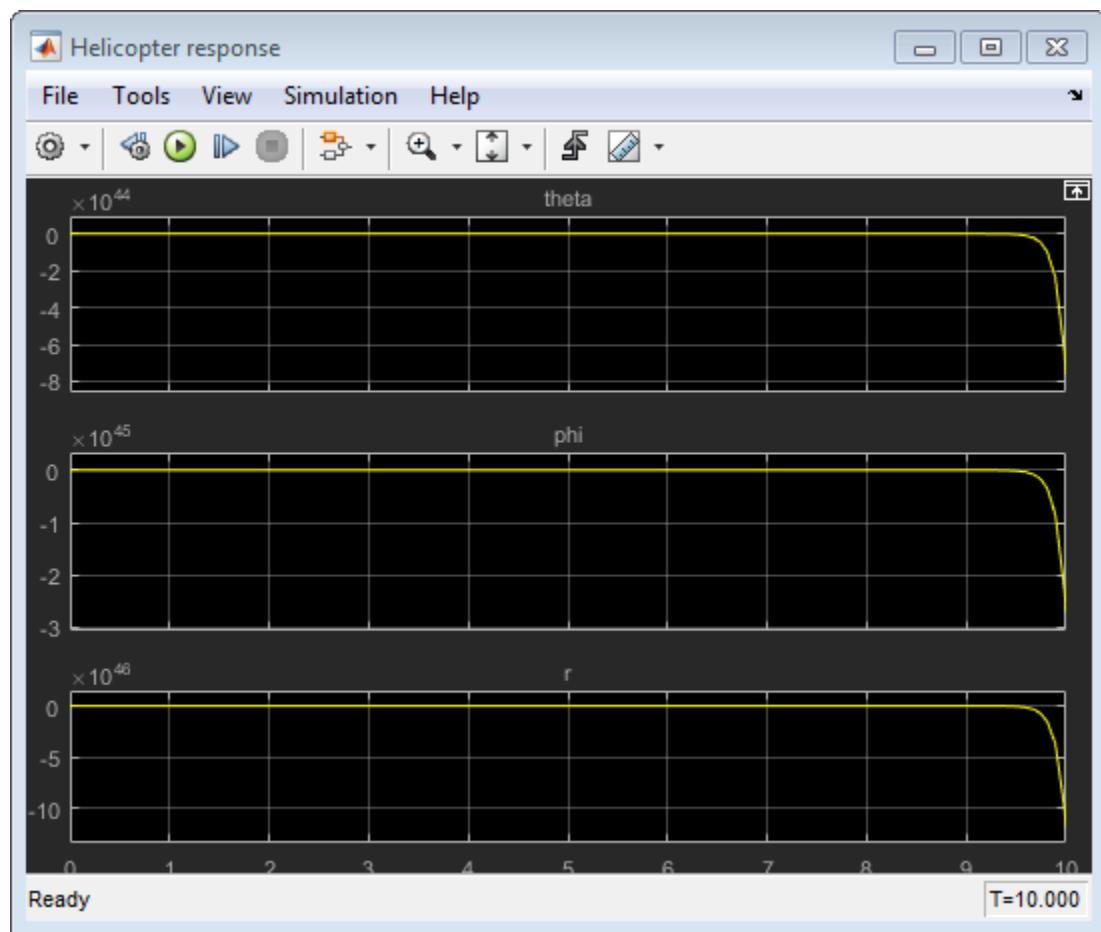
Click **Add Blocks**. Control System Tuner analyzes your model to find blocks that can be tuned. For this example, the controller blocks to tune are the three PI controllers and the gain block. Check the corresponding blocks `PI1`, `PI2`, `PI3`, and `SOF`.

Click **OK**. The Select tuned blocks dialog box now reflects the blocks you added.

When you select a block to tune, Control System Tuner automatically parametrizes the block according to its type and initializes the the parametrization with the block value in the Simulink model. In this example, the PI controllers are initialized to $1 + 1/s$ and the static output-feedback gain is initialized to zero on all channels. Simulating the model shows that the control system is unstable for these initial values.
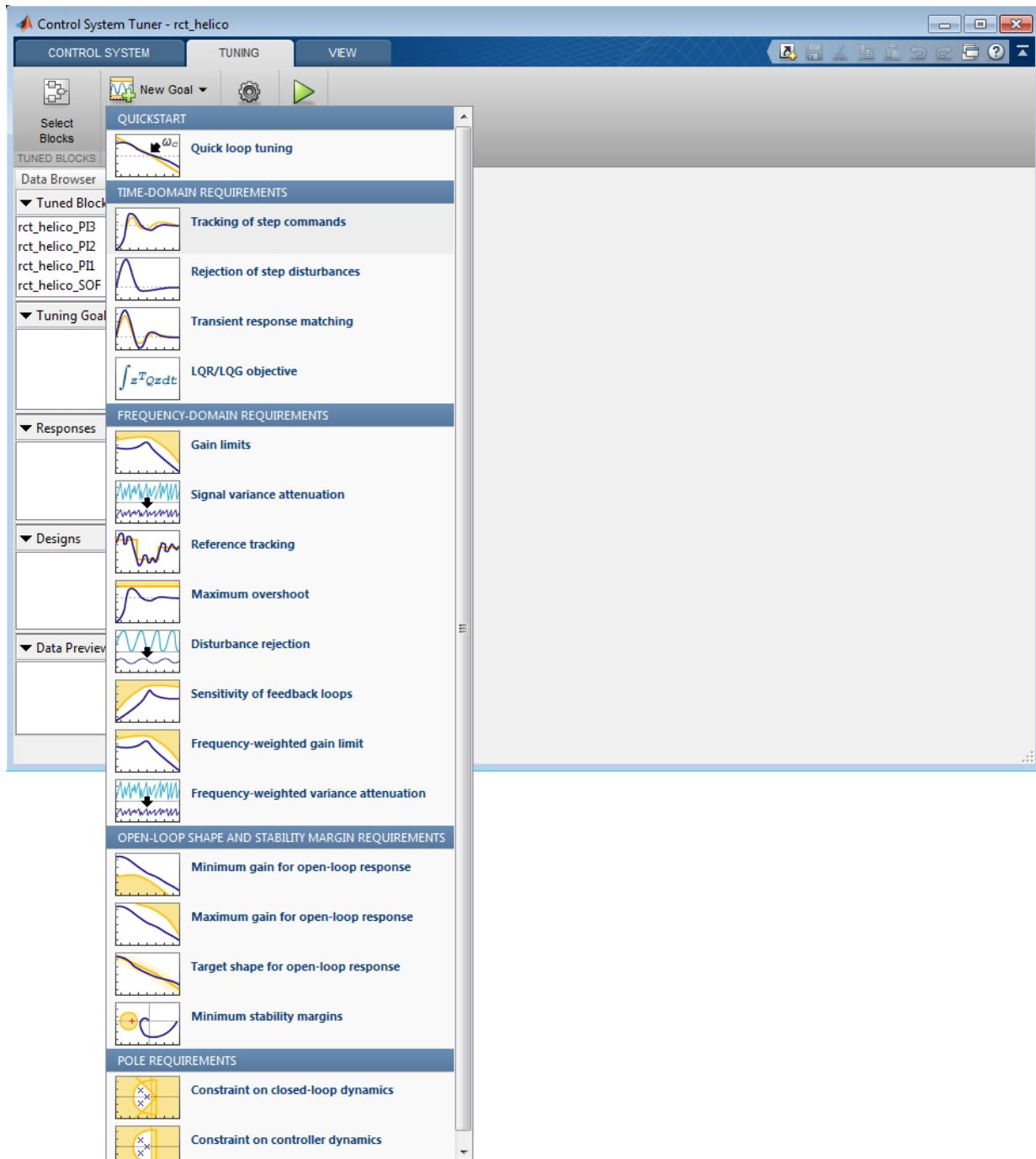
Specify Tuning Goals
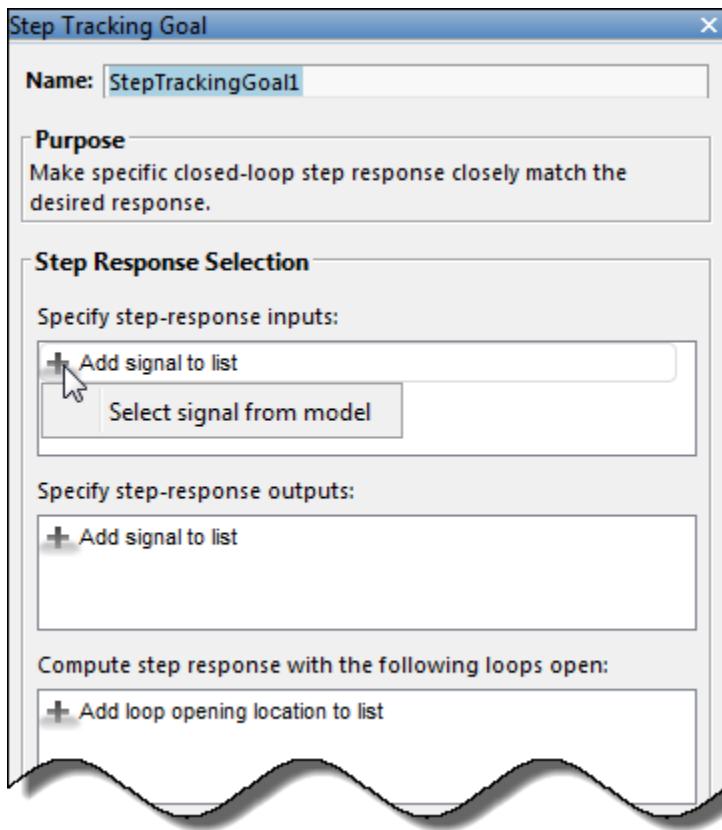
The design requirements for this system, discussed previously, include setpoint tracking, minimum stability margins, and a limit on fast dynamics. In Control System Tuner, you capture design requirements using *tuning goals*.

First, create a tuning goal for the setpoint-tracking requirement on `theta`, `phi`, and `r`. On the **Tuning** tab, in the **New Goal** drop-down list, select `Tracking of step commands`.

In the Step Tracking Goal dialog, specify the reference signals for tracking. Under **Specify step-response inputs**, click **Add signal to list**. Then click **Select signal from model**.
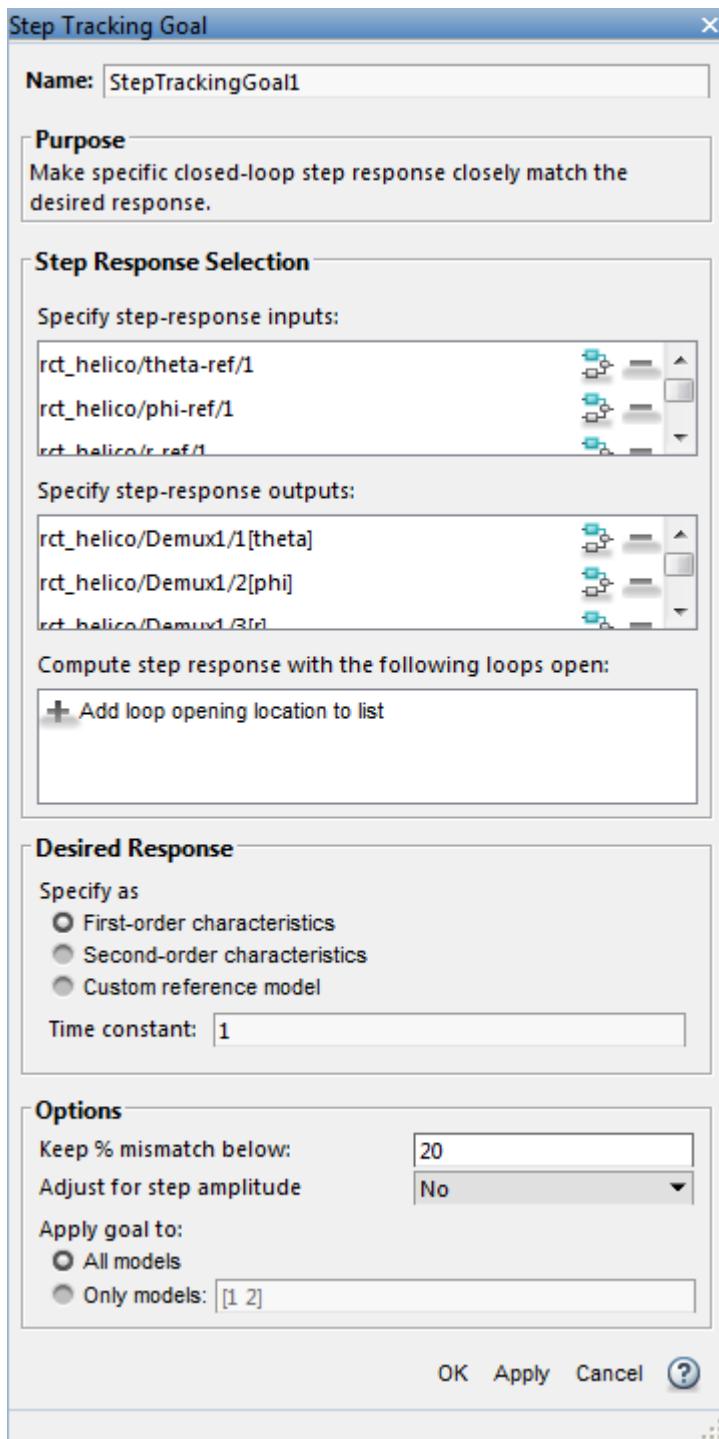

In the Simulink model editor, select the reference signals `theta_ref`, `phi_ref`, and `r_ref`. These signals appear in the Select signals dialog box. Click **Add Signal(s)** to add them to the step tracking goal.

Next, specify the outputs that you want to track those references. Under **Specify step-response outputs**, add the outputs `theta`, `phi`, and `r`.
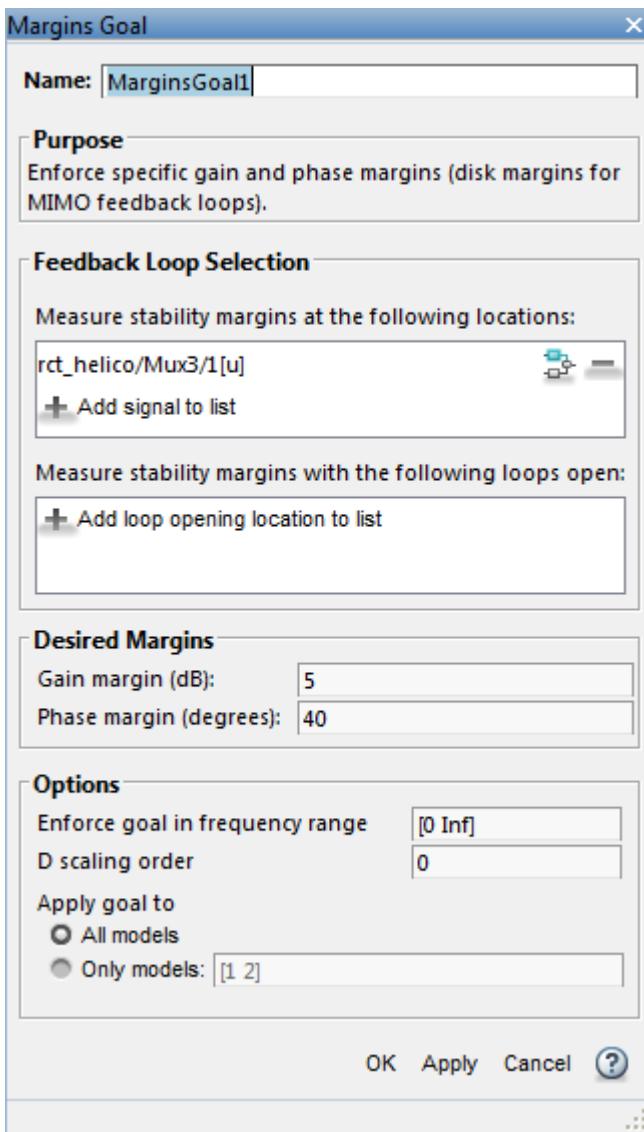
The requirement is that the responses at the outputs track the reference commands with a first-order response that has a one-second time constant. Enter these values in the

Desired Response section of the dialog box. Also, for this example set **Keep mismatch below** to 20. This value sets a 20% relative mismatch between the target first-order response and the tuned response.
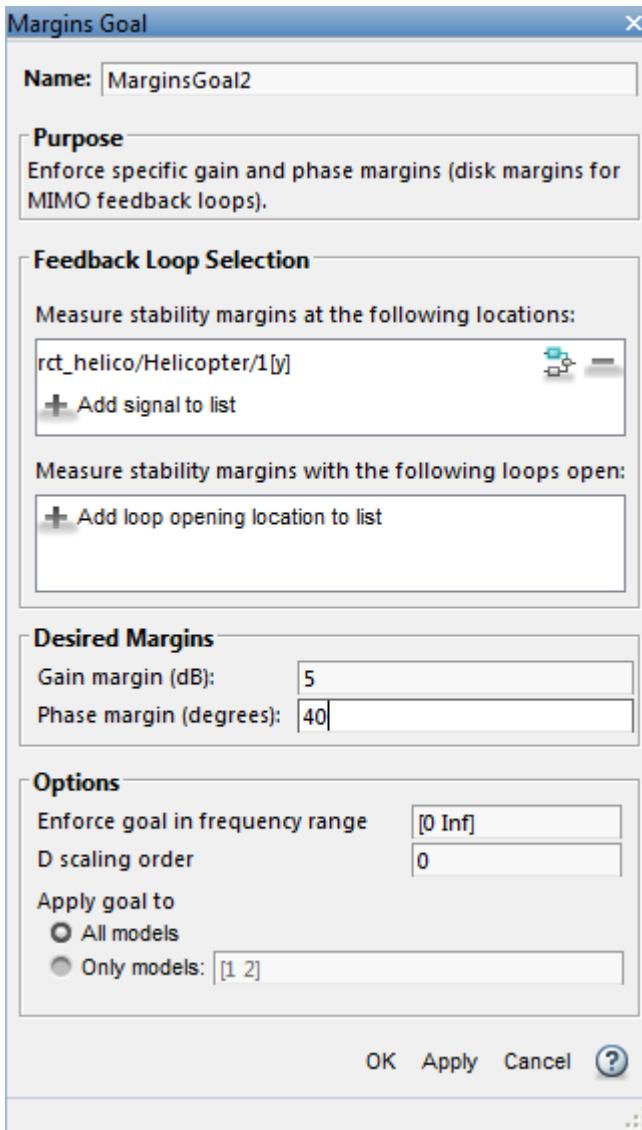
This figure shows the configuration of the Step Tracking Goal dialog box. Click **OK** to save the tuning goal.

Step Tracking Goal   ✕

Name: StepTrackingGoal1

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

Step Response Selection

Specify step-response inputs:

rct_helico/theta-ref/1	
rct_helico/phi-ref/1	
rct_helico/r_ref/1	

Specify step-response outputs:

rct_helico/Demux1/1[theta]	
rct_helico/Demux1/2[phi]	
rct_helico/Demux1/3[r]	

Compute step response with the following loops open:

➕ Add loop opening location to list

Desired Response

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

Time constant: 1

Options

Keep % mismatch below: 20

Adjust for step amplitude: No ▼

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

OK   Apply   Cancel ⑦

Next, create tuning goals for the desired stability margin requirements. For this example, the multivariable gain and phase margins at the plant inputs u and plant outputs y must be at least 5 dB and 40 degrees. Create separate tuning goals for the input and output margin constraints. In the **New Goal** drop-down list, select **Minimum stability margins**. In the Margins Goal dialog box, add the input signal u under **Measure stability margins at the following locations**. Also, enter the gain and phase values 5 and 40 in the **Desired Margins** section of the dialog box. Click **OK** to save the input stability margin goal.
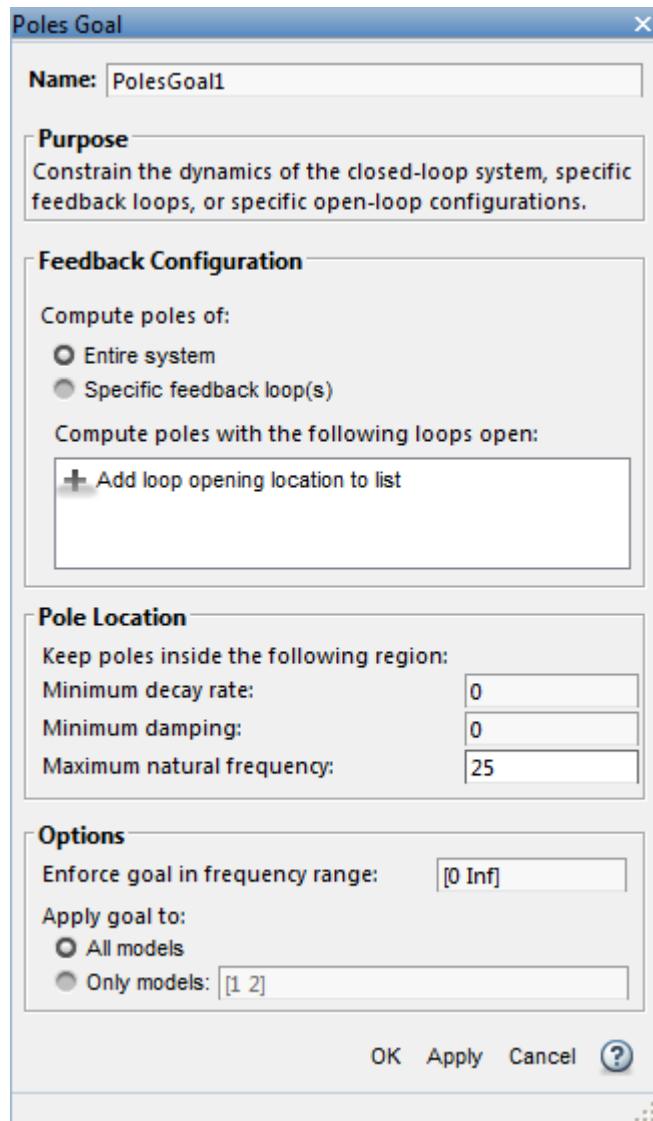
Create another Margins Goal for the output stability margin. Specify the output signal y and the target margins, as shown, and save the output stability margin goal.
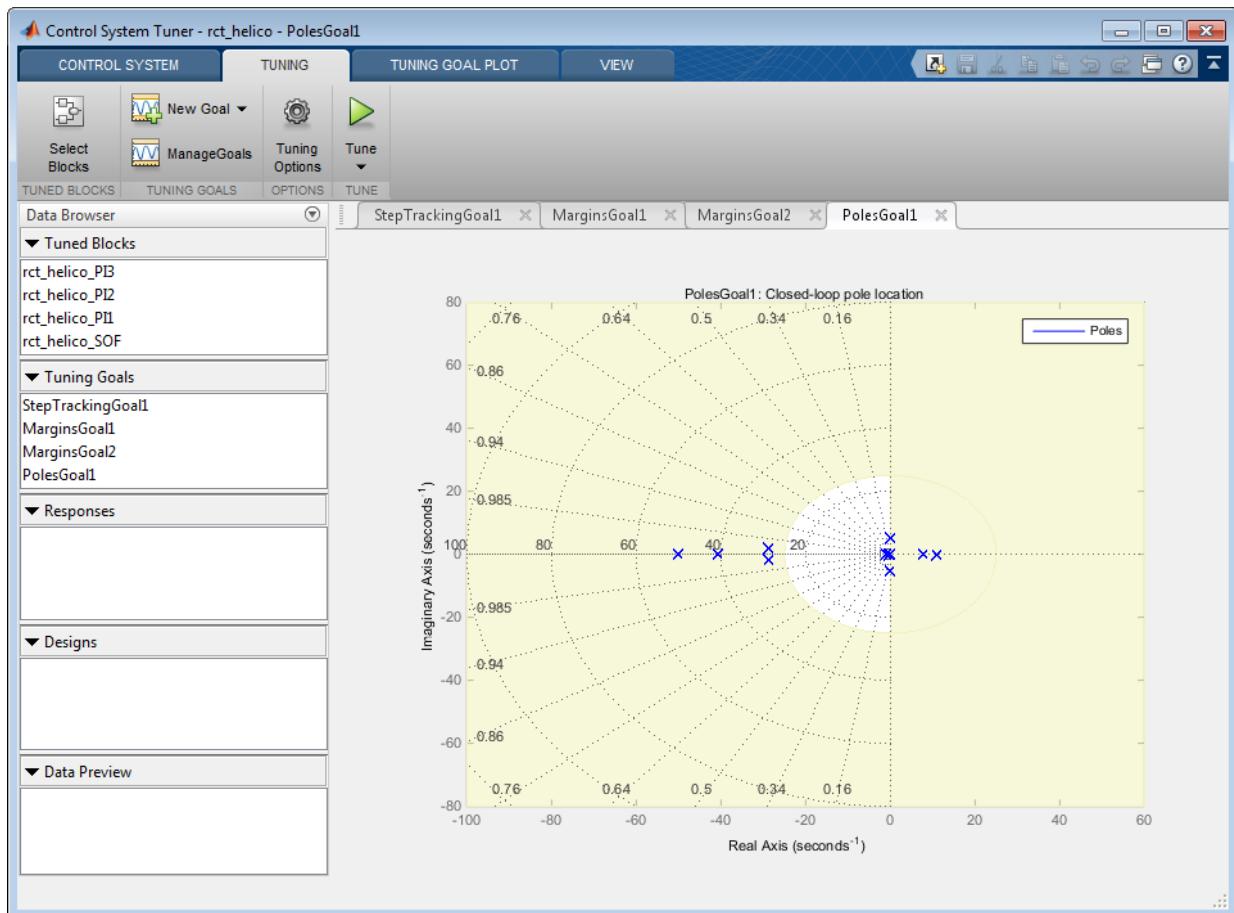
The last requirement is to limit fast dynamics and jerky transients. To achieve this, create a tuning goal that constrains the magnitude of the closed-loop poles to less than 25 rad/s. In the **New Goal** drop-down list, select **Constraint on closed-loop dynamics**.

In the **Poles Goal** dialog box, specify the maximum natural frequency of 25, and click **OK** to save the tuning goal.

As you create each tuning goal, Control System Tuner creates a new figure that displays a graphical representation of the tuning goal. When you tune your control system, you can refer to this figure for a graphical representation of how closely the tuned system satisfies the tuning goal.
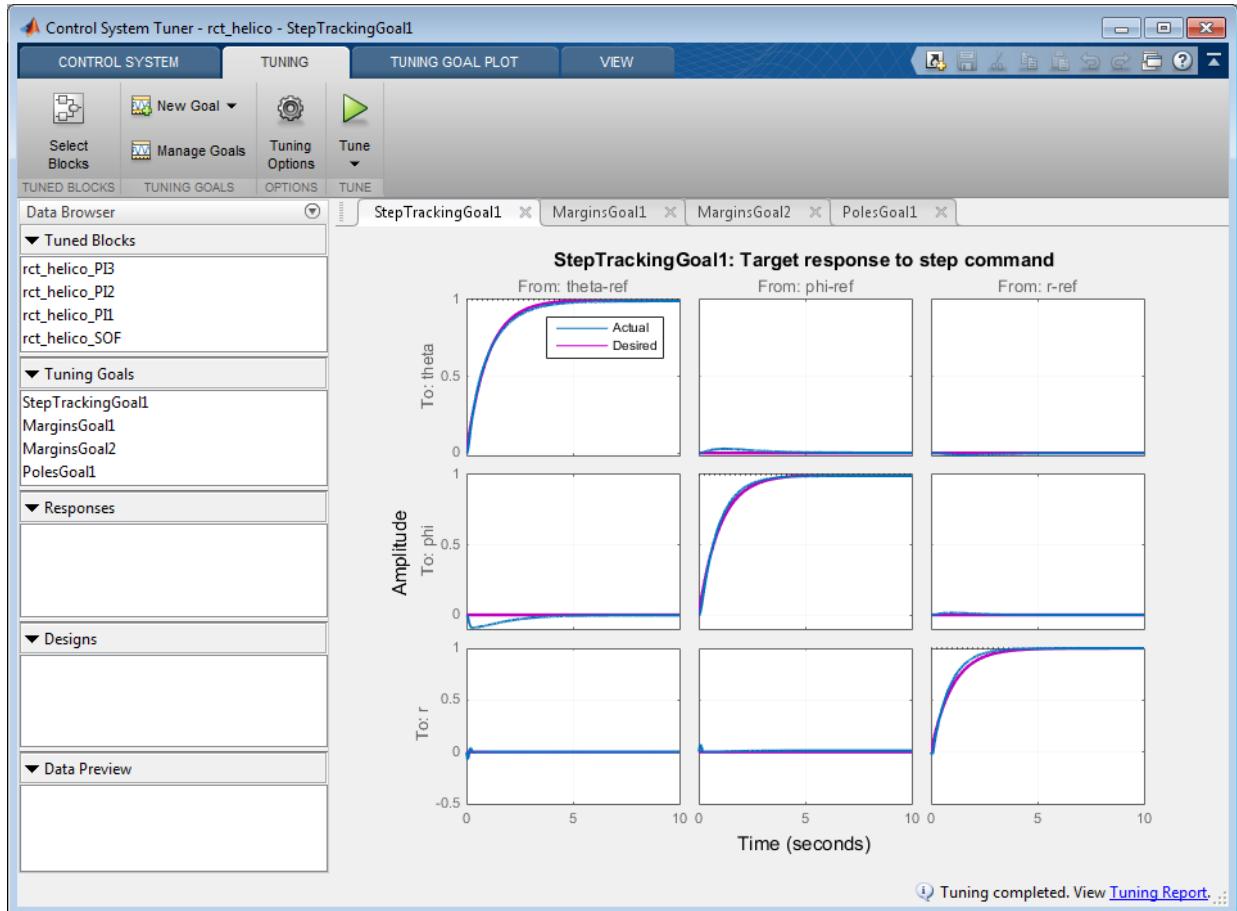
Tune the Control System

Tune the control system to meet the design requirements you have specified.
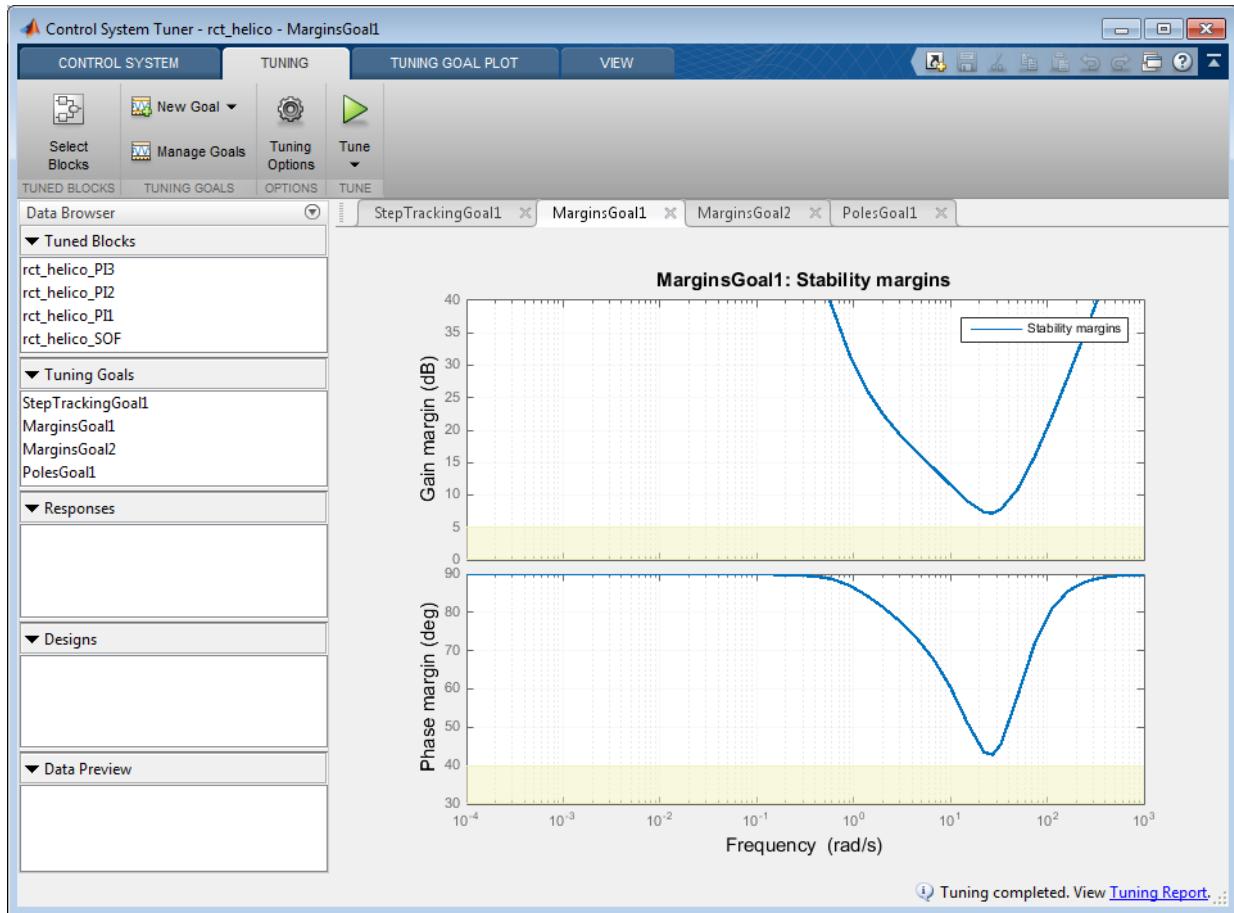
On the **Tuning** tab, click **Tune**. Control System Tuner adjusts the tunable parameters to values that best meet those requirements.

Control System Tuner automatically updates the tuning-goal plots to reflect the tuned parameter values. Examine these plots to see how well the requirements are satisfied by the design. For instance, examine the tuned step responses of tracking requirements.
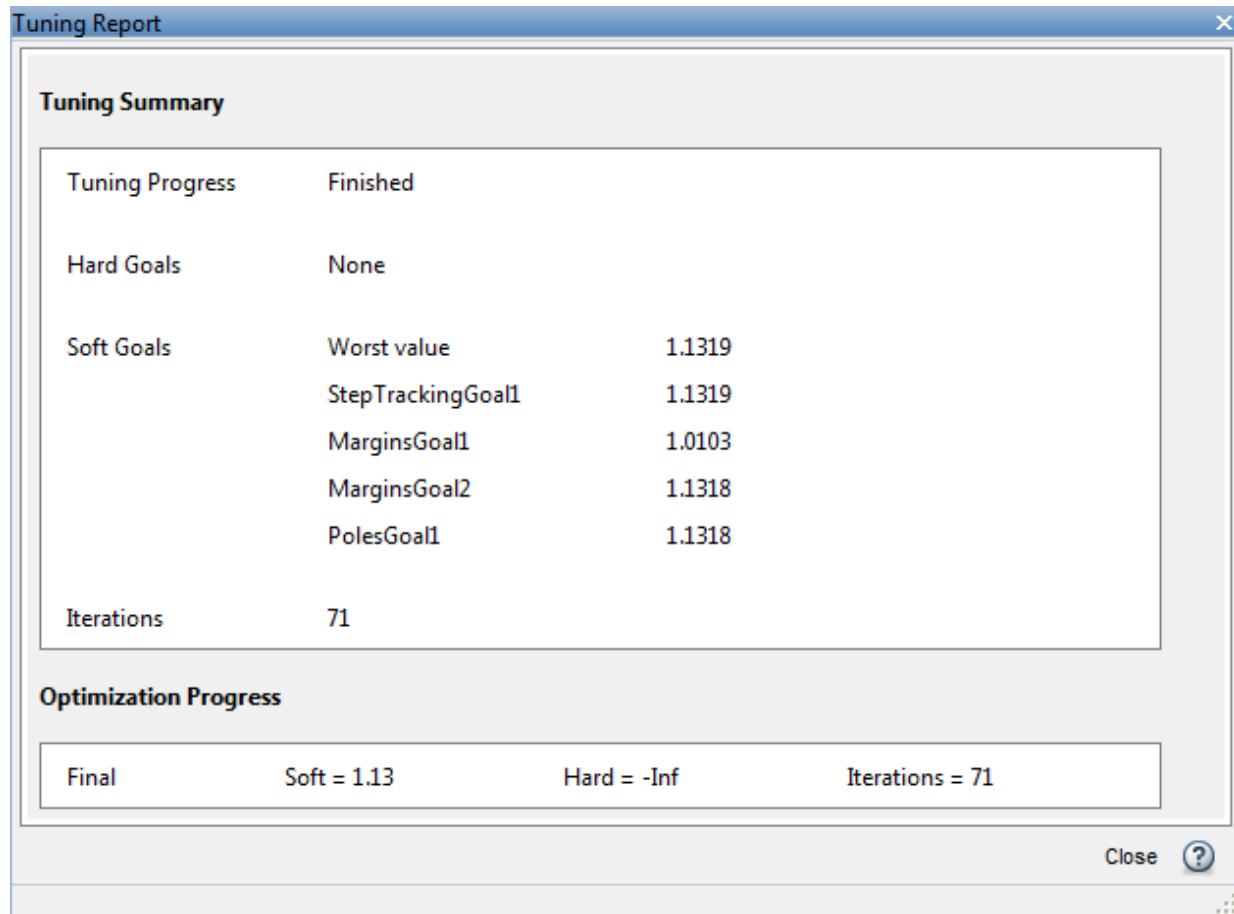
The blue line shows that the tuned response is very close to the target response, in pink. The rise time is about two seconds, and there is no overshoot and little cross-coupling.

Similarly, the `MarginsGoal1` and `MarginsGoal2` plots provide a visual assessment of the multivariable stability margins. (See the `loopmargin` reference page for more information about multivariable stability margins.) These plots show that the stability margin is out of the shaded region, satisfying the requirement at all frequencies.

You can also view a numeric report of the tuning results. Click the **Tuning Report** at the bottom right of Control System Tuner.

Tuning Report   ✕

Tuning Summary

Tuning Progress	Finished	
Hard Goals	None	
Soft Goals	Worst value	1.1319
	StepTrackingGoal1	1.1319
	MarginsGoal1	1.0103
	MarginsGoal2	1.1318
	PolesGoal1	1.1318
Iterations	71	

Optimization Progress

Final	Soft = 1.13	Hard = -Inf	Iterations = 71

Close   ⑦
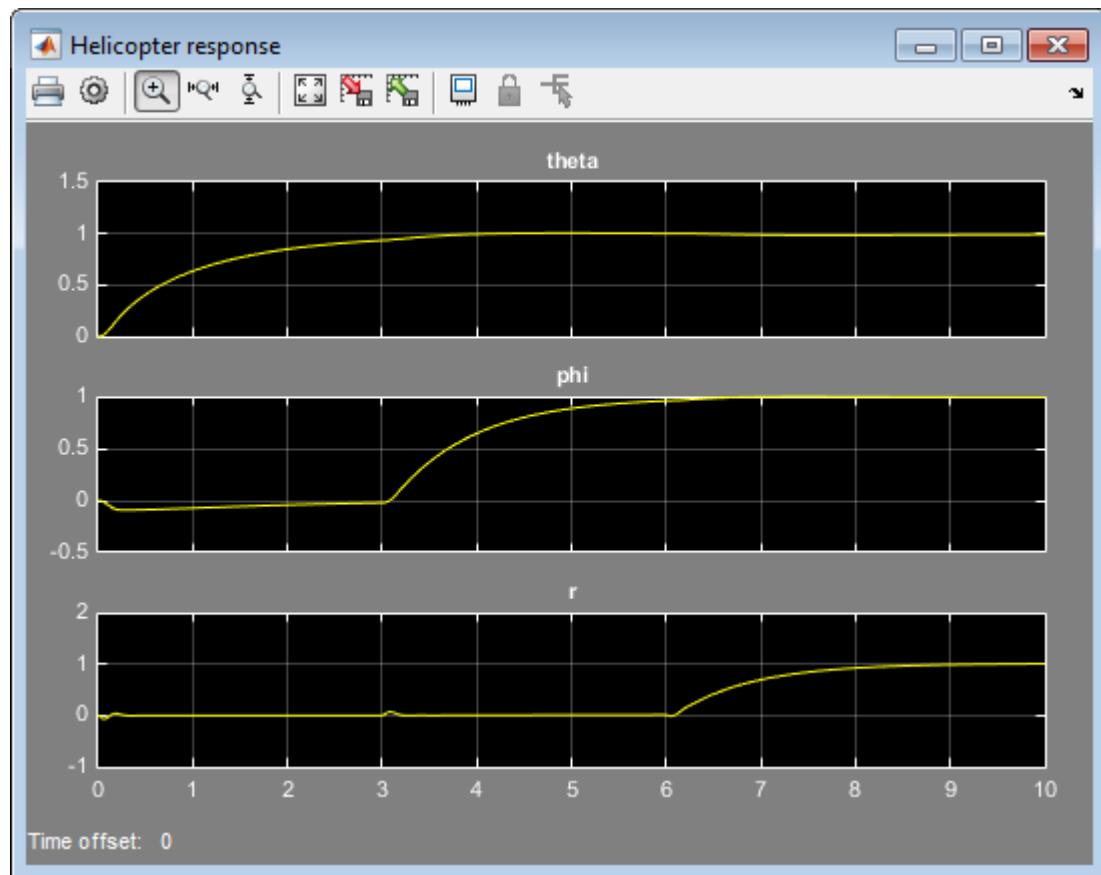
When you tune the model, Control System Tuner converts each tuning goal to a function of the tunable parameters of the system and adjusts the parameters to minimize the value of those functions. For this example, the tuning report shows that the final values for all tuning goals are close to 1, which indicates that all the requirements are nearly met.

Validate the Tuned Design

Control System Tuner tunes the parameters for a linearization of the control system. Therefore, it is important to validate the design on the full nonlinear Simulink model.

Write the tuned parameter values back to the Simulink model. On the **Control System** tab, click **Update Blocks**. In the Simulink model window, simulate the model with the new parameter values. Observe the response to the step changes in setpoint commands, `theta-ref`, `phi-ref`, and `r-ref` at 0, 3, and 6 seconds respectively.

The responses of the nonlinear system are satisfactory. The rise time of each response is about 2 seconds with no overshoot, no steady-state error, and minimal cross-coupling, as specified in the design requirements.

Validating Results

This example shows how to interpret and validate tuning results from `systune`.

Background
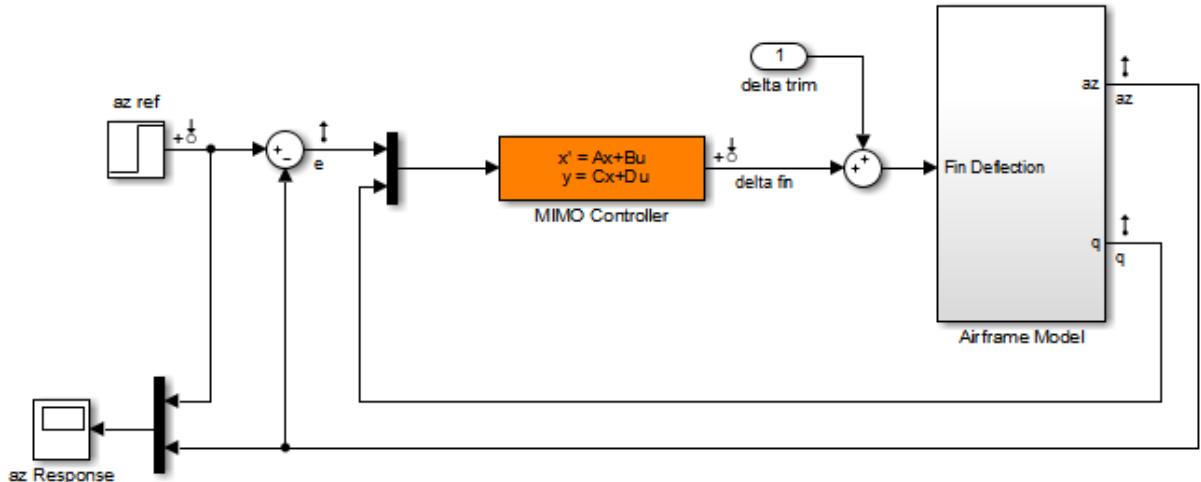
You can tune the parameters of your control system with `systune` or `looptune`. The design specifications are captured using `TuningGoal` requirement objects. This example shows how to interpret the results from `systune`, graphically verify the design requirements, and perform additional open- and closed-loop analysis.

Controller Tuning with SYSTUNE

We use an autopilot tuning application as illustration, see the *"Tuning of a Two-Loop Autopilot"* example for details. The tuned compensator is the "MIMO Controller" block highlighted in orange in the model below.

```
open_system('rct_airframe2')
```

Two-loop autopilot for controlling the vertical acceleration of an airframe

The setup and tuning steps are repeated below for completeness.

```
STO = slTuner('rct_airframe2','MIMO Controller');

% Compensator parameterization
```

```
CO = tunableSS('C',2,1,2);
CO.d.Value(1) = 0;   CO.d.Free(1) = false;
setBlockParam(STO,'MIMO Controller',CO)

% Requirements
Req1 = TuningGoal.Tracking('az ref','az',1);                  % tracking
Req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0])); % roll-off
Req3 = TuningGoal.Margins('delta fin',7,45);                  % margins
MaxGain = frd([2 200 200],[0.02 2 200]);
Req4 = TuningGoal.Gain('delta fin','az',MaxGain);  % disturbance rejection

% Tuning
Opt = systuneOptions('RandomStart',3);
rng('default')
[ST1,fSoft] = systune(STO,[Req1,Req2,Req3,Req4],Opt);

Final: Soft = 1.5, Hard = -Inf, Iterations = 63
Final: Soft = 1.49, Hard = -Inf, Iterations = 108
Final: Soft = 1.15, Hard = -Inf, Iterations = 69
Final: Soft = 1.15, Hard = -Inf, Iterations = 105
```

Interpreting Results

`systune` run three optimizations from three different starting points and returned the best overall result. The first output `ST` is an `slTuner` interface representing the tuned control system. The second output `fSoft` contains the final values of the four requirements for the best design.

`fSoft`
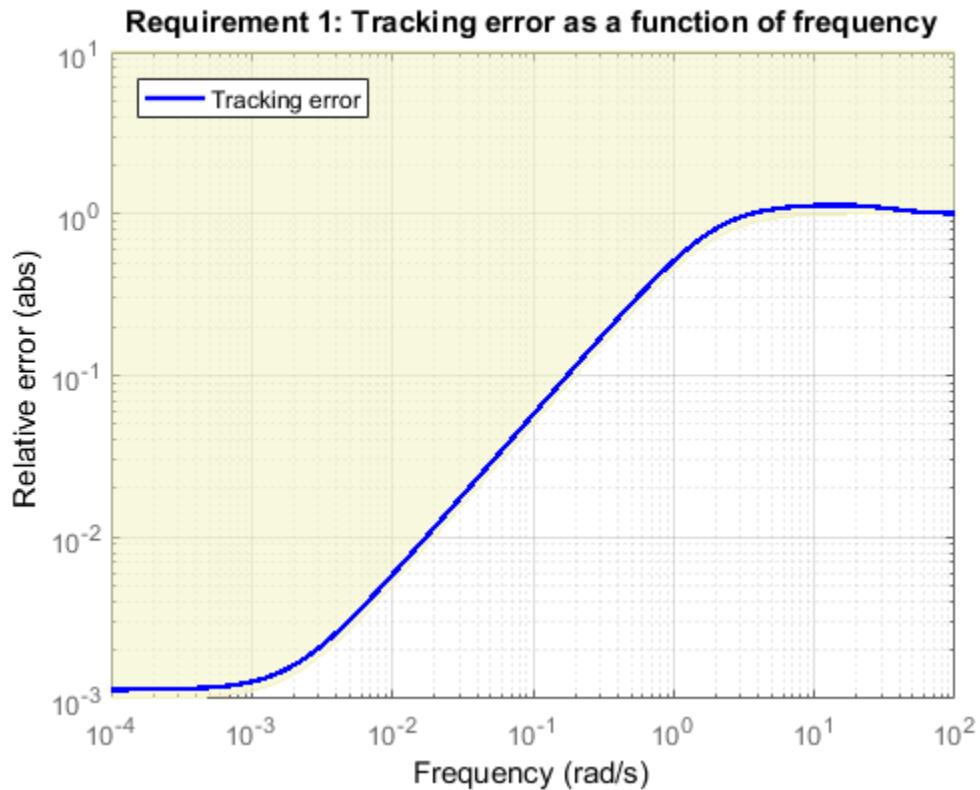
```
fSoft =

    1.1477    1.1477    0.5458    1.1477
```

Requirements are normalized so a requirement is satisfied if and only if its value is less than 1. Inspection of `fSoft` reveals that Requirements 1,2,4 are active and slightly violated while Requirement 3 (stability margins) is satisfied.

Verifying Requirements
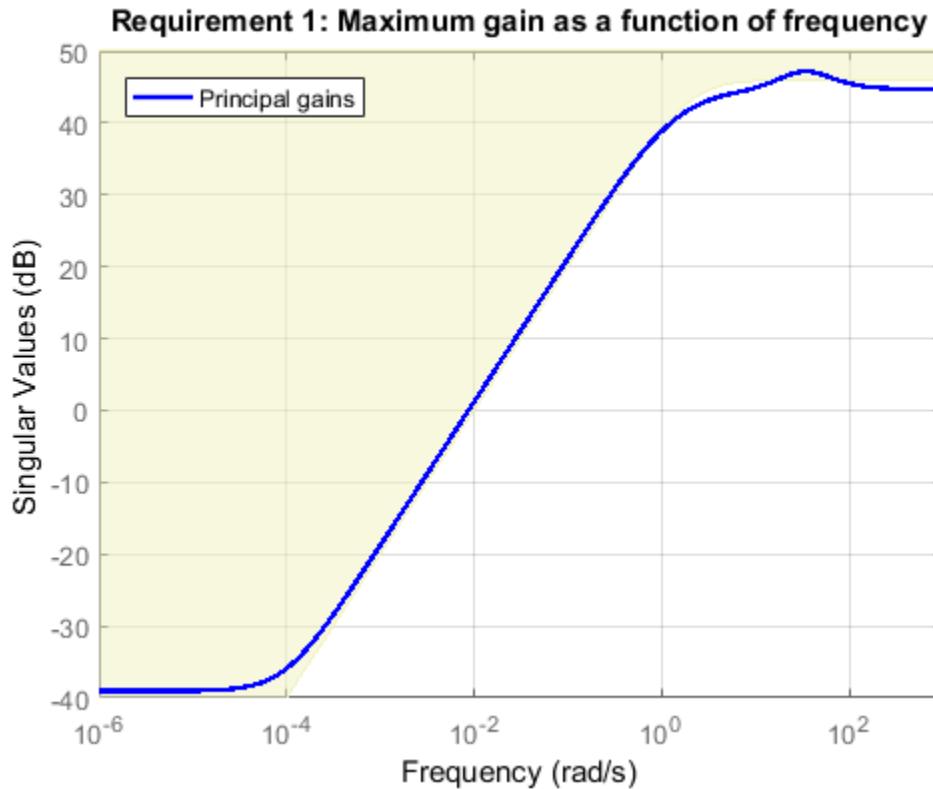
Use `viewSpec` to graphically inspect each requirement. This is useful to understand whether small violations are acceptable or what causes large violations. First verify the tracking requirement.

```
viewSpec(Req1,ST1)
```


Requirement 1: Tracking error as a function of frequency
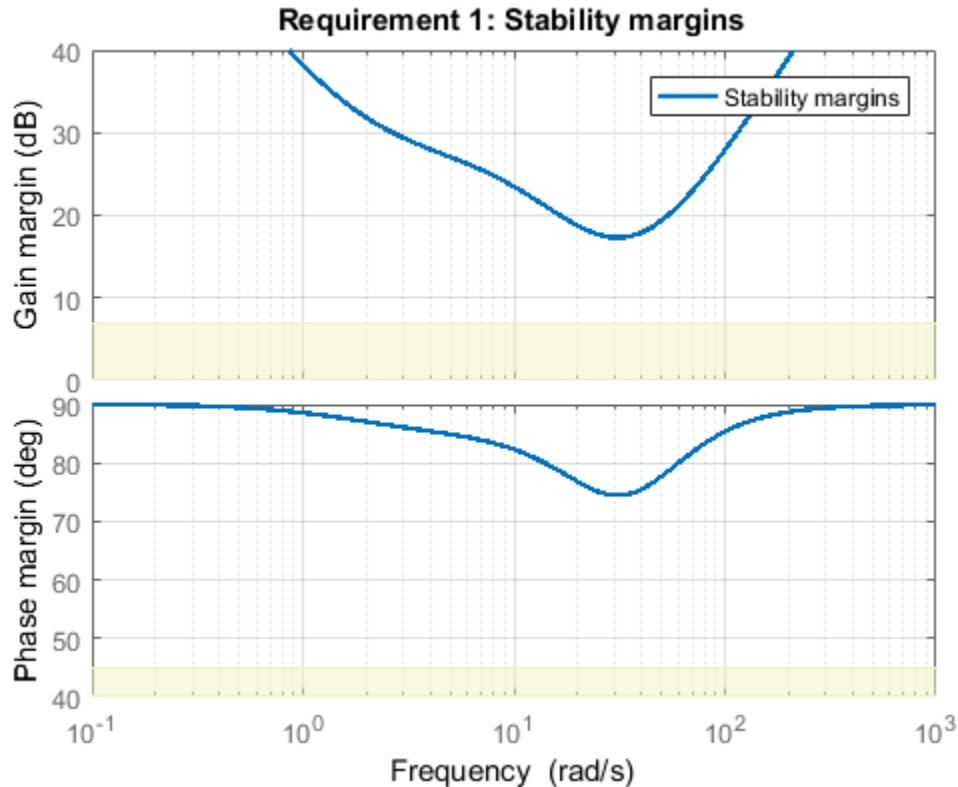
We observe a slight violation across frequency, suggesting that setpoint tracking will perform close to expectations. Similarly, verify the disturbance rejection requirement.

```
viewSpec(Req4,ST1)
legend('location','NorthWest')
```

Requirement 1: Maximum gain as a function of frequency

Most of the violation is at low frequency with a small bump near 35 rad/s, suggesting possible damped oscillations at this frequency. Finally, verify the stability margin requirement.

```
viewSpec(Req3,ST1)
```

Requirement 1: Stability margins

This requirement is satisfied at all frequencies, with the smallest margins achieved near the crossover frequency as expected.

Evaluating Requirements

You can also use `evalSpec` to evaluate each requirement, that is, compute its contribution to the soft and hard constraints. For example

```
[H1,f1] = evalSpec(Req1,ST1);
```

returns the value `f1` of the requirement and the underlying frequency-weighted transfer function `H1` used to computed it. You can verify that `f1` matches the first entry of `fSoft` and coincides with the peak gain of `H1`.
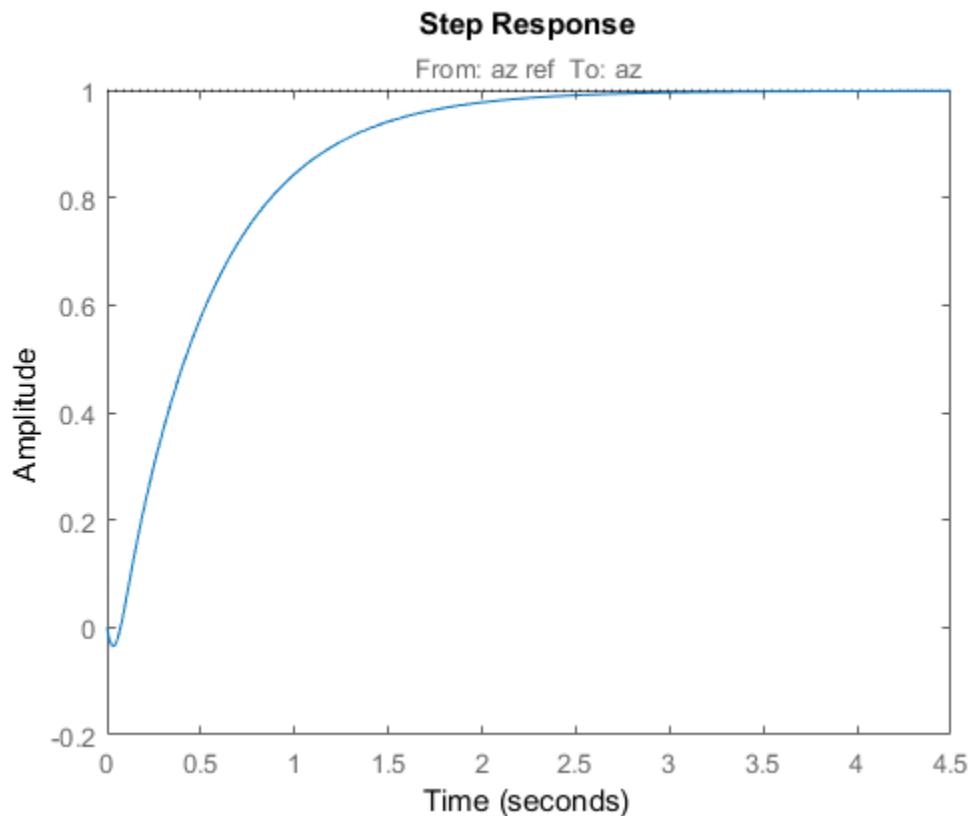
```
[f1 fSoft(1) getPeakGain(H1,1e-6)]


ans =

    1.1477    1.1477    1.1477
```

Analyzing System Responses
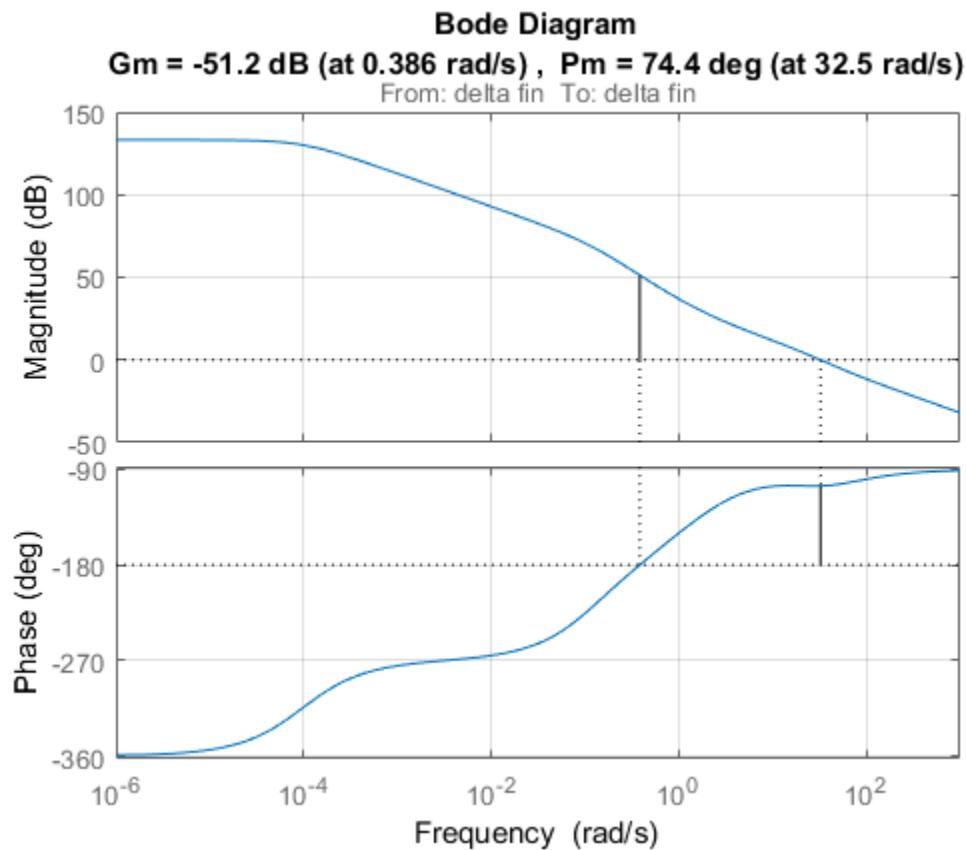
In addition to verifying requirements, you can perform basic open- and closed-loop analysis using `getIOTransfer` and `getLoopTransfer`. For example, verify tracking performance in the time domain by plotting the response `az` to a step command `azref` for the tuned system `ST1`.

```
T = ST1.getIOTransfer('az ref','az');
step(T)
```

Also plot the open-loop response measured at the plant input `delta fin`. You can use this plot to assess the classical gain and phase margins at the plant input.

```
L = ST1.getLoopTransfer('delta fin',-1);  % negative-feedback loop transfer
margin(L), grid
```

Bode Diagram
Gm = -51.2 dB (at 0.386 rad/s) , Pm = 74.4 deg (at 32.5 rad/s)

Soft vs Hard Requirements

So far we have treated all four requirements equally in the objective function. Alternatively, you can use a mix of soft and hard constraints to differentiate between must-have and nice-to-have requirements. For example, you could treat Requirements 3,4 as hard constraints and optimize the first two requirements subject to these constraints. For best results, do this only after obtaining a reasonable design with all requirements treated equally.

```
[ST2,fSoft,gHard] = systune(ST1,[Req1 Req2],[Req3 Req4]);

Final: Soft = 1.31, Hard = 0.99987, Iterations = 141

fSoft
```

```
fSoft =

    1.2253    1.3079


gHard


gHard =

    0.4756    0.9999
```

Here `fSoft` contains the final values of the first two requirements (soft constraints) and `gHard` contains the final values of the last two requirements (hard constraints). The hard constraints are satisfied since all entries of `gHard` are less than 1. As expected, the best value of the first two requirements went up as the optimizer strived to strictly enforce the fourth requirement.

Using Parallel Computing to Accelerate Tuning

This example shows how to leverage the Parallel Computing Toolbox™ to accelerate multi-start strategies for tuning fixed-structure control systems.

Background

Both `systune` and `looptune` use local optimization methods for tuning the control architecture at hand. To mitigate the risk of ending up with a locally optimal but globally poor design, it is recommended to run several optimizations starting from different randomly generated initial points. If you have a multi-core machine or have access to distributed computing resources, you can significantly speed up this process using the Parallel Computing Toolbox.
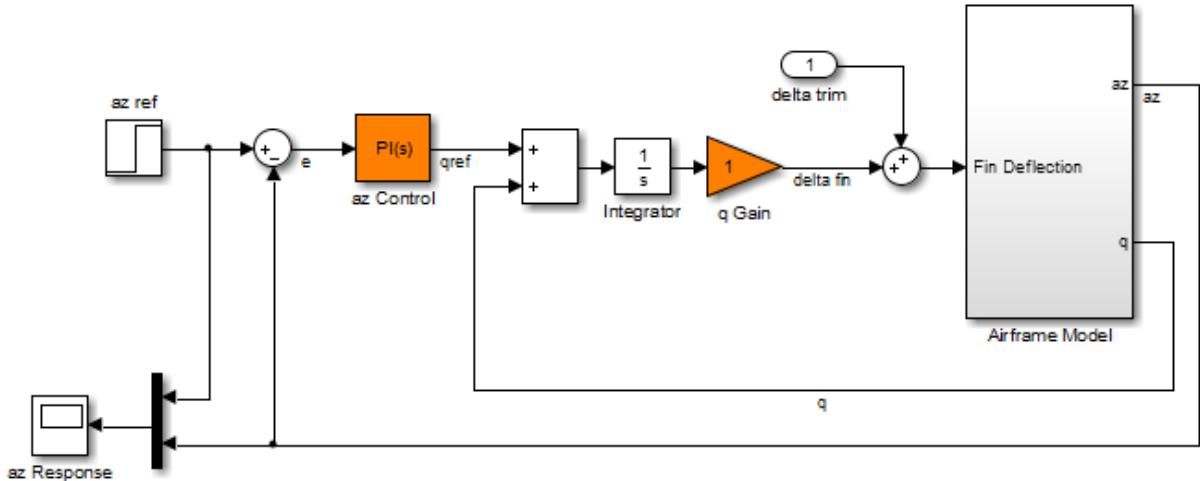
This example shows how to parallelize the tuning of an airframe autopilot with `looptune`. See the example "Tuning of a Two-Loop Autopilot" for more details about this application of `looptune`.

Autopilot Tuning

The airframe dynamics and autopilot are modeled in Simulink.

```
open_system('rct_airframe1')
```

Two-loop autopilot for controlling the vertical acceleration of an airframe

See AutopilotExample to see how you can tune the autopilot gains with the
LOOPTUNE command.

The autopilot consists of two cascaded loops whose tunable elements include two PI
controller gains ("az Control" block) and one gain in the pitch-rate loop ("q Gain" block).
The vertical acceleration az should track the command azref with a 1 second response
time. Use slTuner to configure this tuning task (see "Tuning of a Two-Loop Autopilot"
example for details):

```
STO = slTuner('rct_airframe1',{'az Control','q Gain'});
addPoint(STO,{'az ref','delta fin','az','q'})

% Design requirements
wc = [3,12];    % bandwidth
TrackReq = TuningGoal.Tracking('az ref','az',1);   % tracking
```

Parallel Tuning with LOOPTUNE

We are ready to tune the autopilot gains with looptune. To minimize the risk of getting
a poor-quality local minimum, run 30 optimizations starting from 30 randomly generated
values of the three gains. Configure the looptune options to enable parallel processing
of these 30 runs:

```
rng('default')
Options = looptuneOptions('RandomStart',30,'UseParallel',true);
```

Next call `looptune` to launch the tuning algorithm. The 30 runs are automatically distributed across available computing resources:

```
Controls = 'delta fin';
Measurements = {'az','q'};
[ST,gam,Info] = looptune(STO,Controls,Measurements,wc,TrackReq,Options);

Starting parallel pool (parpool) using the 'local' profile ... connected to 4 workers.
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.042)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.039)
Final: Peak gain = 1.23, Iterations = 53
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 62, Iterations = 128
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 135
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Peak gain = 1.23, Iterations = 132
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Peak gain = 62, Iterations = 92
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.04)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.044)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.039)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 61.9, Iterations = 79
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 106
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.051)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 42
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
```

Most runs return 1.23 as optimal gain value, suggesting that this local minimum has a wide region of attraction and is likely to be the global optimum. Use `showBlockValue` to see the corresponding gain values:
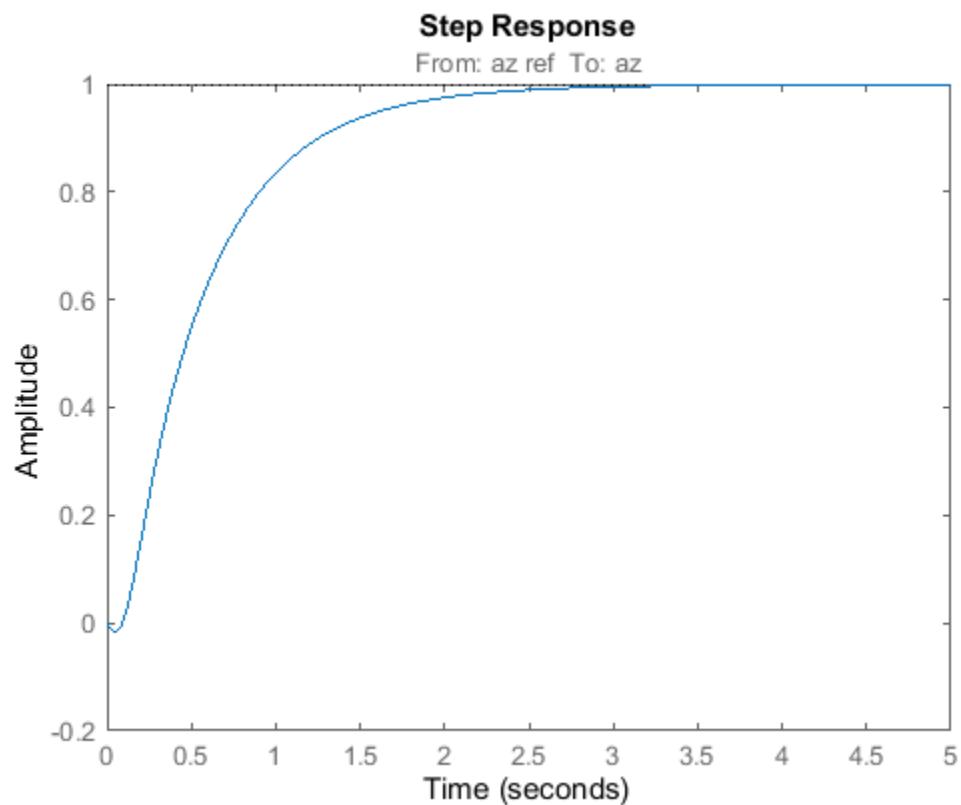
```
showBlockValue(ST)

AnalysisPoints_ =

  d =
       u1  u2  u3  u4
   y1   1   0   0   0
   y2   0   1   0   0
   y3   0   0   1   0
   y4   0   0   0   1

Name: AnalysisPoints_
Static gain.
----------------------------------
az_Control =

            1
  Kp + Ki * ---
            s

  with Kp = 0.00165, Ki = 0.00166

Name: az_Control
Continuous-time PI controller in parallel form.
----------------------------------
q_Gain =

  d =
          u1
   y1  1.985

Name: q_Gain
Static gain.
```

Plot the closed-loop response for this set of gains:

```
T = getIOTransfer(ST,'az ref','az');
step(T,5)
```

Control of a Linear Electric Actuator Using Control System Tuner

This example shows how to use the Control System Tuner app to tune the current and velocity loops in a linear electric actuator with saturation limits.

Linear Electric Actuator Model

Open the Simulink model of the linear electric actuator:

```
open_system('rct_linact')
```


Linear Electric Actuator (System-Level Model)

This linear actuator consists of a DC motor driving a worm gear which in turn drives a lead screw to produce linear motion. This model is a system-level model suitable for motor current and speed controller tuning. The Simulation Mode parameter is set to Averaged for both the Controlled PWM Voltage and H-Bridge blocks, resulting in fast simulation and correct linearization.

The electrical and mechanical components are modeled using Simscape Electronics and Simscape Multibody. The control system consists of two cascaded feedback loops controlling the driving current and angular speed of the DC motor.

Current Controller

Speed Controller

Figure 1: Current and Speed Controllers.

Note that the inner-loop (current) controller is a proportional gain while the outer-loop (speed) controller has proportional and integral actions. The output of both controllers is limited to plus/minus 5.

Design Specifications

We need to tune the proportional and integral gains to respond to a 2000 rpm speed demand in about 0.1 seconds with minimum overshoot. The initial gain settings in the model are P=50 and PI(s)=0.2+0.1/s and the corresponding response is shown in Figure 2. This response is too slow and too sensitive to load disturbances.

Figure 2: Untuned Response.

Control System Tuning

You can use Control System Tuner to jointly tune both feedback loops. First, open Control System Tuner from Simulink Analysis Menu.
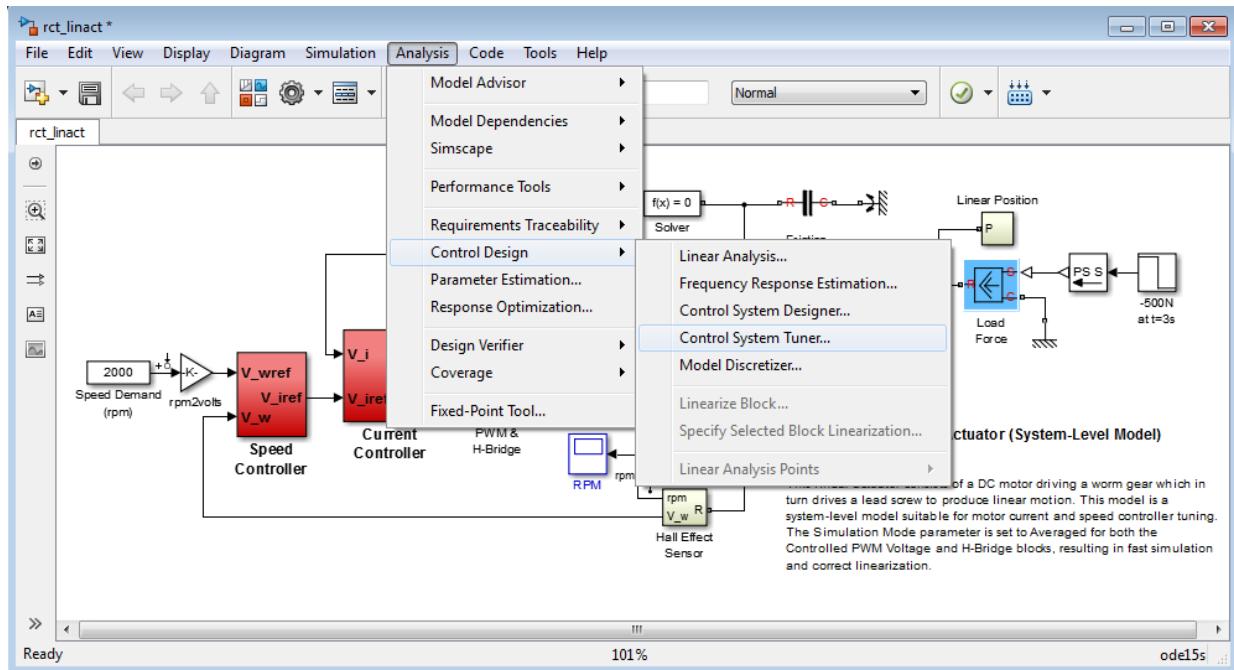
Figure 3: Opening Control System Tuner.

This opens Control System Tuner.
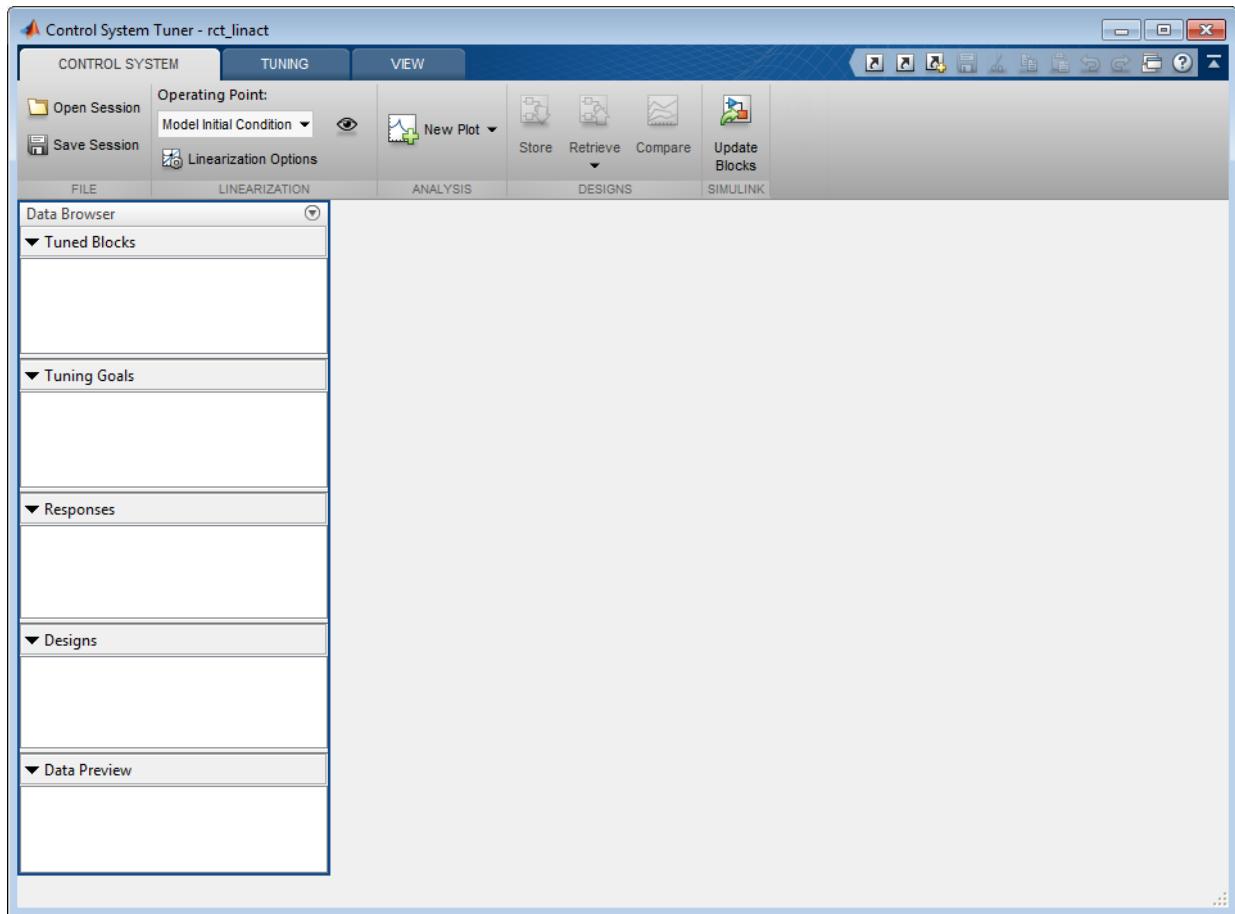
Figure 4: Control System Tuner.

You linearize the model at t=0.5 to avoid discontinuities in some derivatives at t=0. You can set the operating point in `Linearize At...`.
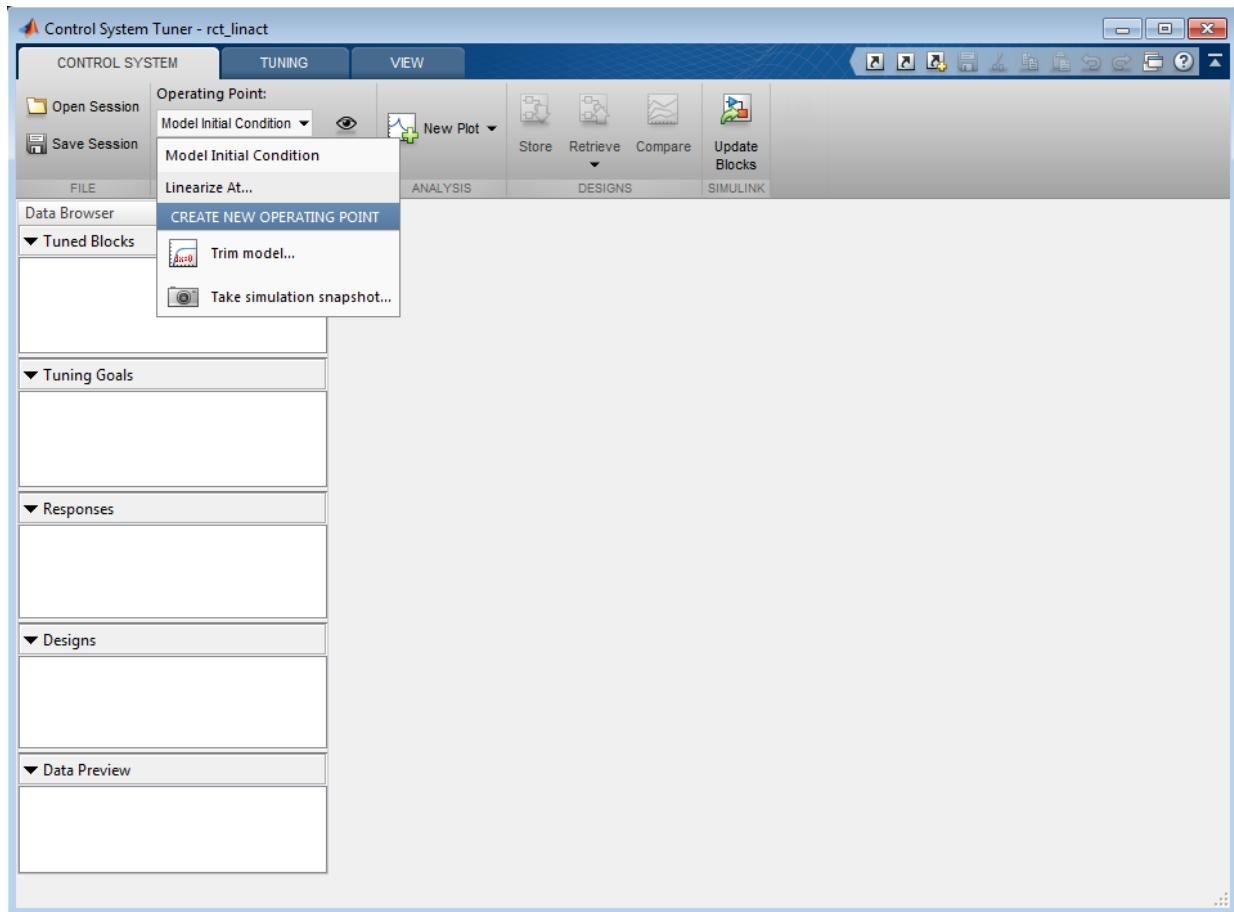
Figure 5: Setting Operating Point for Linearization.

Set the linearization snapshot time at t=0.5.
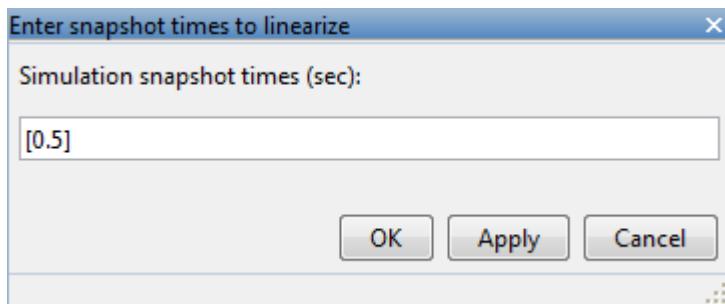
Figure 6: Setting the Linearization Snapshot Time.

In order to set the tuned blocks of the control system, open `Select Blocks` from `Tuning` tab.
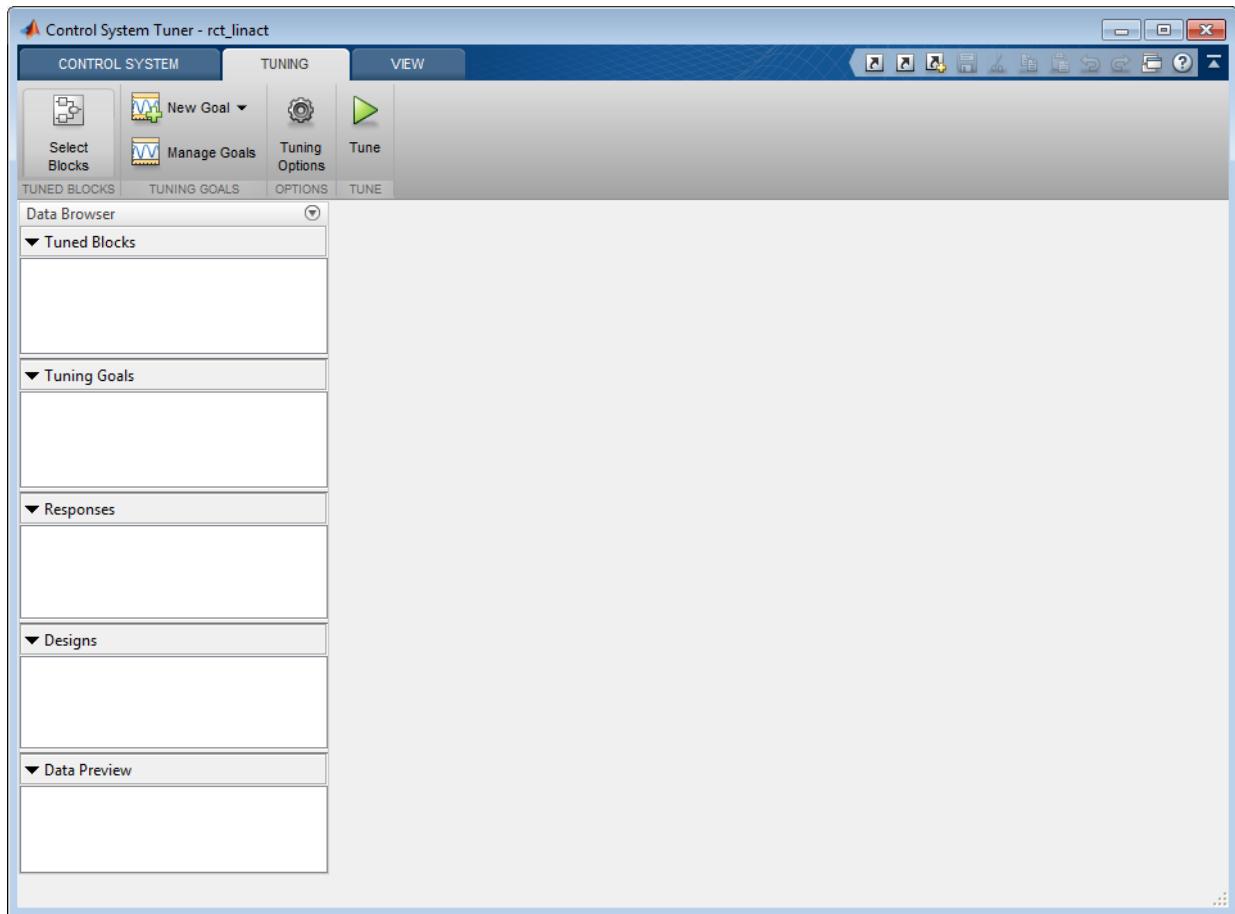
Figure 7: Tuning Tab of Control System Tuner.

This shows the editor for tuned blocks where you can Add Blocks.
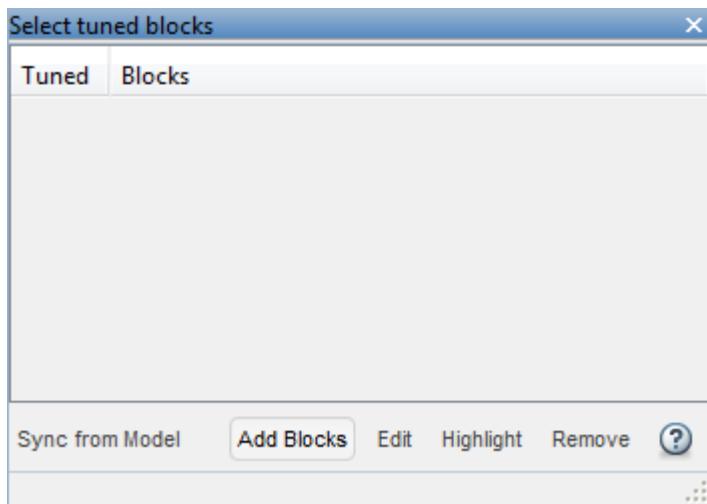
Figure 8: Editor for Tuned Blocks.

Set the tuned blocks Current PID and Speed PID by navigating through the tree on the left.
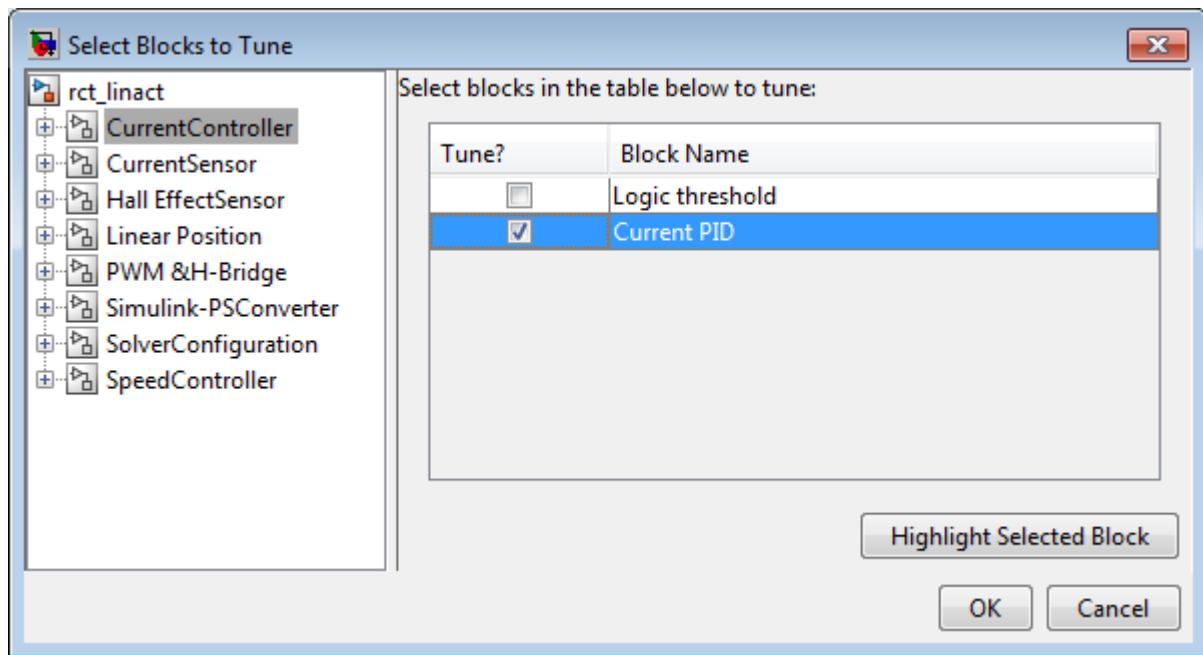
Figure 9: Selecting Tuned Block `Current PID`.
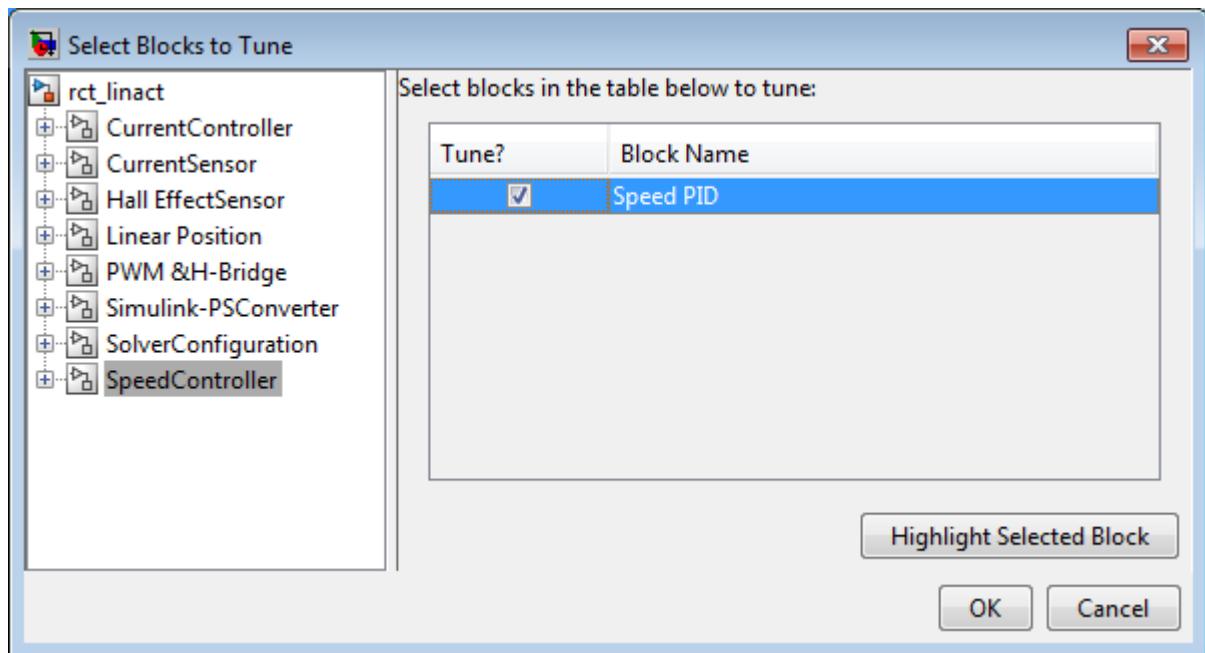
Figure 10: Selecting Tuned Block `Speed PID`.

Selected tuned blocks `Current PID` and `Speed PID` show in the editor for tuned blocks.
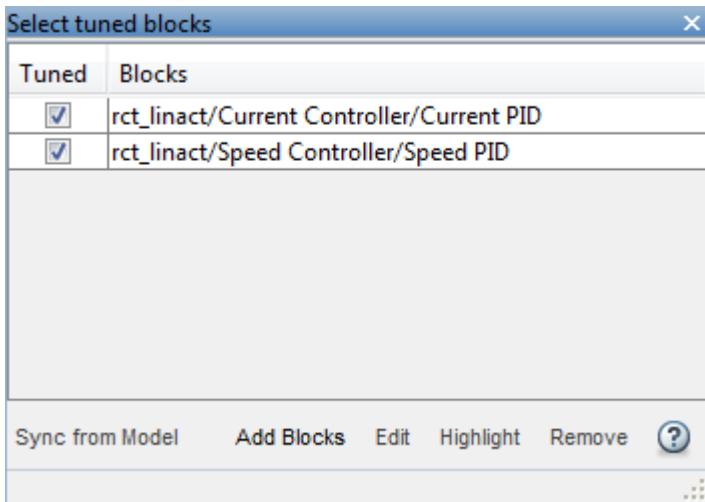
Figure 11: Editor Updated with Selected Tuned Blocks.

They also appear in the Tuned Blocks section of Data Browser on the left side of Control System Tuner.
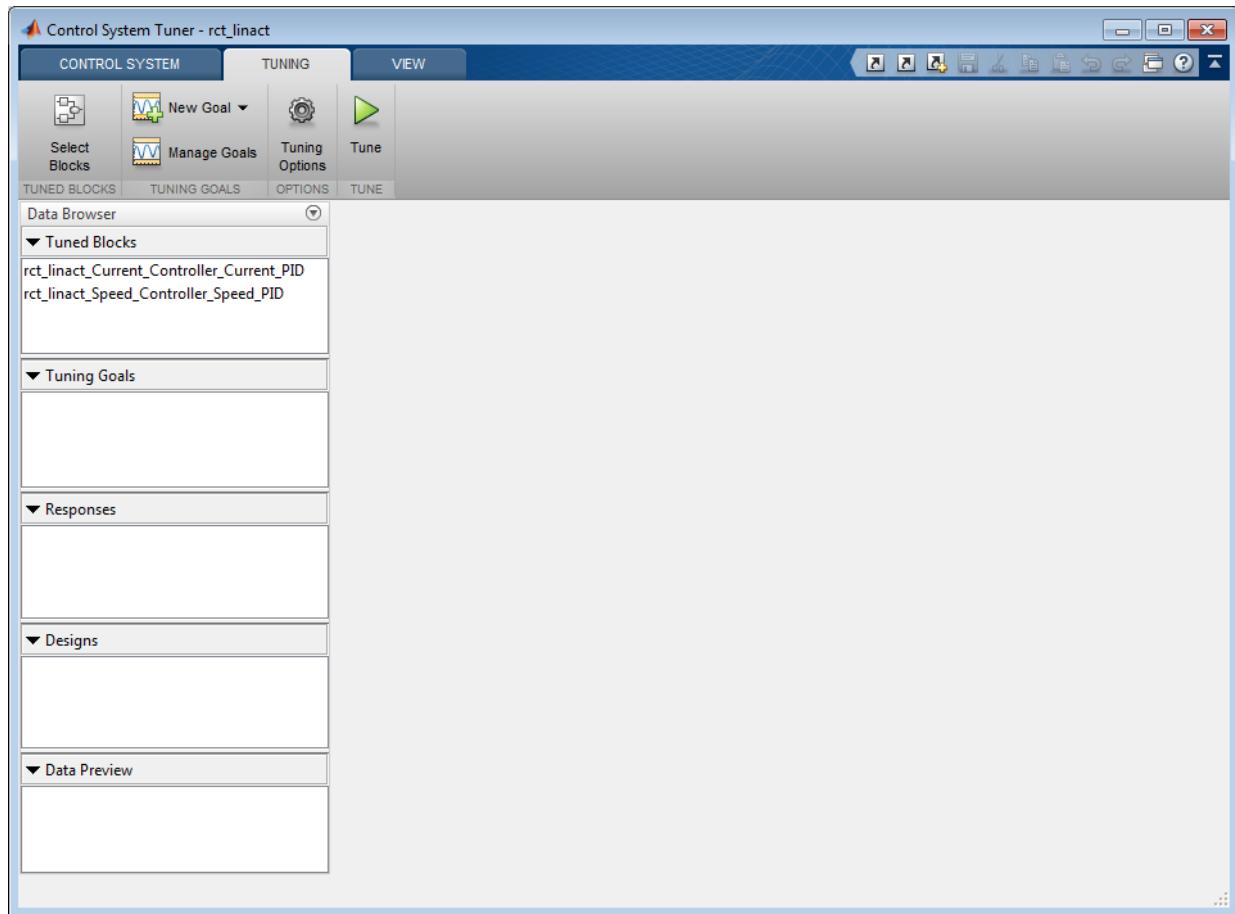
Figure 12: Updated Tuned Blocks in Control System Tuner.

Next specify the tracking goal that the DC motor should follow a 2000 rpm speed demand in 0.1 seconds. See different types of goals under New Goal and select Reference Tracking.
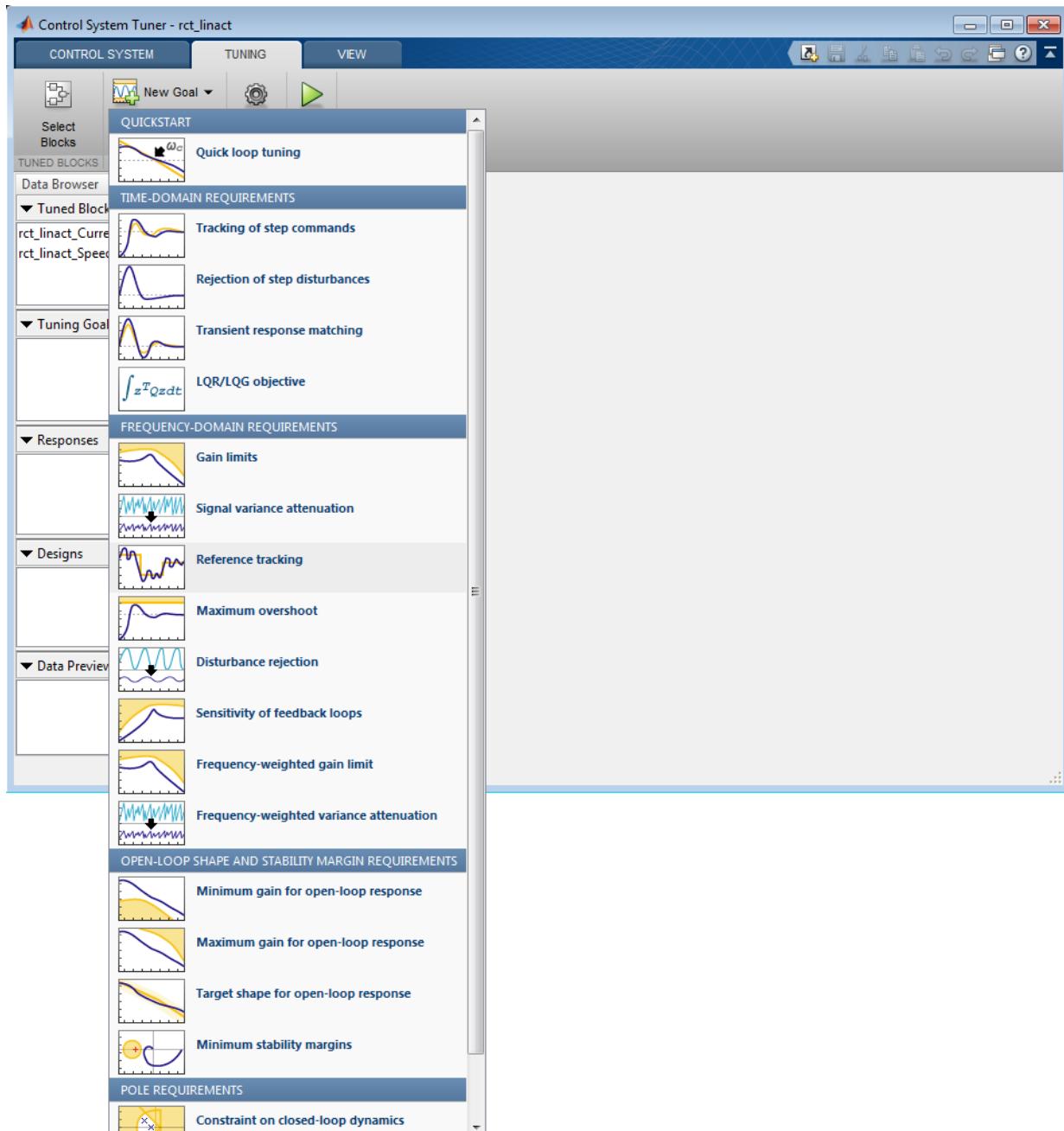
Figure 13: Available Goals for Selection in Control System Tuner.

Name the tracking goal as `TR`, specify the tracking goal from the reference input `rct_linact/Speed Demand(rpm)/1` to the reference-tracking output `rct_linact/Hall Effect Sensor/1[rpm]` with the response time 0.1 seconds.

Reference Tracking Goal　　　　　　　　　　　　　　×

Name: TR

Purpose

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

Response Selection

Specify reference inputs:

rct_linact/Speed Demand (rpm)/1

➕ Add signal to list

Specify reference-tracking outputs:

➕ Add signal to list

> rct_linact/Current Sensor/1[i]
>
> rct_linact/Hall Effect Sensor/1[rpm]
>
> rct_linact/Speed Controller/Speed PID/1　　loops open:
>
> rct_linact/Speed Demand (rpm)/1
>
> Select signal from model

Tracking Performance

Specify as

　⦿ Response time, DC error, and peak error

　◯ Maximum error as a function of frequency

Response Time:　　　　　　　　　0.1

Steady-state error (%):　　　　　0.1

Peak error across frequency (%):　100

Options

Enforce goal in frequency range　　[0 Inf]

Adjust for signal amplitude　　　　No ▼

Apply goal to

　⦿ All models

　◯ Only models: [1 2]

OK　Apply　Cancel　⑦

Figure 14: Reference Tracking Dialog in Control System Tuner.

The plot for specified tracking goal appears in Control System Tuner and `Tuning Goals` section of Data Browser on the left side is updated.

Figure 15: Tracking Tuning Goal in Control System Tuner.

You can now tune the proportional and integral gains with Control System Tuner from clicking `Tune` button. The plot for tracking goal is updated
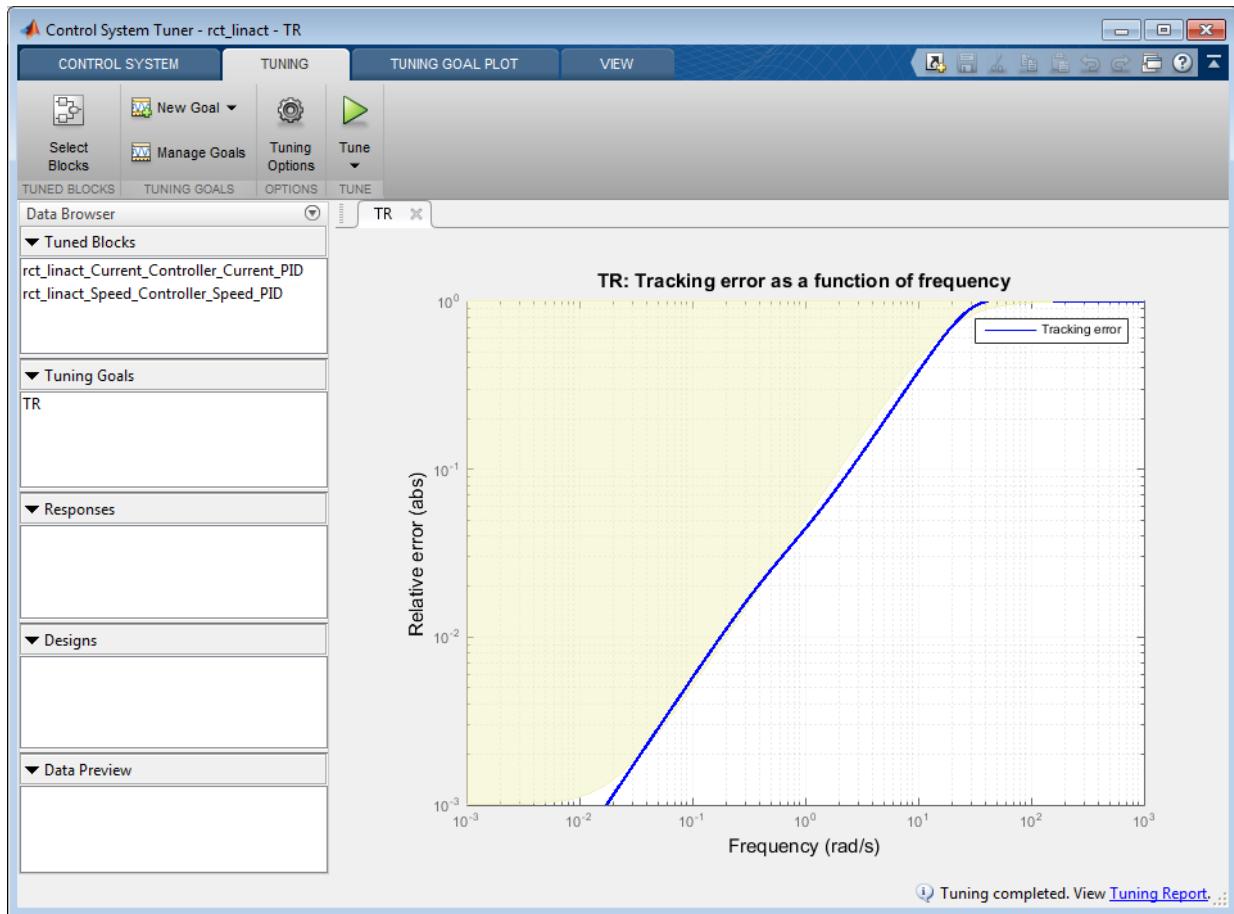
Figure 16: Updated Tracking Goal Plot with Tuned Blocks in Control System Tuner.

Tuned blocks are updated with the tuned gain values. To validate this design, plot the closed-loop response from speed demand to speed from `New Plot` of `Control System` Tab.
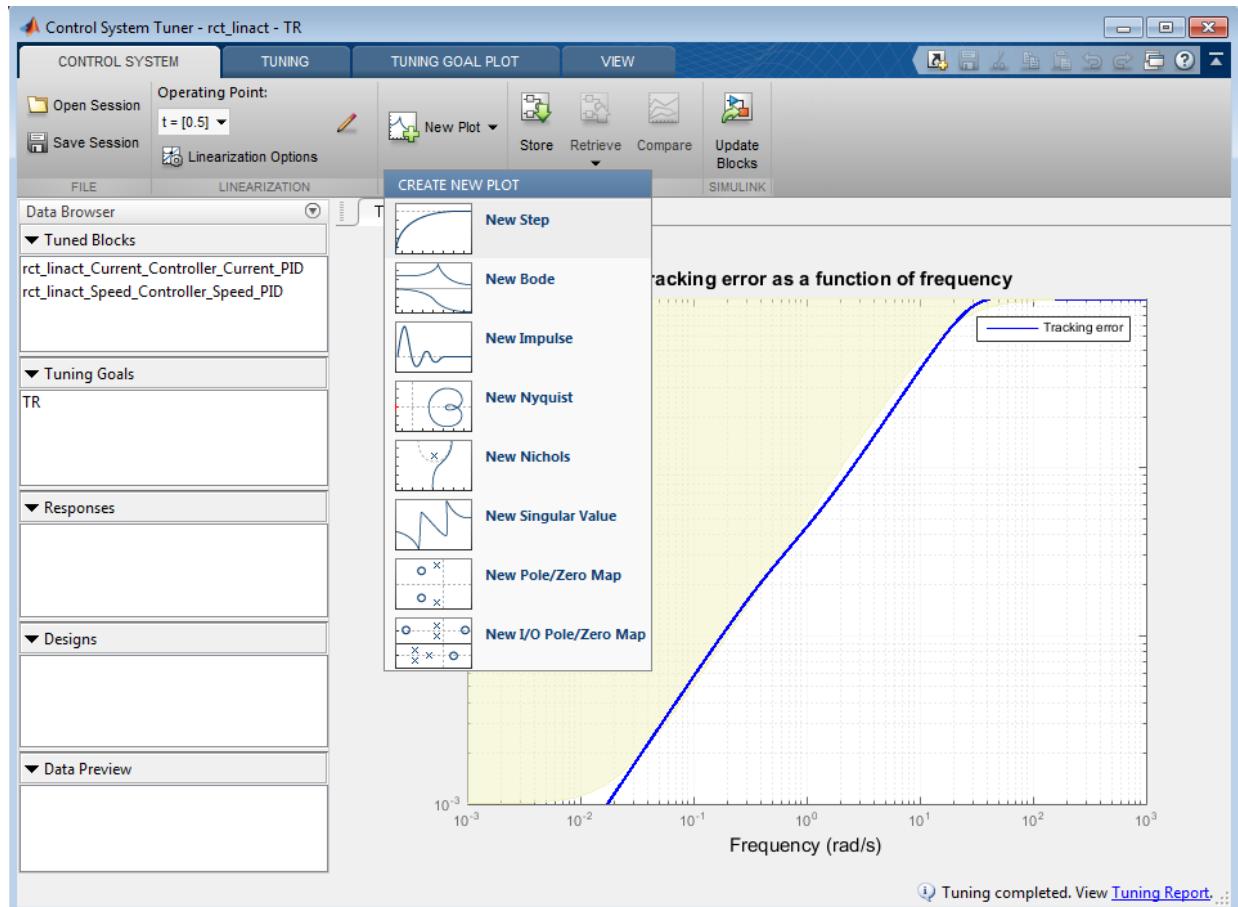
Figure 17: New Plot in Control System Tuner.

Specify the closed-loop response from speed demand to speed by the step plot dialog.
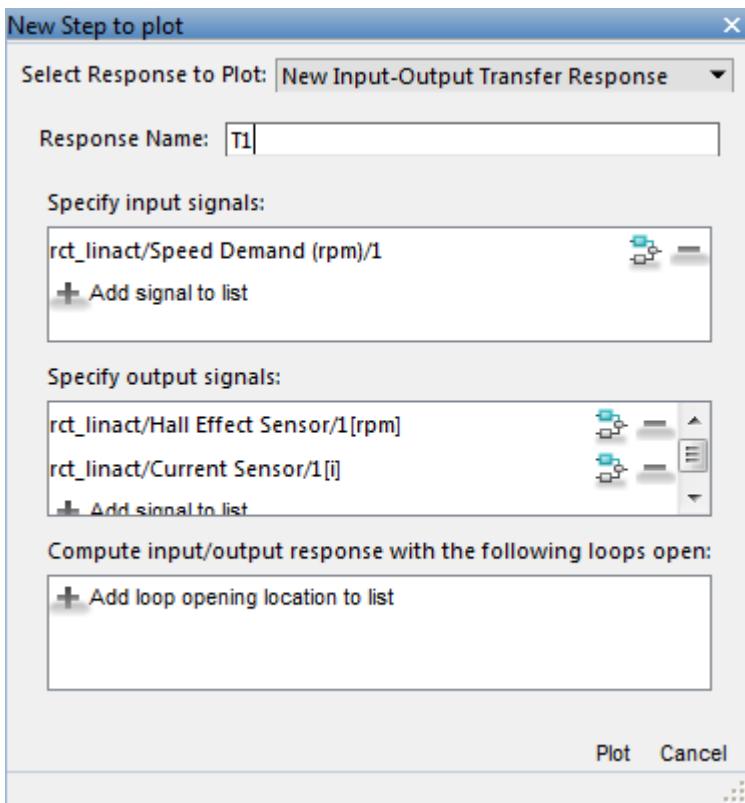
Figure 18: Step Plot Dialog in Control System Tuner.

You see the step plot of the response in Control System Tuner.
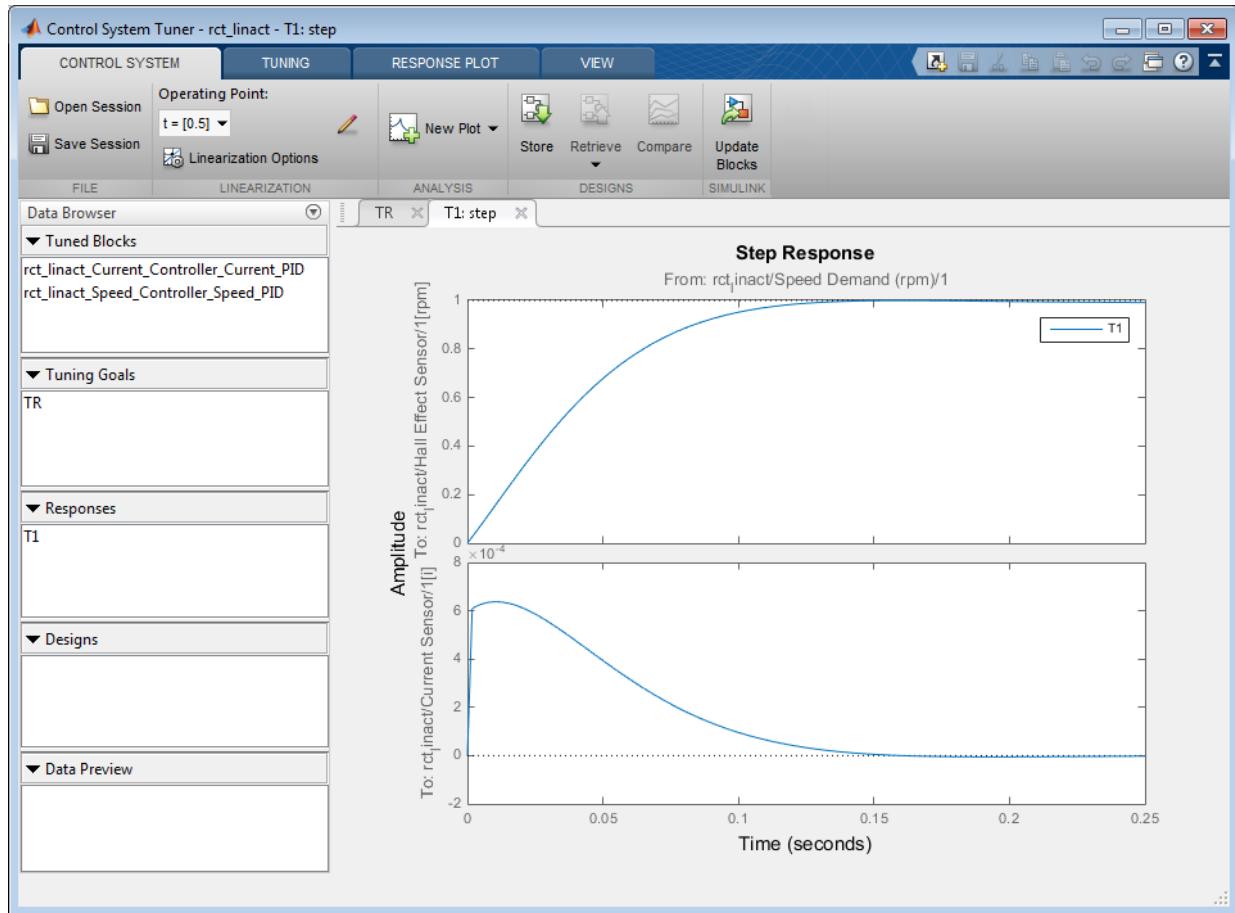
Figure 19: Step Plot in Control System Tuner.

The response looks good in the linear domain so first store the current design by clicking Store and push the tuned gain values to Simulink by clicking Update Blocks and further validate the design in the nonlinear model.
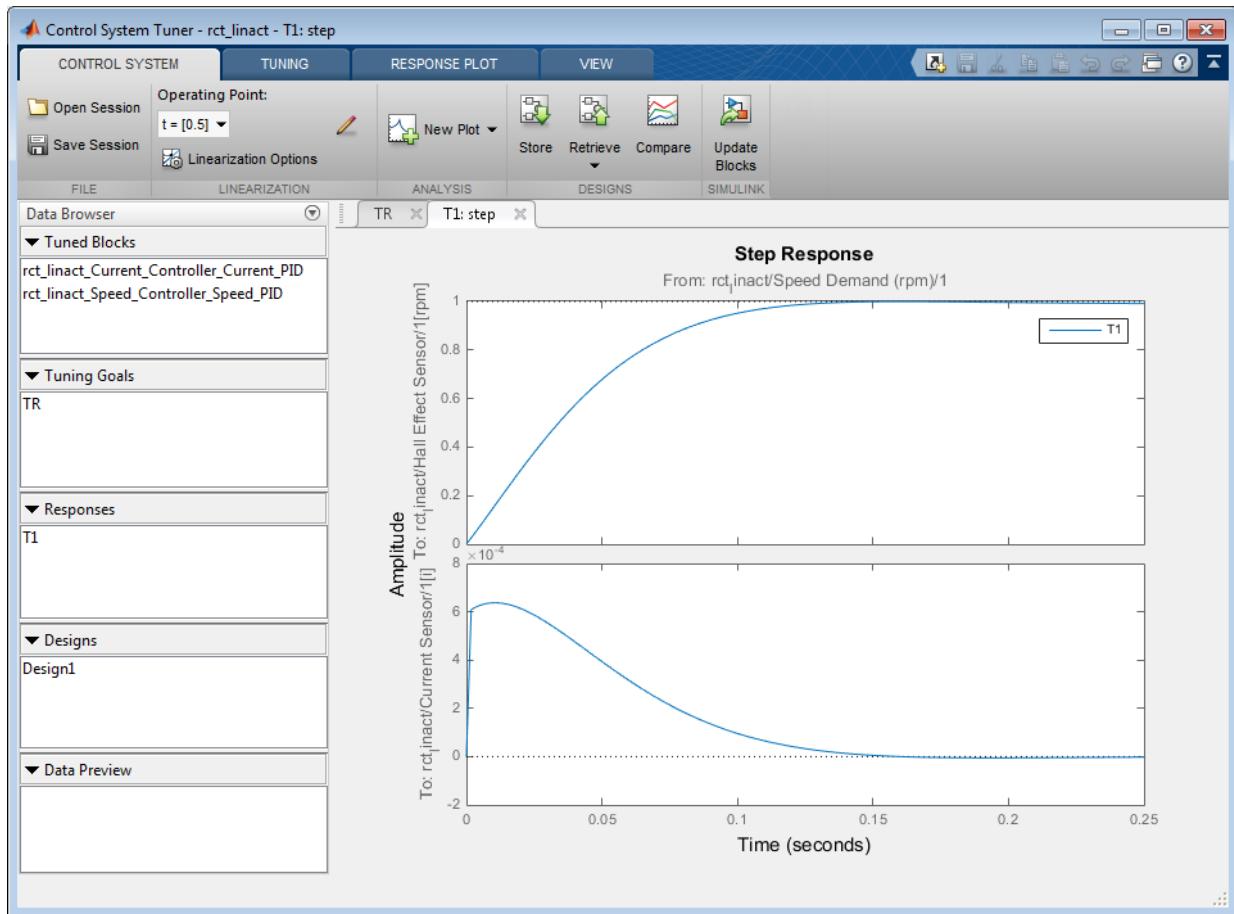
Figure 20: Stored Values of Tuned Blocks in Control System Tuner.

The nonlinear simulation results appear in Figure 21. The nonlinear behavior is far worse than the linear approximation, a discrepancy that can be traced to saturations in the inner loop (see Figure 22).
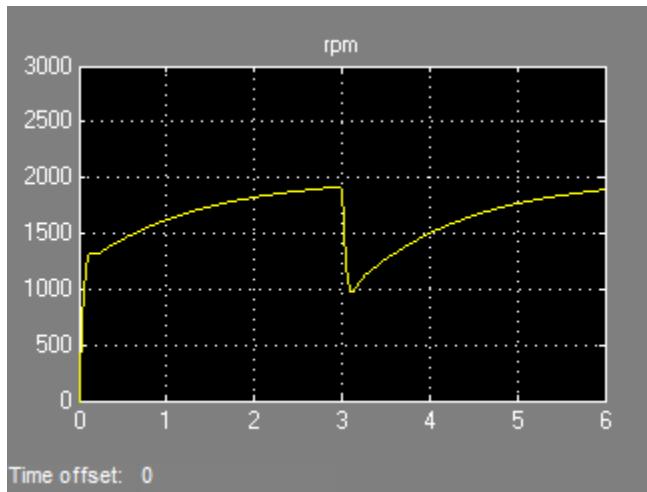
Figure 21: Nonlinear Simulation of Tuned Controller.
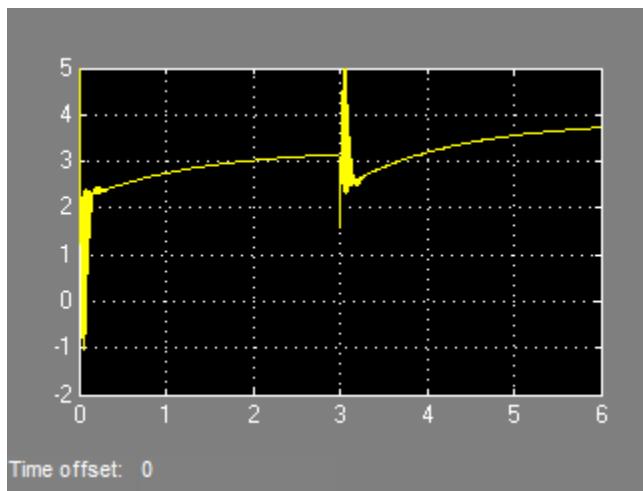
Figure 22: Current Controller Output (limited to plus/minus 5).

Preventing Saturations

So far we have only specified a desired response time for the outer (speed) loop. This leaves `systune` free to allocate the control effort between the inner and outer loops.

Saturations in the inner loop suggest that the proportional gain is too high and that some rebalancing is needed. One possible remedy is to explicitly limit the gain from the speed command to the outputs of the P and PI controllers. For a speed reference of 2000 rpm and saturation limits of plus/minus 5, the average gain should not exceed 5/2000 = 0.0025. To be conservative, we can try to keep the gain from speed reference to controller outputs below 0.001. To do this, add two gain requirements and retune the controller gains with all three requirements in place.

Limit gain from speed demand to control signals to avoid saturation by specifying two new goals from Tuning tab. You need to select control signals from Simulink model since they are not defined previously.

Gain Goal

Name: MG1

Purpose

Limit gain of a specific input/output transfer function.

I/O Transfer Selection

Specify input signals:

rct_linact/Speed Demand (rpm)/1

+ Add signal to list

Specify output signals:

rct_linact/Speed Controller/Speed PID/1

+ Add signal to list

Compute input/output gain with the following loops open:

+ Add loop opening location to list

Options

Limit gain to	0.001
Stabilize I/O transfer	Yes
Enforce goal in frequency range	[0 Inf]
Adjust for signal amplitude	No

Apply goal to

○ All models

○ Only models: [1 2]

OK   Apply   Cancel   ?

Figure 23: Gain Goal Dialog from Speed Demand to Control Signal of Speed PID.

Gain Goal

Name: MG2

Purpose

Limit gain of a specific input/output transfer function.

I/O Transfer Selection

Specify input signals:

rct_linact/Speed Demand (rpm)/1

+ Add signal to list

Specify output signals:

rct_linact/Current Controller/Current PID/1

+ Add signal to list

Compute input/output gain with the following loops open:

+ Add loop opening location to list

Options

Limit gain to	0.001
Stabilize I/O transfer	Yes
Enforce goal in frequency range	[0 Inf]
Adjust for signal amplitude	No

Apply goal to

○ All models

○ Only models: [1 2]

OK   Apply   Cancel   ?

Figure 24: Gain Goal Dialog from Speed Demand to Control Signal of Current PID.

New gain goals appear in Tuning Goals section of Control System Tuner.
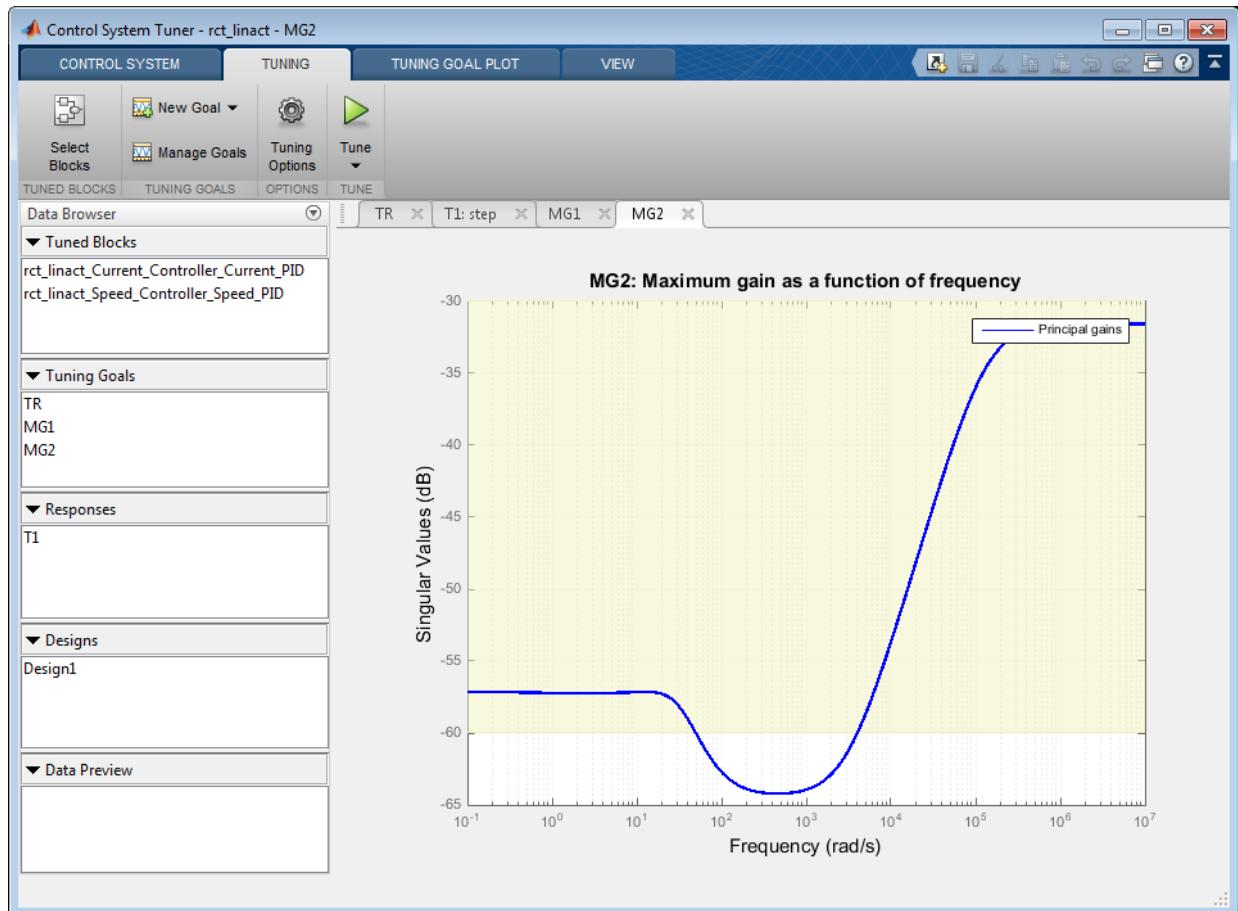
Figure 25: Two Gain Goals Added to Control System Tuner.

Retune with these additional requirements. Tuning Report accessed at the bottom right of the tool shows the worst gain 1.39 indicating that the requirements are nearly but not exactly met (all requirements are met when the final gain is less than 1).

Tuning Report ✕

Tuning Summary

Tuning Progress	Finished	
Hard Goals	None	
Soft Goals	Worst value	1.3897
	TR	1.3897
	MG1	0.93028
	MG2	1.3897
Iterations	42	

Optimization Progress

Final	Soft = 1.39	Hard = -Inf	Iterations = 42

Close ⑦
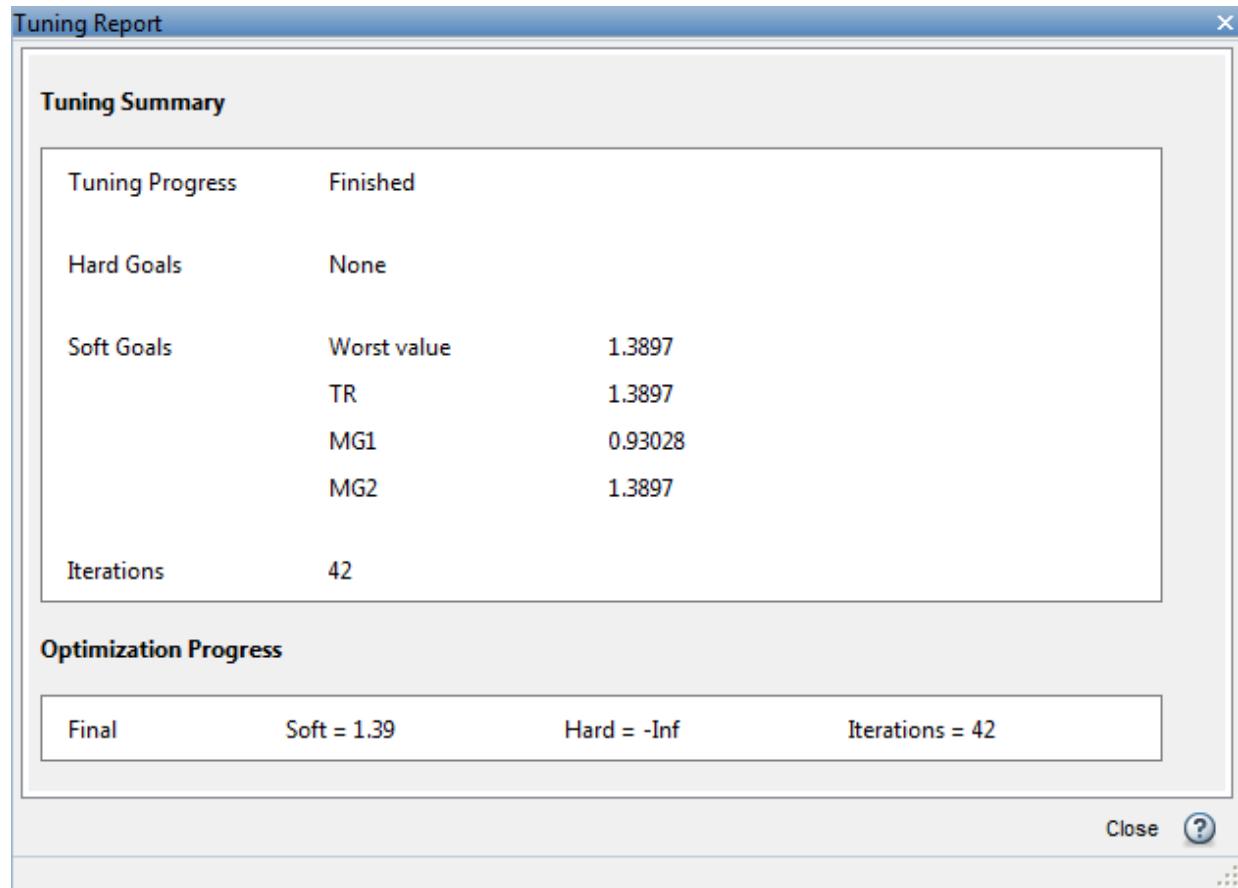
Figure 26: Tuning Report After Retuning.

Next compare the two designs in the linear domain by clicking `Compare` in `Control System` tab.
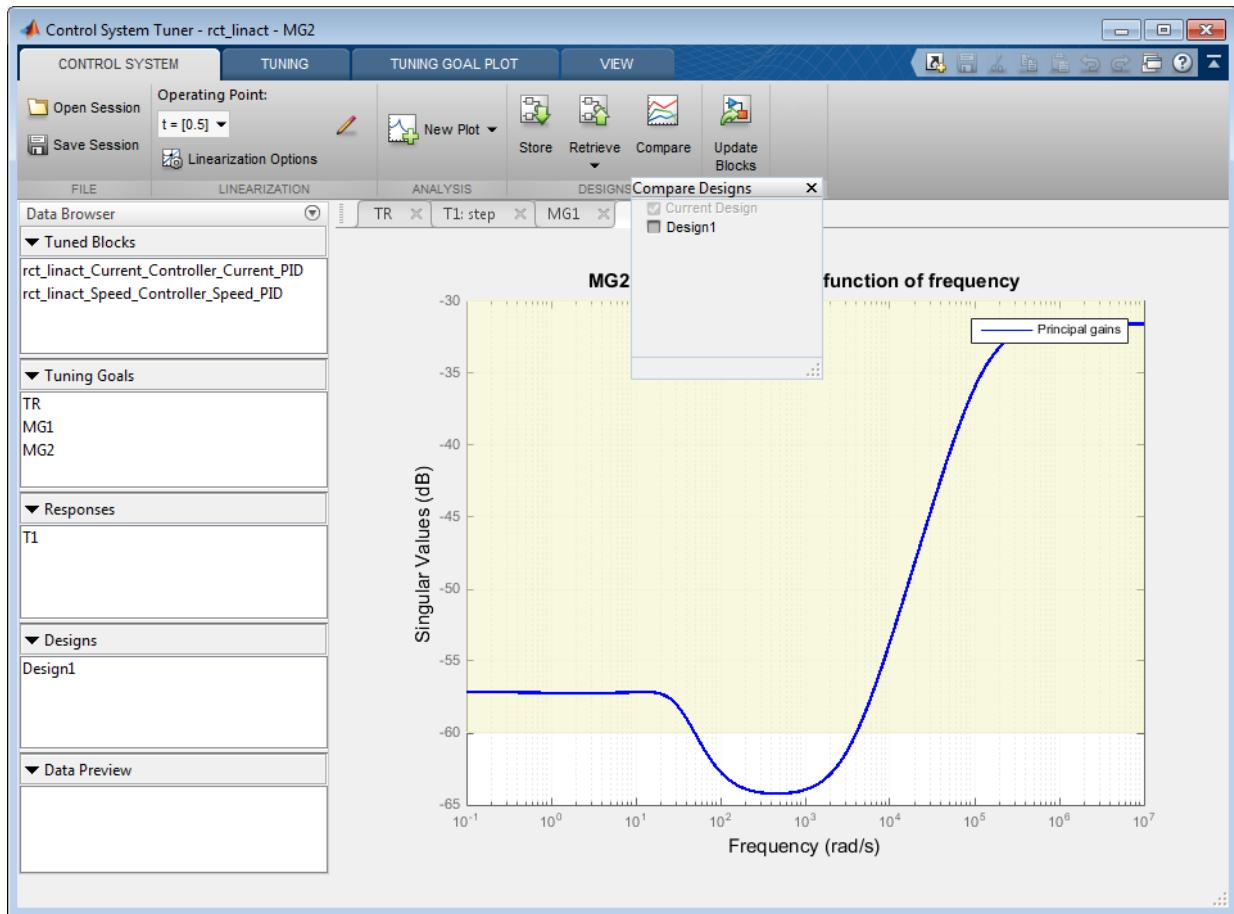
Figure 27: Comparing Two Designs.

The second design is less aggressive but still meets the response time requirement.
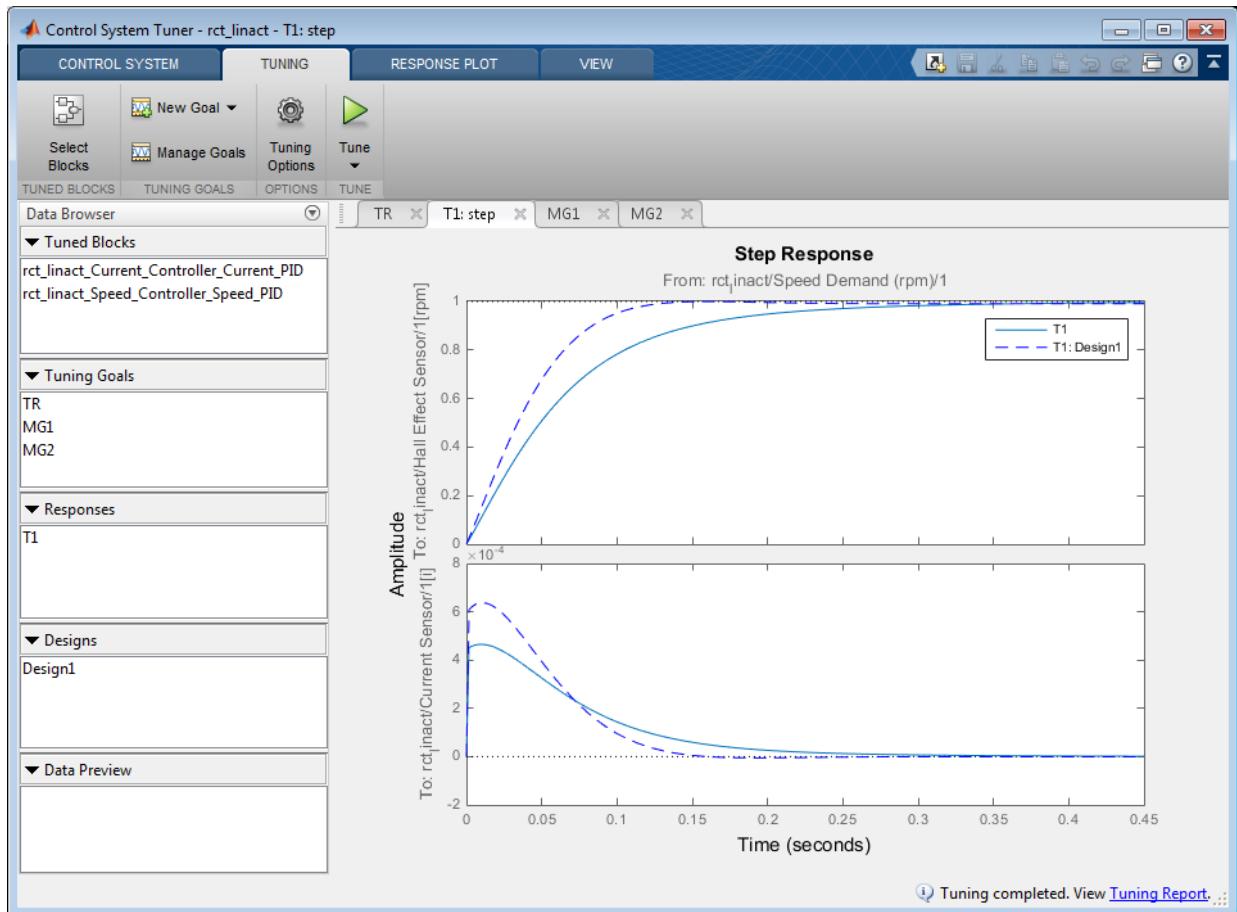
Figure 28: Step Responses of Two Designs.

Finally, push the new tuned gain values to the Simulink model by `Update Blocks` and simulate the response to a 2000 rpm speed demand and 500 N load disturbance. The simulation results appear in Figure 29 and the current controller output is shown in Figure 30.
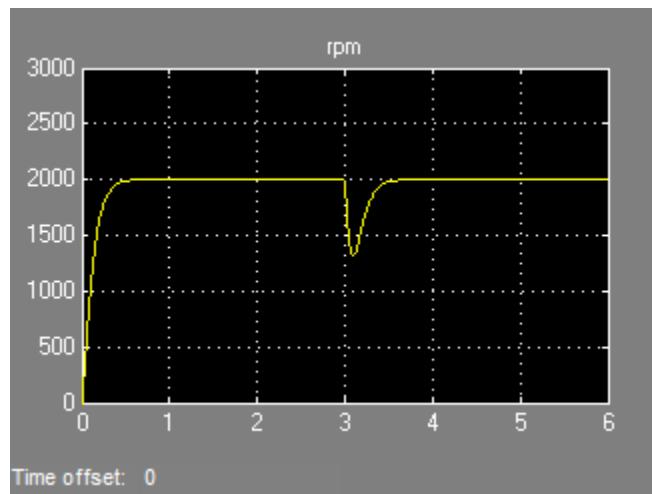
Figure 29: Nonlinear Response of Tuning with Gain Constraints.
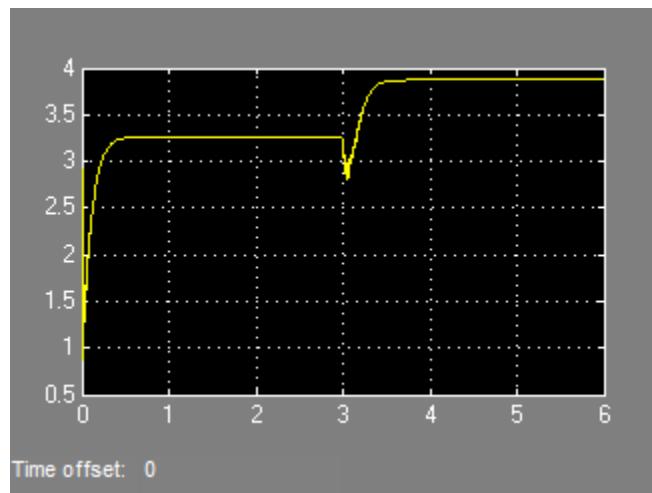
Figure 30: Current Controller Output.

The nonlinear responses are now satisfactory and the current loop is no longer saturating. The additional gain constraints have forced systune to re-distribute the control effort between the inner and outer loops so as to avoid saturation.
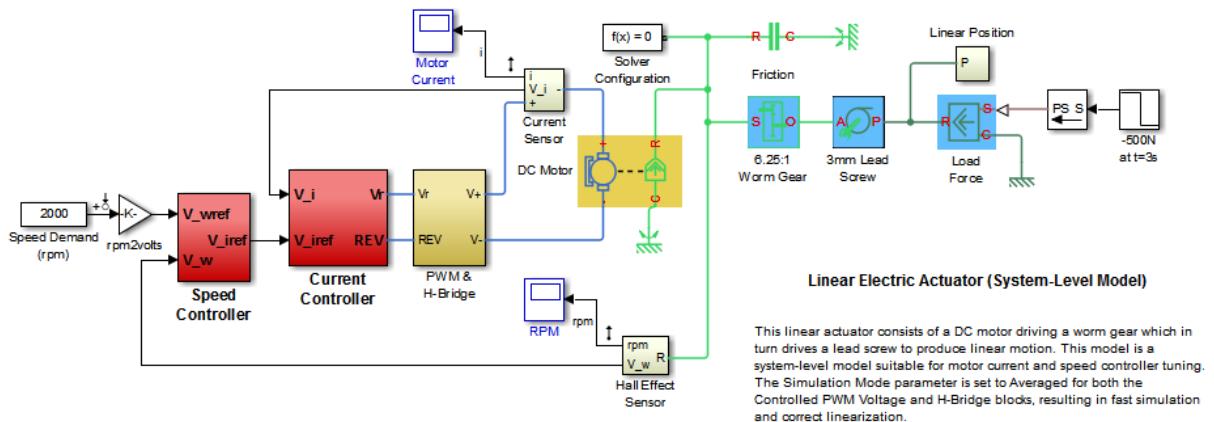
Control of a Linear Electric Actuator

This example shows how to use `slTuner` and `systune` to tune the current and velocity loops in a linear electric actuator with saturation limits.
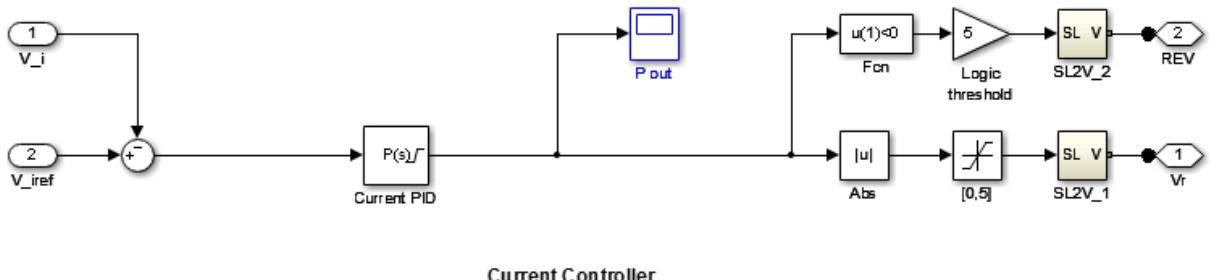
Linear Electric Actuator Model

Open the Simulink model of the linear electric actuator:

```
open_system('rct_linact')
```


The electrical and mechanical components are modeled using Simscape Electronics and Simscape Multibody. The control system consists of two cascaded feedback loops controlling the driving current and angular speed of the DC motor.

Current Controller

Speed Controller

Figure 1: Current and Speed Controllers.

Note that the inner-loop (current) controller is a proportional gain while the outer-loop (speed) controller has proportional and integral actions. The output of both controllers is limited to plus/minus 5.

Design Specifications

We need to tune the proportional and integral gains to respond to a 2000 rpm speed demand in about 0.1 seconds with minimum overshoot. The initial gain settings in the model are P=50 and PI(s)=0.2+0.1/s and the corresponding response is shown in Figure 2. This response is too slow and too sensitive to load disturbances.

Figure 2: Untuned Response.

Control System Tuning

You can use `systune` to jointly tune both feedback loops. To set up the design, create an instance of the `slTuner` interface with the list of tuned blocks. All blocks and signals are specified by their names in the model. The model is linearized at t=0.5 to avoid discontinuities in some derivatives at t=0.

```
TunedBlocks = {'Current PID','Speed PID'};
tLinearize = 0.5;  % linearize at t=0.5

% Create tuning interface
ST0 = slTuner('rct_linact',TunedBlocks,tLinearize);
```

The data structure `ST0` contains a description of the control system and its tunable elements. Next specify that the DC motor should follow a 2000 rpm speed demand in 0.1 seconds:

```
TR = TuningGoal.Tracking('Speed Demand (rpm)','rpm',0.1);
```
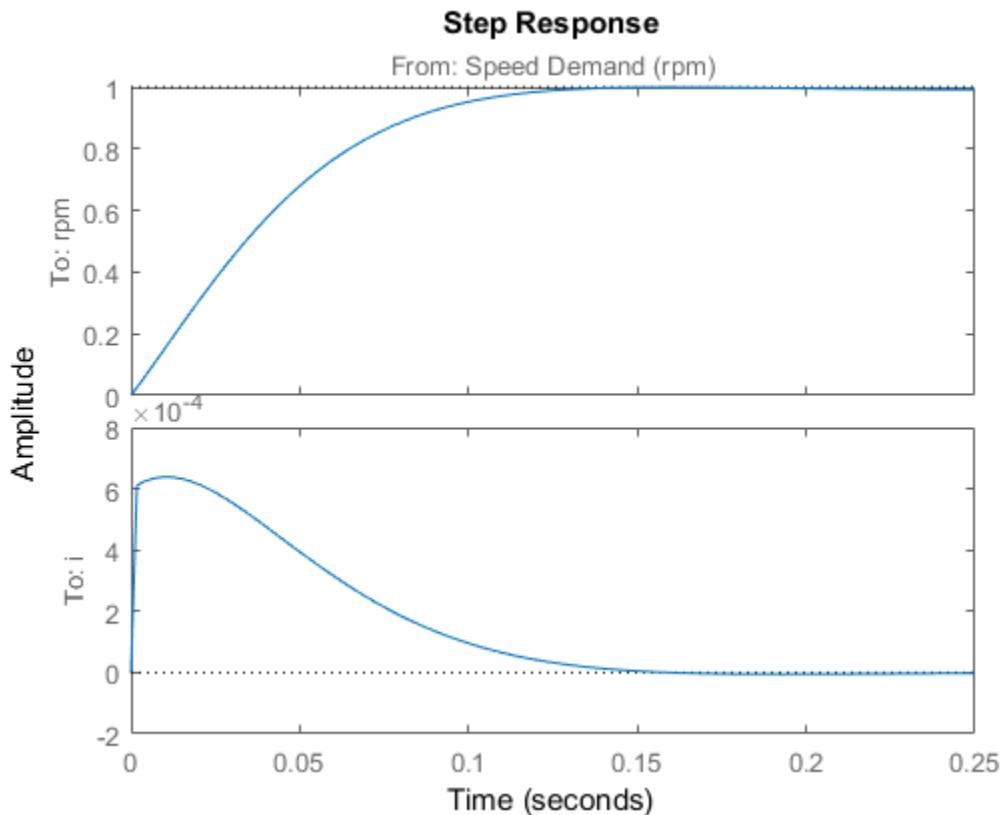
You can now tune the proportional and integral gains with `looptune`:

```
ST1 = systune(ST0,TR);

Final: Soft = 1.12, Hard = -Inf, Iterations = 34
```

13-343

This returns an updated description ST1 containing the tuned gain values. To validate this design, plot the closed-loop response from speed demand to speed:

```
T1 = getIOTransfer(ST1,'Speed Demand (rpm)',{'rpm','i'});
figure
step(T1)
```


The response looks good in the linear domain so push the tuned gain values to Simulink and further validate the design in the nonlinear model.

```
writeBlockValue(ST1)
```

The nonlinear simulation results appear in Figure 3. The nonlinear behavior is far worse than the linear approximation, a discrepancy that can be traced to saturations in the inner loop (see Figure 4).
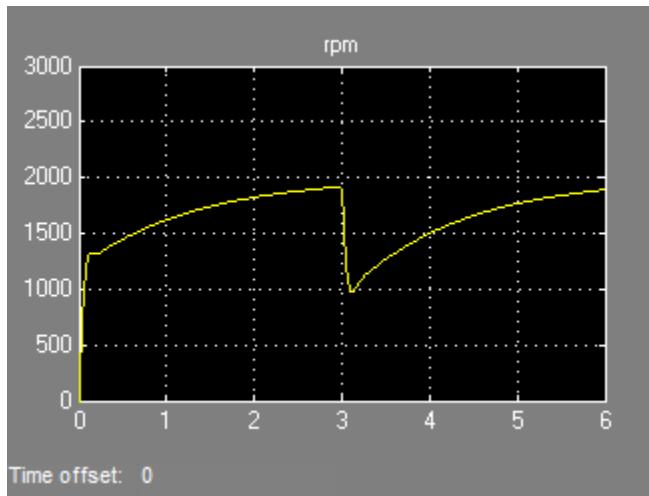
Figure 3: Nonlinear Simulation of Tuned Controller.
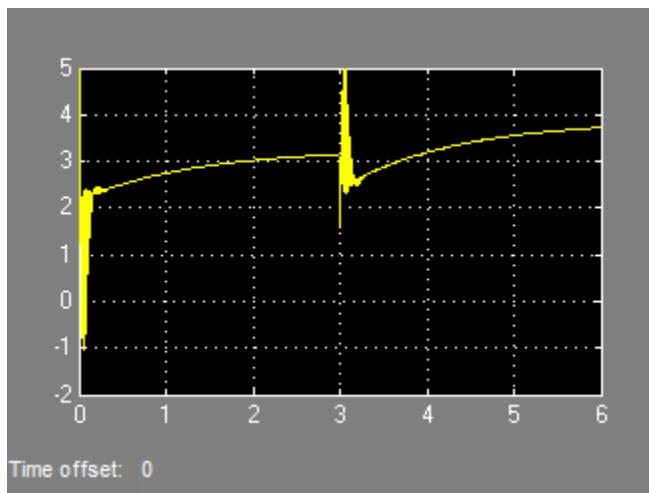
Figure 4: Current Controller Output (limited to plus/minus 5).

Preventing Saturations

So far we have only specified a desired response time for the outer (speed) loop. This leaves systune free to allocate the control effort between the inner and outer loops.

Saturations in the inner loop suggest that the proportional gain is too high and that some rebalancing is needed. One possible remedy is to explicitly limit the gain from the speed command to the outputs of the P and PI controllers. For a speed reference of 2000 rpm and saturation limits of plus/minus 5, the average gain should not exceed 5/2000 = 0.0025. To be conservative, we can try to keep the gain from speed reference to controller outputs below 0.001. To do this, add two gain requirements and retune the controller gains with all three requirements in place.

```
% Mark the control signals as points of interest so that they can be
% referenced in the gain requirements
addPoint(ST0,{'Current PID','Speed PID'})

% Limit gain from speed demand to control signals to avoid saturation
MG1 = TuningGoal.Gain('Speed Demand (rpm)','Speed PID',0.001);
MG2 = TuningGoal.Gain('Speed Demand (rpm)','Current PID',0.001);

% Retune with these additional requirements
[ST2,~,~,info] = systune(ST0,[TR,MG1,MG2]);

Final: Soft = 1.39, Hard = -Inf, Iterations = 45
```
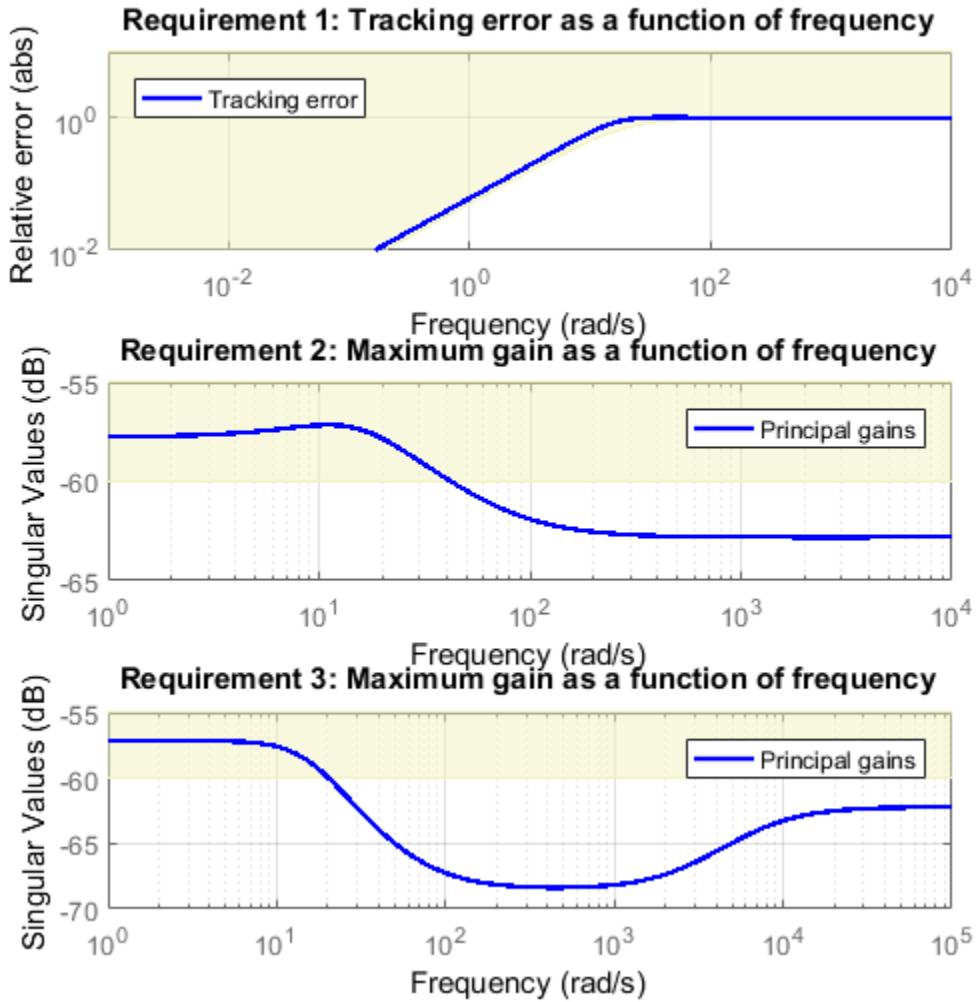
The final gain 1.39 indicates that the requirements are nearly but not exactly met (all requirements are met when the final gain is less than 1). Use `viewSpec` to inspect how the tuned controllers fare against each requirement.
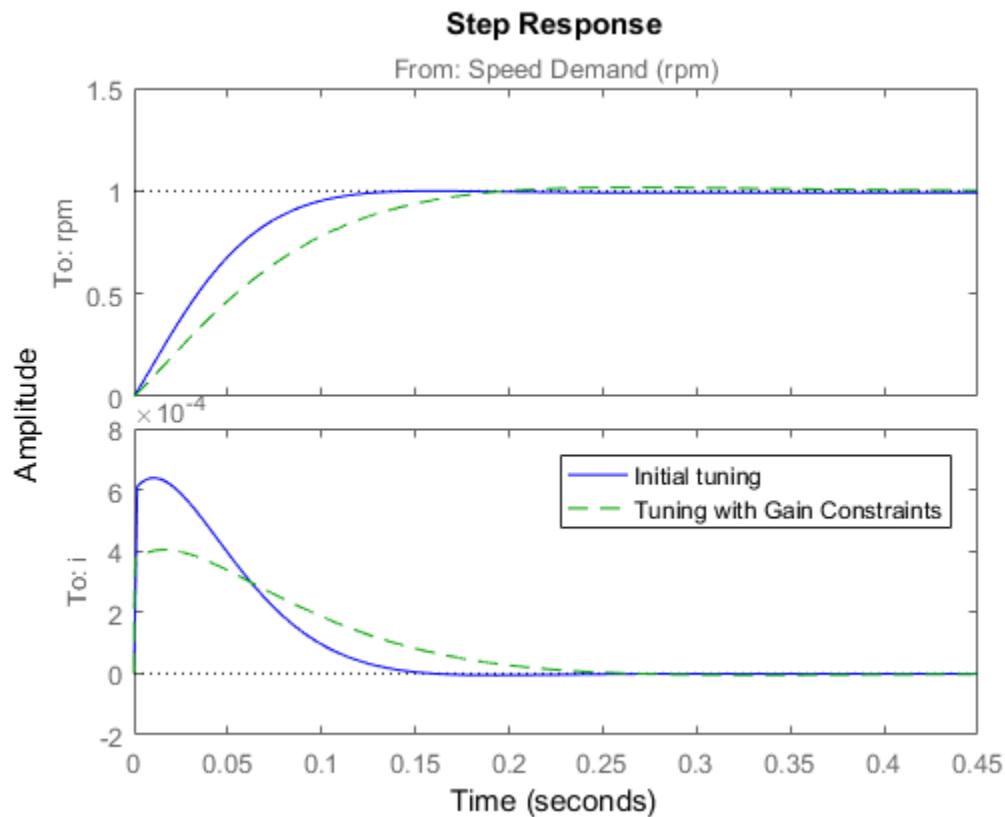
```
figure('Position',[100,100,560,550])
viewSpec([TR,MG1,MG2],ST2,info)
```

Next compare the two designs in the linear domain.

```
T2 = getIOTransfer(ST2,'Speed Demand (rpm)',{'rpm','i'});
figure
step(T1,'b',T2,'g--')
```

```
legend('Initial tuning','Tuning with Gain Constraints')
```


The second design is less aggressive but still meets the response time requirement. Finally, push the new tuned gain values to the Simulink model and simulate the response to a 2000 rpm speed demand and 500 N load disturbance. The simulation results appear in Figure 5 and the current controller output is shown in Figure 6.

```
writeBlockValue(ST2)
```
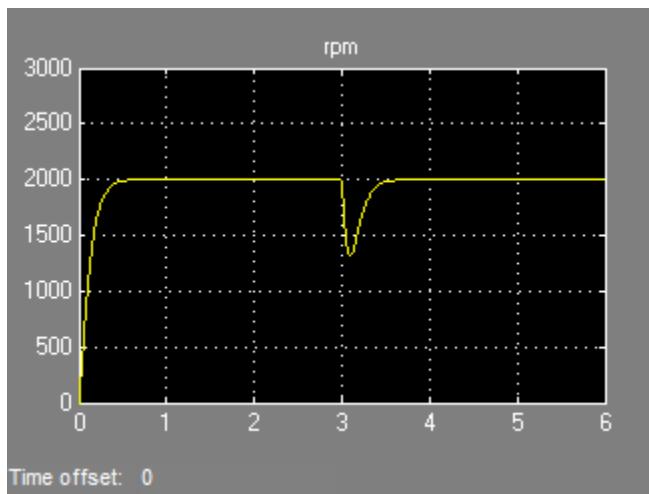
**Figure 5: Nonlinear Response of Tuning with Gain Constraints.**
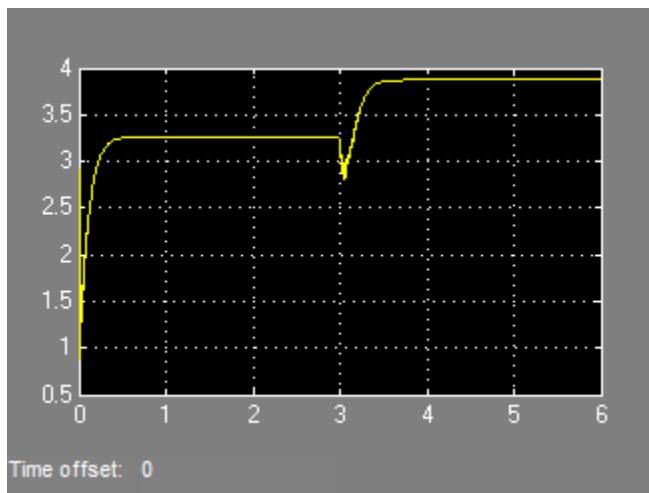
Figure 6: Current Controller Output.

The nonlinear responses are now satisfactory and the current loop is no longer saturating. The additional gain constraints have forced `systune` to re-distribute the control effort between the inner and outer loops so as to avoid saturation.

PID Tuning for Setpoint Tracking vs. Disturbance Rejection

This example uses `systune` to explore trade-offs between setpoint tracking and disturbance rejection when tuning PID controllers.

PID Tuning Trade-Offs

When tuning 1-DOF PID controllers, it is often impossible to achieve good tracking and fast disturbance rejection at the same time. Assuming the control bandwidth is fixed, faster disturbance rejection requires more gain inside the bandwidth, which can only be achieved by increasing the slope near the crossover frequency. Because a larger slope means a smaller phase margin, this typically comes at the expense of more overshoot in the response to setpoint changes.

Figure 1: Trade-off in 1-DOF PID Tuning.

This example uses `systune` to explore this trade-off and find the right compromise for your application. See also `pidtool` for a more direct way to make such trade-off (see "Design Focus" under Controller Options).

Tuning Setup

Consider the PID loop of Figure 2 with a load disturbance at the plant input.

Figure 2: PID Control Loop.

For this example we use the plant model

$$G(s) = \frac{10(s+5)}{(s+1)(s+2)(s+10)}.$$

The target control bandwidth is 10 rad/s. Create a tunable PID controller and fix its derivative filter time constant to $T_f = 0.01$ (10 times the bandwidth) so that there are only three gains to tune (proportional, integral, and derivative gains).

```
G = zpk(-5,[-1 -2 -10],10);
C = tunablePID('C','pid');
C.Tf.Value = 0.01;  C.Tf.Free = false;  % fix Tf=0.01
```

Construct a tunable model TO of the closed-loop transfer from r to y. Use an "analysis point" block to mark the location u where the disturbance enters.

```
LS = AnalysisPoint('u');
TO = feedback(G*LS*C,1);
TO.u = 'r'; TO.y = 'y';
```

The gain of the open-loop response $L = GC$ is a key indicator of the feedback loop behavior. The open-loop gain should be high (greater than one) inside the control

bandwidth to ensure good disturbance rejection, and should be low (less than one) outside the control bandwidth to be insensitive to measurement noise and unmodeled plant dynamics. Accordingly, use three requirements to express the control objectives:

- "Tracking" requirement to specify a response time of about 0.2 seconds to step changes in r

- "MaxLoopGain" requirement to force a roll-off of -20 dB/decade past the crossover frequency 10 rad/s

- "MinLoopGain" requirement to adjust the integral gain at frequencies below 0.1 rad/s.

```
s = tf('s');
wc = 10; % target crossover frequency

% Tracking
R1 = TuningGoal.Tracking('r','y',2/wc);

% Bandwidth and roll-off
R2 = TuningGoal.MaxLoopGain('u',wc/s);

% Disturbance rejection
R3 = TuningGoal.MinLoopGain('u',wc/s);
R3.Focus = [0 0.1];
```

Tuning of 1-DOF PID Controller

Use `systune` to tune the PID gains to meet these requirements. Treat the bandwidth and disturbance rejection goals as hard constraints and optimize tracking subject to these constraints.

```
T1 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.12, Hard = 0.99976, Iterations = 144
```

Verify that all three requirements are nearly met. The blue curves are the achieved values and the yellow patches highlight regions where the requirements are violated.

```
figure('Position',[100,100,560,580])
viewSpec([R1 R2 R3],T1)
```

Tracking vs. Rejection

To gain insight into the trade-off between tracking and disturbance rejection, increase the minimum loop gain in the frequency band [0,0.1] rad/s by a factor α. Re-tune the PID gains for the values $\alpha = 2, 4$.

```
% Increase loop gain by factor 2
alpha = 2;
R3.MinGain = alpha*wc/s;
T2 = systune(T0,R1,[R2 R3]);
```

```
Final: Soft = 1.17, Hard = 0.99973, Iterations = 117
```

```
% Increase loop gain by factor 4
alpha = 4;
R3.MinGain = alpha*wc/s;
T3 = systune(T0,R1,[R2 R3]);
```

```
Final: Soft = 1.27, Hard = 0.9994, Iterations = 161
```

Compare the responses to a step command r and to a step disturbance d entering at the plant input u.

```
figure, step(T1,T2,T3,3)
title('Setpoint tracking')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```

```
% Compute closed-loop transfer from u to y
D1 = getIOTransfer(T1,'u','y');
D2 = getIOTransfer(T2,'u','y');
D3 = getIOTransfer(T3,'u','y');
step(D1,D2,D3,10)
title('Disturbance rejection')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```

Disturbance rejection

From: u To: y

Note how disturbance rejection improves as `alpha` increases, but at the expense of increased overshoot in setpoint tracking. Plot the open-loop responses for the three designs, and note how the slope before crossover (0dB) increases with `alpha`.

```
L1 = getLoopTransfer(T1,'u');
L2 = getLoopTransfer(T2,'u');
L3 = getLoopTransfer(T3,'u');
bodemag(L1,L2,L3,{1e-2,1e2}), grid
title('Open-loop response')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```

Open-loop response

From: u To: u

Which design is most suitable depends on the primary purpose of the feedback loop you are tuning.

Tuning of 2-DOF PID Controller

If you cannot compromise tracking to improve disturbance rejection, consider using a 2-DOF architecture instead. A 2-DOF PID controller is capable of fast disturbance rejection without significant increase of overshoot in setpoint tracking.

Figure 3: 2-DOF PID Control Loop.

Use the `tunablePID2` object to parameterize the 2-DOF PID controller and construct a tunable model `T0` of the closed-loop system in Figure 3.

```
C = tunablePID2('C','pid');
C.Tf.Value = 0.01;  C.Tf.Free = false;  % fix Tf=0.01

T0 = feedback(G*LS*C,1,2,1,+1);
T0 = T0(:,1);
T0.u = 'r';  T0.y = 'y';
```

Next tune the 2-DOF PI controller for the largest loop gain tried earlier ($\alpha = 4$).

```
% Minimum loop gain inside bandwidth (for disturbance rejection)
alpha = 4;
R3.MinGain = alpha*wc/s;

% Tune 2-DOF PI controller
T4 = systune(T0,R1,[R2 R3]);

Final: Soft = 1.09, Hard = 0.99942, Iterations = 157
```

Compare the setpoint tracking and disturbance rejection properties of the 1-DOF and 2-DOF designs for $\alpha = 4$.

```
clf, step(T3,'b',T4,'g--',4)
title('Setpoint tracking')
legend('1-DOF','2-DOF')
```

```
D4 = getIOTransfer(T4,'u','y');
step(D3,'b',D4,'g--',4)
title('Disturbance rejection')
legend('1-DOF','2-DOF')
```

The responses to a step disturbance are similar but the 2-DOF controller eliminates the overshoot in the response to a setpoint change. You can use `showTunable` to compare the tuned gains in the 1-DOF and 2-DOF controllers.

```
showTunable(T3)   % 1-DOF PI

C =

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

  with Kp = 9.64, Ki = 16.1, Kd = 0.885, Tf = 0.01
```

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

```
showTunable(T4)   % 2-DOF PI
```

```
C =

                      1                  s
  u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                      s              Tf*s+1

  with Kp = 6.75, Ki = 19.7, Kd = 0.971, Tf = 0.01, b = 0.74, c = 1.13

Name: C
Continuous-time 2-DOF PIDF controller in parallel form.
```

Active Vibration Control in Three-Story Building

This example uses `systune` to control seismic vibrations in a three-story building.

Background

This example considers an Active Mass Driver (AMD) control system for vibration isolation in a three-story experimental structure. This setup is used to assess control design techniques for increasing safety of civil engineering structures during earthquakes. The structure consists of three stories with an active mass driver on the top floor which is used to attenuate ground disturbances. This application is borrowed from *"Benchmark Problems in Structural Control: Part I - Active Mass Driver System,"* B.F. Spencer Jr., S.J. Dyke, and H.S. Deoskar, Earthquake Engineering and Structural Dynamics, 27(11), 1998, pp. 1127-1139.
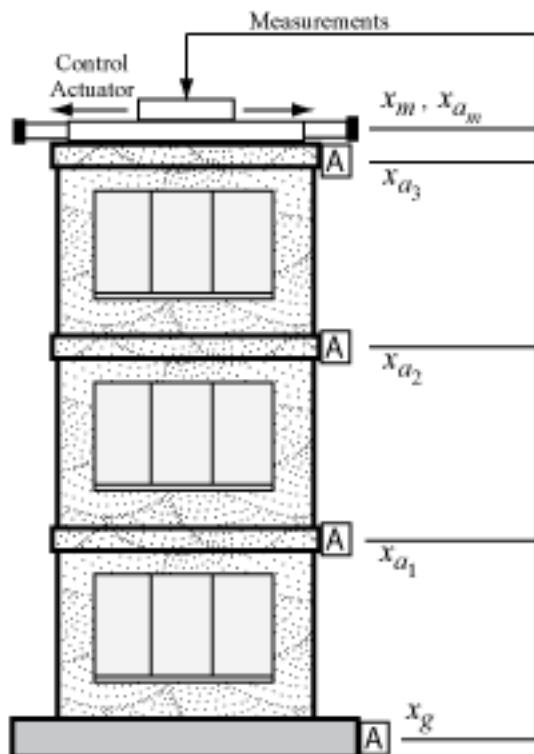
Figure 1: Active Mass Driver Control System

The plant P is a 28-state model with the following state variables:

- x(i): displacement of i-th floor relative to the ground (cm)
- xm: displacement of AMD relative to 3rd floor (cm)
- xv(i): velocity of i-th floor relative to the ground (cm/s)
- xvm: velocity of AMD relative to the ground (cm/s)
- xa(i): acceleration of i-th floor relative to the ground (g)
- xam: acceleration of AMD relative to the ground (g)
- d(1)=x(1), d(2)=x(2)-x(1), d(3)=x(3)-x(2): inter-story drifts

The inputs are the ground acceleration xag (in g) and the control signal u. We use 1 g = 981 cm/s^2.

```
load ThreeStoryData
size(P)
```

```
State-space model with 20 outputs, 2 inputs, and 28 states.
```
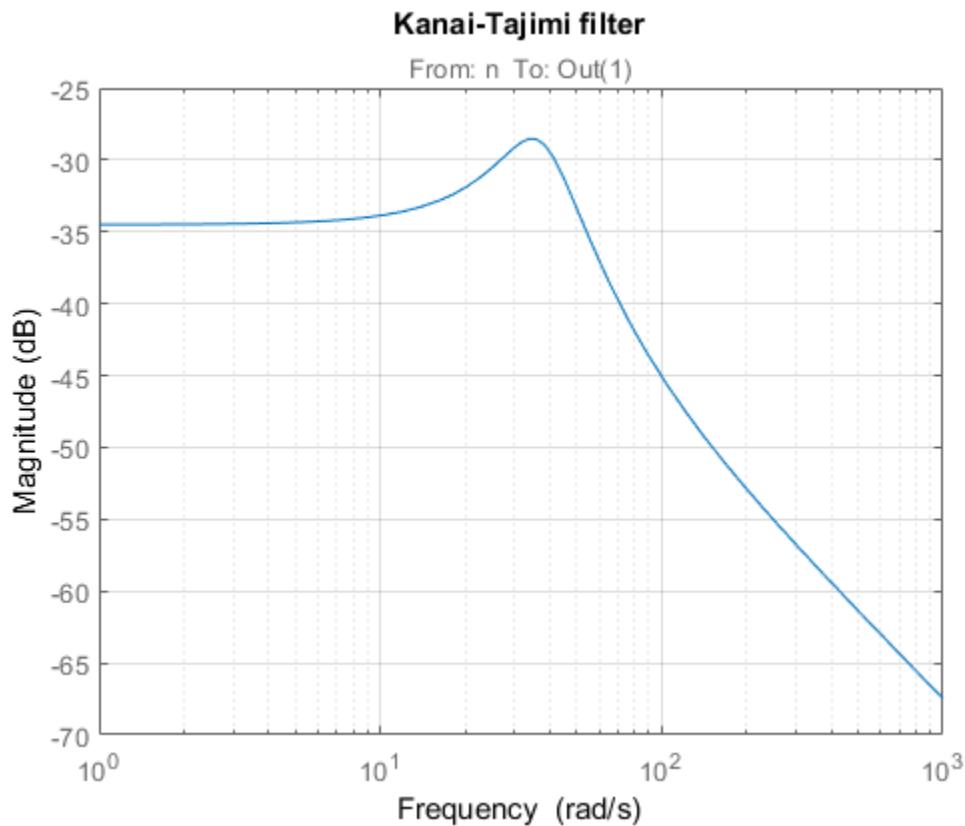
Model of Earthquake Acceleration

The earthquake acceleration is modeled as a white noise process filtered through a Kanai-Tajimi filter.

```
zg = 0.3;   wg = 37.3;
S0 = 0.03*zg/(pi*wg*(4*zg^2+1));
num = sqrt(S0)*[2*zg*wg wg^2];
den = [1 2*zg*wg wg^2];

F = sqrt(2*pi)*tf(num,den);
F.InputName = 'n';   % white noise input

bodemag(F), grid, title('Kanai-Tajimi filter')
```

Kanai-Tajimi filter
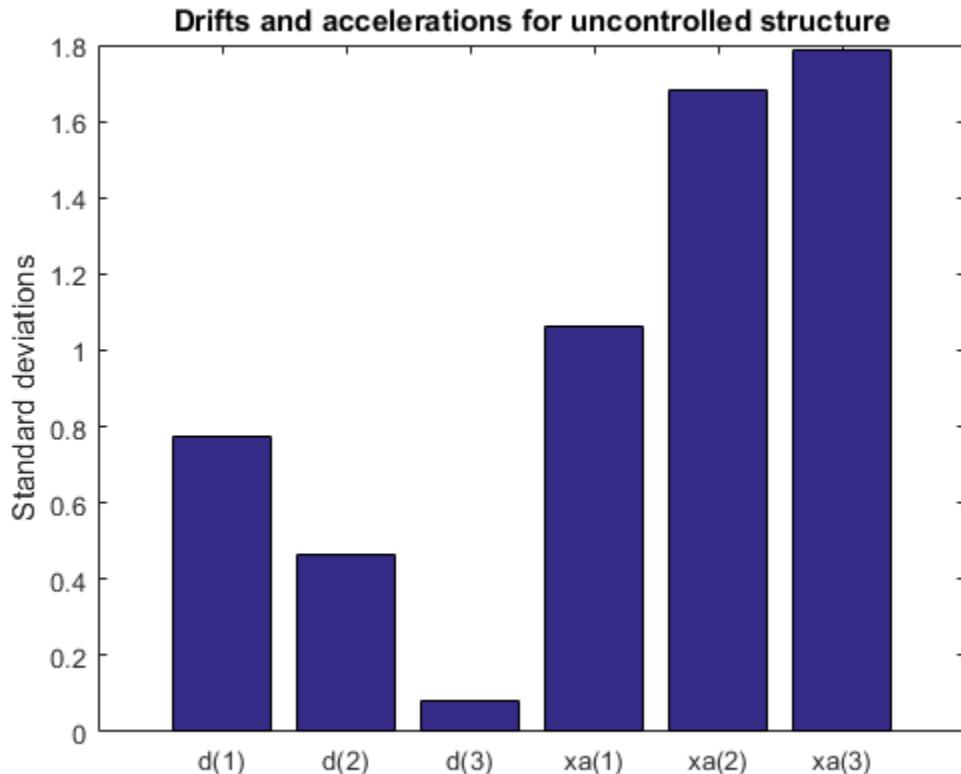
Open-Loop Characteristics

The effect of an earthquake on the uncontrolled structure can be simulated by injecting a white noise input `n` into the plant-filter combination. You can also use `covar` to directly compute the standard deviations of the resulting inter-story drifts and accelerations.

```
% Add Kanai-Tajimi filter to the plant
PF = P*append(F , 1);

% Standard deviations of open-loop drifts
CV = covar(PF('d','n'),1);
dO = sqrt(diag(CV));
```

```
% Standard deviations of open-loop acceleration
CV = covar(PF('xa','n'),1);
xa0 = sqrt(diag(CV));

% Plot open-loop RMS values
clf, bar([d0 ; xa0])
title('Drifts and accelerations for uncontrolled structure')
ylabel('Standard deviations')
set(gca,'XTickLabel',{'d(1)','d(2)','d(3)','xa(1)','xa(2)','xa(3)'})
```


Control Structure and Design Requirements

The control structure is depicted in Figure 2.
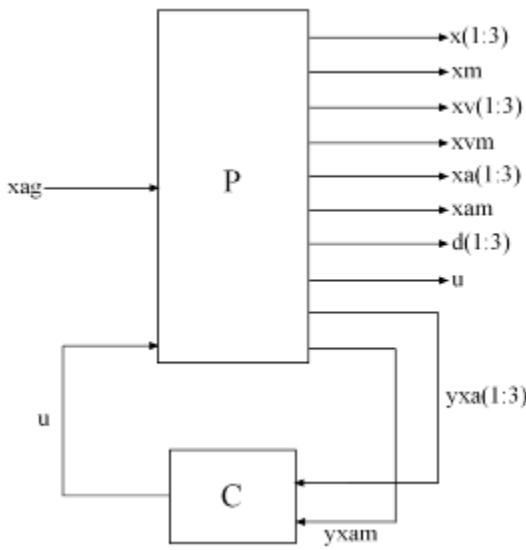
Figure 2: Control Structure

The controller C uses measurements yxa and yxam of xa and xam to generate the control signal u. Physically, the control u is an electrical current driving an hydraulic actuator that moves the masses of the AMD. The design requirements involve:

- Minimization of the inter-story drifts d(i) and accelerations xa(i)
- Hard constraints on control effort in terms of mass displacement xm, mass acceleration xam, and control effort u

All design requirements are assessed in terms of standard deviations of the corresponding signals. Use TuningGoal.Variance to express these requirements and scale each variable by its open-loop standard deviation to seek uniform relative improvement in all variables.

```
% Soft requirements on drifts and accelerations
Soft = [...
   TuningGoal.Variance('n','d(1)', d0(1)) ; ...
   TuningGoal.Variance('n','d(2)', d0(2)) ; ...
   TuningGoal.Variance('n','d(3)', d0(3)) ; ...
   TuningGoal.Variance('n','xa(1)', xa0(1)) ; ...
   TuningGoal.Variance('n','xa(2)', xa0(2)) ; ...
   TuningGoal.Variance('n','xa(3)', xa0(3))];
```

```
% Hard requirements on control effort
Hard = [...
   TuningGoal.Variance('n','xm', 3) ; ...
   TuningGoal.Variance('n','xam', 2) ; ...
   TuningGoal.Variance('n','u', 1)];
```

Controller Tuning

`systune` lets you tune virtually any controller structure subject to these requirements. The controller complexity can be adjusted by trial-and-error, starting with sufficiently high order to gauge the limits of performance, then reducing the order until you observe a noticeable performance degradation. For this example, start with a 5th-order controller with no feedthrough term.

```
C = tunableSS('C',5,1,4);
C.d.Value = 0;
C.d.Free = false;    % Fix feedthrough to zero
```

Construct a tunable model `TO` of the closed-loop system of Figure 2 and tune the controller parameters with `systune`.

```
% Build tunable closed-loop model
TO = lft(PF,C);

% Tune controller parameters
[T,fSoft,gHard]  = systune(TO,Soft,Hard);

Final: Soft = 0.601, Hard = 0.97239, Iterations = 160
```

The summary indicates that we achieved an overall reduction of 40% in standard deviations (`Soft = 0.6`) while meeting all hard constraints (`Hard < 1`).
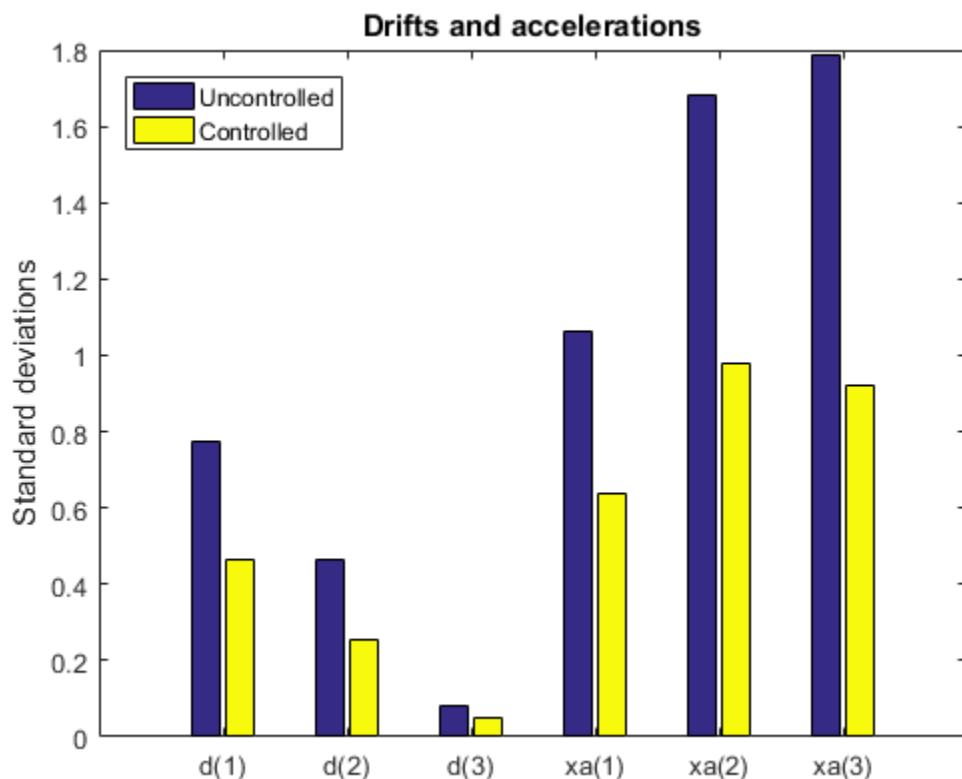
Validation

Compute the standard deviations of the drifts and accelerations for the controlled structure and compare with the uncontrolled results. The AMD control system yields significant reduction of both drift and acceleration.

```
% Standard deviations of closed-loop drifts
CV = covar(T('d','n'),1);
d = sqrt(diag(CV));

% Standard deviations of closed-loop acceleration
CV = covar(T('xa','n'),1);
```

```
xa = sqrt(diag(CV));

% Compare open- and closed-loop values
clf, bar([dO d; xaO xa])
title('Drifts and accelerations')
ylabel('Standard deviations')
set(gca,'XTickLabel',{'d(1)','d(2)','d(3)','xa(1)','xa(2)','xa(3)'})
legend('Uncontrolled','Controlled','location','NorthWest')
```


Simulate the response of the 3-story structure to an earthquake-like excitation in both open and closed loop. The earthquake acceleration is modeled as a white noise process colored by the Kanai-Tajimi filter.
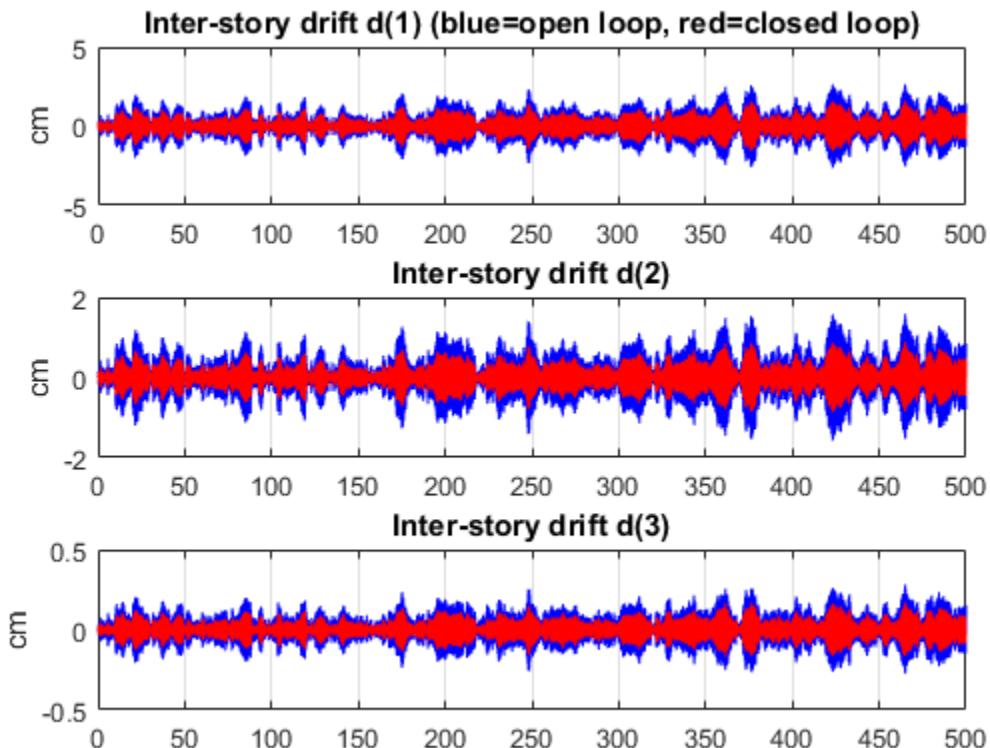
```
% Sampled white noise process
```

```matlab
rng('default')
dt = 1e-3;
t = 0:dt:500;
n = randn(1,length(t))/sqrt(dt); % white noise signal

% Open-loop simulation
ysimOL = lsim(PF(:,1), n , t);

% Closed-loop simulation
ysimCL = lsim(T, n , t);

% Drifts
clf, subplot(311); plot(t,ysimOL(:,13),'b',t,ysimCL(:,13),'r'); grid;
title('Inter-story drift d(1) (blue=open loop, red=closed loop)');  ylabel('cm');
subplot(312); plot(t,ysimOL(:,14),'b',t,ysimCL(:,14),'r'); grid;
title('Inter-story drift d(2)');  ylabel('cm');
subplot(313); plot(t,ysimOL(:,15),'b',t,ysimCL(:,15),'r'); grid;
title('Inter-story drift d(3)');  ylabel('cm');
```
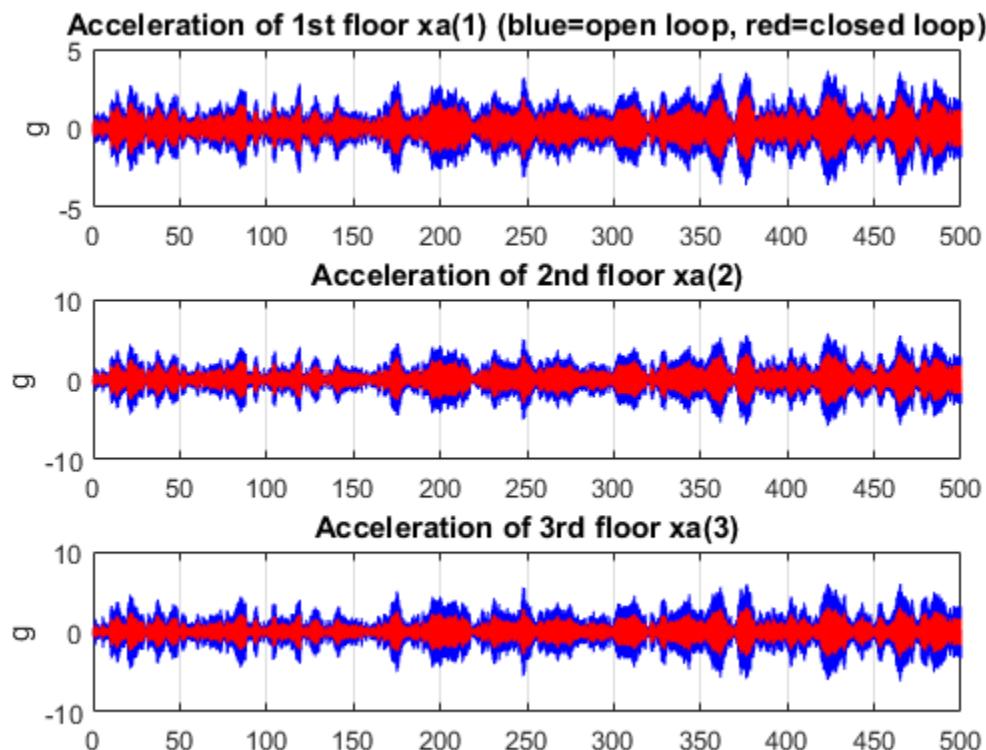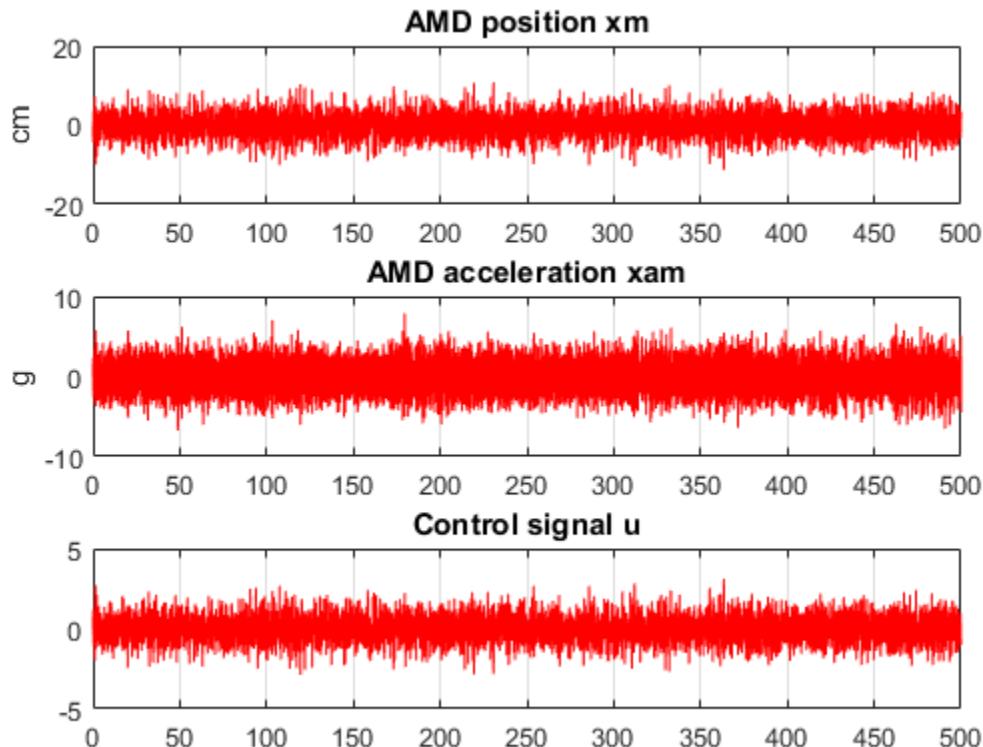
```
% Accelerations
clf, subplot(311); plot(t,ysimOL(:,9),'b',t,ysimCL(:,9),'r'); grid;
title('Acceleration of 1st floor xa(1) (blue=open loop, red=closed loop)');  ylabel('g
subplot(312); plot(t,ysimOL(:,10),'b',t,ysimCL(:,10),'r'); grid;
title('Acceleration of 2nd floor xa(2)');  ylabel('g');
subplot(313); plot(t,ysimOL(:,11),'b',t,ysimCL(:,11),'r'); grid;
title('Acceleration of 3rd floor xa(3)');  ylabel('g');
```
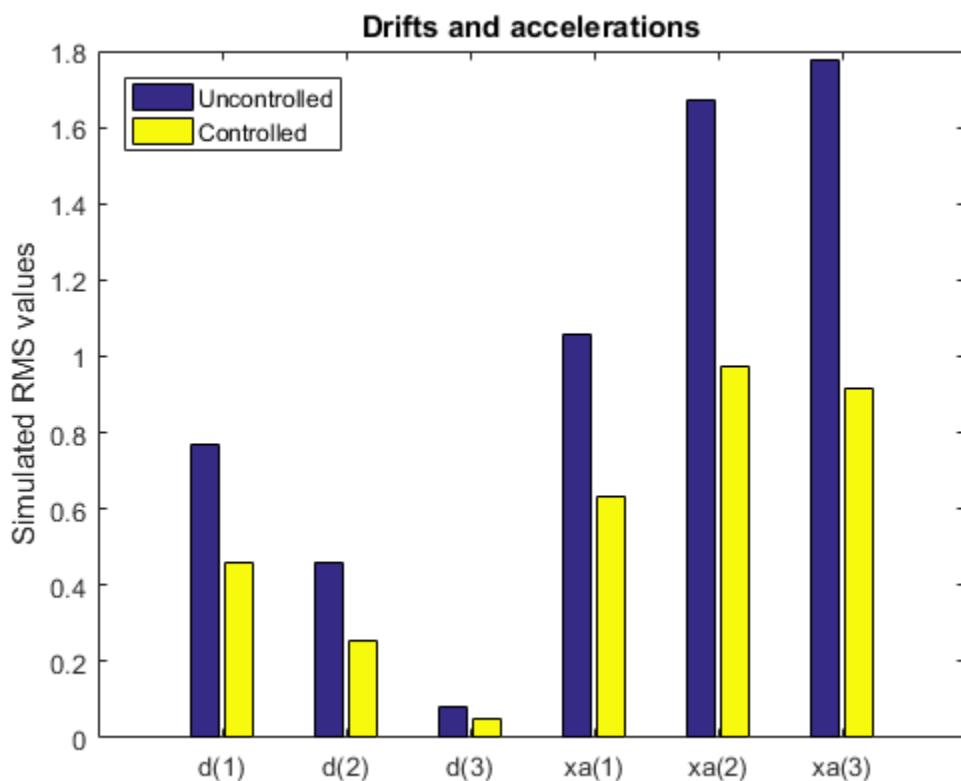
```
% Control variables
clf, subplot(311); plot(t,ysimCL(:,4),'r'); grid;
title('AMD position xm');  ylabel('cm');
subplot(312); plot(t,ysimCL(:,12),'r'); grid;
title('AMD acceleration xam');  ylabel('g');
subplot(313); plot(t,ysimCL(:,16),'r'); grid;
title('Control signal u');
```

Plot the root-mean-square (RMS) of the simulated signals for both the controlled and uncontrolled scenarios. Assuming ergodicity, the RMS performance can be estimated from a single sufficiently long simulation of the process and coincides with the standard deviations computed earlier. Indeed the RMS plot closely matches the standard deviation plot obtained earlier.

```
clf, bar([std(ysimOL(:,13:15)) std(ysimOL(:,9:11)) ; ...
    std(ysimCL(:,13:15)) std(ysimCL(:,9:11))]')
title('Drifts and accelerations')
ylabel('Simulated RMS values')
set(gca,'XTickLabel',{'d(1)','d(2)','d(3)','xa(1)','xa(2)','xa(3)'})
legend('Uncontrolled','Controlled','location','NorthWest')
```

Overall, the controller achieves significant reduction of ground vibration both in terms of drift and acceleration for all stories while meeting the hard constraints on control effort and mass displacement.
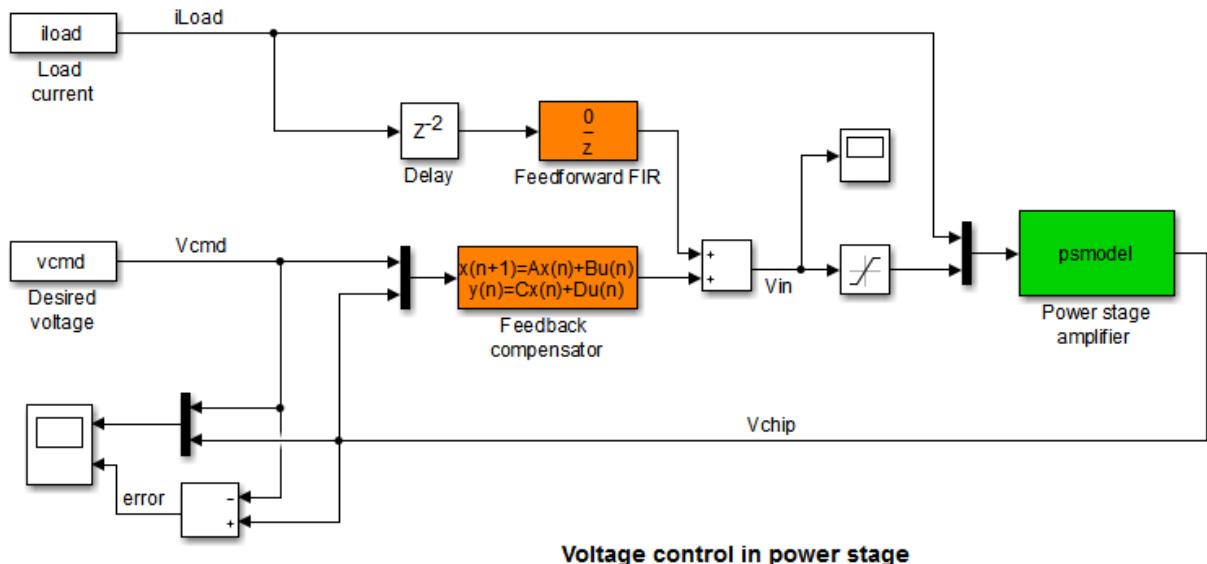
Digital Control of Power Stage Voltage

This example shows how to tune a high-performance digital controller with bandwidth close to the sampling frequency.

Voltage Regulation in Power Stage
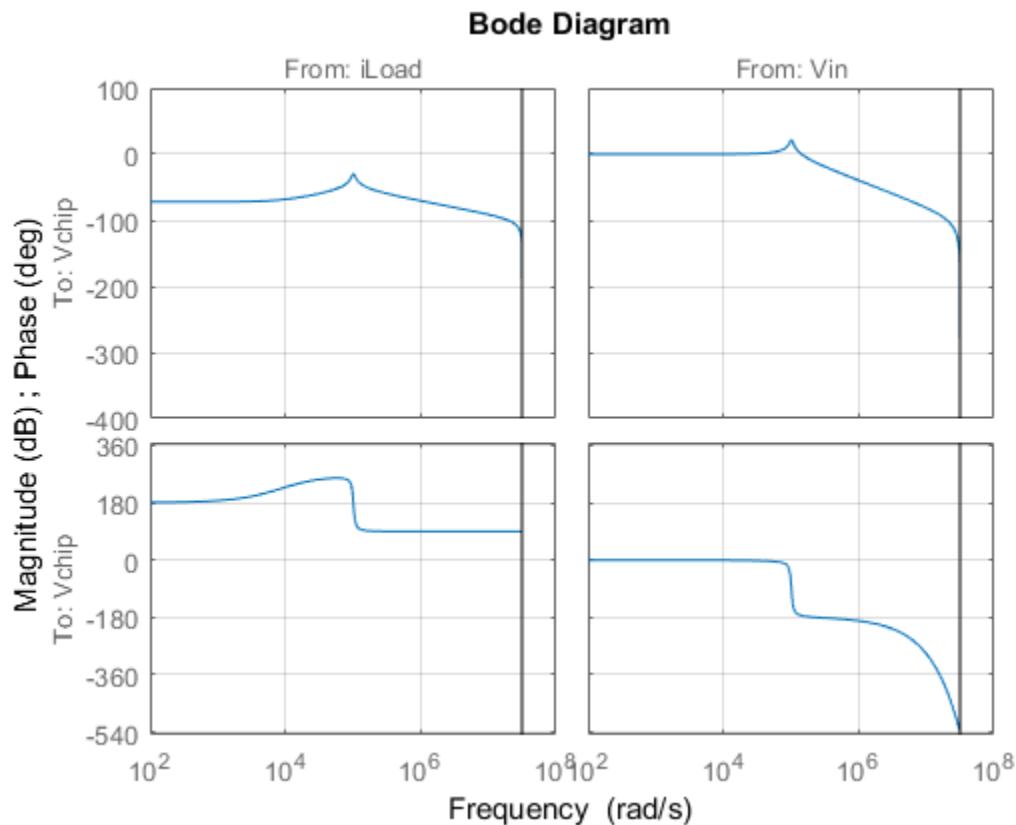
We use Simulink to model the voltage controller in the power stage for an electronic device:

```
open_system('rct_powerstage')
```


Voltage control in power stage

The power stage amplifier is modeled as a second-order linear system with the following frequency response:

```
bode(psmodel), grid
```
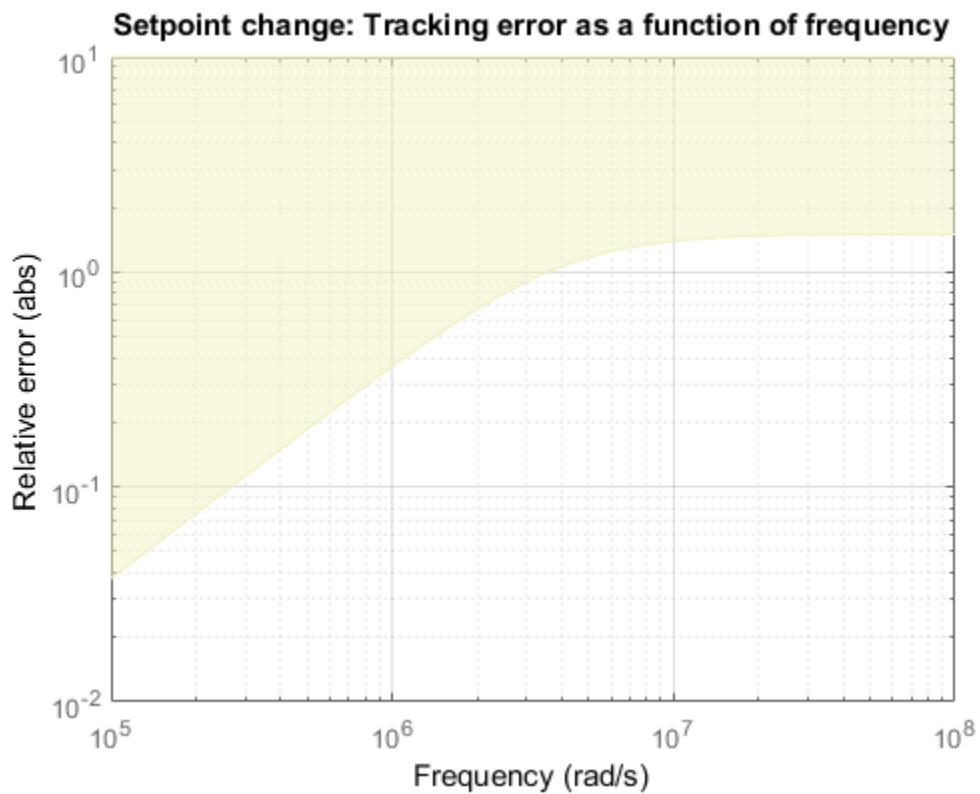
The controller must regulate the voltage Vchip delivered to the device to track the setpoint Vcmd and be insensitive to variations in load current iLoad. The control structure consists of a feedback compensator and a disturbance feedforward compensator. The voltage Vin going into the amplifier is limited to $V_{\max} = 12V$. The controller sampling rate is 10 MHz (sample time Tm is 1e-7 seconds).

Performance Requirements

This application is challenging because the controller bandwidth must approach the Nyquist frequency pi/Tm = 31.4 MHz. To avoid aliasing troubles when discretizing continuous-time controllers, it is preferable to tune the controller directly in discrete time.

The power stage should respond to a setpoint change in desired voltage `Vcmd` in about 5 sampling periods with a peak error (across frequency) of 50%. Use a tracking requirement to capture this objective.

```
Req1 = TuningGoal.Tracking('Vcmd','Vchip',5*Tm,0,1.5);
Req1.Name = 'Setpoint change';

viewSpec(Req1)
```


The power stage should also quickly reject load disturbances `iLoad`. Express this requirement in terms of gain from `iLoad` to `Vchip`. This gain should be low at low frequency for good disturbance rejection.

```
s = tf('s');
```

```
nf = pi/Tm;    % Nyquist frequency

Req2 = TuningGoal.Gain('iLoad','Vchip',1.5e-3 * s/nf);
Req2.Focus = [nf/1e4, nf];
Req2.Name = 'Load disturbance';
```

High-performance demands may lead to high control effort and saturation. For the ramp profile vcmd specified in the Simulink model (from 0 to 1 in about 250 sampling periods), we want to avoid hitting the saturation constraint $V_{in} \leq V_{max}$. Use a rate-limiting filter to model the ramp command, and require that the gain from the rate-limiter input to V_{in} be less than V_{max}.

```
RateLimiter = 1/(250*Tm*s); % models ramp command in Simulink

% |RateLimiter * (Vcmd->Vin)| < Vmax
Req3 = TuningGoal.Gain('Vcmd','Vin',Vmax/RateLimiter);
Req3.Focus = [nf/1000, nf];
Req3.Name = 'Saturation';
```

To ensure adequate robustness, require at least 7 dB gain margin and 45 degrees phase margin at the plant input.

```
Req4 = TuningGoal.Margins('Vin',7,45);
Req4.Name = 'Margins';
```

Finally, the feedback compensator has a tendency to cancel the plant resonance by notching it out. Such plant inversion may lead to poor results when the resonant frequency is not exactly known or subject to variations. To prevent this, impose a minimum closed-loop damping of 0.5 to actively damp of the plant's resonant mode.

```
Req5 = TuningGoal.Poles(0,0.5,3*nf);
Req5.Name = 'Damping';
```

Tuning

Next use systune to tune the controller parameters subject to the requirements defined above. First use the slTuner interface to configure the Simulink model for tuning. In particular, specify that there are two tunable blocks and that the model should be linearized and tuned at the sample time Tm.

```
TunedBlocks = {'compensator','FIR'};
ST0 = slTuner('rct_powerstage',TunedBlocks);
```

```
STO.Ts = Tm;

% Register points of interest for open- and closed-loop analysis
addPoint(STO,{'Vcmd','iLoad','Vchip','Vin'});
```

We want to use an FIR filter as feedforward compensator. To do this, create a parameterization of a first-order FIR filter and assign it to the "Feedforward FIR" block in Simulink.

```
FIR = tunableTF('FIR',1,1,Tm);
% Fix denominator to z^n
FIR.den.Value = [1 0];
FIR.den.Free = false;
setBlockParam(STO,'FIR',FIR);
```
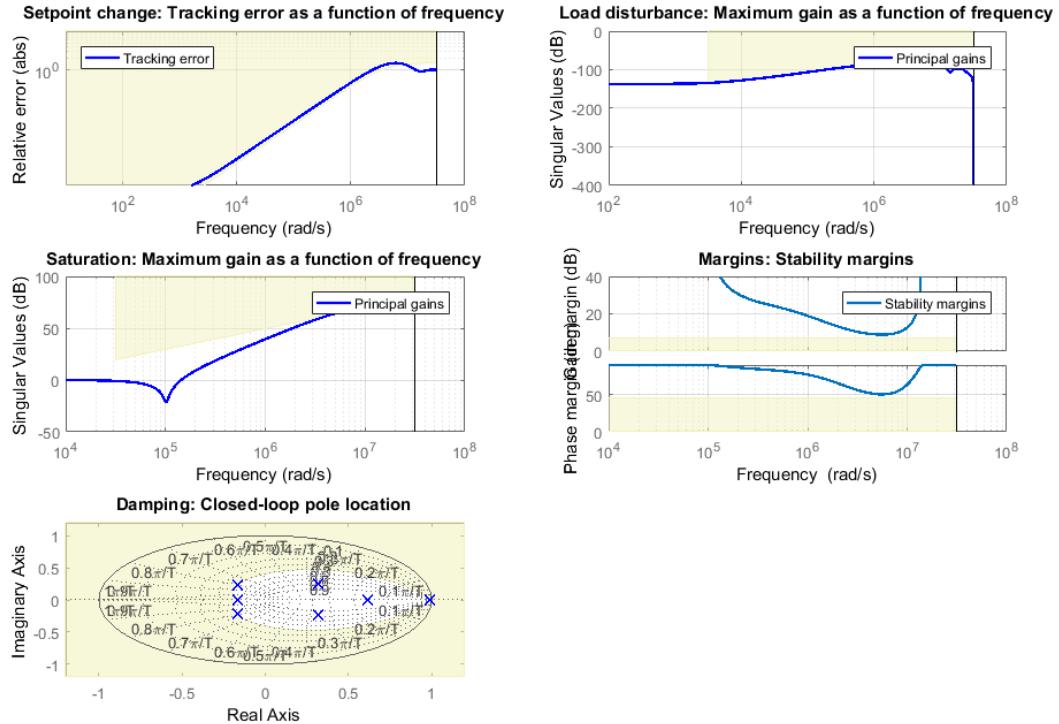
Note that `slTuner` automatically parameterizes the feedback compensator as a third-order state-space model (the order specified in the Simulink block). Next tune the feedforward and feedback compensators with `systune`. Treat the damping and margin requirements as hard constraints and try to best meet the remaining requirements.

```
rng(0)
topt = systuneOptions('RandomStart',6);
ST = systune(STO,[Req1 Req2 Req3],[Req4 Req5],topt);

Final: Soft = 1.28, Hard = 0.99934, Iterations = 452
Final: Soft = 1.69, Hard = 0.99222, Iterations = 368
Final: Soft = 1.29, Hard = 0.99396, Iterations = 420
Final: Soft = 1.29, Hard = 0.99958, Iterations = 285
Final: Soft = 1.77, Hard = 0.99949, Iterations = 420
Final: Soft = 1.3, Hard = 0.99574, Iterations = 463
Final: Soft = 1.38, Hard = 0.95232, Iterations = 383
```

The best design satisfies the hard constraints (`Hard` less than 1) and nearly satisfies the other constraints (`Soft` close to 1). Verify this graphically by plotting the tuned responses for each requirement.
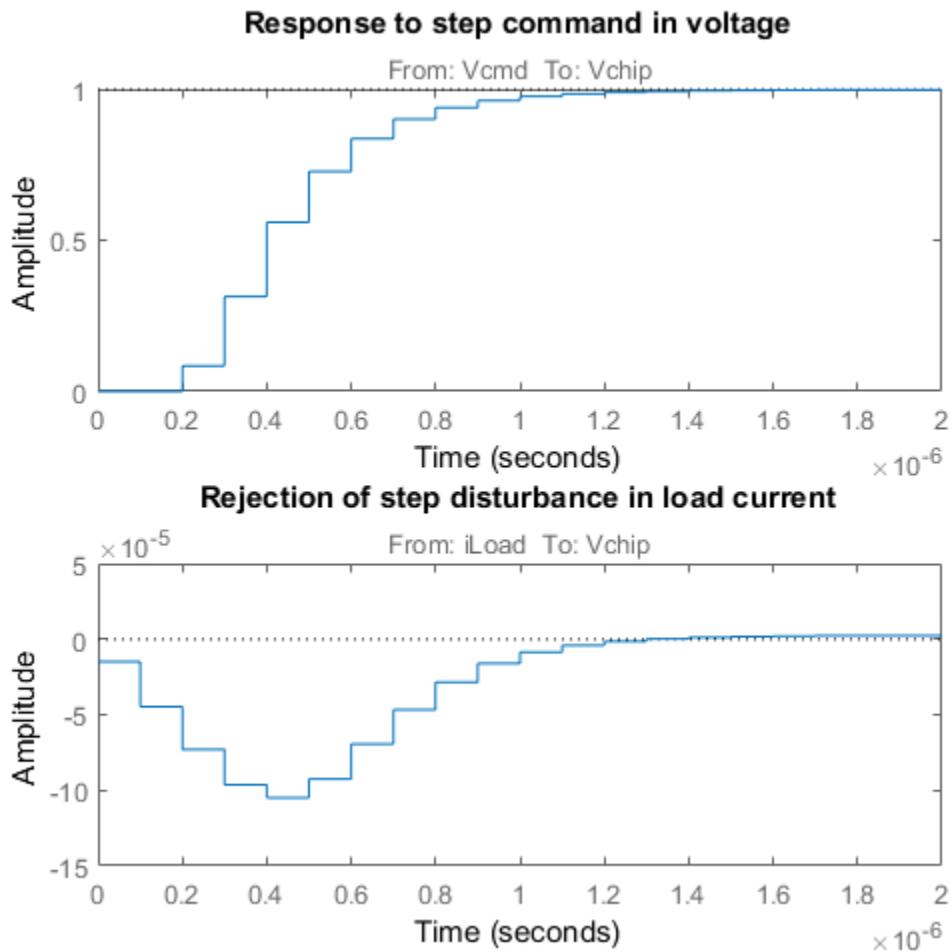
```
figure('Position',[10,10,1071,714])
viewSpec([Req1 Req2 Req3 Req4 Req5],ST)
```

Validation
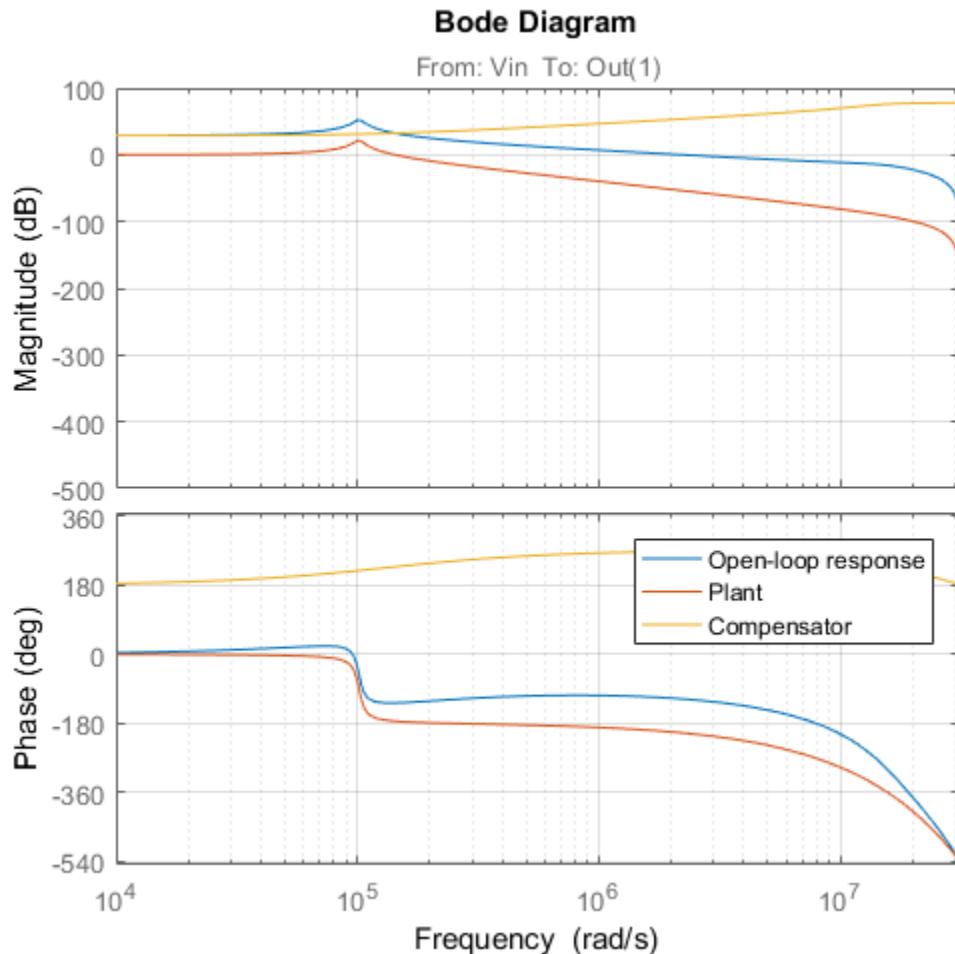
First validate the design in the linear domain using the `slTuner` interface. Plot the closed-loop response to a step command `Vcmd` and a step disturbance `iLoad`.

```
figure('Position',[100,100,560,500])
subplot(211), step(getIOTransfer(ST,'Vcmd','Vchip'),20*Tm)
title('Response to step command in voltage')
subplot(212), step(getIOTransfer(ST,'iLoad','Vchip'),20*Tm)
title('Rejection of step disturbance in load current')
```

Use `getLoopTransfer` to compute the open-loop response at the plant input and superimpose the plant and feedback compensator responses.

```
clf
L = getLoopTransfer(ST,'Vin',-1);
C = getBlockValue(ST,'compensator');
bodeplot(L,psmodel(2),C(2),{1e-3/Tm pi/Tm}), grid
legend('Open-loop response','Plant','Compensator')
```

Bode Diagram

From: Vin To: Out(1)

The controller achieves the desired bandwidth and the responses are fast enough. Apply the tuned parameter values to the Simulink model and simulate the tuned responses.

```
writeBlockValue(ST)
```

The results from the nonlinear simulation appear below. Note that the control signal `Vin` remains approximately within $\pm 12V$ saturation bounds for the setpoint tracking portion of the simulation.
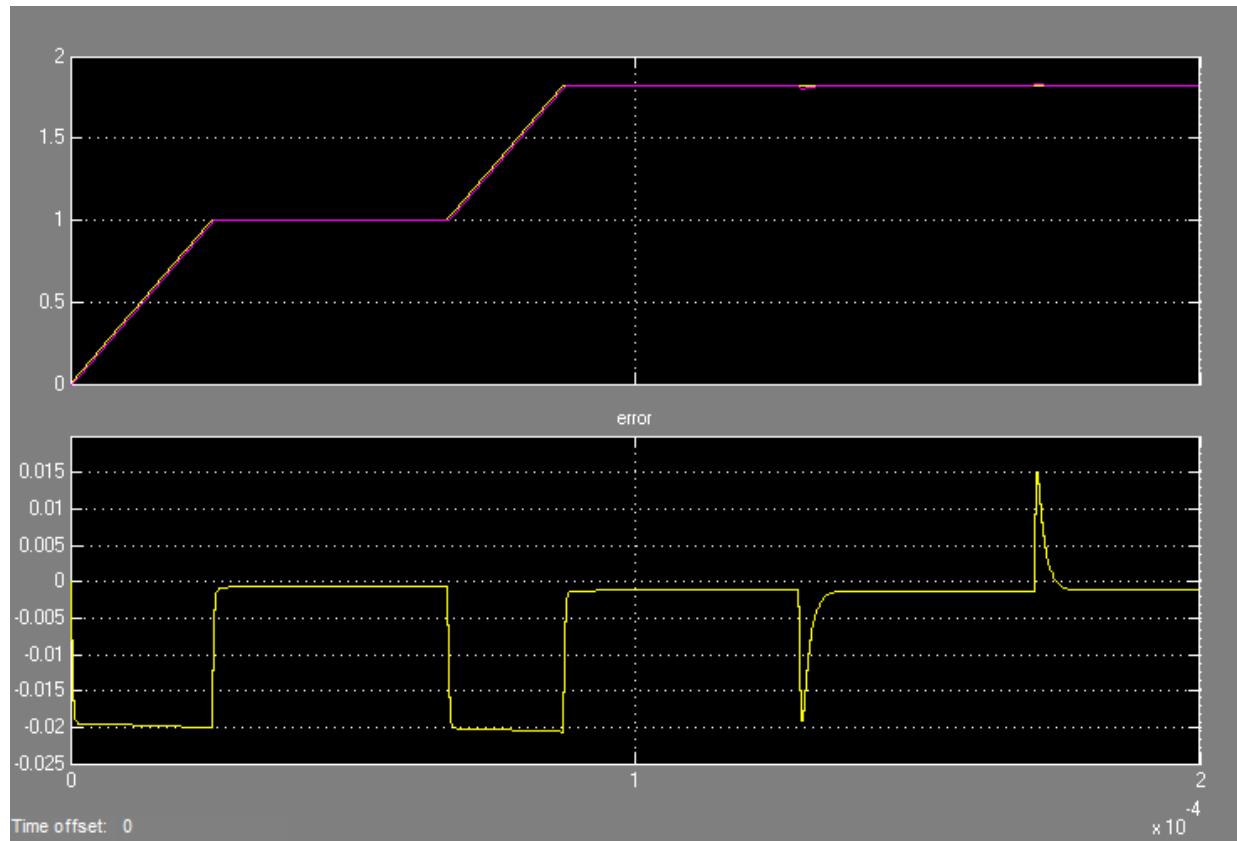
Figure 1: Response to ramp command and step load disturbances.
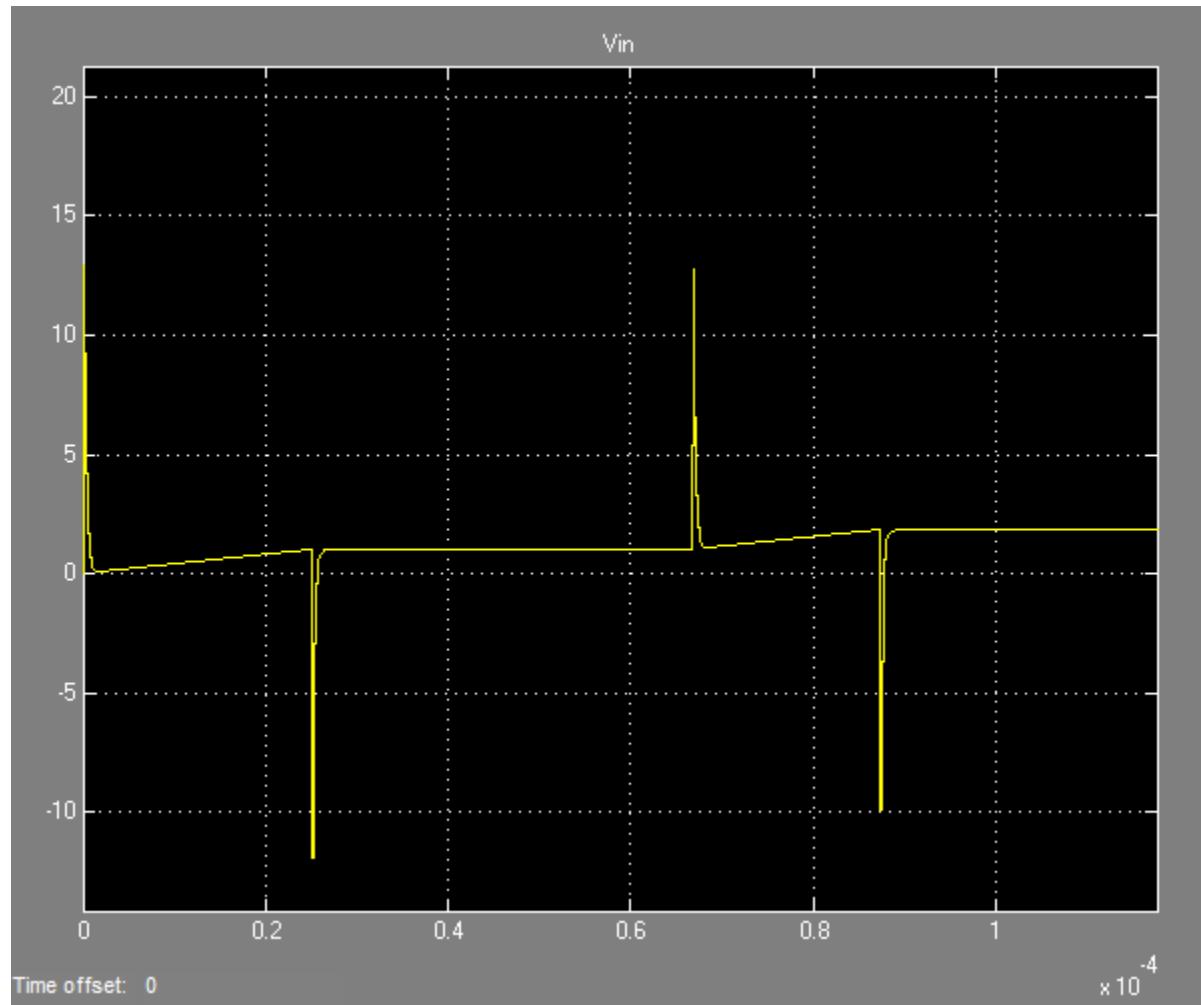
Figure 2: Amplitude of input voltage `Vin` during setpoint tracking phase.
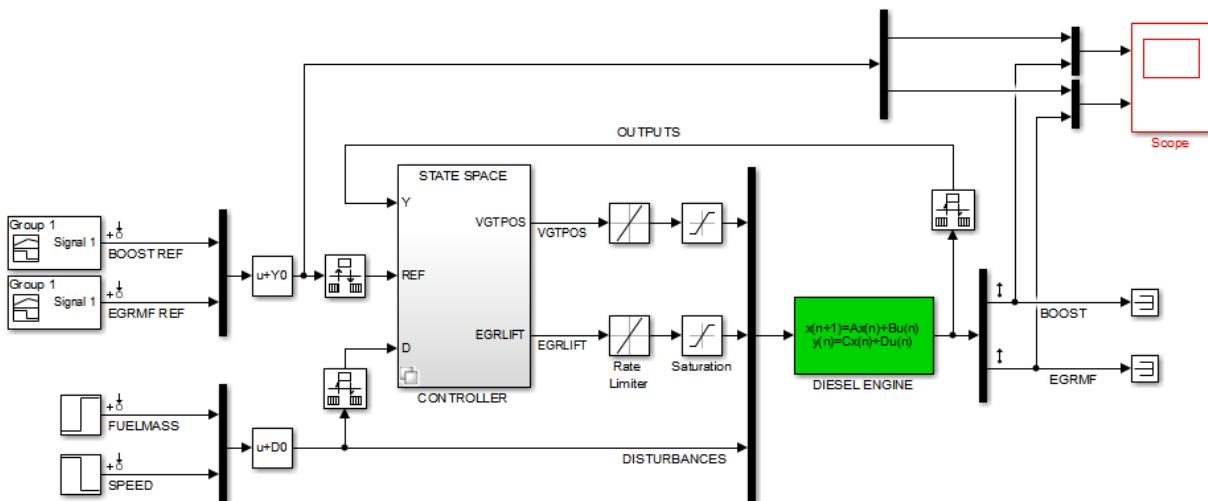
MIMO Control of Diesel Engine

This example uses `systune` to design and tune a MIMO controller for a Diesel engine. The controller is tuned in discrete time for a single operating condition.

Diesel Engine Model

Modern Diesel engines use a variable geometry turbocharger (VGT) and exhaust gas recirculation (EGR) to reduce emissions. Tight control of the VGT boost pressure and EGR massflow is necessary to meet strict emission targets. This example shows how to design and tune a MIMO controller that regulates these two variables when the engine operates at 2100 rpm with a fuel mass of 12 mg per injection-cylinder.
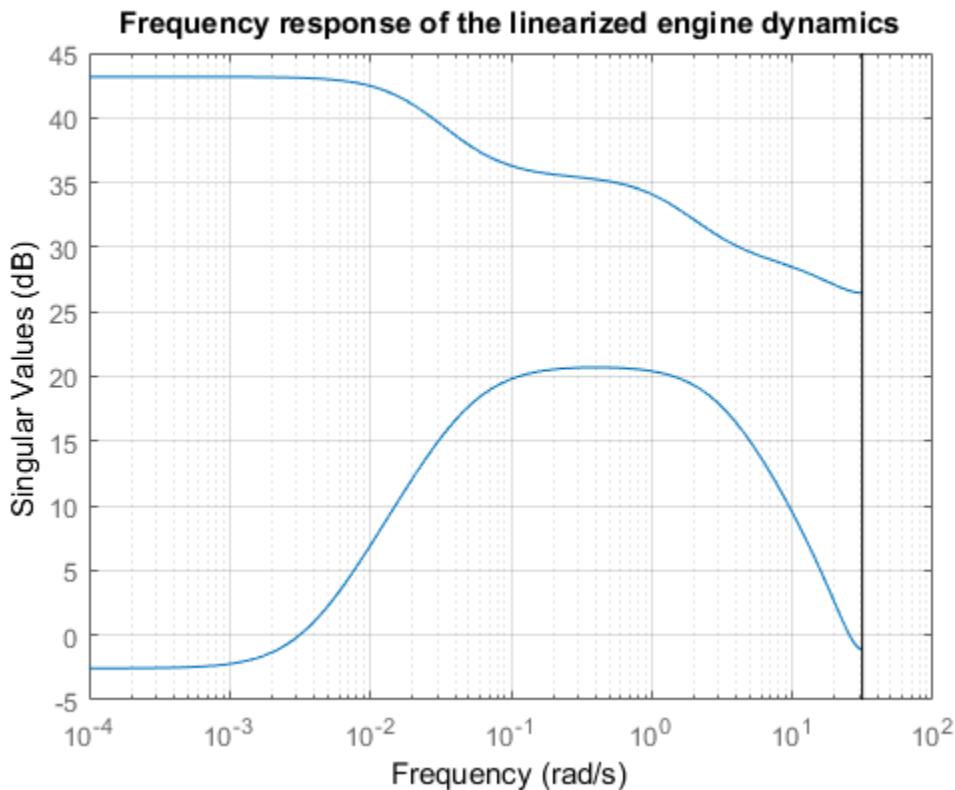
```
open_system('rct_diesel')
```


The VGT/EGR control system is modeled in Simulink. The controller adjusts the positions `EGRLIFT` and `VGTPOS` of the EGR and VGT valves. It has access to the boost pressure and EGR massflow targets and measured values, as well as fuel mass and engine speed measurements. Both valves have rate and saturation limits. The plant model is sampled every 0.1 seconds and the control signals `EGRLIFT` and `VGTPOS` are refreshed every 0.2 seconds. This example considers step changes of +10 KPa in boost pressure and +3 g/s in EGR massflow, and disturbances of +5 mg in fuel mass and -200 rpm in speed.

For the operating condition under consideration, we used System Identification to derive a linear model of the engine from experimental data. The frequency response from the manipulated variables EGRLIFT and VGTPOS to the controlled variables BOOST and EGR MF appears below. Note that the plant is ill conditioned at low frequency which makes independent control of boost pressure and EGR massflow difficult.

```
sigma(Plant(:,1:2)), grid
title('Frequency response of the linearized engine dynamics')
```


Control Objectives

There are two main control objectives:

1 Respond to step changes in boost pressure and EGR massflow in about 5 seconds with minimum cross-coupling

2 Be insensitive to (small) variations in speed and fuel mass.

Use a tracking requirement for the first objective. Specify the amplitudes of the step changes to ensure that cross-couplings are small *relative* to these changes.

```
% 5-second response time, steady-state error less than 5%
TR = TuningGoal.Tracking({'BOOST REF';'EGRMF REF'},{'BOOST';'EGRMF'},5,0.05);
TR.Name = 'Setpoint tracking';
TR.InputScaling = [10 3];
```

For the second objective, treat the speed and fuel mass variations as step disturbances and specify the peak amplitude and settling time of the resulting variations in boost pressure and EGR massflow. Also specify the signal amplitudes to properly reflect the relative contribution of each disturbance.

```
% Peak<0.5, settling time<5
DR = TuningGoal.StepRejection({'FUELMASS';'SPEED'},{'BOOST';'EGRMF'},0.5,5);
DR.Name = 'Disturbance rejection';
DR.InputScaling = [5 200];
DR.OutputScaling = [10 3];
```

To provide adequate robustness to unmodeled dynamics and aliasing, limit the control bandwidth and impose sufficient stability margins at both the plant inputs and outputs. Because we are dealing with a 2-by-2 MIMO feedback loops, these stability margins are interpreted as disk margins (see `loopmargin` and `TuningGoal.Margins` for details).
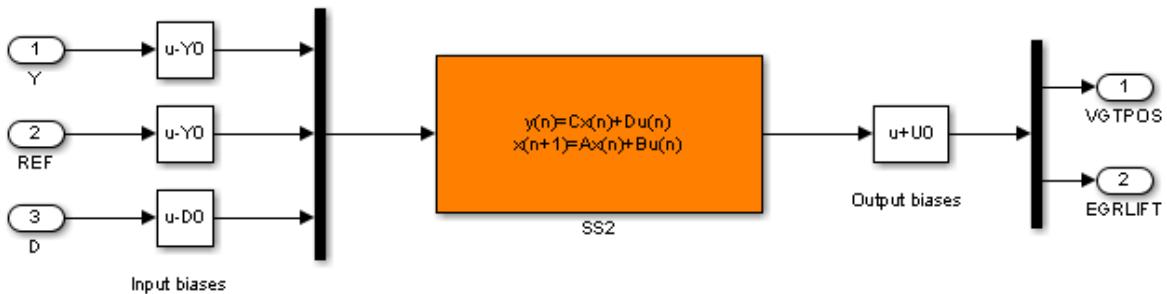
```
% Roll off of -20 dB/dec past 1 rad/s
RO = TuningGoal.MaxLoopGain({'EGRLIFT','VGTPOS'},1,1);
RO.LoopScaling = 'off';
RO.Name = 'Roll-off';

% 7 dB of gain margin and 45 degrees of phase margin
M1 = TuningGoal.Margins({'EGRLIFT','VGTPOS'},7,45);
M1.Name = 'Plant input';
M2 = TuningGoal.Margins('DIESEL ENGINE',7,45);
M2.Name = 'Plant output';
```

Tuning of Blackbox MIMO Controller

Without a-priori knowledge of a suitable control structure, first try "blackbox" state-space controllers of various orders. The plant model has four states, so try a controller of order

four or less. Here we tune a second-order controller since the "SS2" block in the Simulink model has two states.

2-DOF MIMO PID with disturbance feedforward

Figure 1: Second-order blackbox controller.

Use the `slTuner` interface to configure the Simulink model for tuning. Mark the block "SS2" as tunable, register the locations where to assess margins and loop shapes, and specify that linearization and tuning should be performed at the controller sampling rate.

```
ST0 = slTuner('rct_diesel','SS2');
ST0.Ts = 0.2;
addPoint(ST0,{'EGRLIFT','VGTPOS','DIESEL ENGINE'})
```
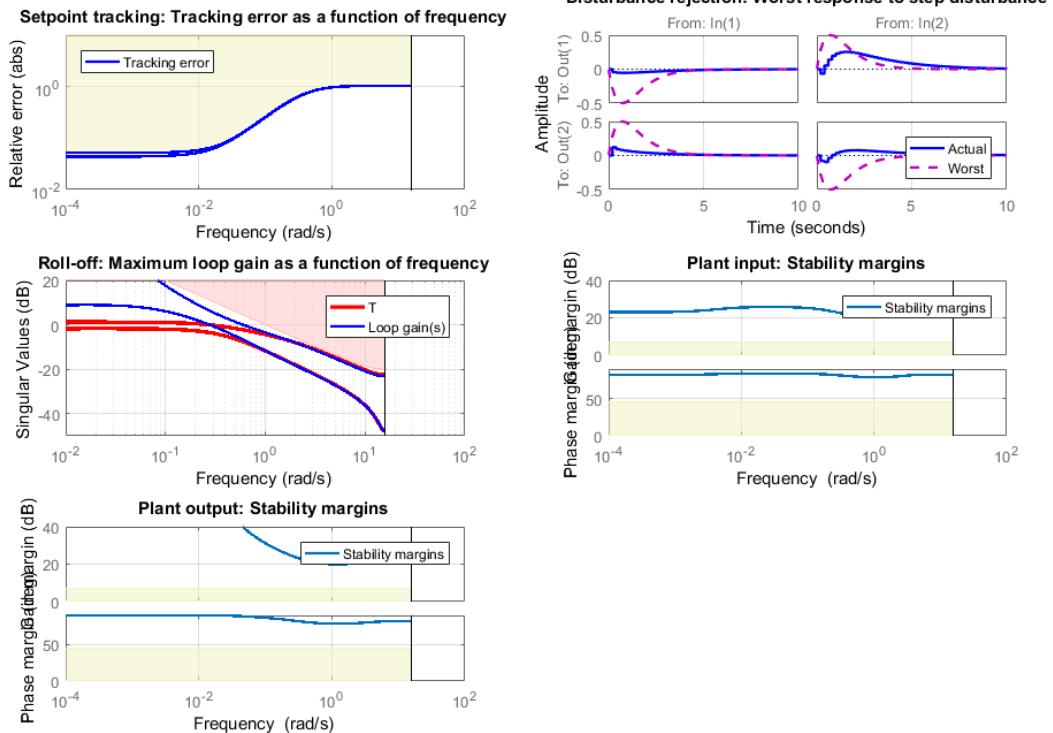
Now use `systune` to tune the state-space controller subject to our control objectives. Treat the stability margins and roll-off target as hard constraints and try to best meet the remaining objectives (soft goals). Randomize the starting point to reduce exposure to undesirable local minima.

```
Opt = systuneOptions('RandomStart',2);
rng(0), ST1 = systune(ST0,[TR DR],[M1 M2 RO],Opt);
```

```
Final: Soft = 1.05, Hard = 0.99589, Iterations = 584
Final: Soft = 1.05, Hard = 0.88839, Iterations = 509
Final: Soft = 1.05, Hard = 0.96032, Iterations = 502
```
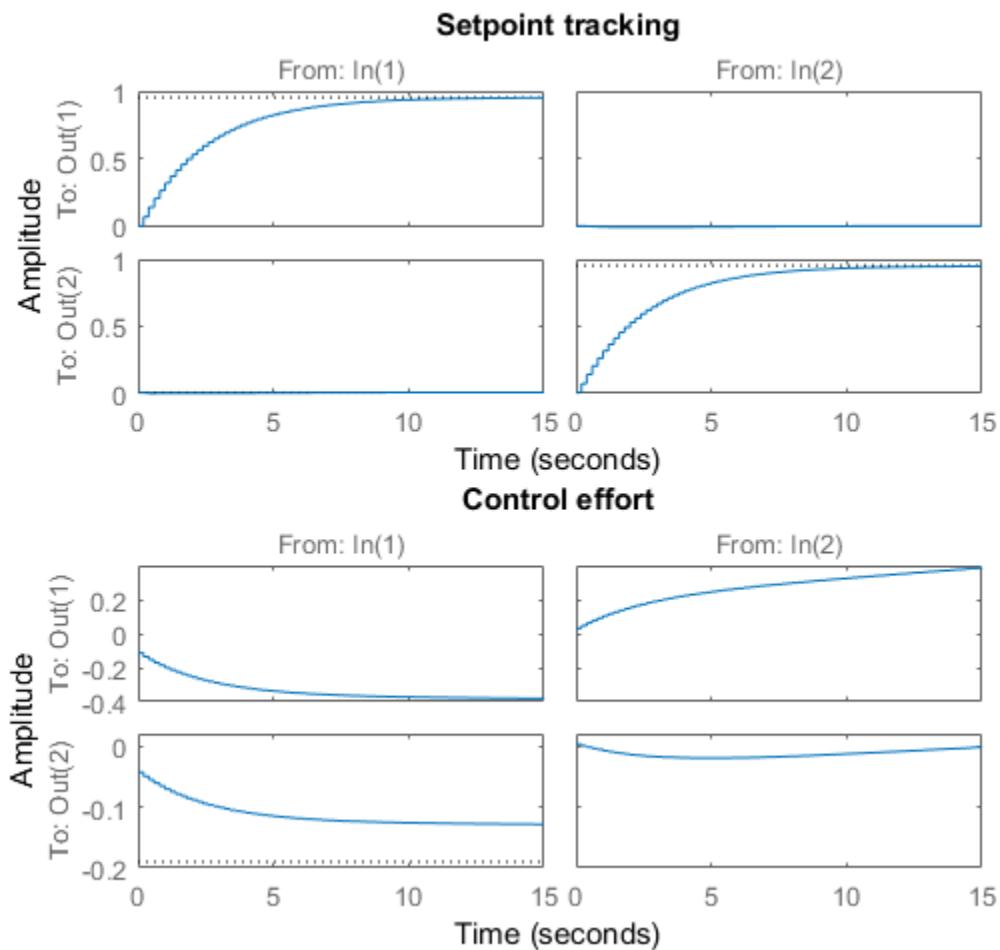
All requirements are nearly met (a requirement is satisfied when its normalized value is less than 1). Verify this graphically.

```
figure('Position',[10,10,1071,714])
viewSpec([TR DR RO M1 M2],ST1)
```
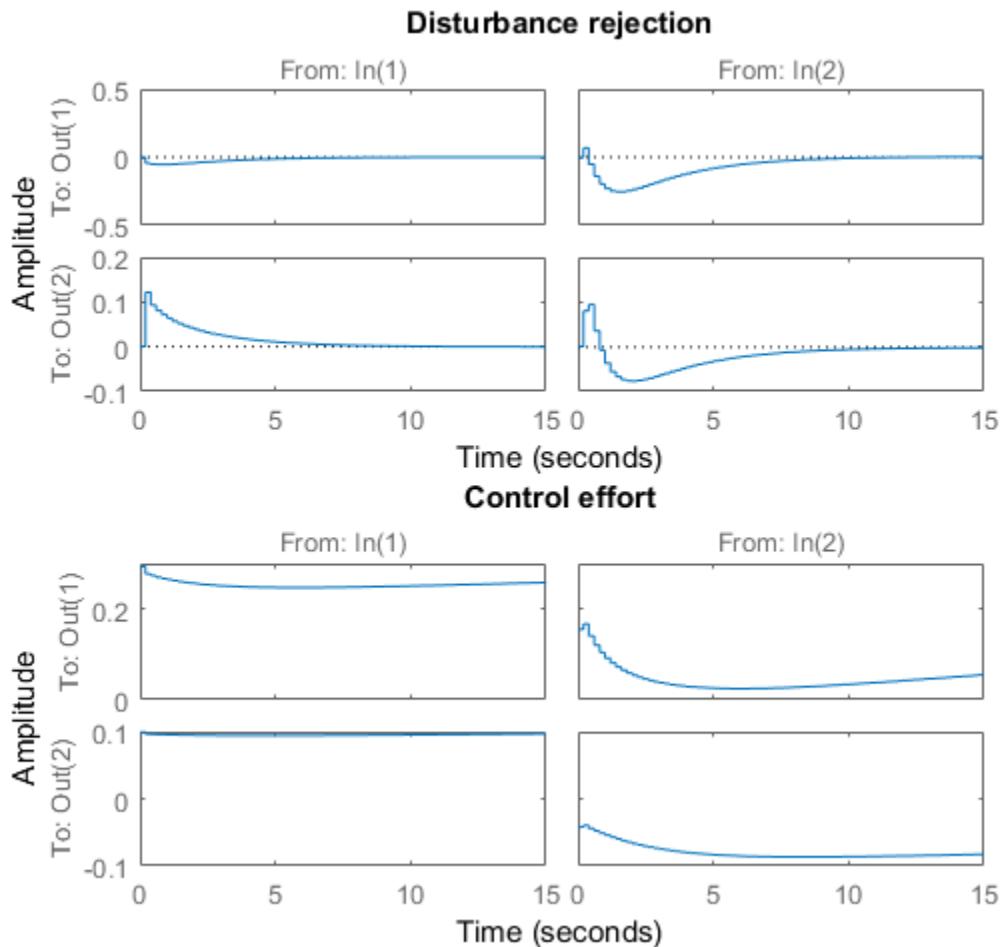

Plot the setpoint tracking and disturbance rejection responses. Scale by the signal amplitudes to show normalized effects (boost pressure changes by +10 KPa, EGR massflow by +3 g/s, fuel mass by +5 mg, and speed by -200 rpm).

```
figure('Position',[100,100,560,500])
T1 = getIOTransfer(ST1,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
T1 = diag([1/10 1/3 1 1]) * T1 * diag([10 3]);
subplot(211), step(T1(1:2,:),15), title('Setpoint tracking')
subplot(212), step(T1(3:4,:),15), title('Control effort')
```

Setpoint tracking

Control effort


```
D1 = getIOTransfer(ST1,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
D1 = diag([1/10 1/3 1 1]) * D1 * diag([5 -200]);
subplot(211), step(D1(1:2,:),15), title('Disturbance rejection')
subplot(212), step(D1(3:4,:),15), title('Control effort')
```
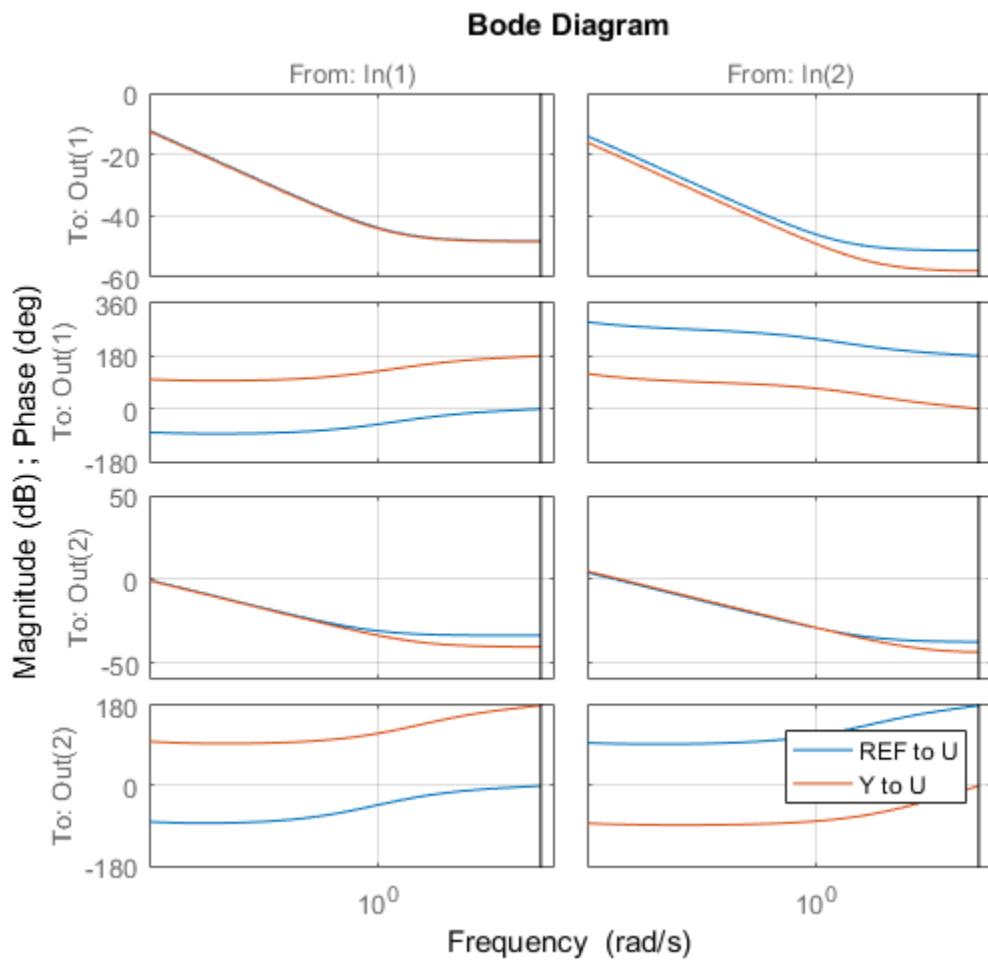
The controller responds in less than 5 seconds with minimum cross-coupling between the BOOST and EGRMF variables.

Tuning of Simplified Control Structure

The state-space controller could be implemented as is, but it is often desirable to boil it down to a simpler, more familiar structure. To do this, get the tuned controller and inspect its frequency response
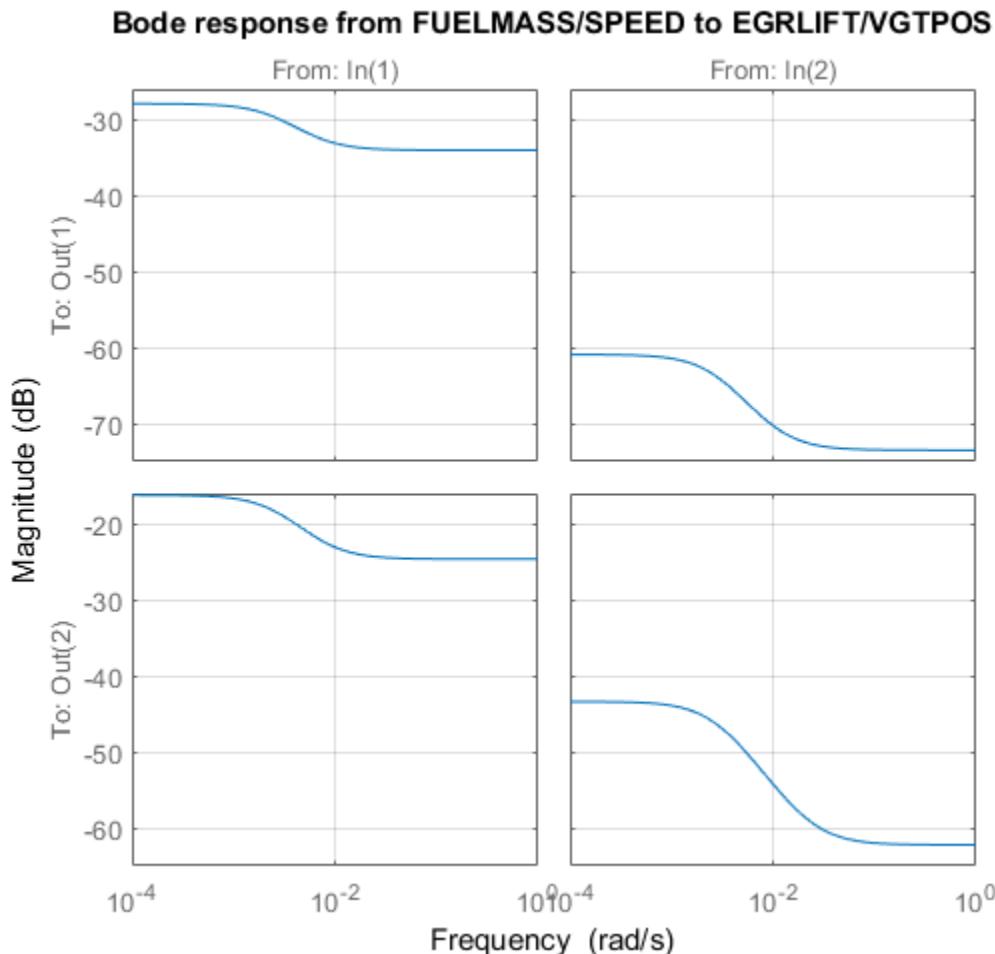
```
C = getBlockValue(ST1,'SS2');

clf
bode(C(:,1:2),C(:,3:4),{.02 20}), grid
legend('REF to U','Y to U')
```


Bode Diagram
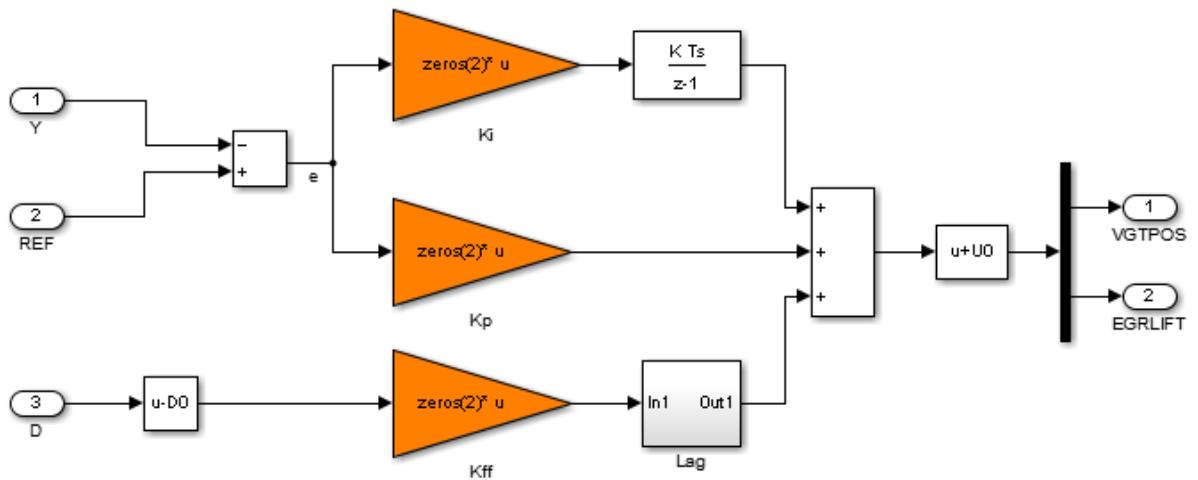
```
bodemag(C(:,5:6)), grid
```

```
title('Bode response from FUELMASS/SPEED to EGRLIFT/VGTPOS')
```


The first plot suggests that the controller essentially behaves like a PI controller acting on REF-Y (the difference between the target and actual values of the controlled variables). The second plot suggests that the transfer from measured disturbance to manipulated variables could be replaced by a gain in series with a lag network. Altogether this suggests the following simplified control structure consisting of a MIMO PI controller with a first-order disturbance feedforward.

MIMO PID with disturbance feedforward

Figure 2: Simplified control structure.

Using variant subsystems, you can implement both control structures in the same Simulink model and use a variable to switch between them. Here setting `MODE=2` selects the MIMO PI structure. As before, use `systune` to tune the three 2-by-2 gain matrices `Kp`, `Ki`, `Kff` in the simplified control structure.

```
% Select "MIMO PI" variant in "CONTROLLER" block
MODE = 2;

% Configure tuning interface
ST0 = slTuner('rct_diesel',{'Kp','Ki','Kff'});
ST0.Ts = 0.2;
addPoint(ST0,{'EGRLIFT','VGTPOS','DIESEL ENGINE'})

% Tune MIMO PI controller.
ST2 = systune(ST0,[TR DR],[M1 M2 RO]);

Final: Soft = 1.09, Hard = 0.99687, Iterations = 322
```
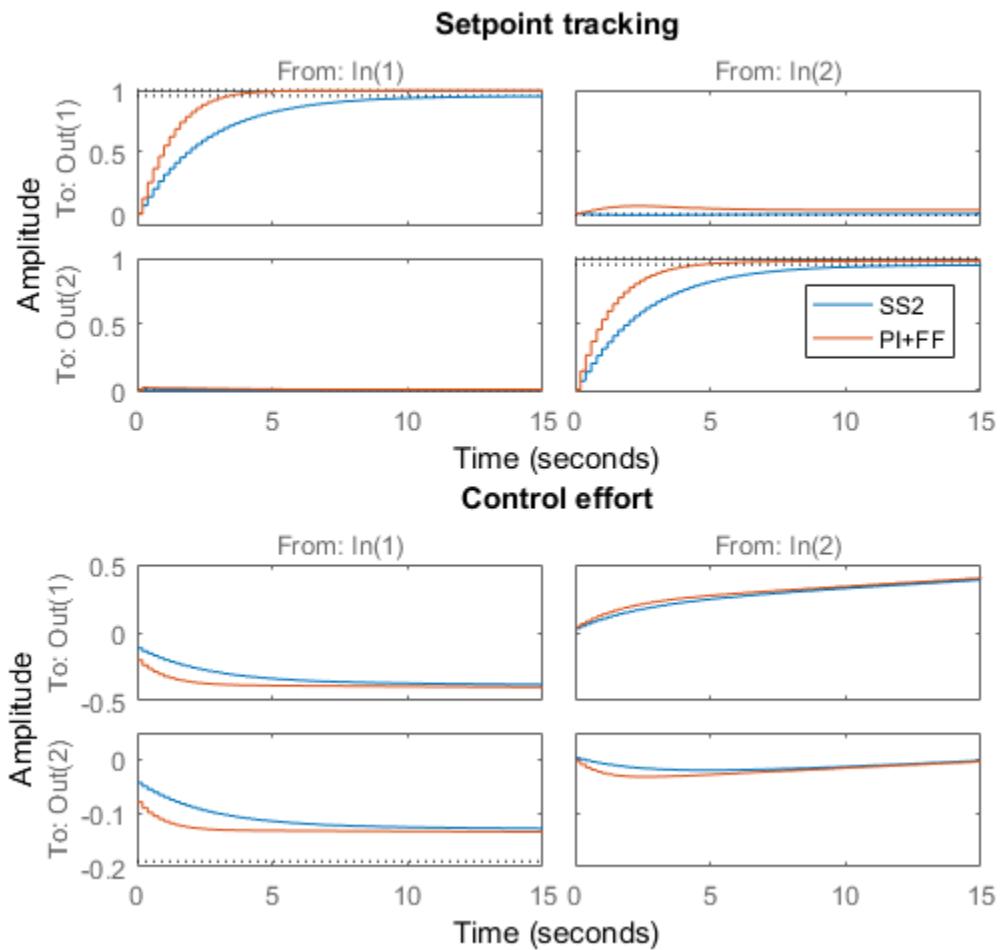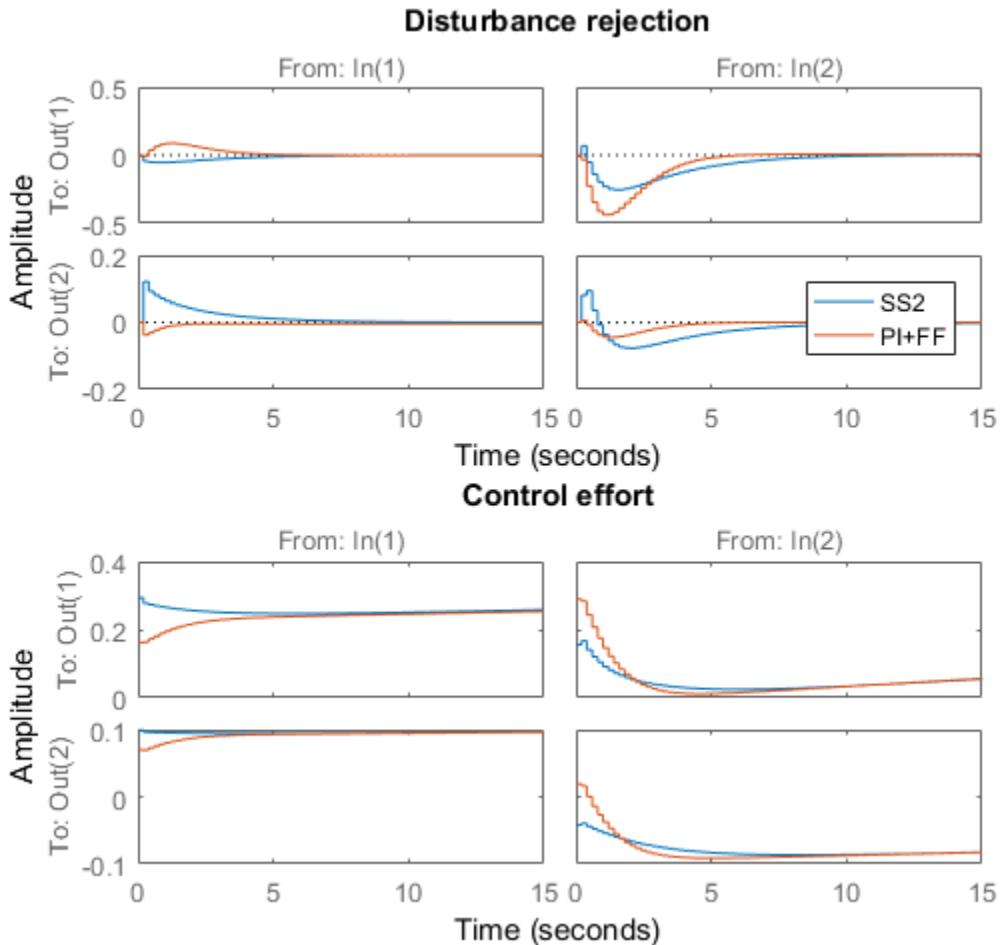
Again all requirements are nearly met. Plot the closed-loop responses and compare with the state-space design.

```
clf
T2 = getIOTransfer(ST2,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
T2 = diag([1/10 1/3 1 1]) * T2 * diag([10 3]);
subplot(211), step(T1(1:2,:),T2(1:2,:),15), title('Setpoint tracking')
legend('SS2','PI+FF')
subplot(212), step(T1(3:4,:),T2(3:4,:),15), title('Control effort')
```

```
D2 = getIOTransfer(ST2,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
D2 = diag([1/10 1/3 1 1]) * D2 * diag([5 -200]);
subplot(211), step(D1(1:2,:),D2(1:2,:),15), title('Disturbance rejection')
legend('SS2','PI+FF')
subplot(212), step(D1(3:4,:),D2(3:4,:),15), title('Control effort')
```


The blackbox and simplified control structures deliver similar performance. Inspect the tuned values of the PI and feedforward gains.

```
showTunable(ST2)


Block 1: rct_diesel/CONTROLLER/MIMO PID/Kp =

  D =
               u1          u2
    y1   -0.007937  -0.0008536
    y2     -0.0203     0.01402

Name: Kp
Static gain.

----------------------------------

Block 2: rct_diesel/CONTROLLER/MIMO PID/Ki =

  D =
               u1          u2
    y1  -0.01056   -0.01322
    y2  -0.03006    0.04472

Name: Ki
Static gain.

----------------------------------

Block 3: rct_diesel/CONTROLLER/MIMO PID/Kff =

  D =
               u1          u2
    y1     0.01403  -9.443e-05
    y2     0.03251    -0.00145

Name: Kff
Static gain.
```

Nonlinear Validation

To validate the MIMO PI controller in the Simulink model, push the tuned controller parameters to Simulink and run the simulation.

```
writeBlockValue(ST2)
```

The simulation results are shown below and confirm that the controller adequately tracks setpoint changes in boost pressure and EGR massflow and quickly rejects changes in fuel mass (at t=90) and in speed (at t=110).
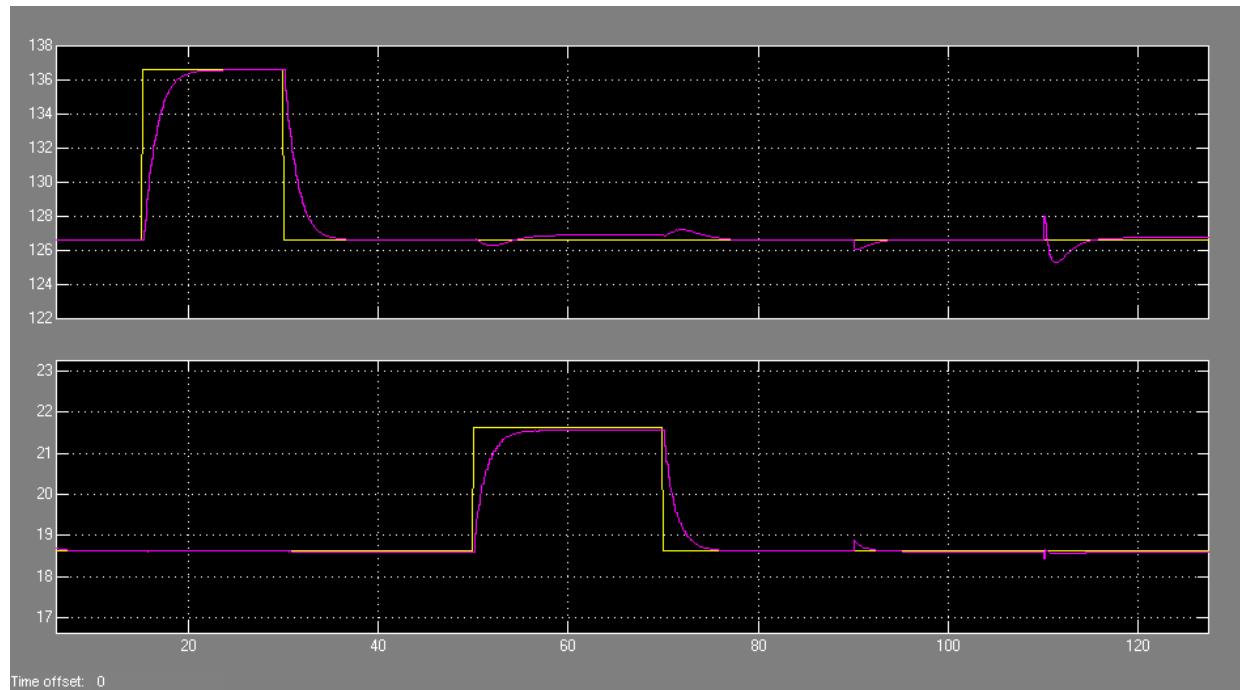
Figure 3: Simulation results with simplified controller.

Tuning of a Two-Loop Autopilot
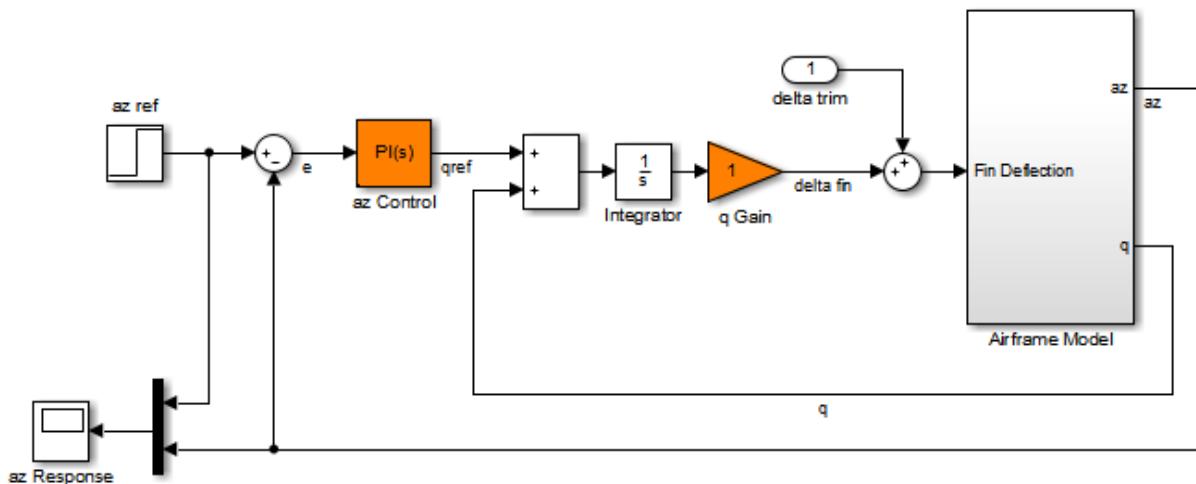
This example shows how to use Simulink Control Design to tune a two-loop autopilot controlling the pitch rate and vertical acceleration of an airframe.

Model of Airframe Autopilot

The airframe dynamics and the autopilot are modeled in Simulink.

```
open_system('rct_airframe1')
```

Two-loop autopilot for controlling the vertical acceleration of an airframe

See AutopilotExample to see how you can tune the autopilot gains with the LOOPTUNE command.

The autopilot consists of two cascaded loops. The inner loop controls the pitch rate q, and the outer loop controls the vertical acceleration `az` in response to the pilot stick command `azref`. In this architecture, the tunable elements include the PI controller gains ("az Control" block) and the pitch-rate gain ("q Gain" block). The autopilot must be tuned to respond to a step command `azref` in about 1 second with minimal overshoot. In this example, we tune the autopilot gains for one flight condition corresponding to zero incidence and a speed of 984 m/s.

To analyze the airframe dynamics, trim the airframe for $\alpha = 0$ and $V = 984m/s$. The trim condition corresponds to zero normal acceleration and pitching moment (w and q steady). Use `findop` to compute the corresponding closed-loop operating condition. Note that we added a "delta trim" input port so that `findop` can adjust the fin deflection to produce the desired equilibrium of forces and moments.

```
opspec = operspec('rct_airframe1');

% Specify trim condition
% Xe,Ze: known, not steady
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
% u,w: known, w steady
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
% theta: known, not steady
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
% q: unknown, steady
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
% integrator states unknown, not steady
opspec.States(5).SteadyState = 0;
opspec.States(6).SteadyState = 0;

op = findop('rct_airframe1',opspec);


 Operating Point Search Report:
---------------------------------

 Operating Report for the Model rct_airframe1.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:           0      dx:           984
      x:     -3.05e+03    dx:             0
(2.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:           0      dx:      -0.00972
(3.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:         984      dx:          22.7
```

```
      x:                0      dx:      -1.44e-11 (0)
(4.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:         -0.00972     dx:      1.72e-16 (0)
(5.) rct_airframe1/Integrator
      x:         0.000708     dx:      -0.00972
(6.) rct_airframe1/az Control/Integrator
      x:                0     dx:      0.000242

Inputs:
----------
(1.) rct_airframe1/delta trim
      u:       0.000708    [-Inf Inf]

Outputs: None
----------
```
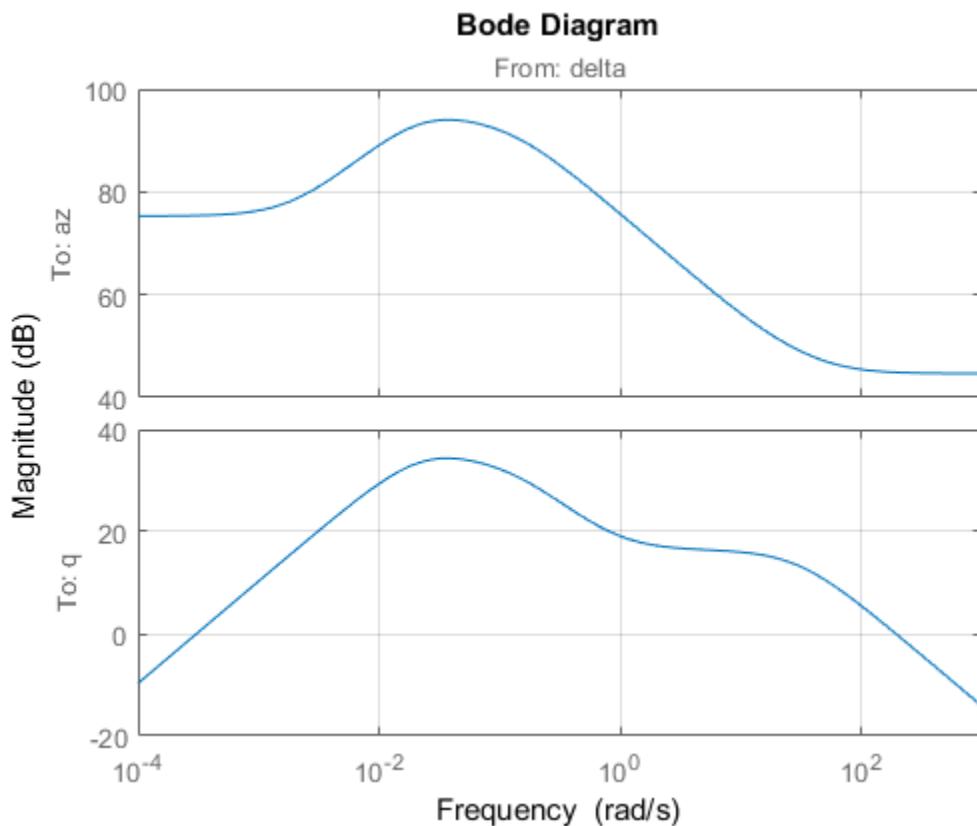
Linearize the "Airframe Model" block for the computed trim condition op and plot the gains from the fin deflection delta to az and q:

```
G = linearize('rct_airframe1','rct_airframe1/Airframe Model',op);
G.InputName = 'delta';
G.OutputName = {'az','q'};

bodemag(G), grid
```

Note that the airframe model has an unstable pole:

```
pole(G)
```

```
ans =

   -0.0320
   -0.0255
    0.1253
  -29.4685
```

Frequency-Domain Tuning with LOOPTUNE

You can use the `looptune` function to automatically tune multi-loop control systems subject to basic requirements such as integral action, adequate stability margins, and desired bandwidth. To apply `looptune` to the autopilot model, create an instance of the `slTuner` interface and designate the Simulink blocks "az Control" and "q Gain" as tunable. Also specify the trim condition `op` to correctly linearize the airframe dynamics.

```
ST0 = slTuner('rct_airframe1',{'az Control','q Gain'},op);
```

Mark the reference, control, and measurement signals as points of interest for analysis and tuning.

```
addPoint(ST0,{'az ref','delta fin','az','q'});
```
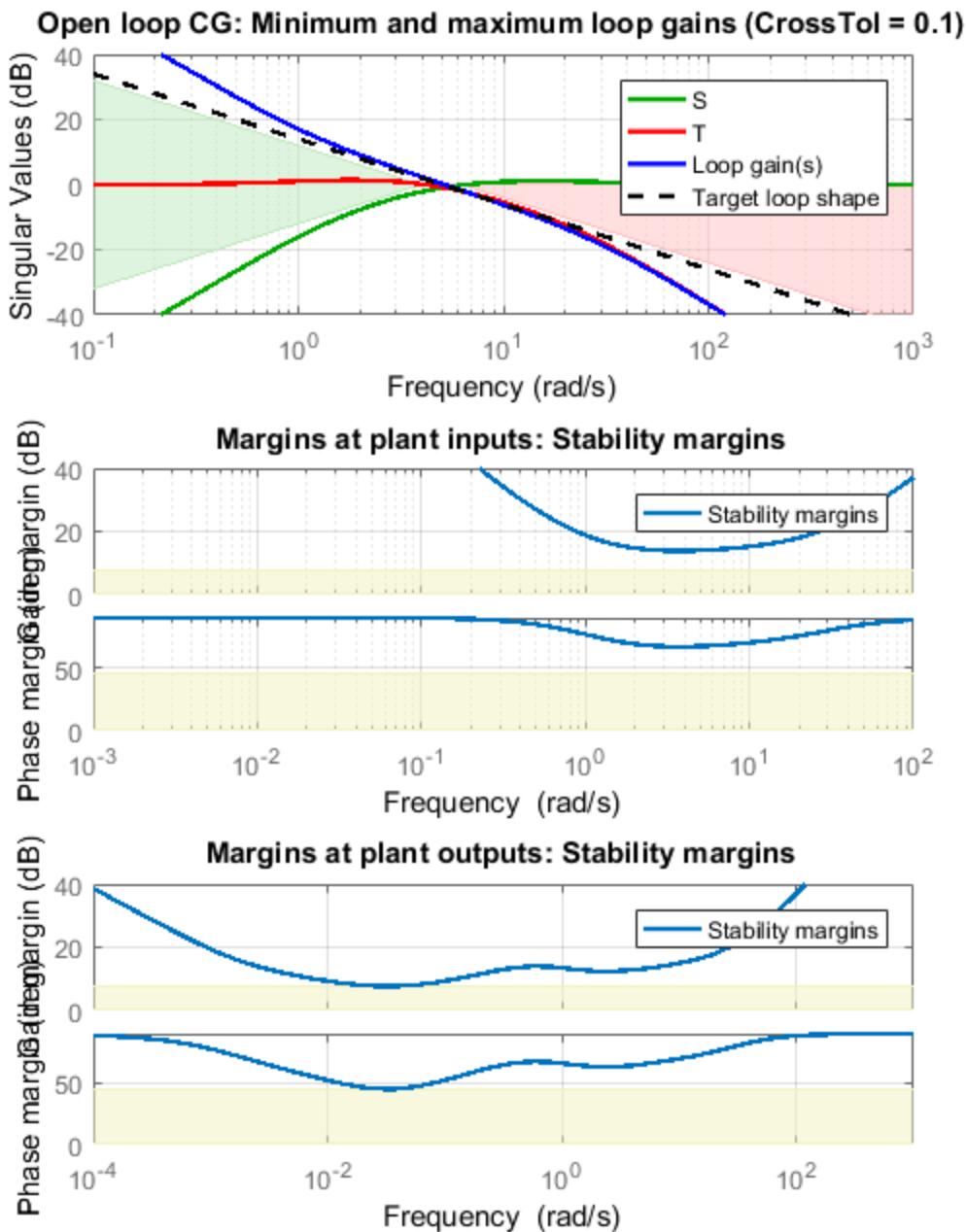
Finally, tune the control system parameters to meet the 1 second response time requirement. In the frequency domain, this roughly corresponds to a gain crossover frequency wc = 5 rad/s for the open-loop response at the plant input "delta fin".

```
wc = 5;
Controls = 'delta fin';
Measurements = {'az','q'};
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc);

Final: Peak gain = 1.01, Iterations = 52
```
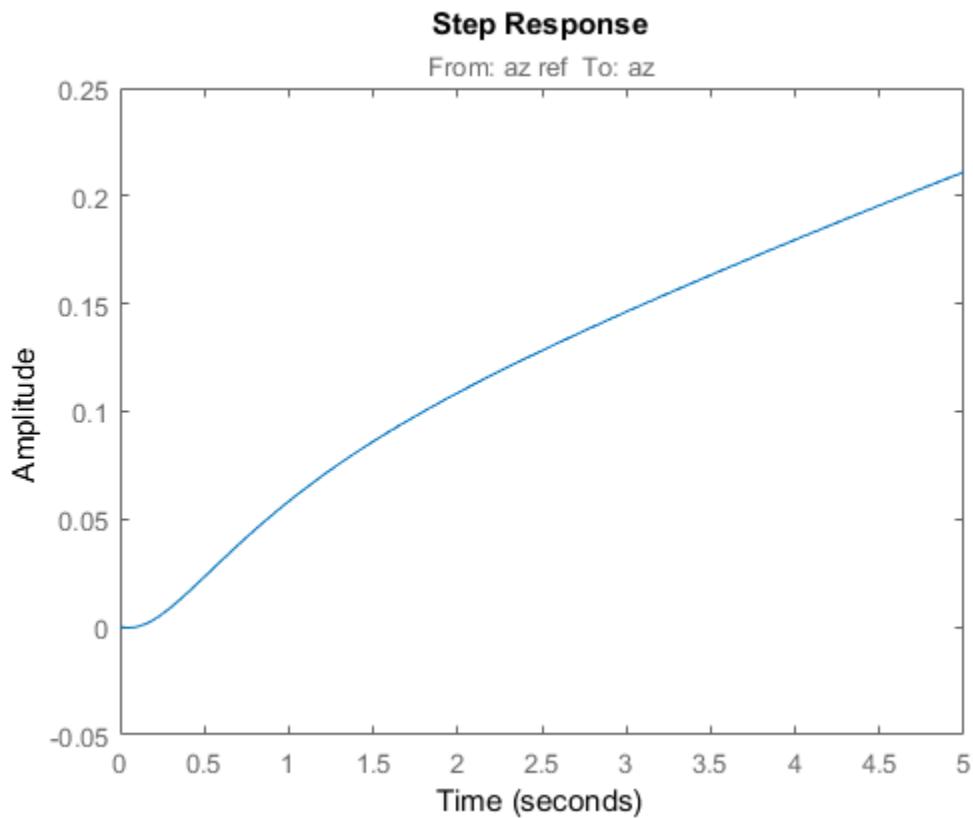
The requirements are normalized so a final value near 1 means that all requirements are met. Confirm this by graphically validating the design.

```
figure('Position',[100,100,560,714])
loopview(ST,Info)
```

Open loop CG: Minimum and maximum loop gains (CrossTol = 0.1)

Margins at plant inputs: Stability margins

Margins at plant outputs: Stability margins

The first plot confirms that the open-loop response has integral action and the desired gain crossover frequency while the second plot shows that the MIMO stability margins are satisfactory (the blue curve should remain below the yellow bound). Next check the response from the step command `azref` to the vertical acceleration `az`:

```
T = getIOTransfer(ST,'az ref','az');
figure, step(T,5)
```

Step Response

From: az ref To: az

The acceleration `az` does not track `azref` despite the presence of an integrator in the loop. This is because the feedback loop acts on the two variables `az` and `q` and we have not specified which one should track `azref`.

Adding a Tracking Requirement
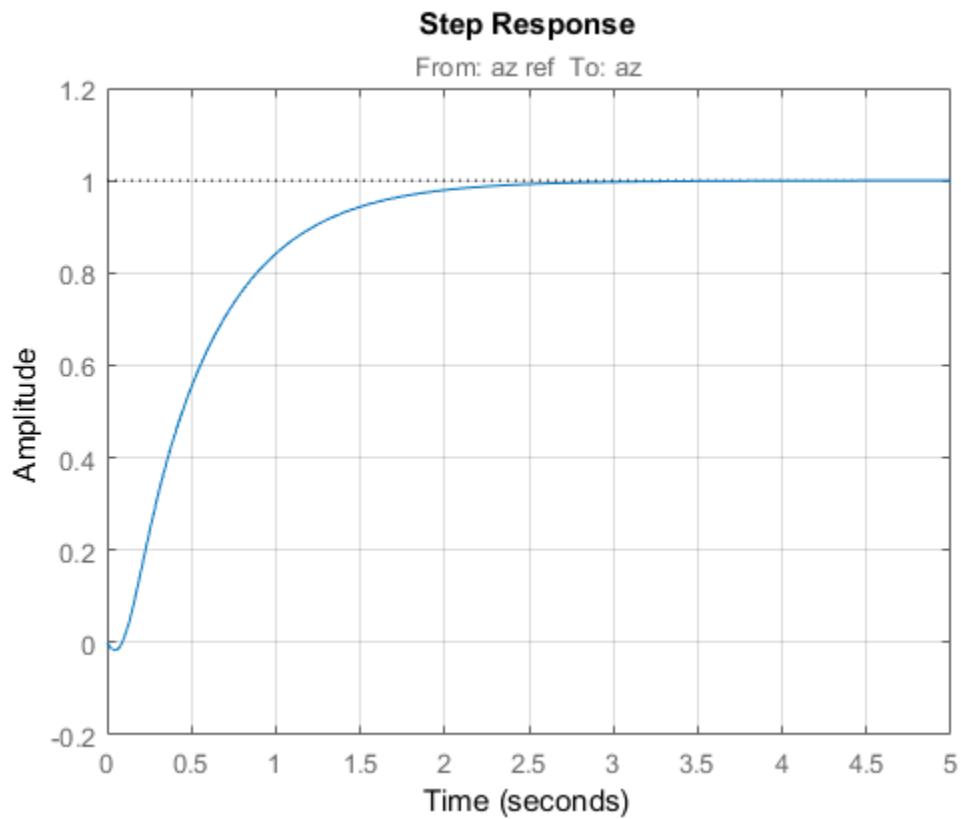
To remedy this issue, add an explicit requirement that `az` should follow the step command `azref` with a 1 second response time. Also relax the gain crossover requirement to the interval [3,12] to let the tuner find the appropriate gain crossover frequency.

```
TrackReq = TuningGoal.Tracking('az ref','az',1);
ST = looptune(STO,Controls,Measurements,[3,12],TrackReq);

Final: Peak gain = 1.23, Iterations = 46
```
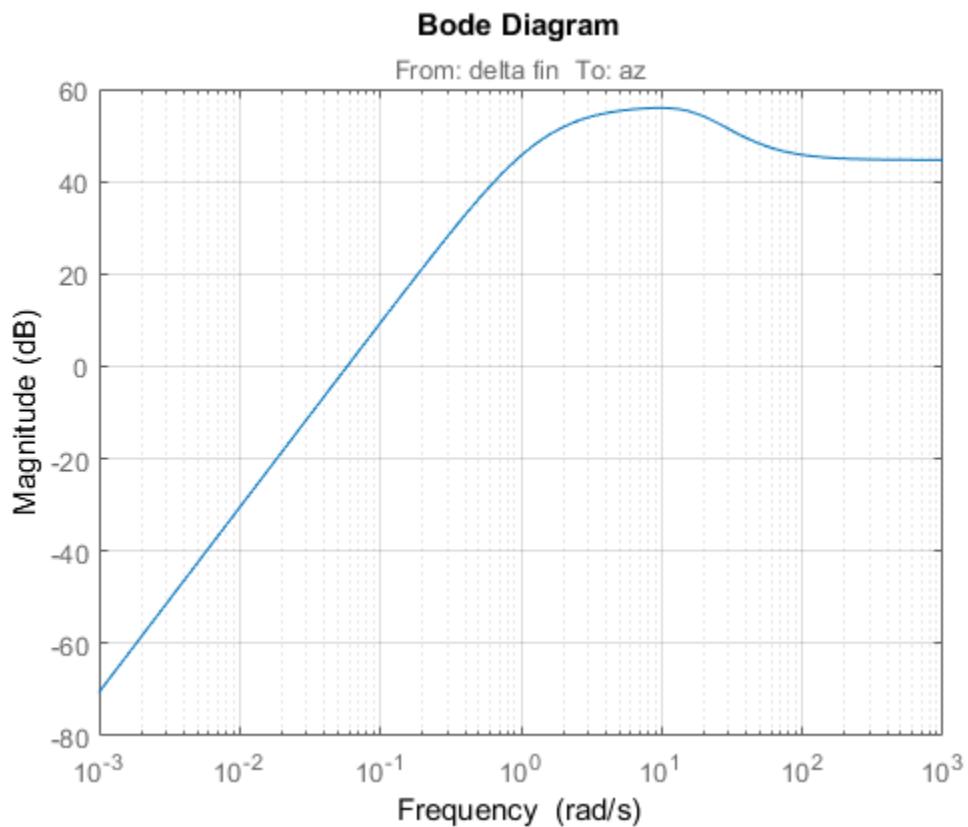
The step response from `azref` to `az` is now satisfactory:

```
Tr1 = getIOTransfer(ST,'az ref','az');
step(Tr1,5), grid
```

Step Response

From: az ref To: az
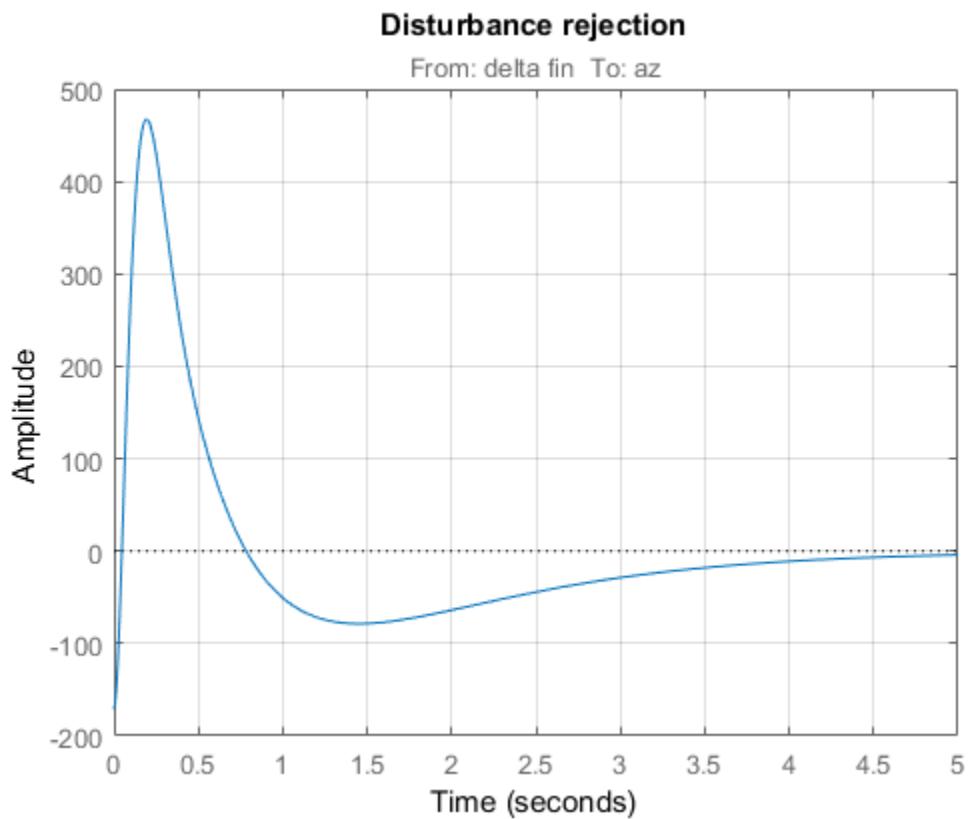
Also check the disturbance rejection characteristics by looking at the responses from a disturbance entering at the plant input

```
Td1 = getIOTransfer(ST,'delta fin','az');
bodemag(Td1), grid
```

Bode Diagram

From: delta fin To: az


```
step(Td1,5), grid, title('Disturbance rejection')
```

Disturbance rejection

From: delta fin To: az

Use `showBlockValue` to see the tuned values of the PI controller and inner-loop gain

```
showBlockValue(ST)

AnalysisPoints_ =

  D =
       u1   u2   u3   u4
    y1   1    0    0    0
    y2   0    1    0    0
    y3   0    0    1    0
    y4   0    0    0    1

Name: AnalysisPoints_
```

```
Static gain.
---------------------------------
az_Control =

              1
  Kp + Ki * ---
              s

  with Kp = 0.00166, Ki = 0.0017

Name: az_Control
Continuous-time PI controller in parallel form.
---------------------------------
q_Gain =

  D =
          u1
    y1  1.987

Name: q_Gain
Static gain.
```

If this design is satisfactory, use `writeBlockValue` to apply the tuned values to the Simulink model and simulate the tuned controller in Simulink.
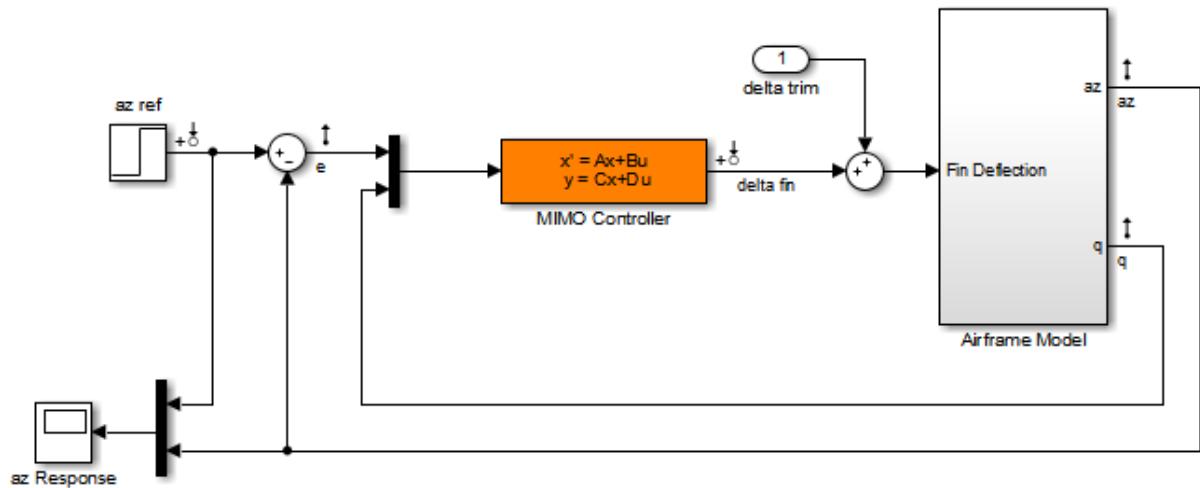
```
writeBlockValue(ST)
```

MIMO Design with SYSTUNE

Cascaded loops are commonly used for autopilots. Yet one may wonder how a single MIMO controller that uses both `az` and `q` to generate the actuator command `delta fin` would compare with the two-loop architecture. Trying new control architectures is easy with `systune` or `looptune`. For variety, we now use `systune` to tune the following MIMO architecture.

```
open_system('rct_airframe2')
```

Two-loop autopilot for controlling the vertical acceleration of an airframe

As before, compute the trim condition for $\alpha = 0$ and $V = 984m/s$.

```matlab
opspec = operspec('rct_airframe2');

% Specify trim condition
% Xe,Ze: known, not steady
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
% u,w: known, w steady
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
% theta: known, not steady
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
% q: unknown, steady
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
% controller states unknown, not steady
opspec.States(5).SteadyState = [0;0];

op = findop('rct_airframe2',opspec);
```

```
 Operating Point Search Report:
 -------------------------------

 Operating Report for the Model rct_airframe2.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
(1.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:             0      dx:             984
      x:      -3.05e+03      dx:              0
(2.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:             0      dx:       -0.00972
(3.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:           984      dx:            22.7
      x:             0      dx:       2.46e-11 (0)
(4.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Mot:
      x:      -0.00972      dx:      -1.15e-16 (0)
(5.) rct_airframe2/MIMO Controller
      x:      0.000654      dx:          -0.009
      x:      1.55e-19      dx:          0.0303

Inputs:
----------
(1.) rct_airframe2/delta trim
      u:      0.000436    [-Inf Inf]

Outputs: None
----------
```

As with `looptune`, use the `slTuner` interface to configure the Simulink model for tuning. Note that the signals of interest are already marked as Linear Analysis points in the Simulink model.

```
ST0 = slTuner('rct_airframe2','MIMO Controller',op);
```

Try a second-order MIMO controller with zero feedthrough from `e` to `delta fin`. To do this, create the desired controller parameterization and associate it with the "MIMO Controller" block using `setBlockParam`:

```
C0 = tunableSS('C',2,1,2);                 % Second-order controller
C0.d.Value(1) = 0;  C0.d.Free(1) = false;  % Fix D(1) to zero
```

```
setBlockParam(STO,'MIMO Controller',C0)
```

Next create the tuning requirements. Here we use the following four requirements:

1 **Tracking**: `az` should respond in about 1 second to the `azref` command
2 **Bandwidth and roll-off**: The loop gain at `delta fin` should roll off after 25 rad/s with a -20 dB/decade slope
3 **Stability margins**: The margins at `delta fin` should exceed 7 dB and 45 degrees
4 **Disturbance rejection**: The attenuation factor for input disturbances should be 40 dB at 1 rad/s increasing to 100 dB at 0.001 rad/s.

```
% Tracking
Req1 = TuningGoal.Tracking('az ref','az',1);

% Bandwidth and roll-off
Req2 = TuningGoal.MaxLoopGain('delta fin',tf(25,[1 0]));

% Margins
Req3 = TuningGoal.Margins('delta fin',7,45);

% Disturbance rejection
% Use an FRD model to sketch the desired attenuation profile with a few points
Freqs = [0 0.001 1];
MinAtt = [100 100 40];  % in dB
Req4 = TuningGoal.Rejection('delta fin',frd(db2mag(MinAtt),Freqs));
Req4.Focus = [0 1];
```

You can now use `systune` to tune the controller parameters subject to these requirements.
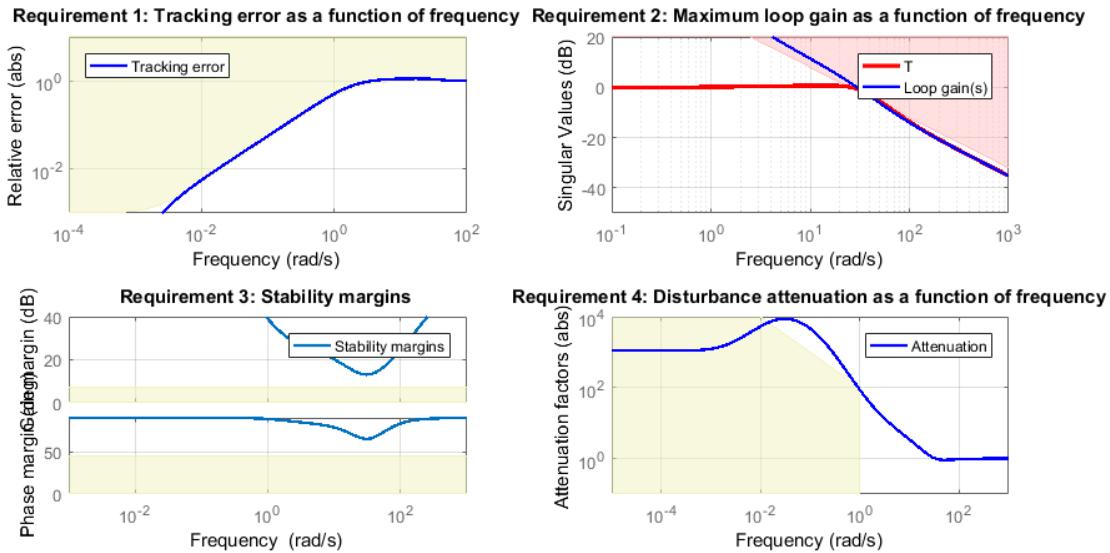
```
AllReqs = [Req1,Req2,Req3 Req4];
Opt = systuneOptions('RandomStart',3);

rng(0)
[ST,fSoft] = systune(STO,AllReqs,Opt);

Final: Soft = 1.14, Hard = -Inf, Iterations = 111
Final: Soft = 1.14, Hard = -Inf, Iterations = 81
Final: Soft = 1.14, Hard = -Inf, Iterations = 102
Final: Soft = 1.15, Hard = -Inf, Iterations = 116
```

The best design has an overall objective value close to 1, indicating that all four requirements are nearly met. Use `viewSpec` to inspect each requirement for the best design.
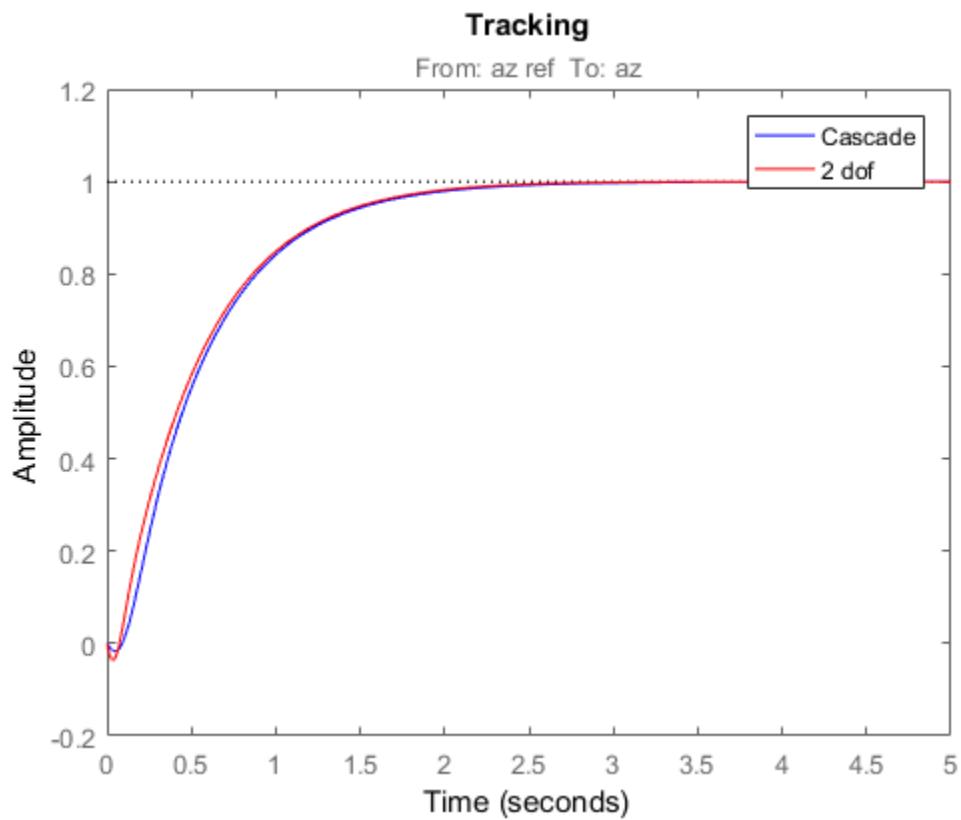
```
figure('Position',[100,100,987,474])
viewSpec(AllReqs,ST)
```
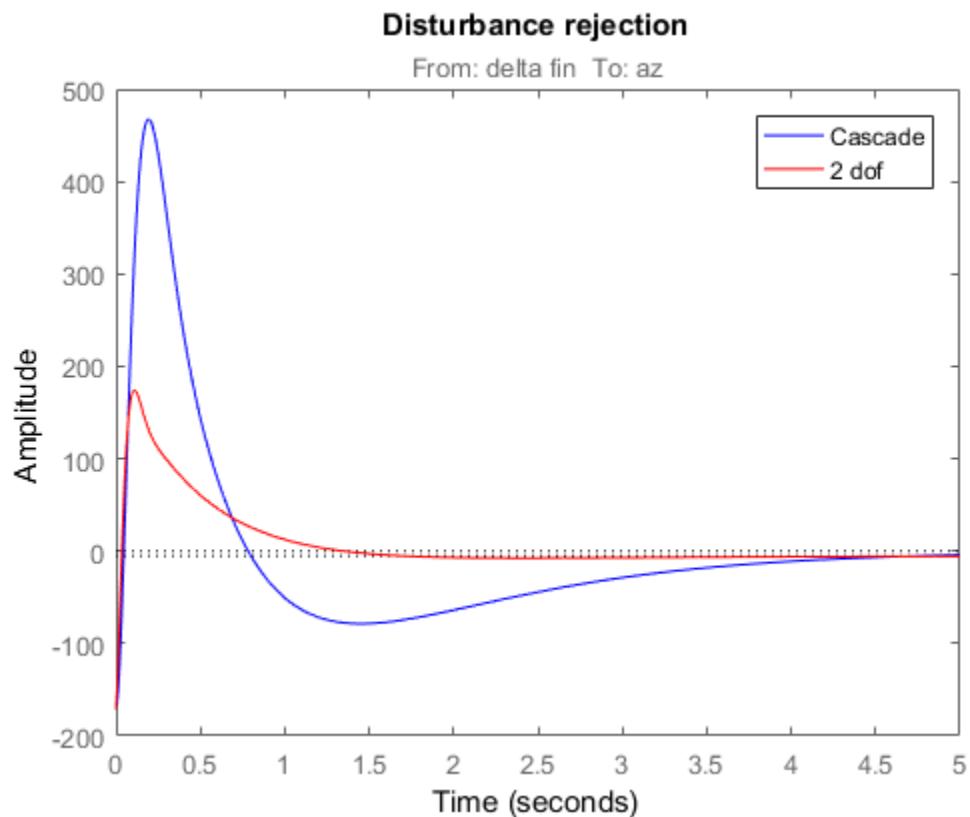

Compute the closed-loop responses and compare with the two-loop design.

```
T = getIOTransfer(ST,{'az ref','delta fin'},'az');

figure
step(Tr1,'b',T(1),'r',5)
title('Tracking'), legend('Cascade','2 dof')
```

```
step(Td1,'b',T(2),'r',5)
title('Disturbance rejection'), legend('Cascade','2 dof')
```

Disturbance rejection

The tracking performance is similar but the second design has better disturbance rejection properties.

Multi-Loop Control of a Helicopter

This example shows how to use `slTuner` and `systune` to tune a multi-loop controller for a rotorcraft.

Helicopter Model

This example uses an 8-state helicopter model at the hovering trim condition. The state vector `x = [u,w,q,theta,v,p,phi,r]` consists of
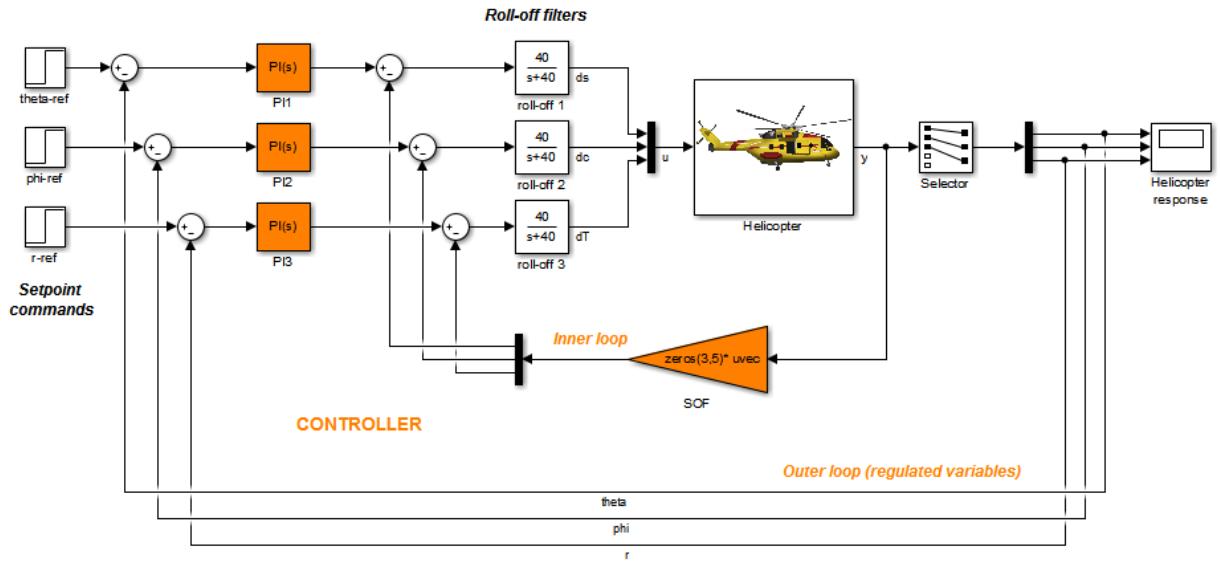
- Longitudinal velocity `u` (m/s)
- Lateral velocity `v` (m/s)
- Normal velocity `w` (m/s)
- Pitch angle `theta` (deg)
- Roll angle `phi` (deg)
- Roll rate `p` (deg/s)
- Pitch rate `q` (deg/s)
- Yaw rate `r` (deg/s).

The controller generates commands `ds,dc,dT` in degrees for the longitudinal cyclic, lateral cyclic, and tail rotor collective using measurements of `theta`, `phi`, `p`, `q`, and `r`.

Control Architecture

The following Simulink model depicts the control architecture:

```
open_system('rct_helico')
```

The control system consists of two feedback loops. The inner loop (static output feedback) provides stability augmentation and decoupling. The outer loop (PI controllers) provides the desired setpoint tracking performance. The main control objectives are as follows:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, rise times of about 2 seconds, minimal overshoot, and minimal cross-coupling
- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise
- Provide strong multivariable gain and phase margins (robustness to simultaneous gain/phase variations at the plant inputs and outputs, see `loopmargin` for details).
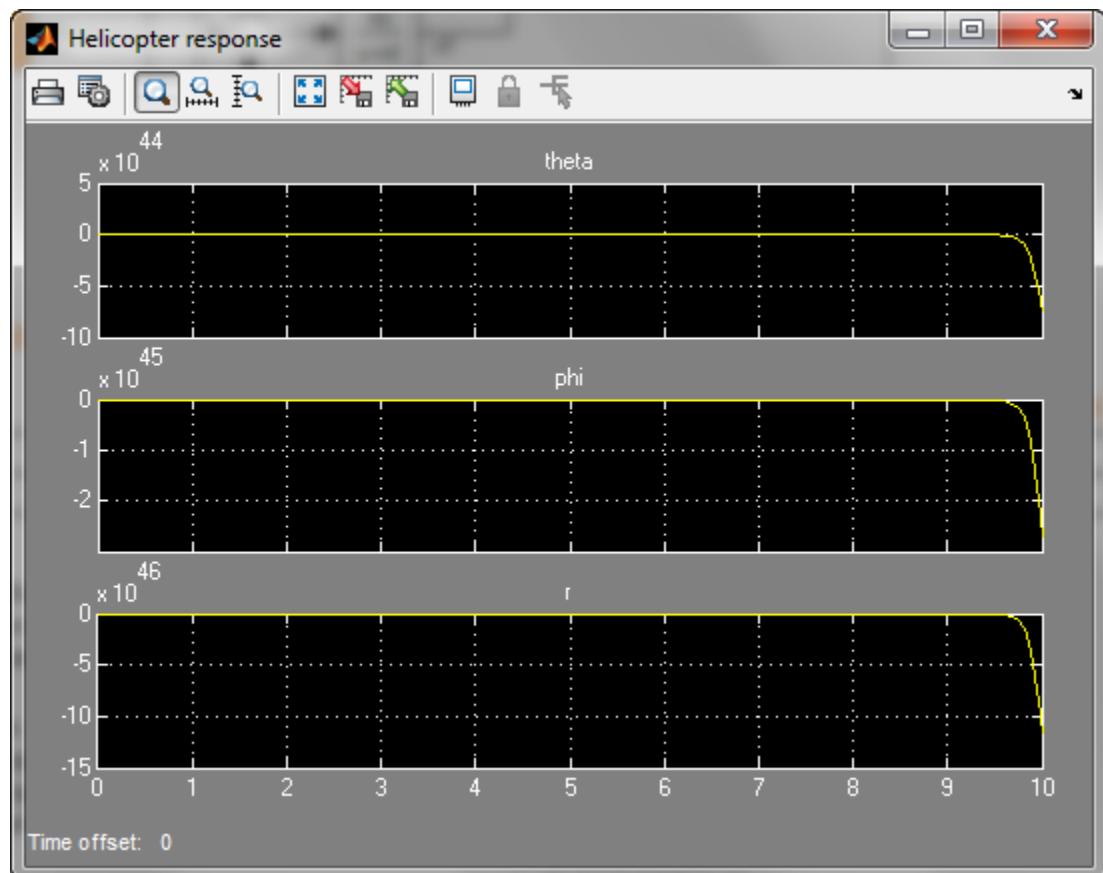
We use lowpass filters with cutoff at 40 rad/s to partially enforce the second objective.

Controller Tuning

You can jointly tune the inner and outer loops with the `systune` command. This command only requires models of the plant and controller along with the desired bandwidth (which is function of the desired response time). When the control system is modeled in Simulink, you can use the `slTuner` interface to quickly set up the tuning task. Create an instance of this interface with the list of blocks to be tuned.

```
ST0 = slTuner('rct_helico',{'PI1','PI2','PI3','SOF'});
```

Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model ($1 + 1/s$ for the PI controllers and zero for the static output-feedback gain). Simulating the model shows that the control system is unstable for these initial values:

Mark the I/O signals of interest for setpoint tracking, and identify the plant inputs and outputs (control and measurement signals) where the stability margin are measured.

```
addPoint(STO,{'theta-ref','phi-ref','r-ref'})    % setpoint commands
addPoint(STO,{'theta','phi','r'})                % corresponding outputs
addPoint(STO,{'u','y'});
```

Finally, capture the design requirements using `TuningGoal` objects. We use the following requirements for this example:

- **Tracking requirement**: The response of `theta`, `phi`, `r` to step commands `theta_ref`, `phi_ref`, `r_ref` must resemble a decoupled first-order response with a one-second time constant

- **Stability margins**: The multivariable gain and phase margins at the plant inputs `u` and plant outputs `y` must be at least 5 dB and 40 degrees

- **Fast dynamics**: The magnitude of the closed-loop poles must not exceed 25 to prevent fast dynamics and jerky transients

```
% Less than 20% mismatch with reference model 1/(s+1)
TrackReq = TuningGoal.StepTracking({'theta-ref','phi-ref','r-ref'},{'theta','phi','r'},
TrackReq.RelGap = 0.2;

% Gain and phase margins at plant inputs and outputs
MarginReq1 = TuningGoal.Margins('u',5,40);
MarginReq2 = TuningGoal.Margins('y',5,40);

% Limit on fast dynamics
MaxFrequency = 25;
PoleReq = TuningGoal.Poles(0,0,MaxFrequency);
```
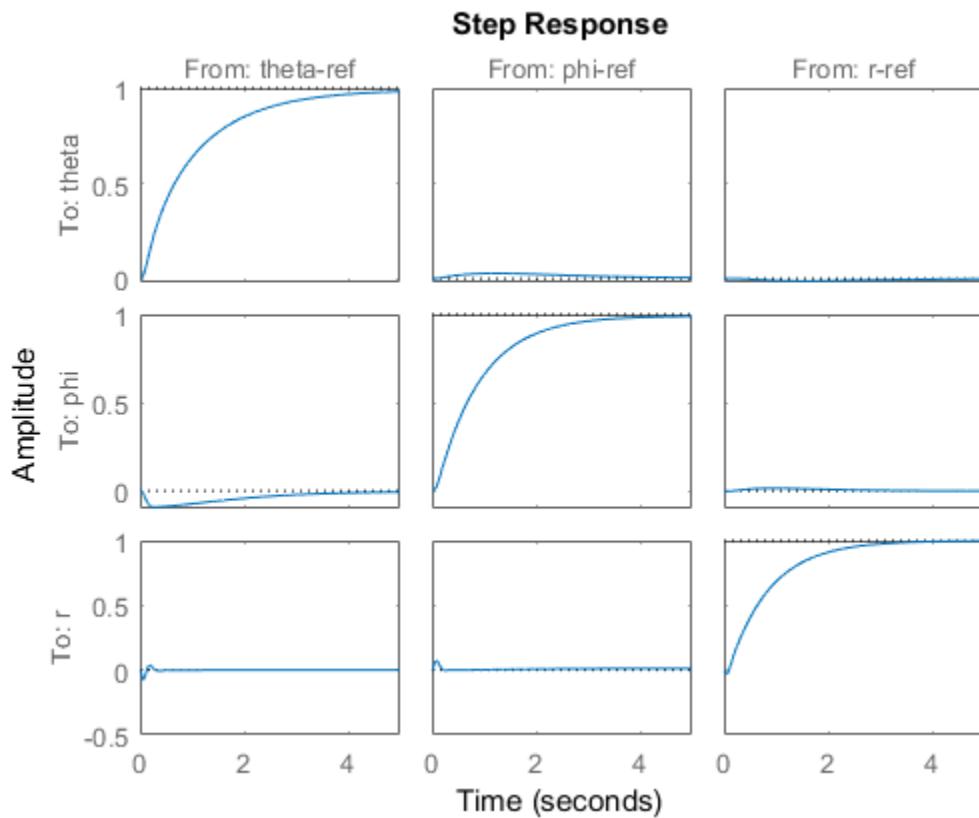
You can now use `systune` to jointly tune all controller parameters. This returns the tuned version `ST1` of the control system `ST0`.

```
AllReqs = [TrackReq,MarginReq1,MarginReq2,PoleReq];
ST1 = systune(ST0,AllReqs);

Final: Soft = 1.13, Hard = -Inf, Iterations = 72
```
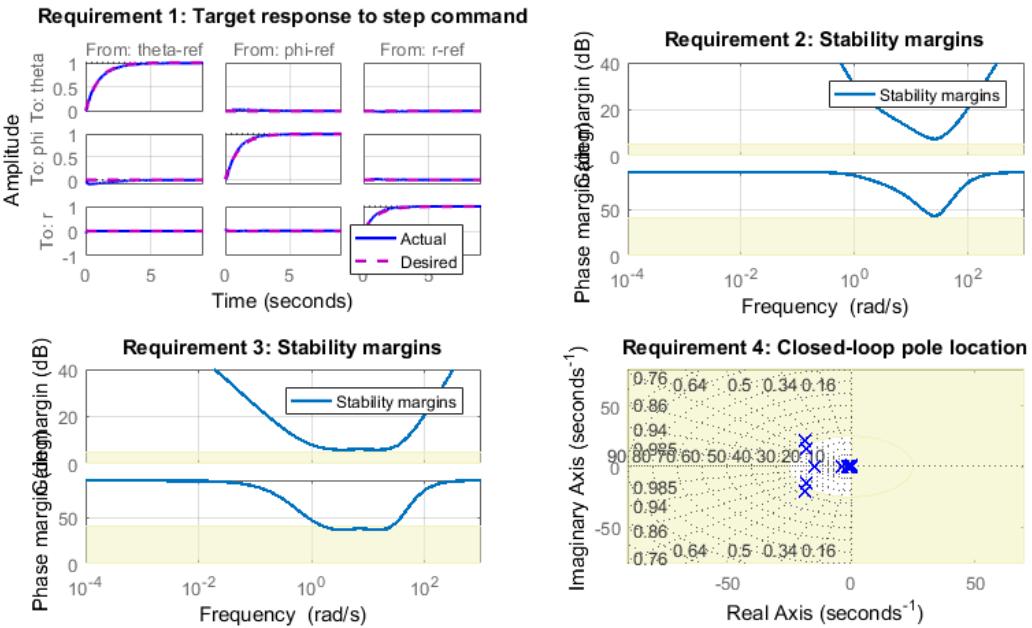
The final value is close to 1 so the requirements are nearly met. Plot the tuned responses to step commands in theta, phi, r:

```
T1 = getIOTransfer(ST1,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
step(T1,5)
```

Step Response

The rise time is about two seconds with no overshoot and little cross-coupling. You can use `viewSpec` for a more thorough validation of each requirement, including a visual assessment of the multivariable stability margins (see `loopmargin` for details):

```
figure('Position',[100,100,900,474])
viewSpec(AllReqs,ST1)
```

Requirement 1: Target response to step command

Requirement 2: Stability margins

Requirement 3: Stability margins

Requirement 4: Closed-loop pole location

Inspect the tuned values of the PI controllers and static output-feedback gain.

```
showTunable(ST1)
```

```
Block 1: rct_helico/PI1 =

            1
  Kp + Ki * ---
            s

  with Kp = 0.732, Ki = 1.6

Name: PI1
Continuous-time PI controller in parallel form.

-----------------------------------

Block 2: rct_helico/PI2 =

            1
```

```
  Kp + Ki * ---
             s

  with Kp = -0.0731, Ki = -1.54

Name: PI2
Continuous-time PI controller in parallel form.

----------------------------------

Block 3: rct_helico/PI3 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.141, Ki = -2.59

Name: PI3
Continuous-time PI controller in parallel form.

----------------------------------

Block 4: rct_helico/SOF =

  D =
               u1         u2         u3         u4         u5
    y1      1.659    -0.1211     0.0938     0.6091  -0.0002313
    y2     -0.291     -1.427    0.02887    -0.07915    -0.1099
    y3   -0.003016    0.01147     -2.261    -0.01255     0.0329

Name: SOF
Static gain.
```

Benefit of the Inner Loop

You may wonder whether the static output feedback is necessary and whether PID controllers aren't enough to control the helicopter. This question is easily answered by re-tuning the controller with the inner loop open. First break the inner loop by adding a loop opening after the SOF block:

```
addOpening(ST0,'SOF')
```

Then remove the SOF block from the tunable block list and re-parameterize the PI blocks as full-blown PIDs with the correct loop signs (as inferred from the first design).

```matlab
PID = pid(0,0.001,0.001,.01);  % initial guess for PID controllers

removeBlock(ST0,'SOF');
setBlockParam(ST0,'PI1',tunablePID('C1',PID));
setBlockParam(ST0,'PI2',tunablePID('C2',-PID));
setBlockParam(ST0,'PI3',tunablePID('C3',-PID));
```
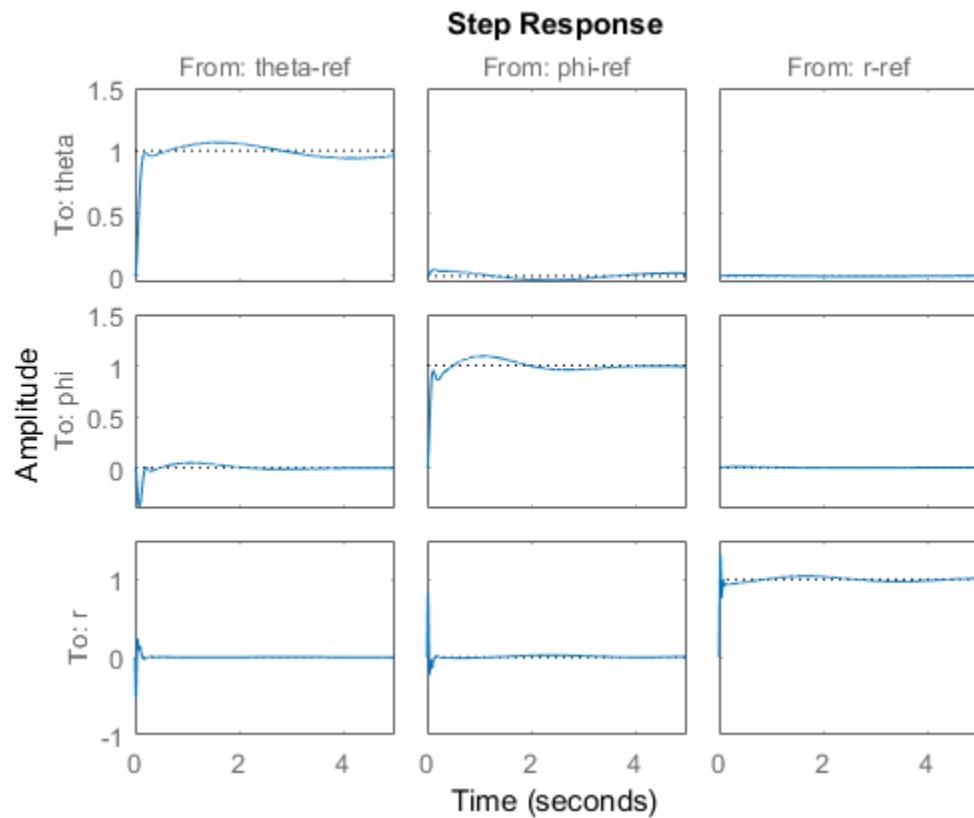
Re-tune the three PID controllers and plot the closed-loop step responses.

```matlab
ST2 = systune(ST0,AllReqs);

Final: Soft = 4.96, Hard = -Inf, Iterations = 69

T2 = getIOTransfer(ST2,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
figure, step(T2,5)
```

Step Response

The final value is no longer close to 1 and the step responses confirm the poorer performance with regard to rise time, overshoot, and decoupling. This suggests that the inner loop has an important stabilizing effect that should be preserved.

Fixed-Structure Autopilot for a Passenger Jet

This example shows how to use `slTuner` and `systune` to tune the standard configuration of a longitudinal autopilot. We thank Professor D. Alazard from Institut Superieur de l'Aeronautique et de l'Espace for providing the aircraft model and Professor Pierre Apkarian from ONERA for developing the example.

Aircraft Model and Autopilot Configuration

The longitudinal autopilot for a supersonic passenger jet flying at Mach 0.7 and 5000 ft is depicted in Figure 1. The autopilot main purpose is to follow vertical acceleration commands N_{zc} issued by the pilot. The feedback structure consists of an inner loop controlling the pitch rate q and an outer loop controlling the vertical acceleration N_z. The autopilot also includes a feedforward component and a reference model $G_{ref}(s)$ that specifies the desired response to a step command N_{zc}. Finally, the second-order roll-off filter

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

is used to attenuate noise and limit the control bandwidth as a safeguard against unmodeled dynamics. The tunable components are highlighted in orange.
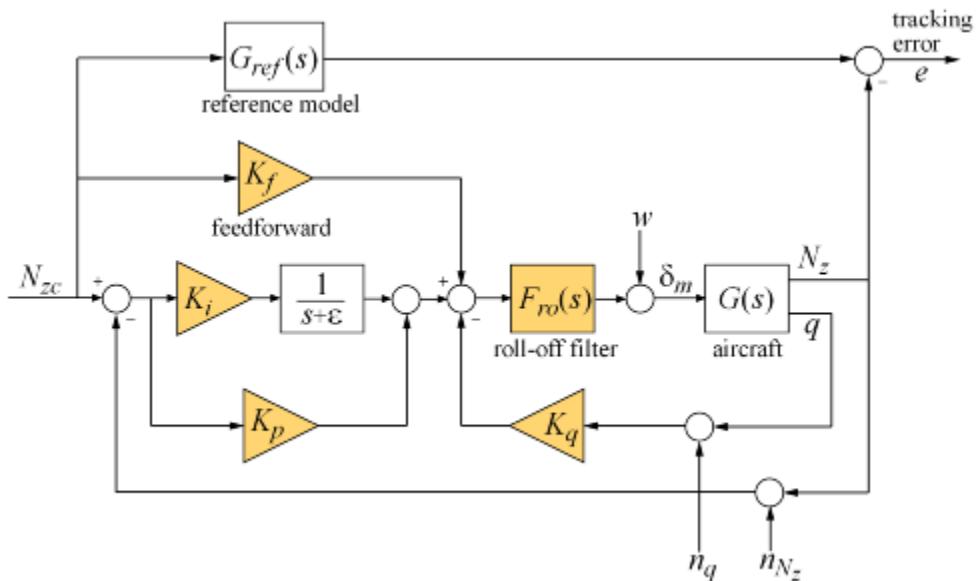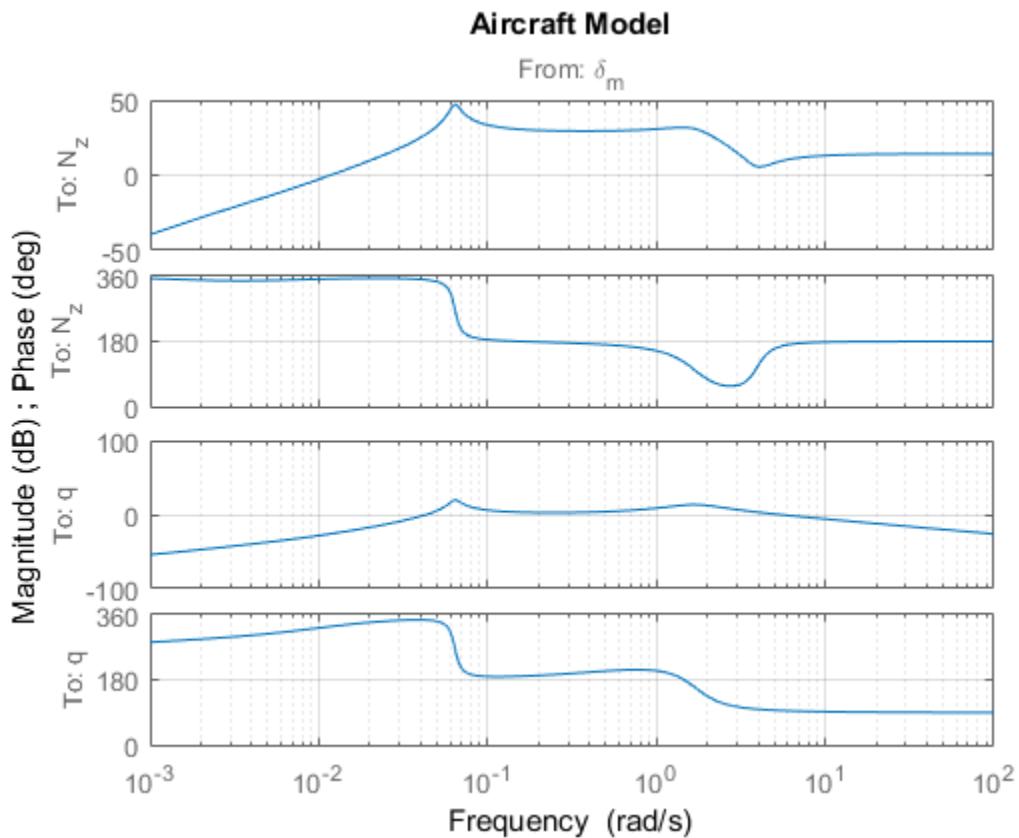
Figure 1: Longitudinal Autopilot Configuration.

The aircraft model $G(s)$ is a 5-state model, the state variables being the aerodynamic speed V (m/s), the climb angle γ (rad), the angle of attack α (rad), the pitch rate q (rad/s), and the altitude H (m). The elevator deflection δ_m (rad) is used to control the vertical load factor N_z. The open-loop dynamics include the α oscillation with frequency and damping ratio ω_n = 1.7 (rad/s) and ζ = 0.33, the phugoid mode ω_n = 0.64 (rad/s) and ζ = 0.06, and the slow altitude mode λ = -0.0026.

```
load ConcordeData G
bode(G,{1e-3,1e2}), grid
title('Aircraft Model')
```
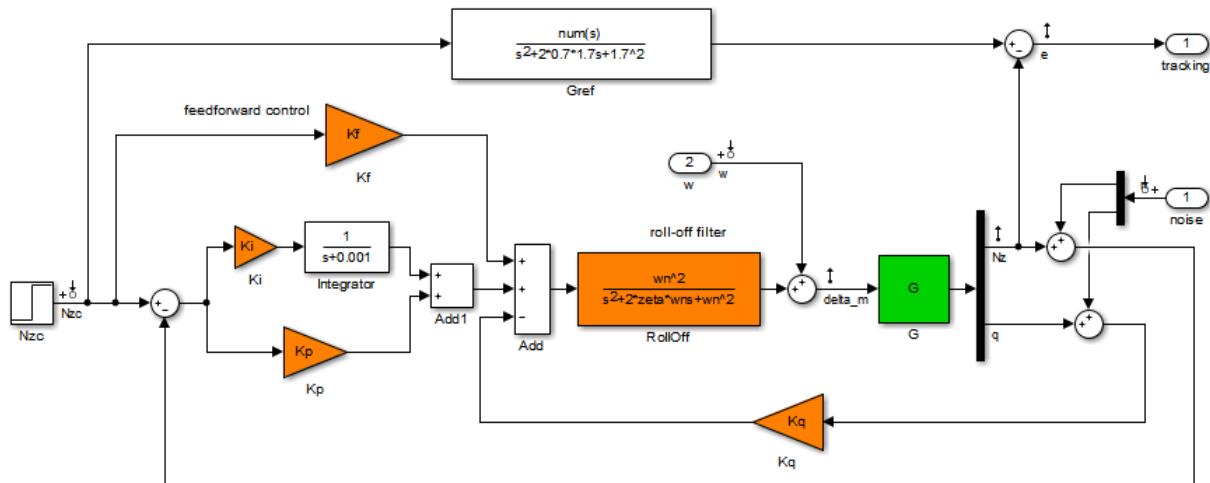
Note the zero at the origin in $G(s)$. Because of this zero, we cannot achieve zero steady-state error and must instead focus on the transient response to acceleration commands. Note that acceleration commands are transient in nature so steady-state behavior is not a concern. This zero at the origin also precludes pure integral action so we use a pseudo-integrator $1/(s + \epsilon)$ with $\epsilon = 0.001$.

Tuning Setup

When the control system is modeled in Simulink, you can use the `slTuner` interface to quickly set up the tuning task. Open the Simulink model of the autopilot.

```
open_system('rct_concorde')
```

Longitudinal autopilot for a passenger jet.

You can tune this autopilot with the SYSTUNE command, see concorde_demo for details

Copyright 2004-2012 The MathWorks, Inc.

Configure the `slTuner` interface by listing the tuned blocks in the Simulink model (highlighted in orange). This automatically picks all Linear Analysis points in the model as points of interest for analysis and tuning.

```
STO = slTuner('rct_concorde',{'Ki','Kp','Kq','Kf','RollOff'});
```

This also parameterizes each tuned block and initializes the block parameters based on their values in the Simulink model. Note that the four gains `Ki,Kp,Kq,Kf` are initialized to zero in this example. By default the roll-off filter $F_{ro}(s)$ is parameterized as a generic second-order transfer function. To parameterize it as

$$F_{ro}(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2},$$

create real parameters ζ, ω_n, build the transfer function shown above, and associate it with the `RollOff` block.

```
wn = realp('wn', 3);                    % natural frequency
```

```matlab
zeta = realp('zeta',0.8);          % damping
Fro = tf(wn^2,[1 2*zeta*wn wn^2]); % parametric transfer function

setBlockParam(ST0,'RollOff',Fro)   % use Fro to parameterize "RollOff" block
```

Design Requirements

The autopilot must be tuned to satisfy three main design requirements:

1. **Setpoint tracking**: The response N_z to the command N_{zc} should closely match the response of the reference model:

$$G_{ref}(s) = \frac{1.7^2}{s^2 + 2 \times 0.7 \times 1.7s + 1.7^2}.$$

This reference model specifies a well-damped response with a 2 second settling time.

2. **High-frequency roll-off**: The closed-loop response from the noise signals to δ_m should roll off past 8 rad/s with a slope of at least -40 dB/decade.

3. **Stability margins**: The stability margins at the plant input δ_m should be at least 7 dB and 45 degrees.

For setpoint tracking, we require that the gain of the closed-loop transfer from the command N_{zc} to the tracking error e be small in the frequency band [0.05,5] rad/s (recall that we cannot drive the steady-state error to zero because of the plant zero at s=0). Using a few frequency points, sketch the maximum tracking error as a function of frequency and use it to limit the gain from N_{zc} to e.
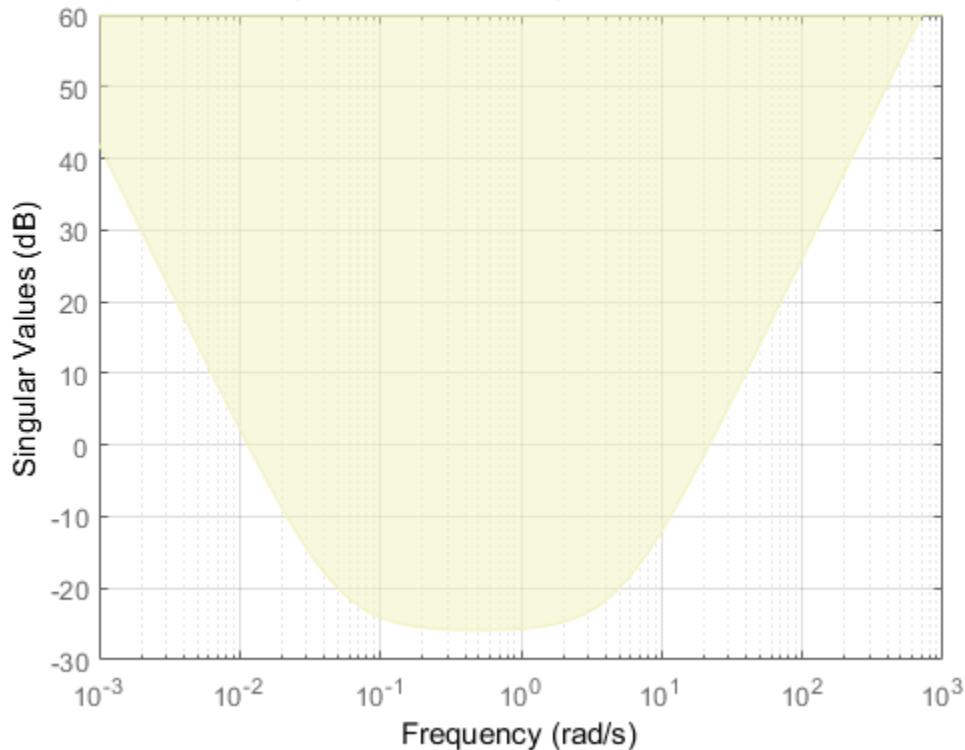
```matlab
Freqs = [0.005 0.05 5 50];
Gains = [5 0.05 0.05 5];
Req1 = TuningGoal.Gain('Nzc','e',frd(Gains,Freqs));
Req1.Name = 'Maximum tracking error';
```
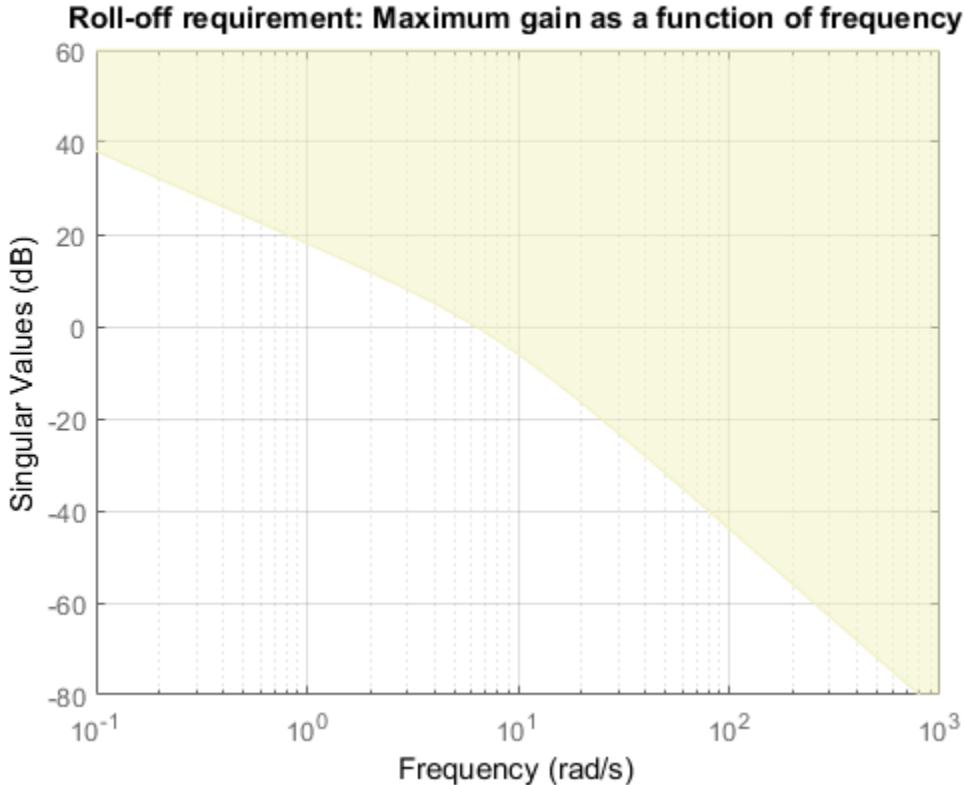
The `TuningGoal.Gain` constructor automatically turns the maximum error sketch into a smooth weighting function. Use `viewSpec` to graphically verify the desired error profile.

```matlab
viewSpec(Req1)
```

Maximum tracking error: Maximum gain as a function of frequency

Repeat the same process to limit the high-frequency gain from the noise inputs to δ_m and enforce a -40 dB/decade slope in the frequency band from 8 to 800 rad/s

```
Freqs = [0.8 8 800];
Gains = [10 1 1e-4];
Req2 = TuningGoal.Gain('n','delta_m',frd(Gains,Freqs));
Req2.Name = 'Roll-off requirement';

viewSpec(Req2)
```

Roll-off requirement: Maximum gain as a function of frequency

Finally, register the plant input δ_m as a site for open-loop analysis and use `TuningGoal.Margins` to capture the stability margin requirement.

```
addPoint(STO,'delta_m')

Req3 = TuningGoal.Margins('delta_m',7,45);
```

Autopilot Tuning

We are now ready to tune the autopilot parameters with `systune`. This command takes the untuned configuration `STO` and the three design requirements and returns the tuned version `ST` of `STO`. All requirements are satisfied when the final value is less than one.

```
[ST,fSoft] = systune(STO,[Req1 Req2 Req3]);
```

```
Final: Soft = 0.965, Hard = -Inf, Iterations = 80
```

Use showTunable to see the tuned block values.

```
showTunable(ST)
```

```
Block 1: rct_concorde/Ki =

  D =
            u1
   y1  -0.02972

Name: Ki
Static gain.

-------------------------------

Block 2: rct_concorde/Kp =

  D =
            u1
   y1  -0.009849

Name: Kp
Static gain.

-------------------------------

Block 3: rct_concorde/Kq =

  D =
            u1
   y1  -0.2854

Name: Kq
Static gain.

-------------------------------

Block 4: rct_concorde/Kf =

  D =
            u1
   y1  -0.0224
```

```
Name: Kf
Static gain.

-----------------------------------

wn = 4.81
-----------------------------------
zeta = 0.51
```

To get the tuned value of $F_{ro}(s)$, use `getBlockValue` to evaluate `Fro` for the tuned parameter values in `ST`:

```
Fro = getBlockValue(ST,'RollOff');
tf(Fro)


ans =

          23.18
  --------------------
  s^2 + 4.909 s + 23.18

Continuous-time transfer function.
```
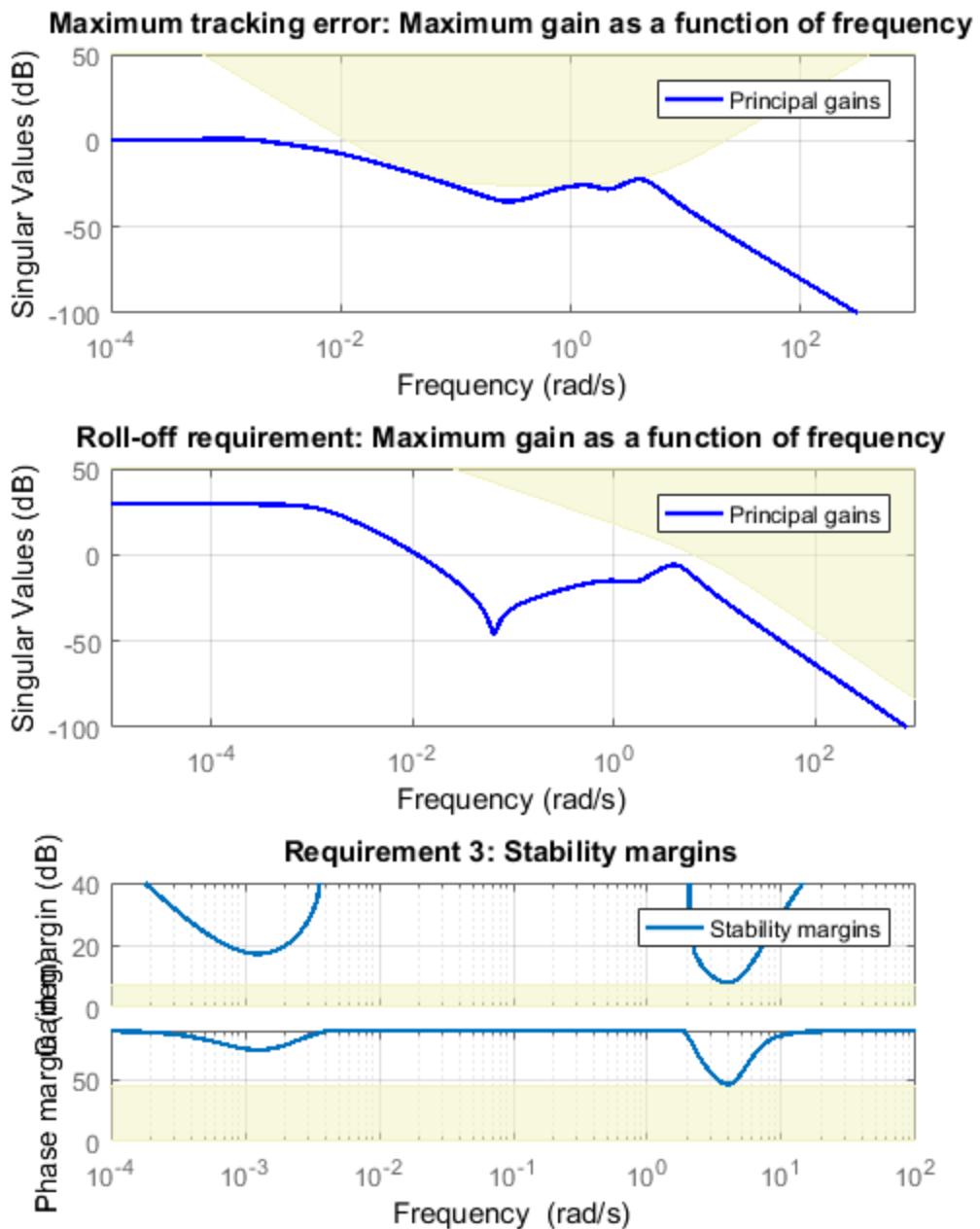
Finally, use `viewSpec` to graphically verify that all requirements are satisfied.
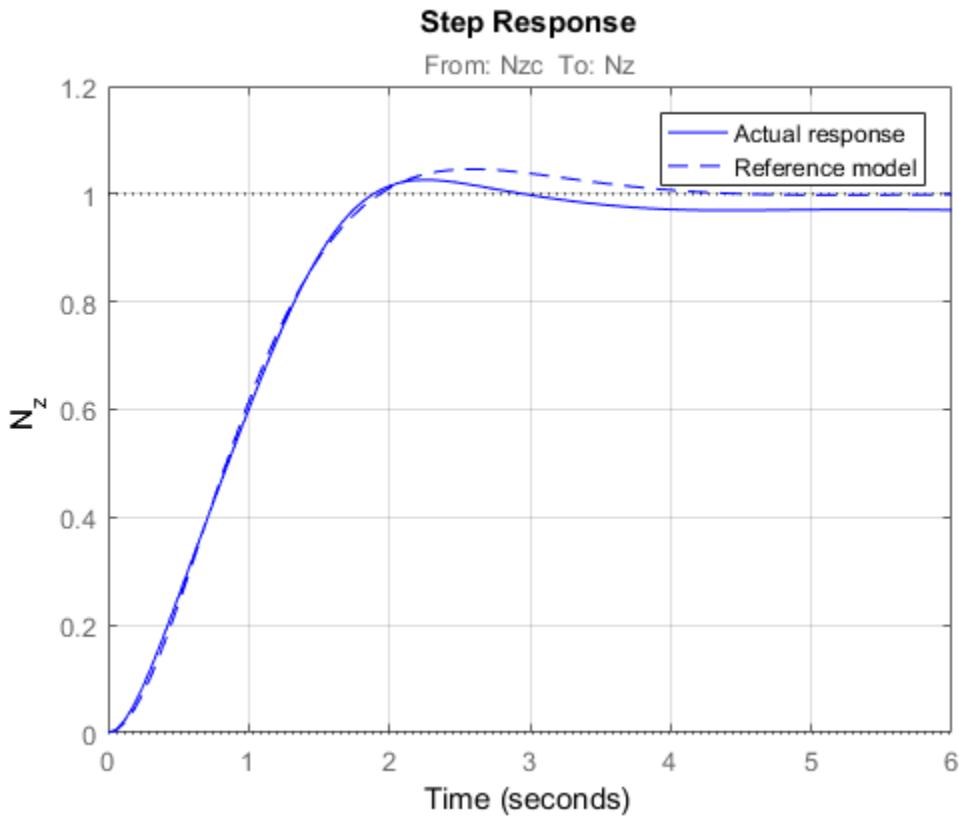
```
figure('Position',[100,100,550,710])
viewSpec([Req1 Req2 Req3],ST)
```

Maximum tracking error: Maximum gain as a function of frequency

Roll-off requirement: Maximum gain as a function of frequency

Requirement 3: Stability margins

Closed-Loop Simulations

We now verify that the tuned autopilot satisfies the design requirements. First compare the step response of N_z with the step response of the reference model $G_{ref}(s)$. Again use `getIOTransfer` to compute the tuned closed-loop transfer from `Nzc` to `Nz`:
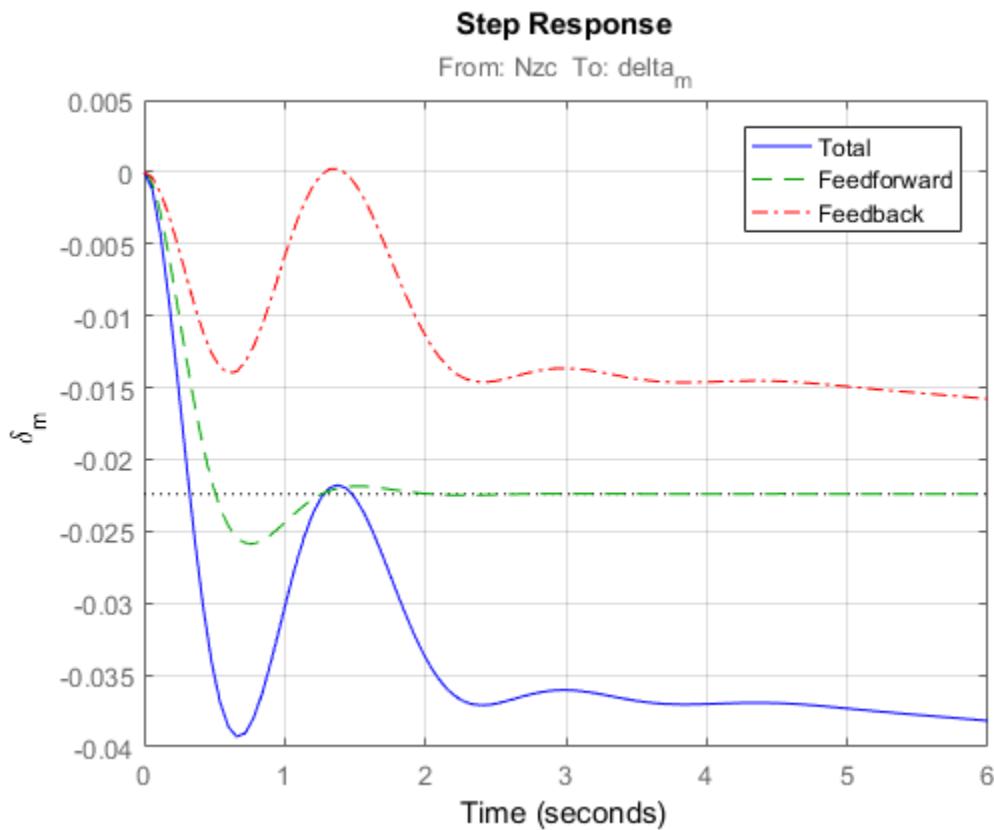
```
Gref = tf(1.7^2,[1 2*0.7*1.7 1.7^2]);      % reference model

T = getIOTransfer(ST,'Nzc','Nz');  % transfer Nzc -> Nz

figure, step(T,'b',Gref,'b--',6), grid,
ylabel('N_z'), legend('Actual response','Reference model')
```


Step Response

From: Nzc To: Nz

Also plot the deflection δ_m and the respective contributions of the feedforward and feedback paths:

```
T = getIOTransfer(ST,'Nzc','delta_m');  % transfer Nzc -> delta_m
Kf = getBlockValue(ST,'Kf');            % tuned value of Kf
Tff = Fro*Kf;           % feedforward contribution to delta_m

step(T,'b',Tff,'g--',T-Tff,'r-.',6), grid
ylabel('\delta_m'), legend('Total','Feedforward','Feedback')
```
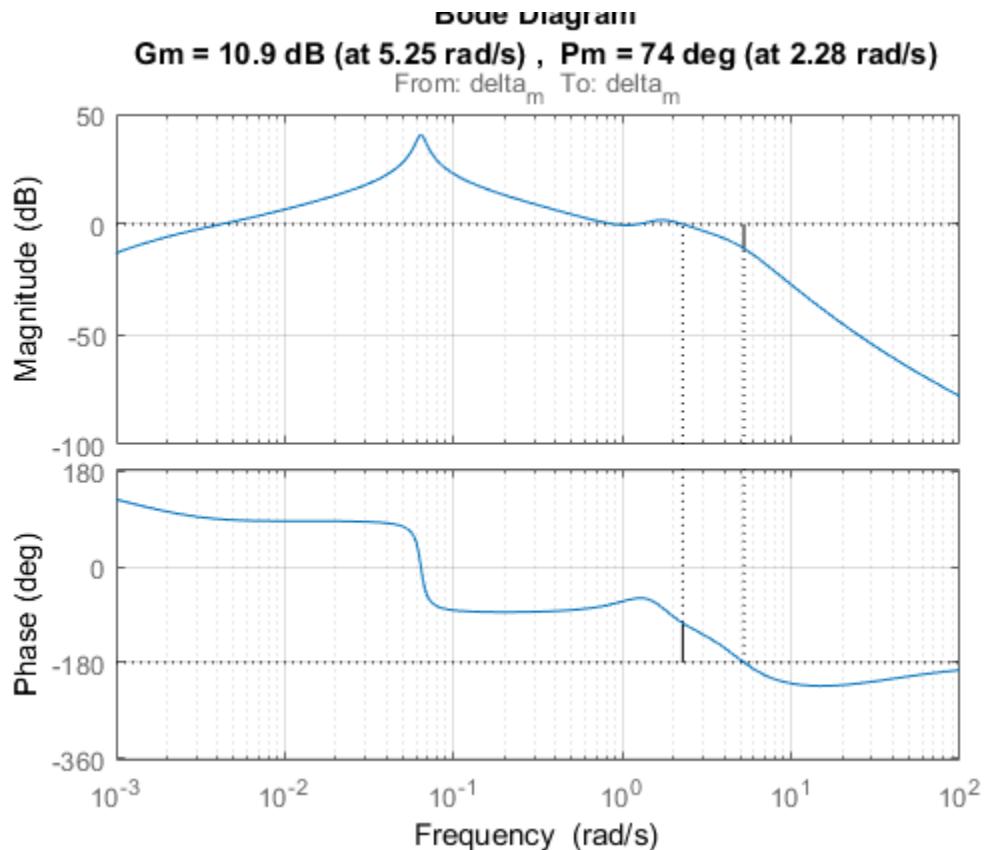

Finally, check the roll-off and stability margin requirements by computing the open-loop response at δ_m.

```
OL = getLoopTransfer(ST,'delta_m',-1); % negative-feedback loop transfer
```

```
margin(OL);
grid;
xlim([1e-3,1e2]);
```


The Bode plot confirms a roll-off of -40 dB/decade past 8 rad/s and indicates gain and phase margins in excess of 10 dB and 70 degrees.

Fault-Tolerant Control of a Passenger Jet

This example shows how to tune a fixed-structure controller for multiple operating modes of the plant.
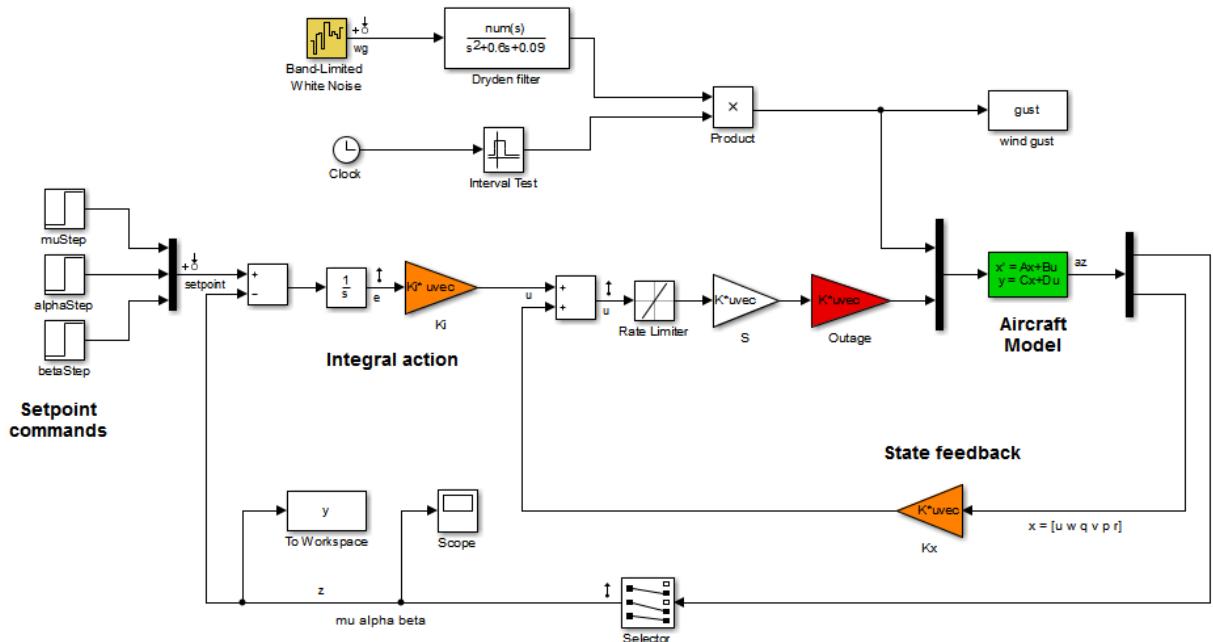
Background

This example deals with fault-tolerant flight control of passenger jet undergoing outages in the elevator and aileron actuators. The flight control system must maintain stability and meet performance and comfort requirements in both nominal operation and degraded conditions where some actuators are no longer effective due to control surface impairment. Wind gusts must be alleviated in all conditions. This application is sometimes called *reliable control* as aircraft safety must be maintained in extreme flight conditions.

Aircraft Model

The control system is modeled in Simulink.

```
open_system('faultTolerantAircraft')
```

The aircraft is modeled as a rigid 6th-order state-space system with the following state variables (units are mph for velocities and deg/s for angular rates):

- u: x-body axis velocity
- w: z-body axis velocity
- q: pitch rate
- v: y-body axis velocity
- p: roll rate
- r: yaw rate

The state vector is available for control as well as the flight-path bank angle rate mu (deg/s), the angle of attack alpha (deg), and the sideslip angle beta (deg). The control inputs are the deflections of the right elevator, left elevator, right aileron, left aileron, and rudder. All deflections are in degrees. Elevators are grouped symmetrically to generate the angle of attack. Ailerons are grouped anti-symmetrically to generate roll motion. This leads to 3 control actions as shown in the Simulink model.

The controller consists of state-feedback control in the inner loop and MIMO integral action in the outer loop. The gain matrices `Ki` and `Kx` are 3-by-3 and 3-by-6, respectively, so the controller has 27 tunable parameters.

Actuator Failures

We use a 9x5 matrix to encode the nominal mode and various actuator failure modes. Each row corresponds to one flight condition, a zero indicating outage of the corresponding deflection surface.

```
OutageCases = [...
   1 1 1 1 1; ... % nominal operational mode
   0 1 1 1 1; ... % right elevator outage
   1 0 1 1 1; ... % left elevator outage
   1 1 0 1 1; ... % right aileron outage
   1 1 1 0 1; ... % left aileron outage
   1 0 0 1 1; ... % left elevator and right aileron outage
   0 1 0 1 1; ... % right elevator and right aileron outage
   0 1 1 0 1; ... % right elevator and left aileron outage
   1 0 1 0 1; ... % left elevator and left aileron outage
   ];
```

Design Requirements

The controller should:

1. Provide good tracking performance in mu, alpha, and beta in nominal operating mode with adequate decoupling of the three axes

2. Maintain performance in the presence of wind gust of 10 mph

3. Limit stability and performance degradation in the face of actuator outage.

To express the first requirement, you can use an LQG-like cost function that penalizes the integrated tracking error `e` and the control effort `u`:

$$J = lim_{T \to \infty} E \left(\frac{1}{T} \int_0^T \|W_e e\|^2 + \|W_u u\|^2 dt \right).$$

The diagonal weights W_e and W_u are the main tuning knobs for trading responsiveness and control effort and emphasizing some channels over others. Use the `WeightedVariance` requirement to express this cost function, and relax the performance weight W_e by a factor 2 for the outage scenarios.

```
We = diag([10 20 15]);    Wu = eye(3);

% Nominal tracking requirement
SoftNom = TuningGoal.WeightedVariance('setpoint',{'e','u'}, blkdiag(We,Wu), []);
SoftNom.Models = 1;    % nominal model

% Tracking requirement for outage conditions
SoftOut = TuningGoal.WeightedVariance('setpoint',{'e','u'}, blkdiag(We/2,Wu), []);
SoftOut.Models = 2:9;  % outage scenarios
```

For wind gust alleviation, limit the variance of the error signal `e` due to the white noise `wg` driving the wind gust model. Again use a less stringent requirement for the outage scenarios.

```
% Nominal gust alleviation requirement
HardNom = TuningGoal.Variance('wg','e',0.02);
HardNom.Models = 1;

% Gust alleviation requirement for outage conditions
HardOut = TuningGoal.Variance('wg','e',0.1);
HardOut.Models = 2:9;
```

Controller Tuning for Nominal Flight

Set the wind gust speed to 10 mph and initialize the tunable state-feedback and integrators gains of the controller.

```
GustSpeed = 10;
Ki = eye(3);
Kx = zeros(3,6);
```

Use the `slTuner` interface to set up the tuning task. List the blocks to be tuned and specify the nine flight conditions by varying the `outage` variable in the Simulink model. Because you can only vary scalar parameters in `slTuner`, independently specify the values taken by each entry of the `outage` vector.
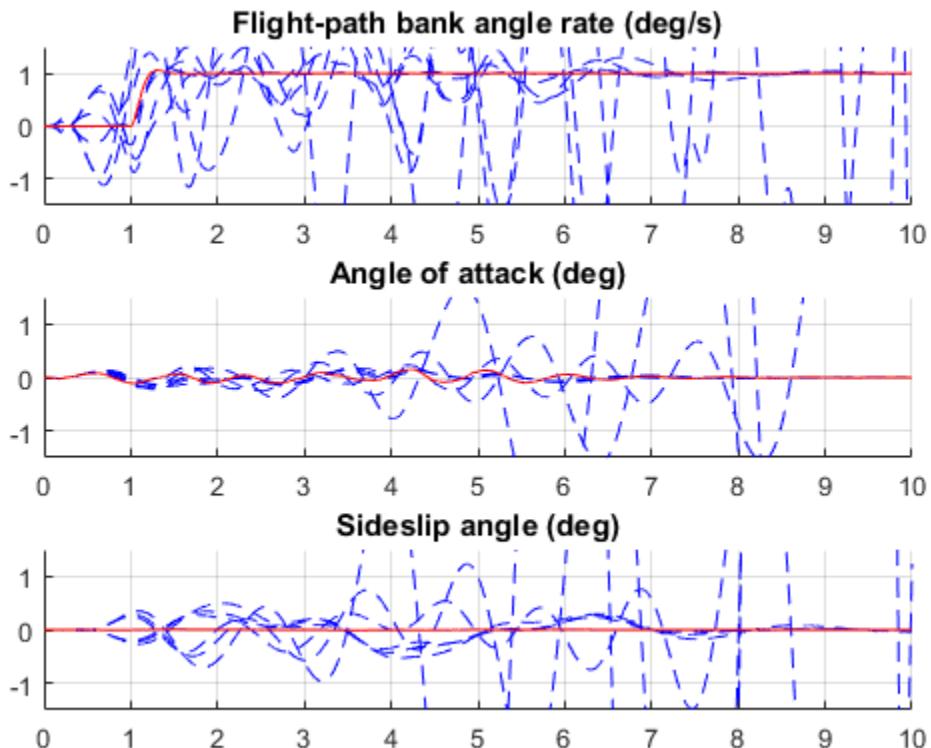
```
OutageData = struct(...
    'Name',{'outage(1)','outage(2)','outage(3)','outage(4)','outage(5)'},...
    'Value',mat2cell(OutageCases,9,[1 1 1 1 1]));
STO = slTuner('faultTolerantAircraft',{'Ki','Kx'},OutageData);
```

Use `systune` to tune the controller gains subject to the nominal requirements. Treat the wind gust alleviation as a hard constraint.

```
[ST,fSoft,gHard]  = systune(STO,SoftNom,HardNom);

Final: Soft = 22.6, Hard = 0.99939, Iterations = 286
```

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);
```

Flight-path bank angle rate (deg/s)

Angle of attack (deg)

Sideslip angle (deg)

```
% 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.
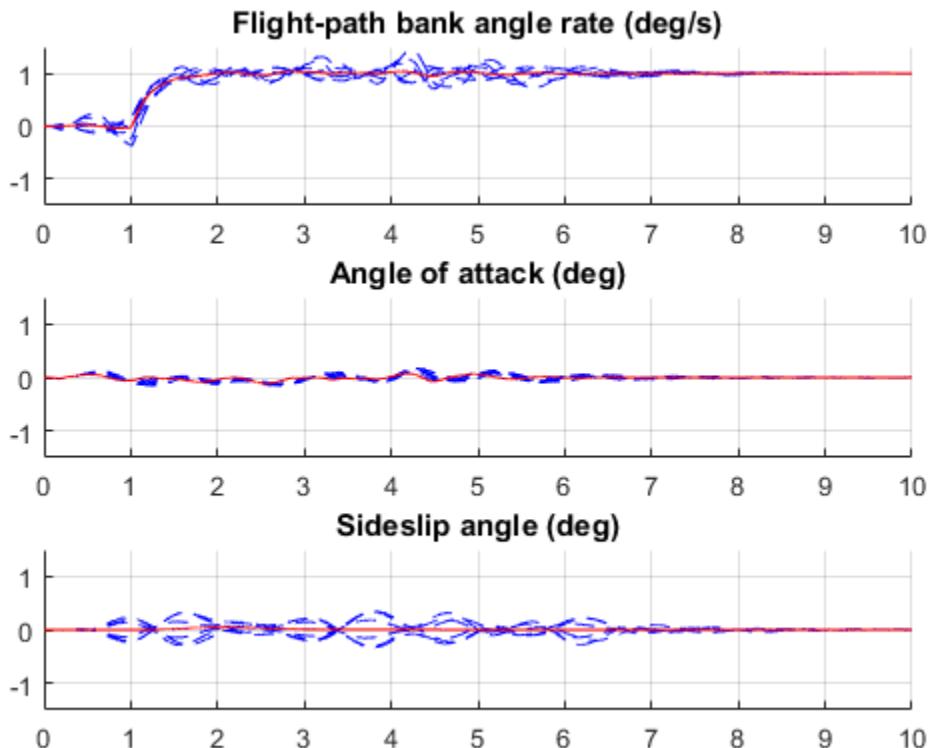
```
[ST,fSoft,gHard]  = systune(ST0,[SoftNom;SoftOut],[HardNom;HardOut]);

Final: Soft = 25.8, Hard = 0.99809, Iterations = 573
```
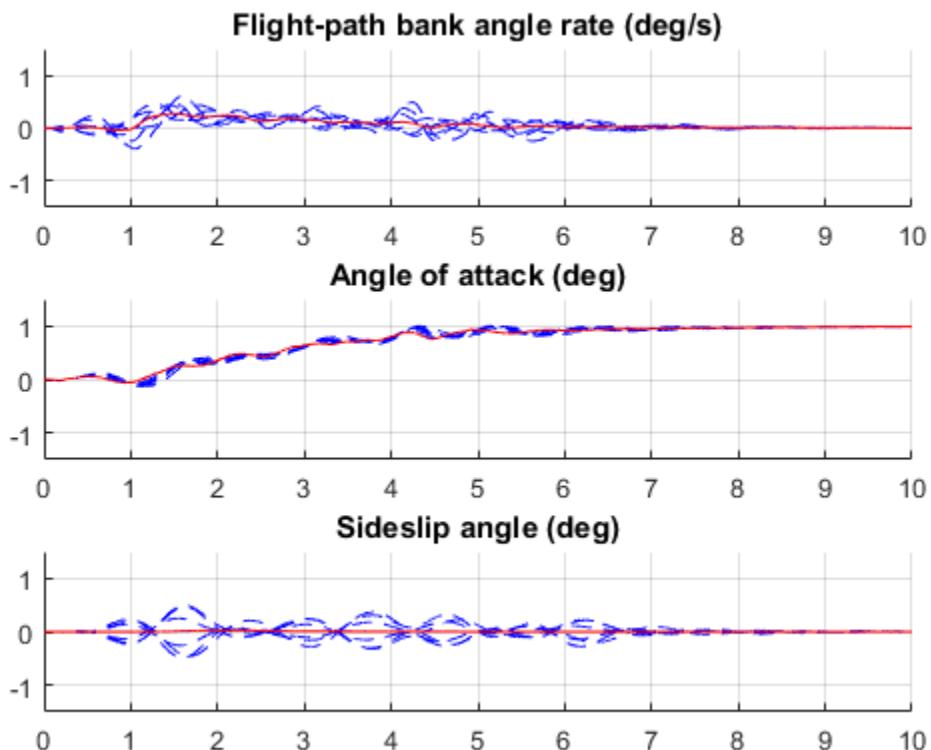
The optimal performance (square root of LQG cost J) is only slightly worse than for the nominal tuning (26 vs. 23). Retrieve the gain values and rerun the simulations (red curve is the nominal response).
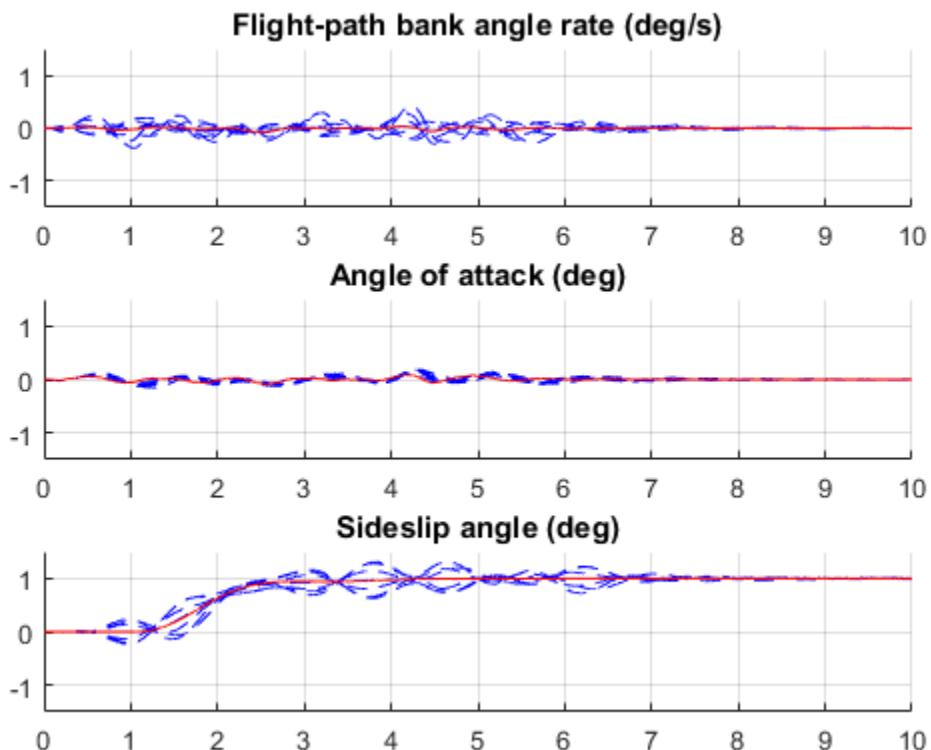
```
Ki = getBlockValue(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% 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.

Passive Control of Water Tank Level
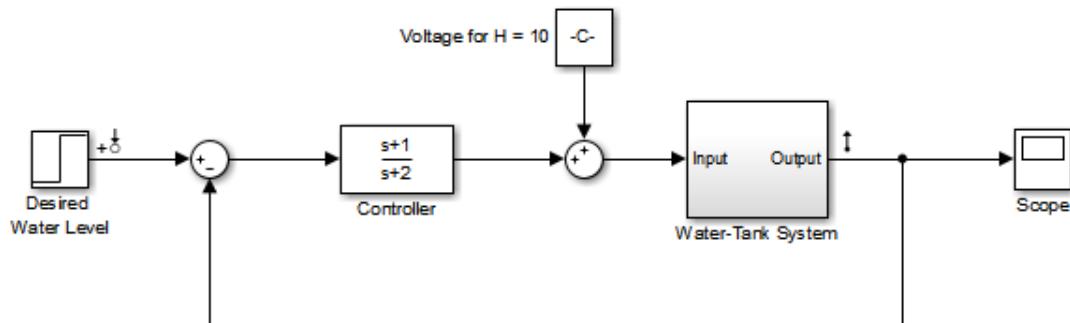
In this example, you learn how to use Control System Tuner app to design a controller for a nonlinear plant modeled in Simulink. You accomplish the following tasks:

- Configure the model and app for compensator tuning
- Tune a first-order compensator using passivity-based design
- Simulate the closed-loop nonlinear response.

Simulink Model of the Control System
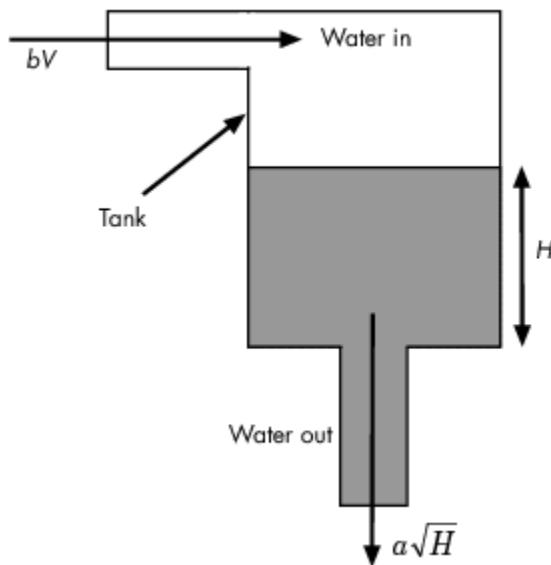
The watertank_comp_design model, shown in the figure below, models a feedback loop for regulating the water level in a water tank. The Controller block contains the first-order compensator to be tuned.

```
mdl = 'watertank_comp_design';
open_system(mdl);
```


Copyright 2004-2015 The MathWorks, Inc.

The Water Tank subsystem models the water-tank dynamics. Water enters the tank from the top at a rate proportional to the voltage, V, applied to the pump. The water leaves through an opening in the tank base at a rate that is proportional to the square root of the water height, H, in the tank. The presence of the square root in the water flow rate makes the plant nonlinear.

The nonlinear model for the water flow is

$$A\dot{x} = bu - a\sqrt{x}$$

$$y = x$$

where
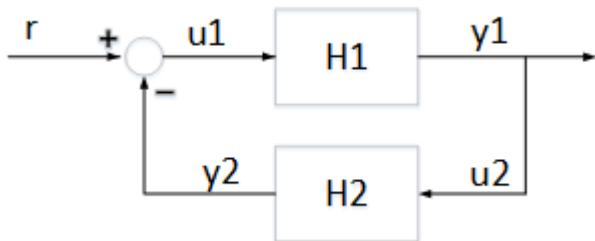
- $x = H$ denotes the height of water in the tank
- u denotes the voltage applied to the pump
- A denotes the cross-sectional area of the tank
- a and b are constants related to the flow rate into and out of the tank

This system is passive with storage function $V(x) = \frac{A}{2b}x^2$ since

$$\dot{V}(x) - uy = -\frac{a}{b}x\sqrt{x} \le 0$$

Passivity-Based Control

By the Passivity Theorem, the negative-feedback interconnection of two strictly passive systems H_1 and H_2 is always stable.
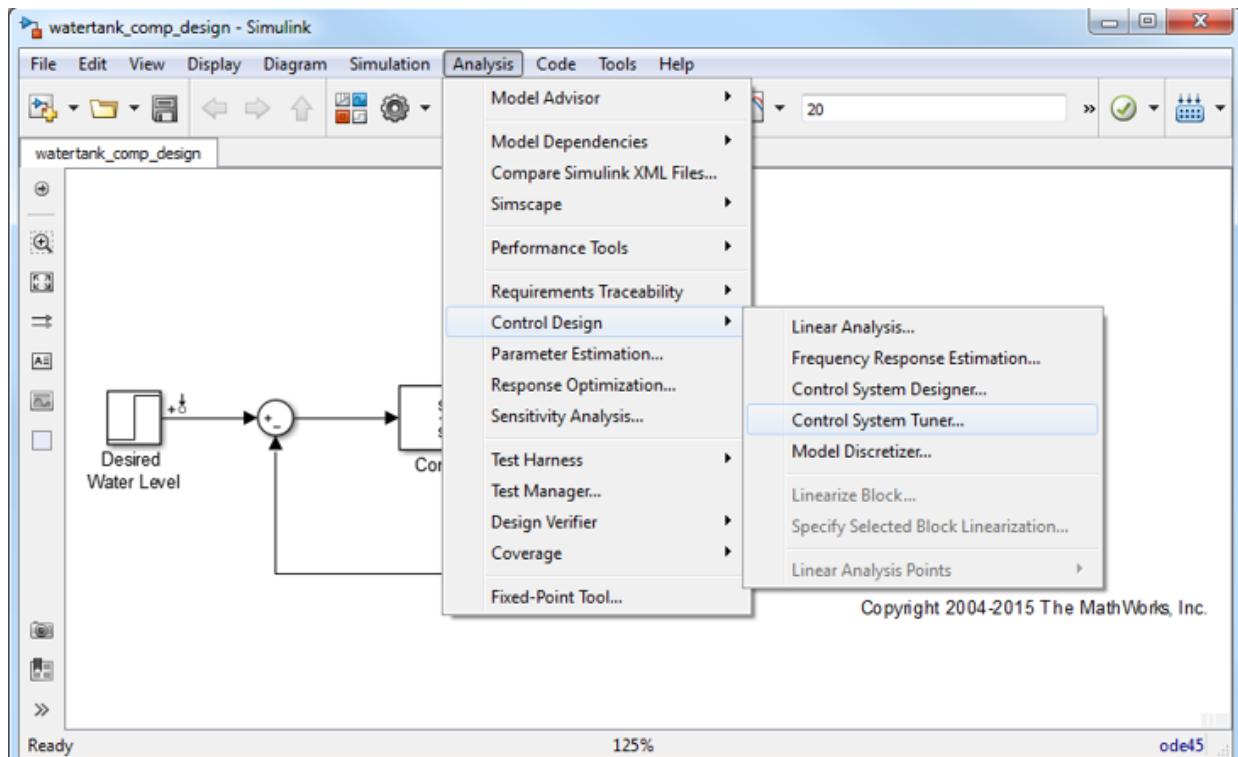
Since the water tank system is passive, it makes sense to require that the controller be strictly passive to guarantee closed-loop stability even when the plant model is inaccurate.

Compensator Tuning using Control System Tuner
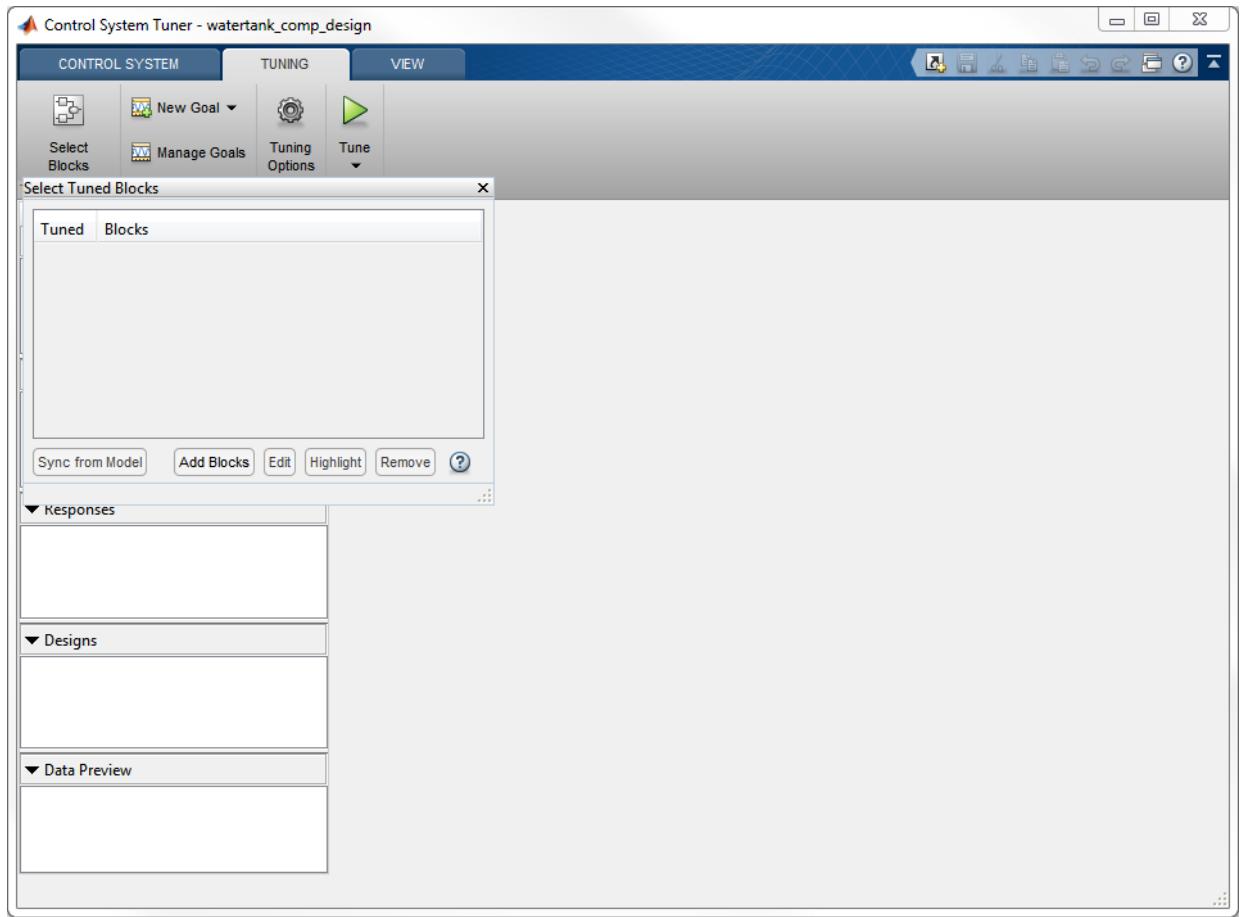
You can use the Control System Tuner app to tune the Controller block.

Step 1: Launch the Control System Tuner app from the Simulink model
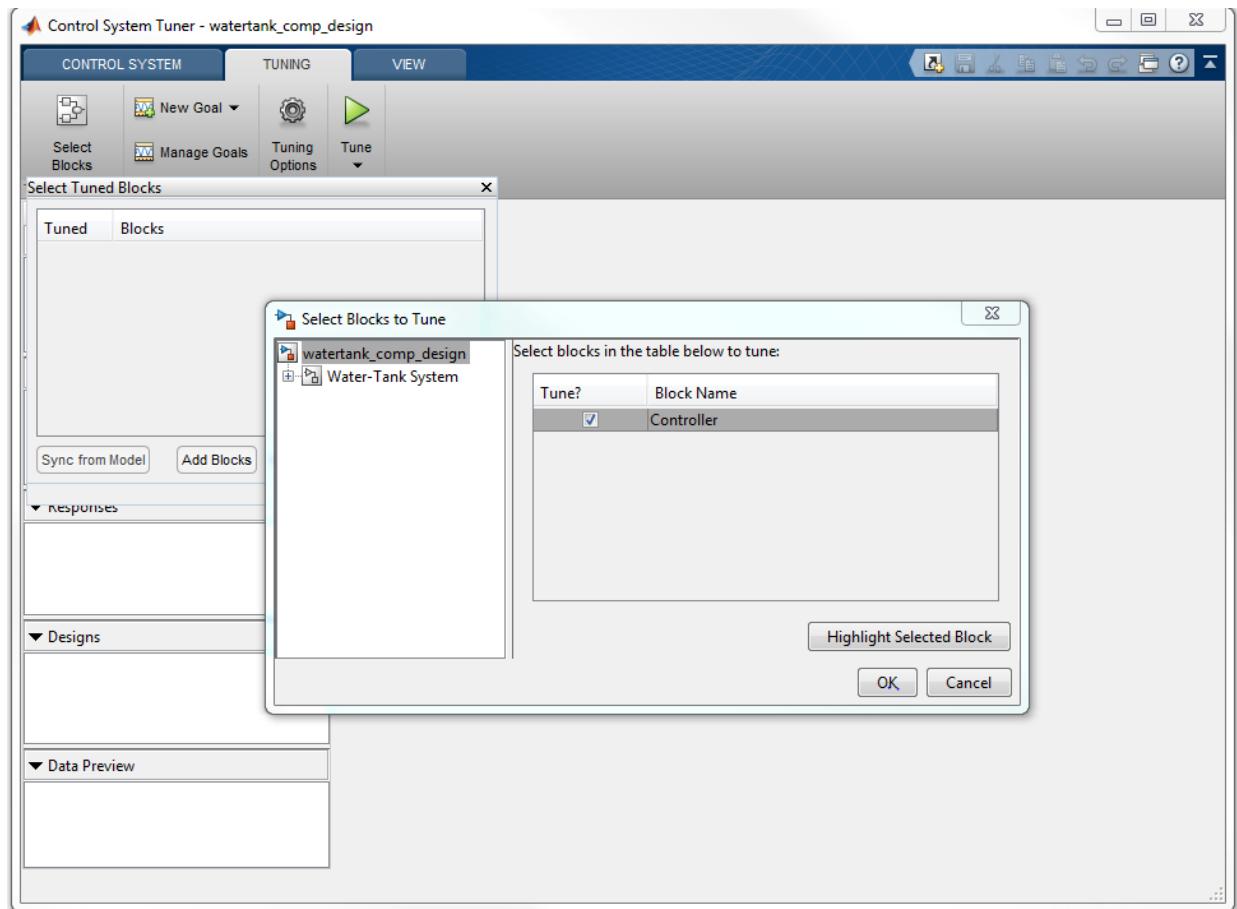
Step 2: Launch the tuned block selector from the **Select Blocks** button in the **Tuning** tab
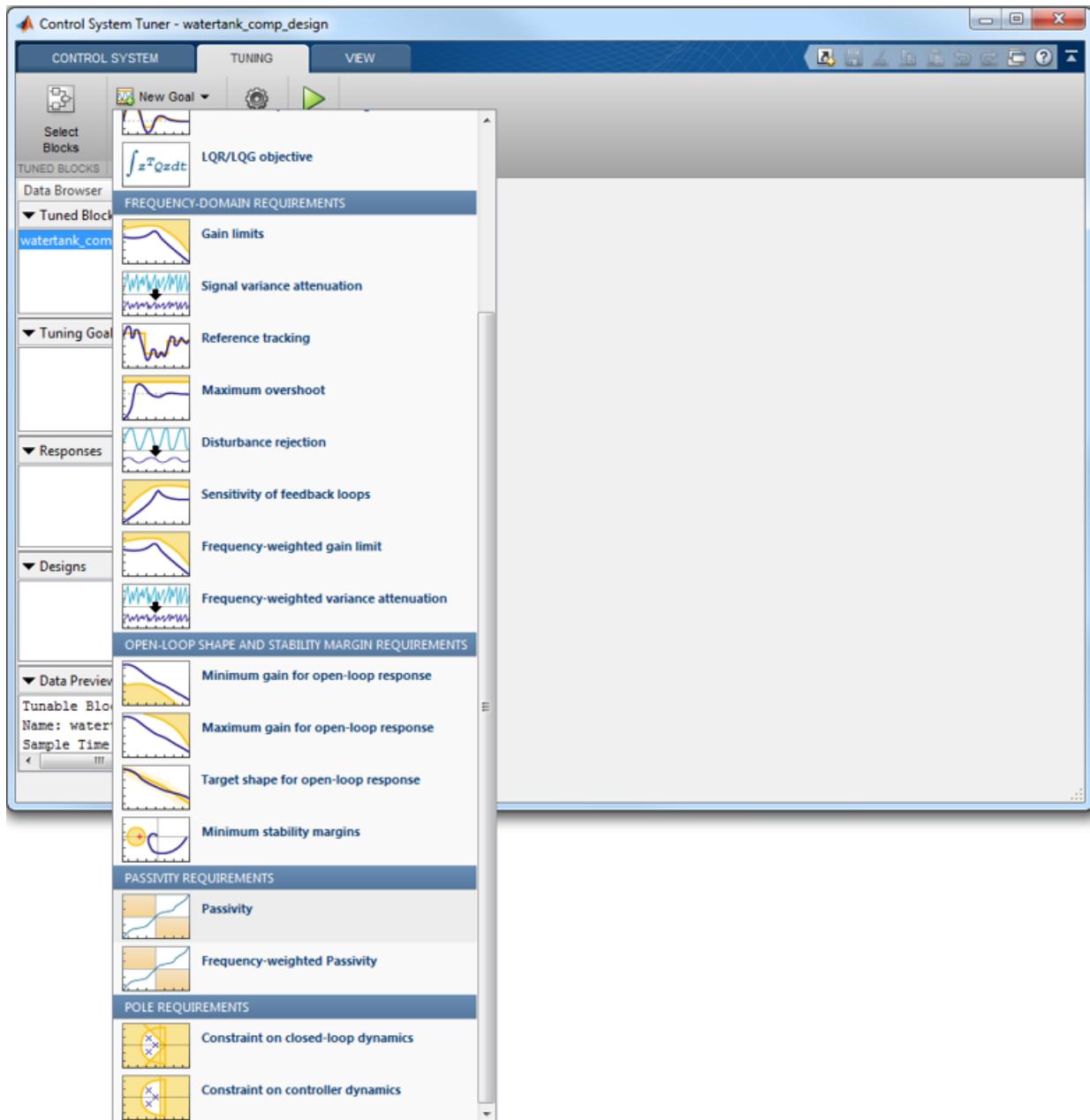
Step 3: Select the Controller block and Click OK. This block now appears in the Tuned Blocks list.
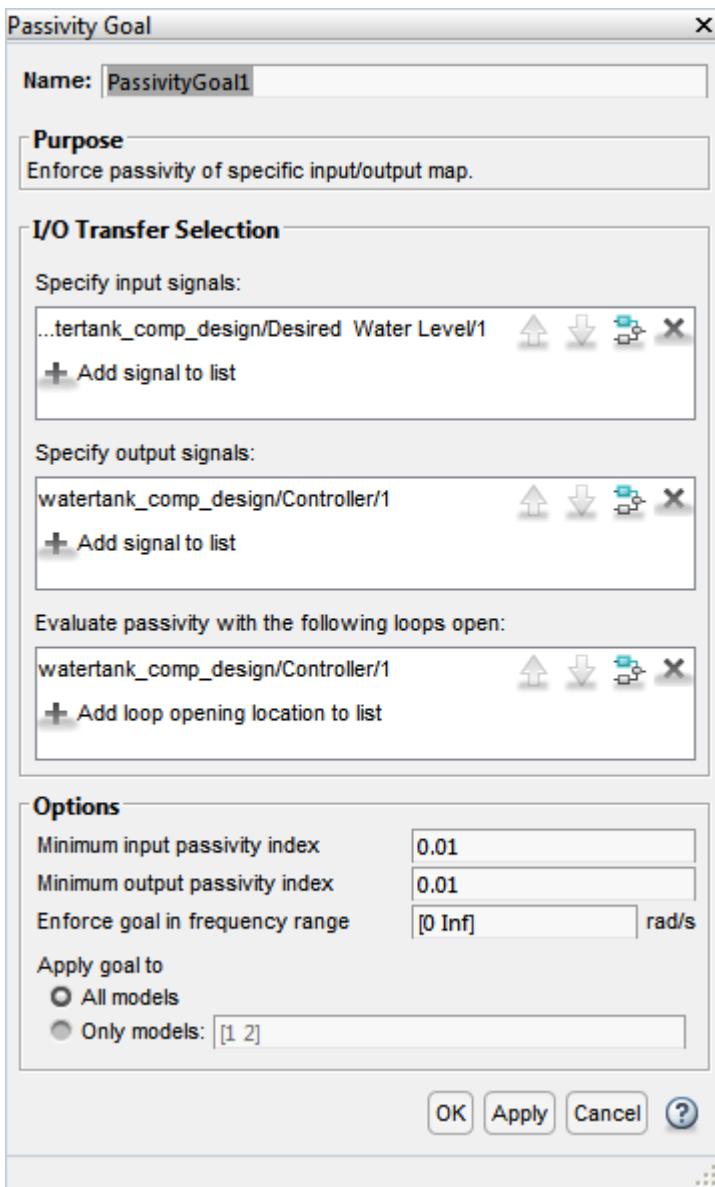
Step 4: Specify the tuning goals. Here there are two main goals:

1 Track step changes in water level

2 Make the controller passive
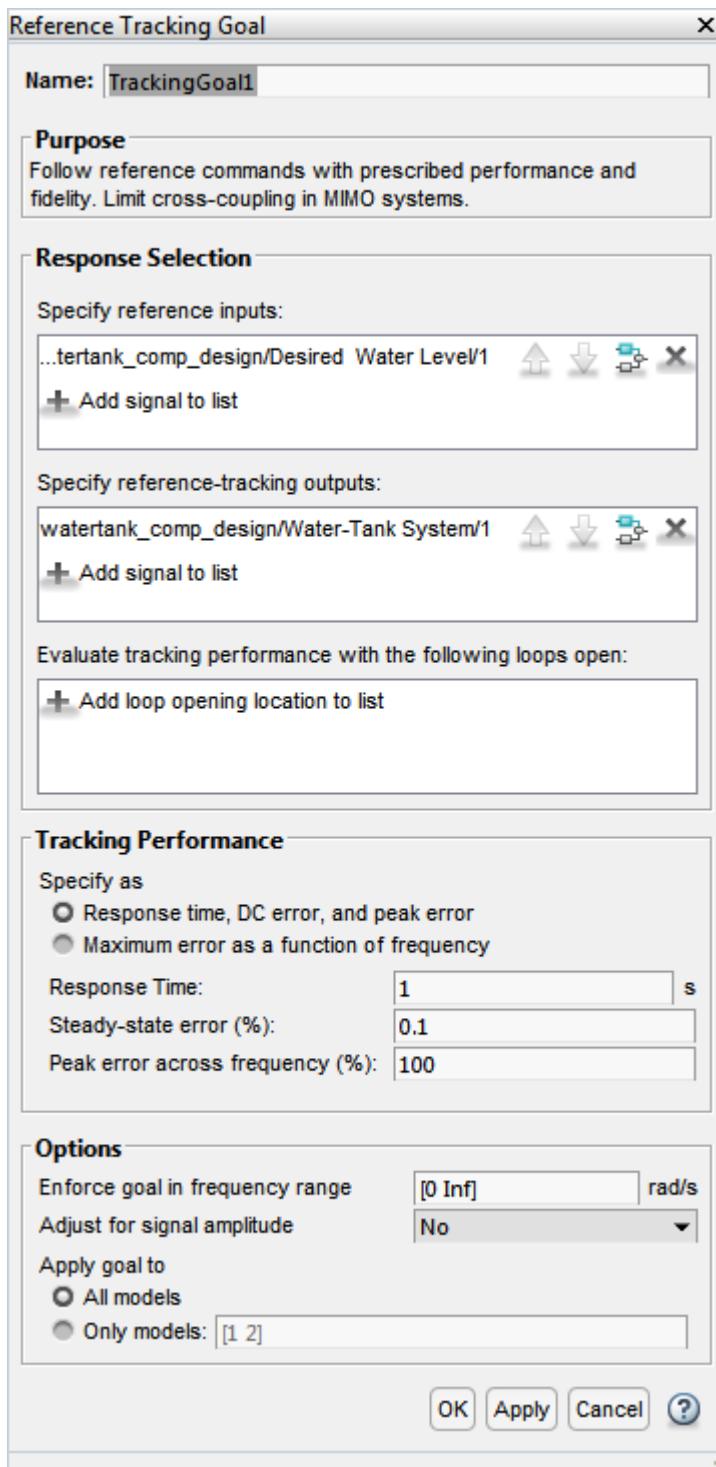
Click on the **New Goal** dropdown list and first add a **Passivity** goal.
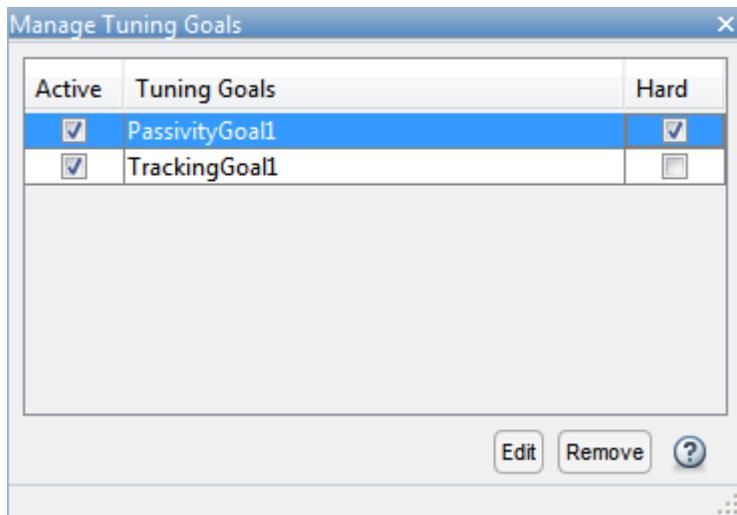
Configure this goal to apply to the Controller block only. This is done by setting the input signal to be the "Desired Water Level", the output signal to be the output of the Controller block, and the loop opening to be at the Controller block output. Also specify minimum passivity indices of 0.01 at the inputs and outputs to enforce strict passivity.

Passivity Goal   ×

Name: PassivityGoal1

Purpose
Enforce passivity of specific input/output map.

I/O Transfer Selection

Specify input signals:

...tertank_comp_design/Desired Water Level/1

+ Add signal to list

Specify output signals:

watertank_comp_design/Controller/1

+ Add signal to list

Evaluate passivity with the following loops open:

watertank_comp_design/Controller/1

+ Add loop opening location to list

Options

Minimum input passivity index   0.01
Minimum output passivity index   0.01
Enforce goal in frequency range   [0 Inf]   rad/s
Apply goal to
  ○ All models
  ○ Only models: [1 2]

OK Apply Cancel ?

Next add a **Reference Tracking** goal from the **New Goal** dropdown list. Configure this goal for a 1 second response time.

Reference Tracking Goal   ✕

Name: TrackingGoal1

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

Response Selection

Specify reference inputs:

...tertank_comp_design/Desired Water Level/1   ⬆ ⬇ 🔁 ✕
➕ Add signal to list

Specify reference-tracking outputs:

watertank_comp_design/Water-Tank System/1   ⬆ ⬇ 🔁 ✕
➕ Add signal to list

Evaluate tracking performance with the following loops open:

➕ Add loop opening location to list

Tracking Performance

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

Response Time:	1	s
Steady-state error (%):	0.1	
Peak error across frequency (%):	100	

Options

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

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

OK   Apply   Cancel   ?

Finally, click on the **Manage Goals** button off the **Tuning** tab and mark the Passivity goal as a hard tuning constraint.

Step 5: You are ready to tune the Controller block. Click the Tune button. You can view the tuning results side by side by selecting **Left/Right** in the **View** tab.
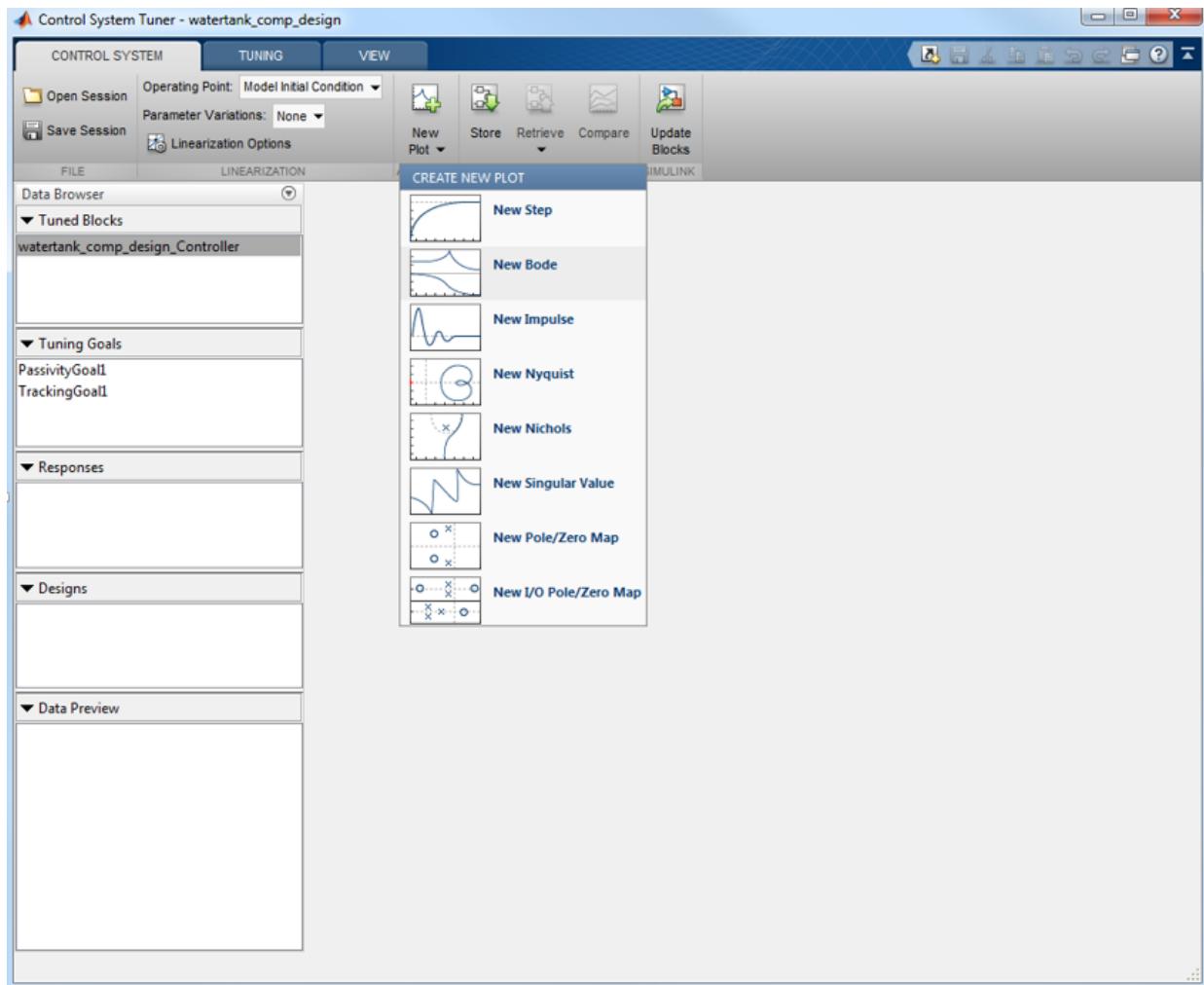
You can further analyze these results by generating a MATLAB script that reproduces this tuning process.
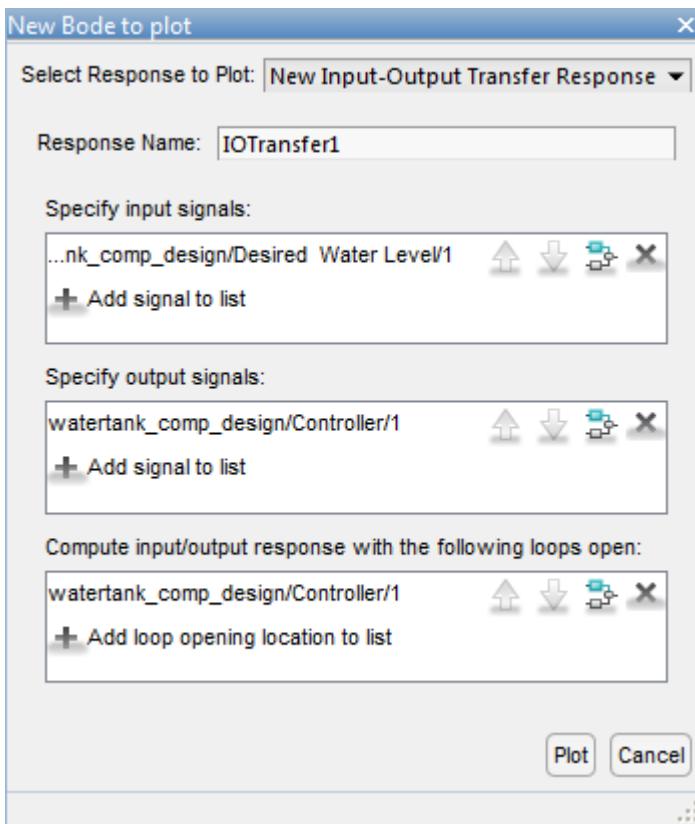
Closed-Loop Simulation
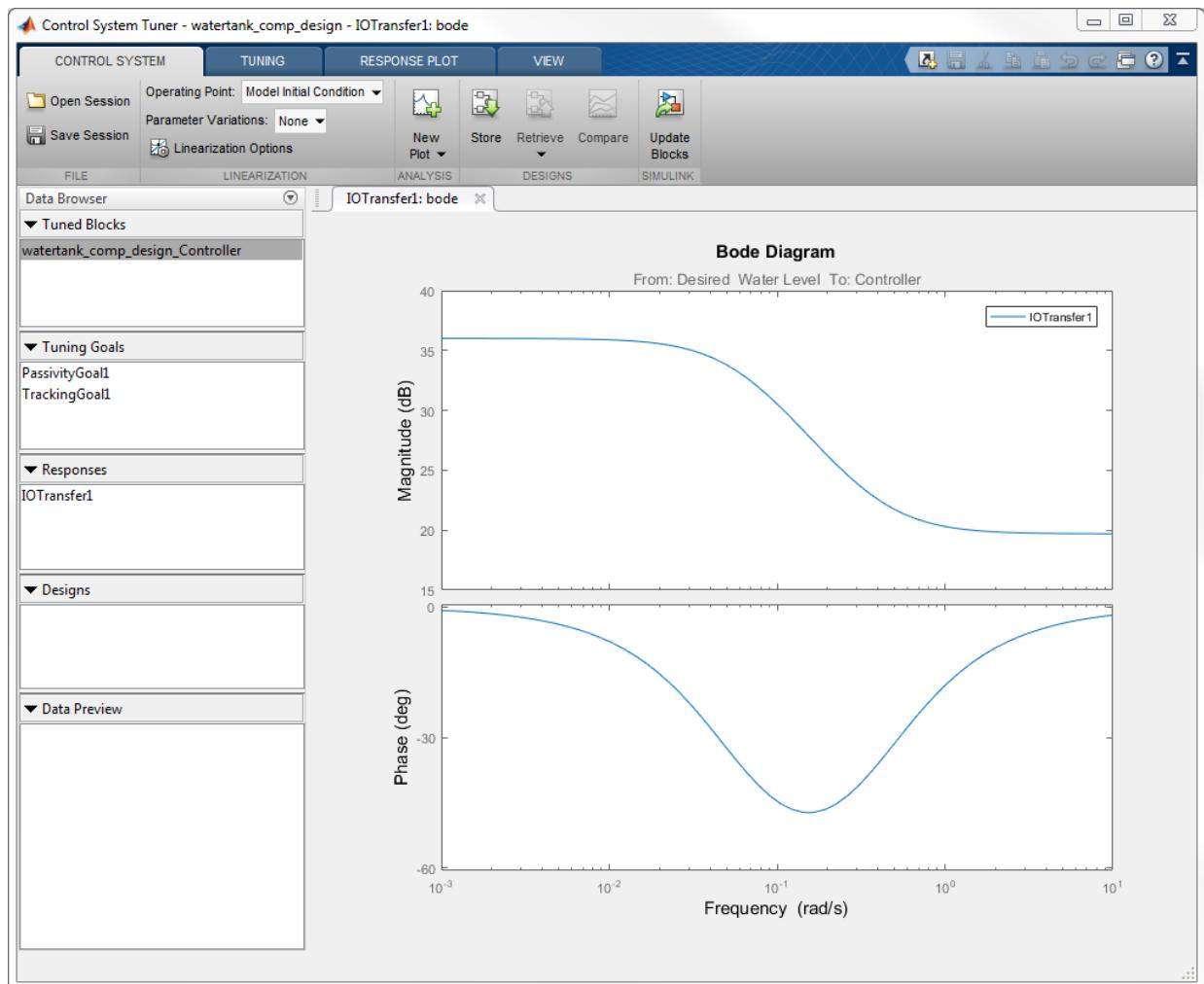
You can view the Bode plot of the tuned controller. Click on the **New Plot** button off the **Control System** tab. Select **New Bode** from dropdown list.
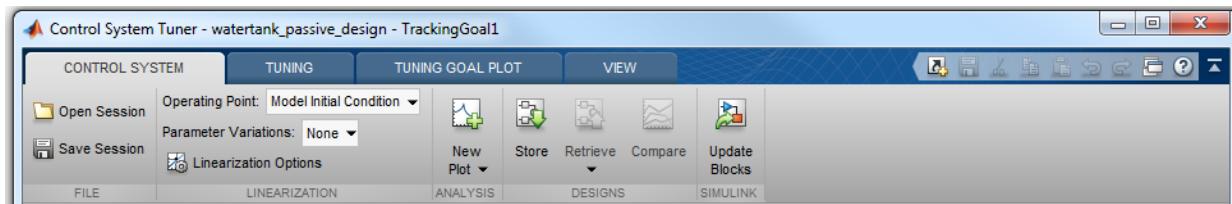
The controller response can be specified as follows.
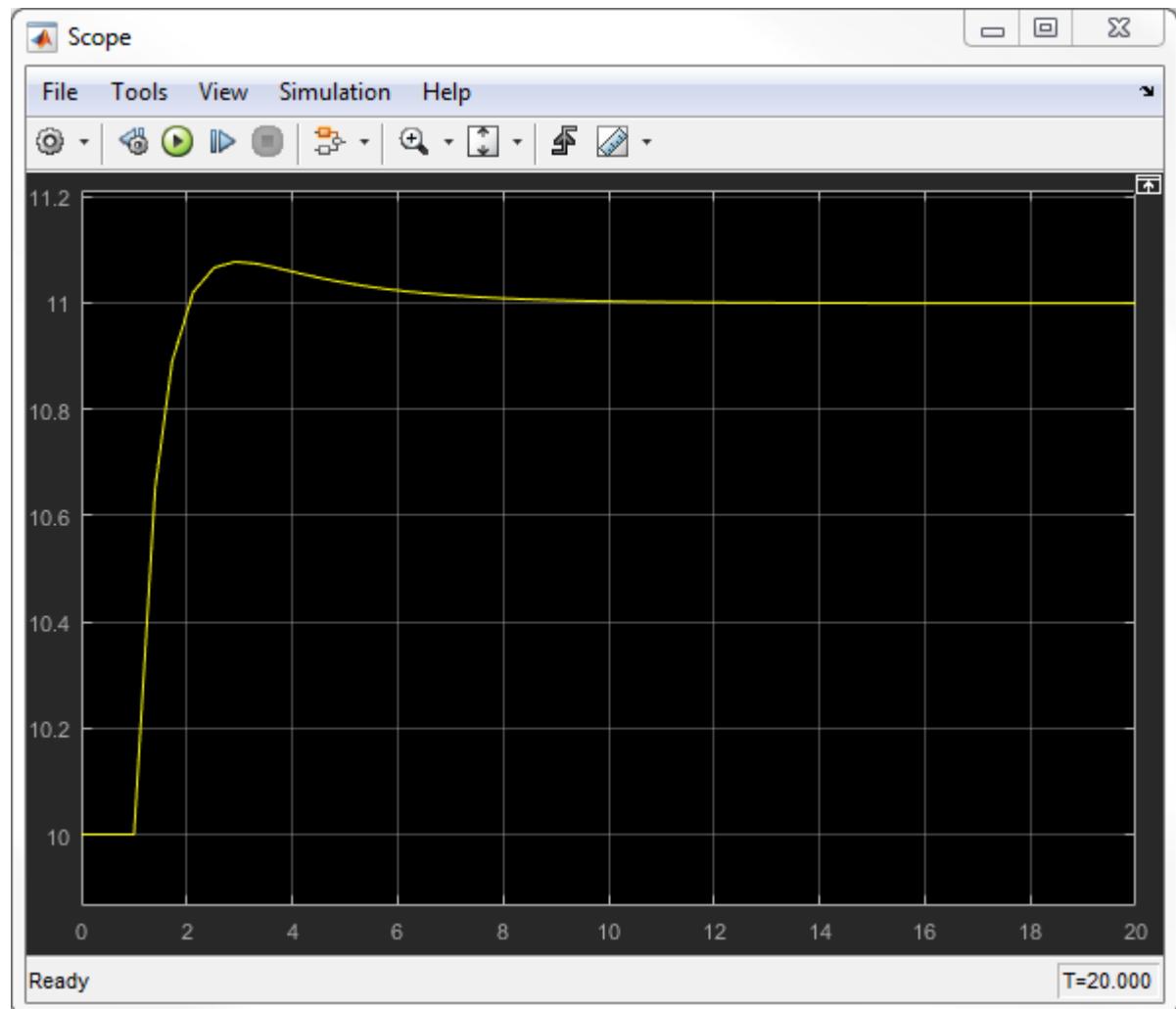
Click on the **Plot** button. The bode plot is shown in the following figure.

You can also simulate the closed-loop nonlinear response with the tuned controller. First, update the Controller block by clicking **Update Blocks** in the **Control System** tab.

In the Simulink model, double click the Scope block to open the Scope window, then simulate the model.

The nonlinear response of the tuned control system appears in the Scope window. This simulation shows that the tracking performance is satisfactory.

Related Examples

*   "Vibration Control in Flexible Beam" on page 13-467

More About

- "About Passivity and Passivity Indices" on page 10-2

Vibration Control in Flexible Beam

This example shows how to tune a controller for reducing vibrations in a flexible beam.

Model of Flexible Beam

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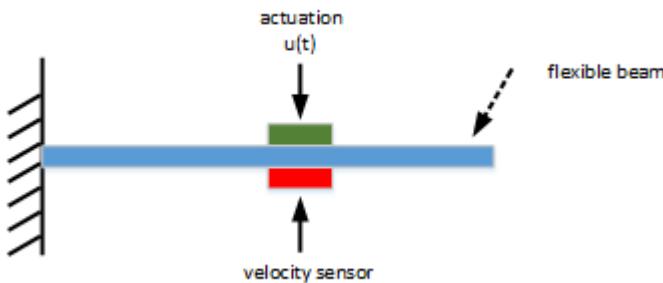
Figure 1: Active control of flexible beam

In this setup, the actuator delivering the force u and the velocity sensor are collocated. We can model the transfer function from control input u to the velocity y using finite-element analysis. Keeping only the first six modes, we obtain a plant model of the form

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

with the following parameter values.

```
% Parameters
xi = 0.05;
alpha = [0.09877, -0.309, -0.891, 0.5878, 0.7071, -0.8091];
w = [1, 4, 9, 16, 25, 36];
```

The resulting beam model for $G(s)$ is given by

```
% Beam model
```

```
G = tf(alpha(1)^2*[1,0],[1, 2*xi*w(1), w(1)^2]) + ...
    tf(alpha(2)^2*[1,0],[1, 2*xi*w(2), w(2)^2]) + ...
    tf(alpha(3)^2*[1,0],[1, 2*xi*w(3), w(3)^2]) + ...
    tf(alpha(4)^2*[1,0],[1, 2*xi*w(4), w(4)^2]) + ...
    tf(alpha(5)^2*[1,0],[1, 2*xi*w(5), w(5)^2]) + ...
    tf(alpha(6)^2*[1,0],[1, 2*xi*w(6), w(6)^2]);

G.InputName = 'uG';  G.OutputName = 'y';
```

With this sensor/actuator configuration, the beam is a passive system:
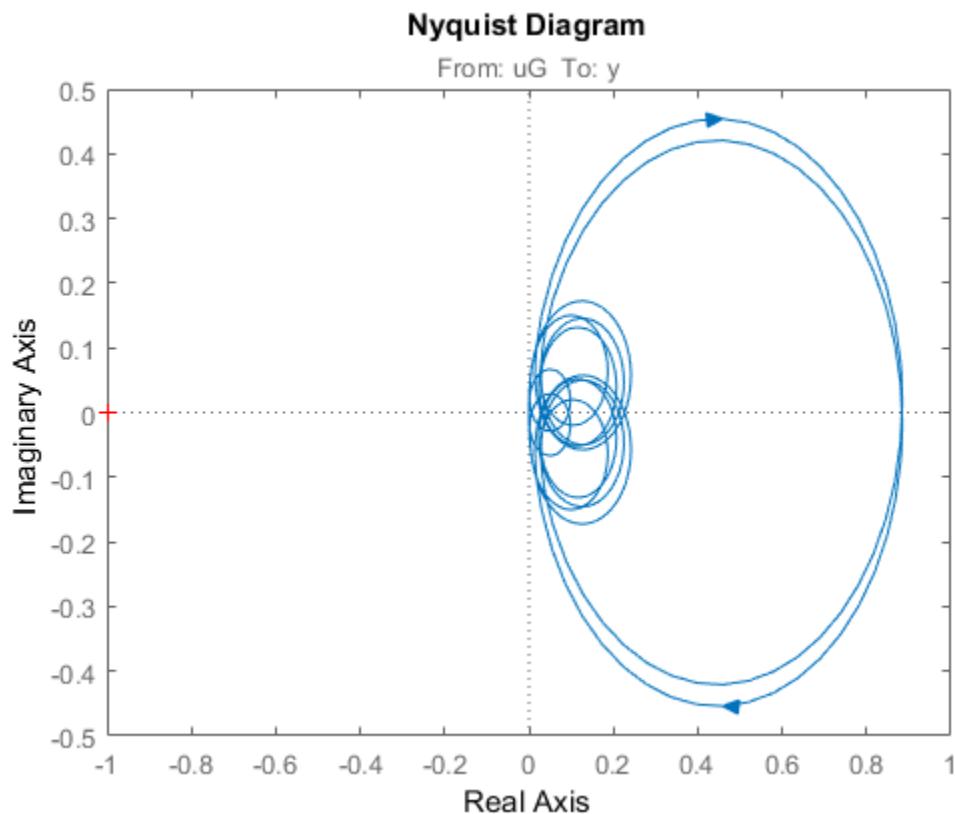
```
isPassive(G)
```

```
ans =

     1
```

This is confirmed by observing that the Nyquist plot of G is positive real.

```
nyquist(G)
```

LQG Controller

LQG control is a natural formulation for active vibration control. The LQG control setup is depicted in Figure 2. The signals d and n are the process and measurement noise, respectively.
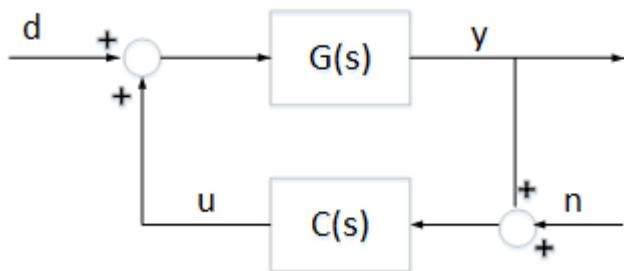
Figure 2: LQG control structure

First use `lqg` to compute the optimal LQG controller for the objective

$$J = \lim_{T \to \infty} E \left(\int_0^T (y^2(t) + 0.001u^2(t)) dt \right)$$

with noise variances:

$$E(d^2(t)) = 1, \quad E(n^2(t)) = 0.01.$$
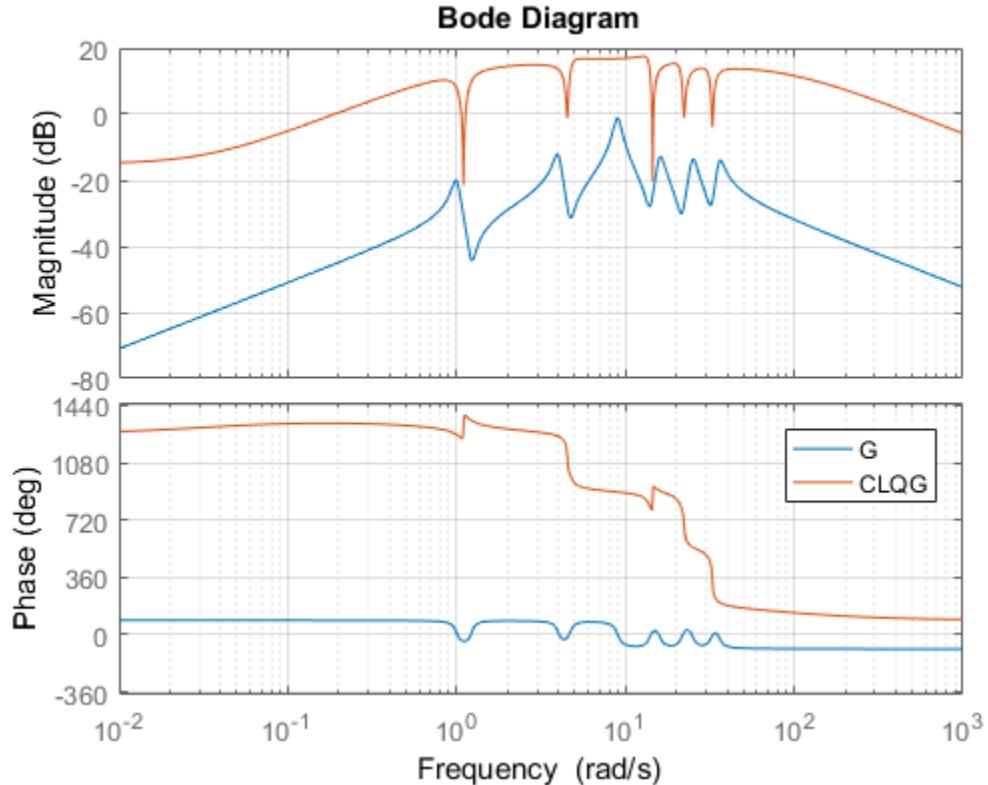
```
[a,b,c,d] = ssdata(G);
M = [c d;zeros(1,12) 1];   % [y;u] = M * [x;u]
QWV = blkdiag(b*b',1e-2);
QXU = M'*diag([1 1e-3])*M;
CLQG = lqg(ss(G),QXU,QWV);
```

The LQG-optimal controller `CLQG` is complex with 12 states and several notching zeros.

```
size(CLQG)
```

State-space model with 1 outputs, 1 inputs, and 12 states.

```
bode(G,CLQG,{1e-2,1e3}), grid, legend('G','CLQG')
```

Bode Diagram

Use the general-purpose tuner `systune` to try and simplify this controller. With `systune`, you are not limited to a full-order controller and can tune controllers of any order. Here for example, let's tune a 2nd-order state-space controller.

```
C = ltiblock.ss('C',2,1,1);
```

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'},{'yn','u'});
```

Use the LQG criterion J above as sole tuning goal. The LQG tuning goal lets you directly specify the performance weights and noise covariances.

```
R1 = TuningGoal.LQG({'d','n'},{'y','u'},diag([1,1e-2]),diag([1 1e-3]));
```

Now tune the controller C to minimize the LQG objective J.

```
[CL1,J1] = systune(CLO,R1);
```

```
Final: Soft = 0.478, Hard = -Inf, Iterations = 42
```

The optimizer found a 2nd-order controller with $J = 0.478$. Compare with the optimal J value for CLQG:
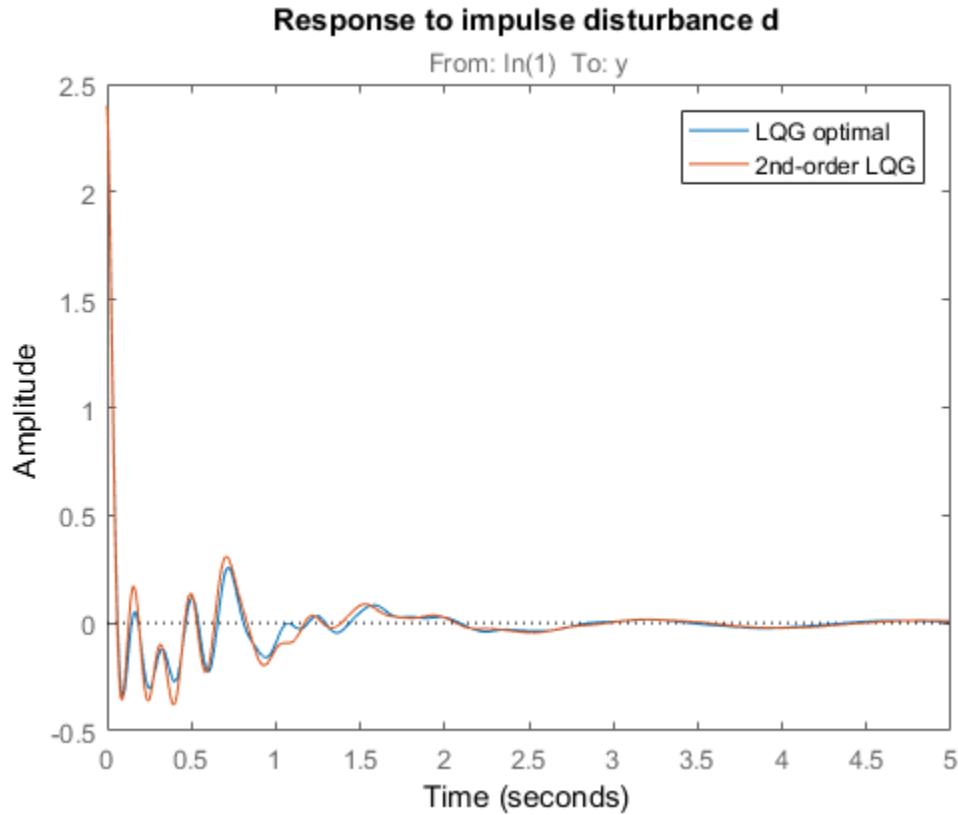
```
[~,Jopt] = evalSpec(R1,replaceBlock(CLO,'C',CLQG))
```

```
Jopt =

    0.4673
```

The performance degradation is less than 5%, and we reduced the controller complexity from 12 to 2 states. Further compare the impulse responses from d to y for the two controllers. The two responses are almost identical. You can therefore obtain near-optimal vibration attenuation with a simple second-order controller.

```
T0 = feedback(G,CLQG,+1);
T1 = getIOTransfer(CL1,'d','y');
impulse(T0,T1,5)
title('Response to impulse disturbance d')
legend('LQG optimal','2nd-order LQG')
```

Response to impulse disturbance d

From: In(1) To: y

Passive LQG Controller

We used an approximate model of the beam to design these two controllers. A priori, there is no guarantee that these controllers will perform well on the real beam. However, we know that the beam is a passive physical system and that the negative feedback interconnection of passive systems is always stable. So if $-C(s)$ is passive, we can be confident that the closed-loop system will be stable.
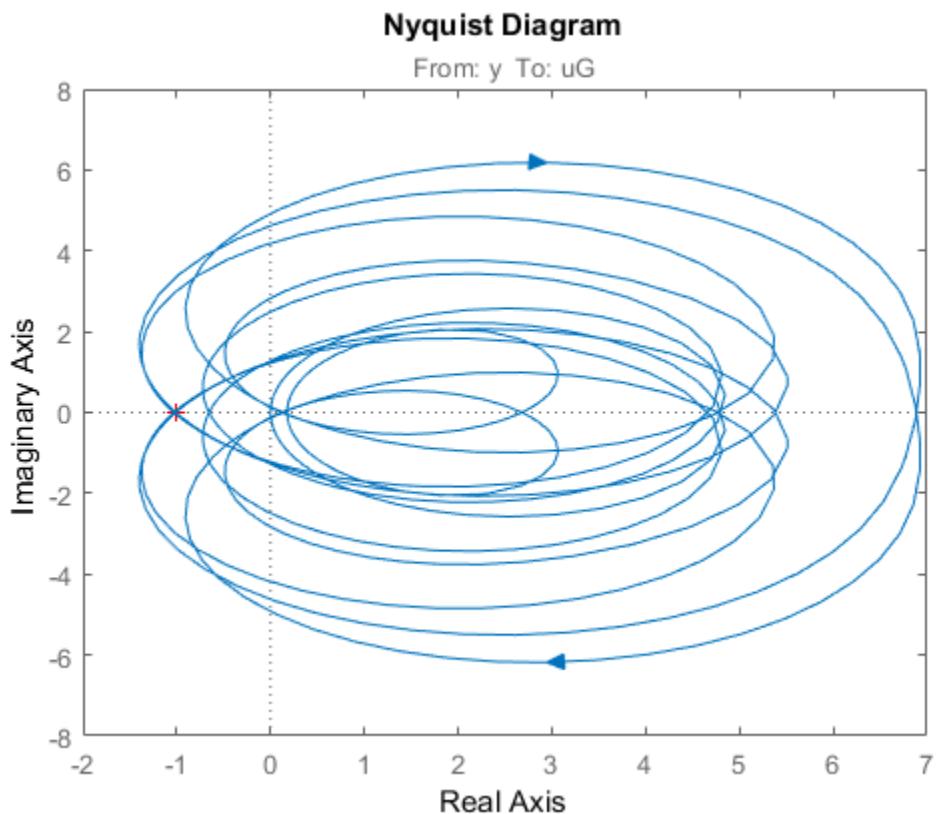
Is any of the two controllers `CLQG` and `C1` passive? The optimal LQG controller is not passive. In fact, its relative passive index is infinite because $1 - CLQG$ is not even minimum phase.

```
getPassiveIndex(-CLQG)
```

```
ans =

   Inf
```

```
nyquist(-CLQG)
```

Nyquist Diagram
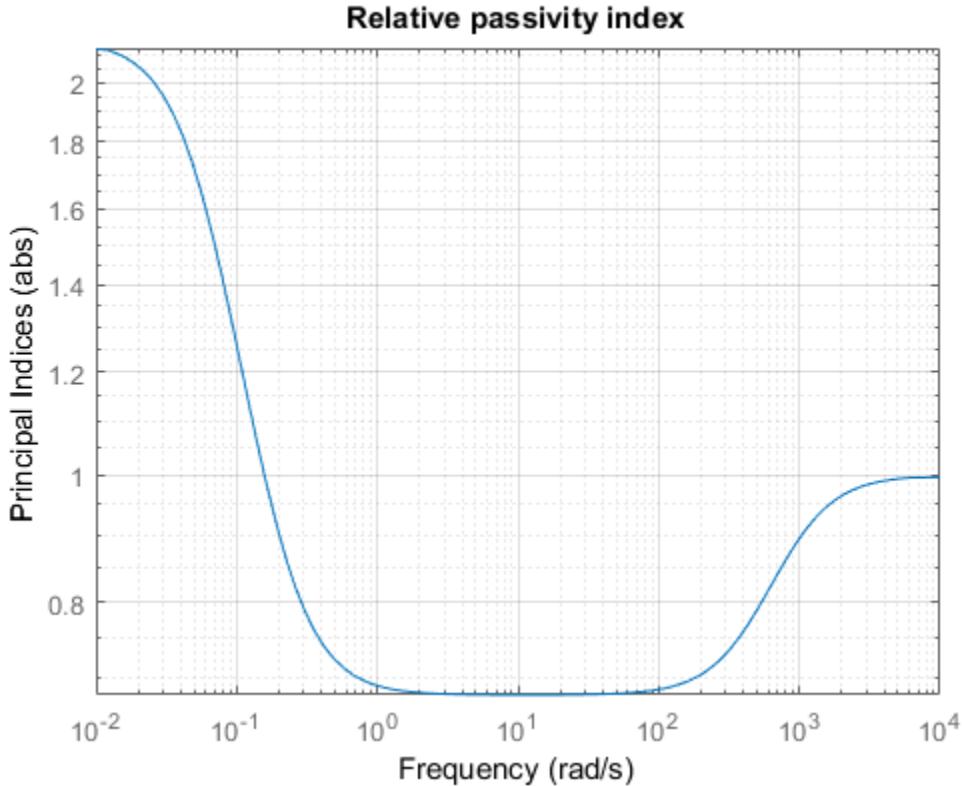
From: y To: uG

The 2nd-order controller `C1` fares better but is not passive (positive real) at low frequencies (its relative index is greater than 1).

```
C1 = getBlockValue(CL1,'C');
passiveplot(-C1)
```

Relative passivity index

Using `systune`, we can re-tune `C1` to try and enforce passivity of $-C(s)$. To do this, create a passivity tuning goal for the open-loop transfer function from `yn` to `u` (which is $C(s)$). Use the "WeightedPassivity" goal to account for the minus sign.

```
R2 = TuningGoal.WeightedPassivity({'yn'},{'u'},-1,1);
R2.Openings = 'u';
```
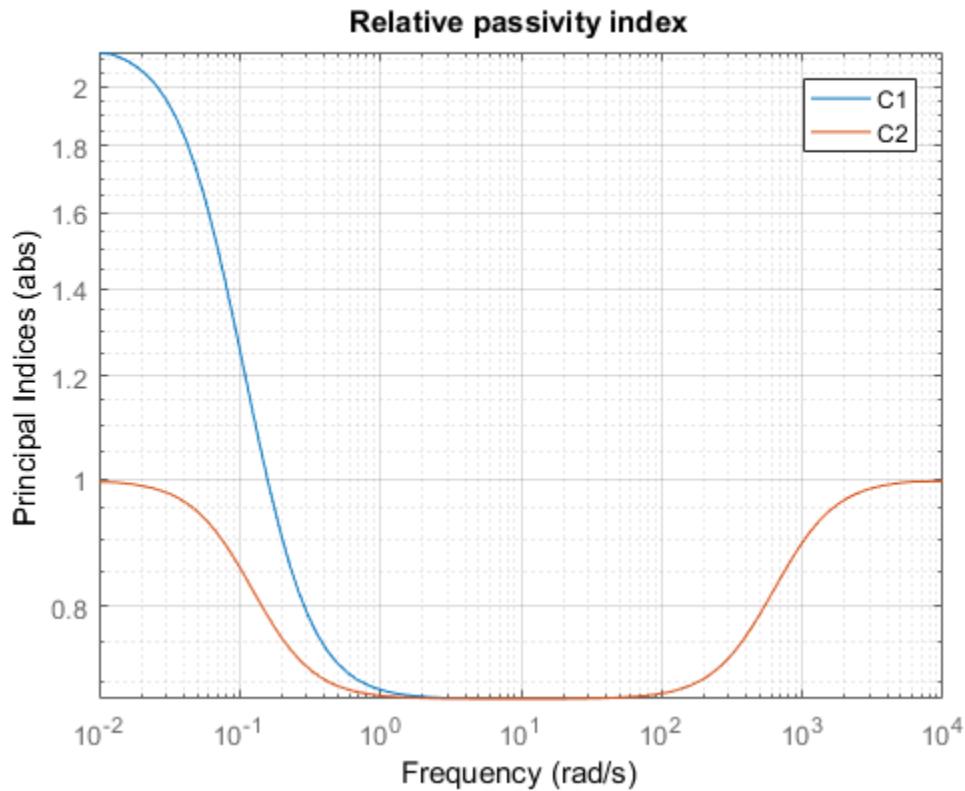
Now re-tune the closed-loop model `CL1` (which uses `C1`) to minimize the LQG objective J subject to $-C(s)$ being passive. Note that the passivity goal `R2` is specified as a hard constraint.

```
[CL2,J2,g] = systune(CL1,R1,R2);
```

```
Final: Soft = 0.478, Hard = 1, Iterations = 126
```
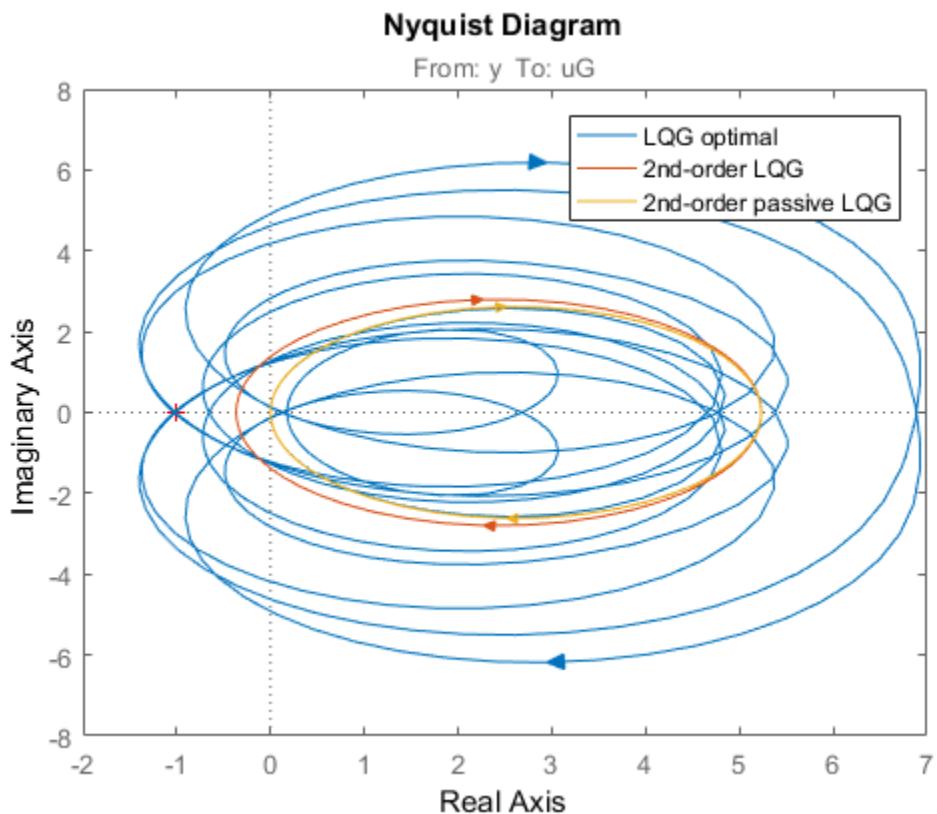
The tuner achieves the same J value as previously, while enforcing passivity (hard constraint less than 1). Compare the resulting controller C2 with C1. Clearly C2 corrects the shortage of passivity at low frequency in C1.

```
C2 = getBlockValue(CL2,'C');
passiveplot(-C1,-C2)
legend('C1','C2')
```
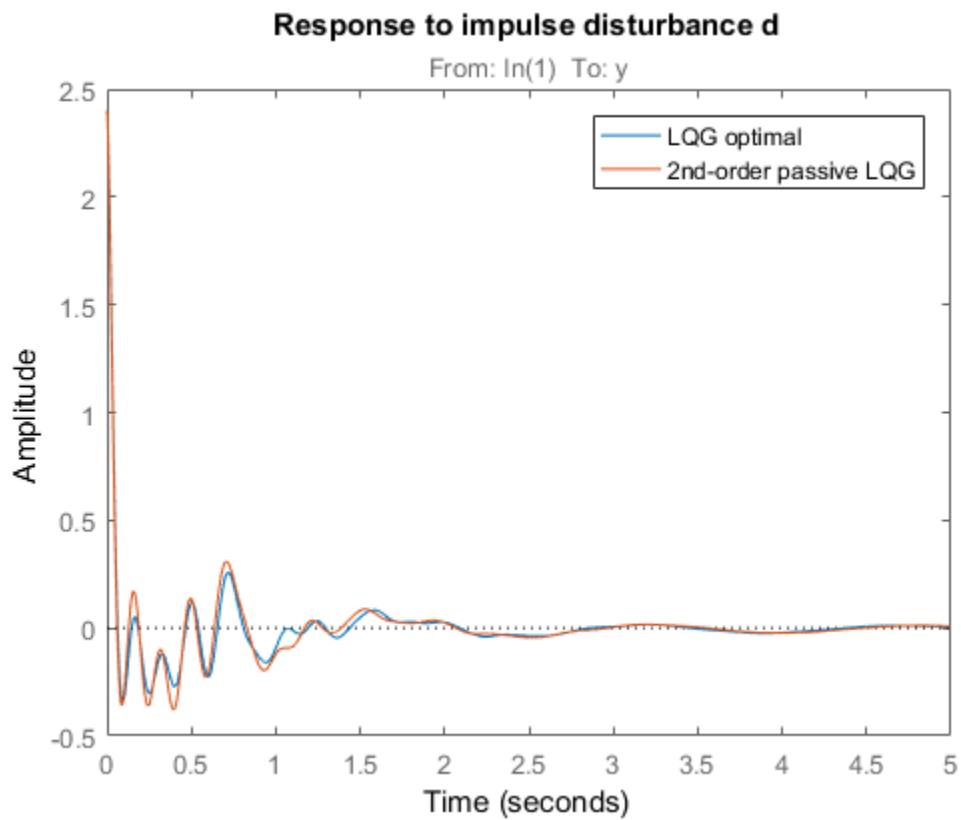

The improvement over the LQG-optimal controller is most visible in the Nyquist plot.

```
nyquist(-CLQG,-C1,-C2)
legend('LQG optimal','2nd-order LQG','2nd-order passive LQG')
```

Finally, compare the impulse responses from d to y.

```
T2 = getIOTransfer(CL2,'d','y');
impulse(T0,T2,5)
title('Response to impulse disturbance d')
legend('LQG optimal','2nd-order passive LQG')
```

Response to impulse disturbance d

Using `systune`, you designed a second-order passive controller with near-optimal LQG performance.

See Also
TuningGoal.WeightedPassivity | TuningGoal.LQG

Related Examples
- "Passive Control of Water Tank Level" on page 13-448
- "Passive Control with Communication Delays" on page 13-480

More About
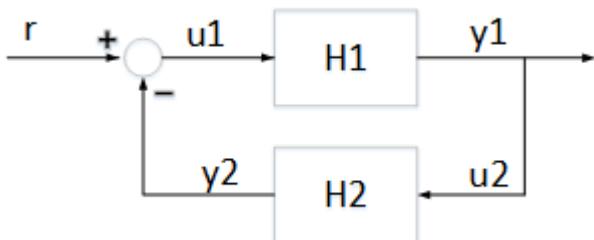
· "About Passivity and Passivity Indices" on page 10-2

Passive Control with Communication Delays

This example shows how to mitigate communication delays in a passive control system.
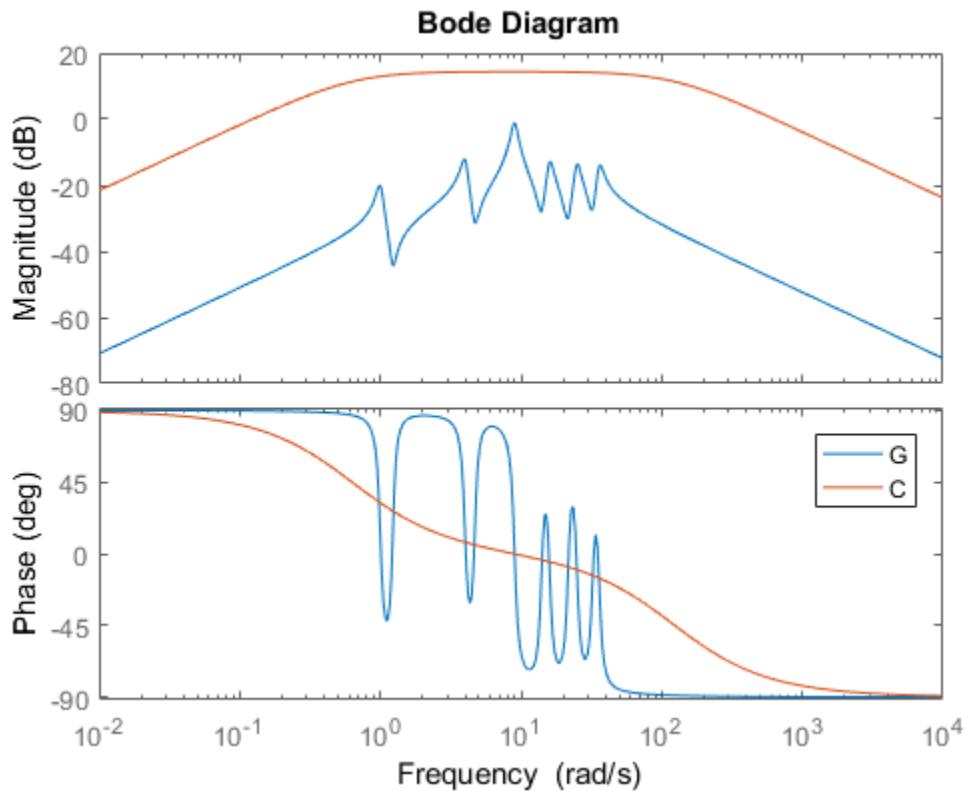
Passivity-Based Control

By the Passivity Theorem, the negative-feedback interconnection of two strictly passive systems H_1 and H_2 is always stable.
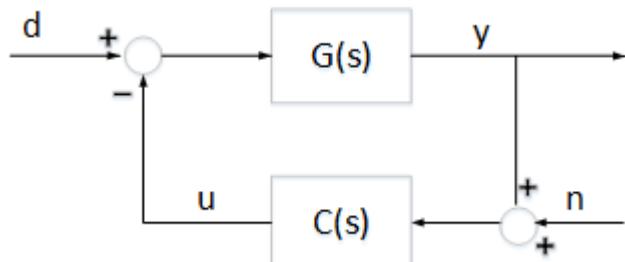
When the physical plant is passive, it is therefore advantageous to use a passive controller for robustness and safety reasons. In networked control systems, however, communication delays can undo the benefits of passivity-based control and lead to instability. To illustrate this point, we use the plant and 2nd-order passive controller from the "Vibration Control in Flexible Beam" example. See this example for background on the underlying control problem. Load the plant model G and passive controller C (note that C corresponds to $-C$ in the other example).
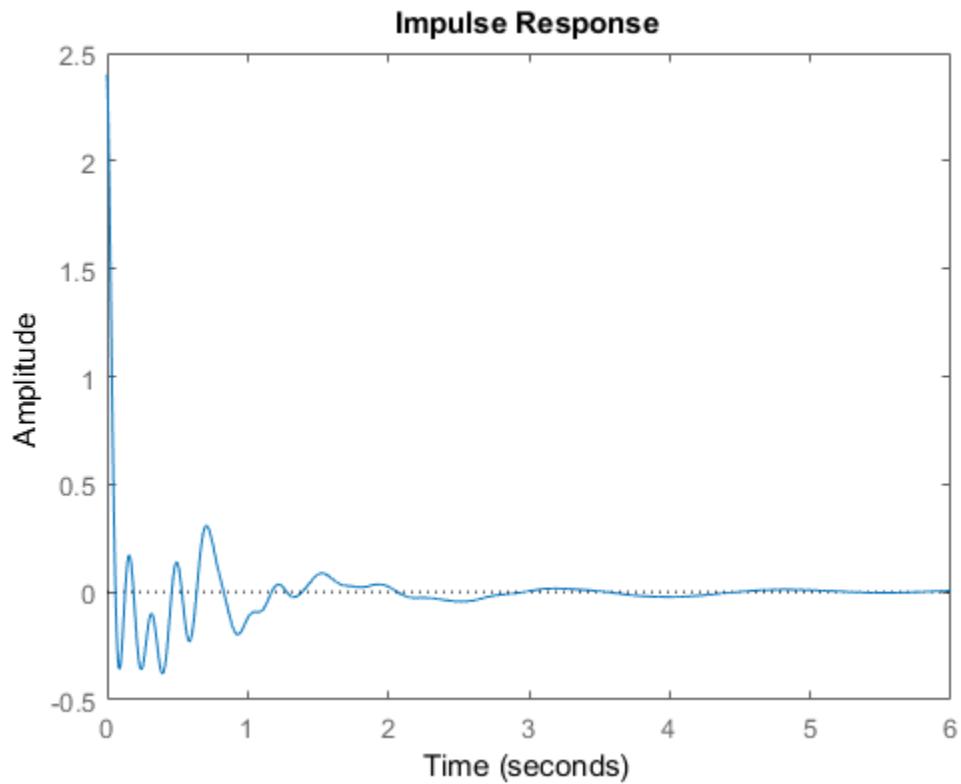
```
load BeamControl G C

bode(G,C,{1e-2,1e4})
legend('G','C')
```

Bode Diagram

The control configuration is shown below as well as the impulse response from d to y.
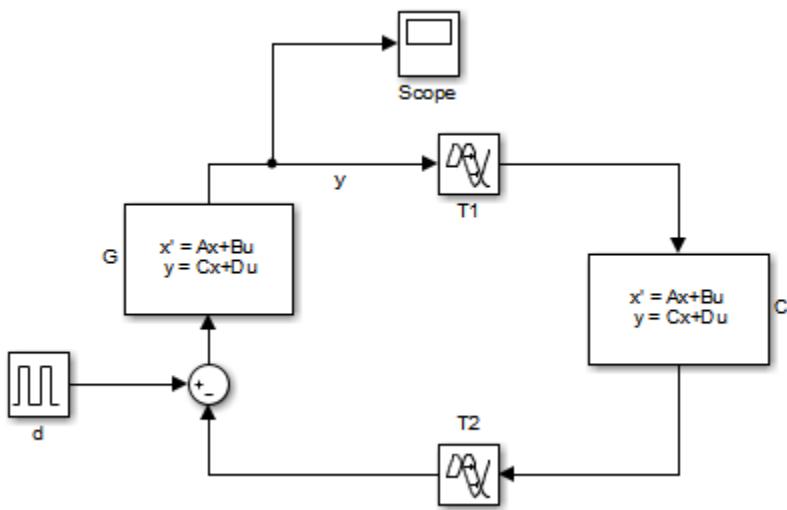

```
impulse(feedback(G,C))
```

Destabilizing Effect of Communication Delays

Now suppose there are substantial communication delays between the sensor and the controller, and between the controller and the actuator. This situation is modeled in Simulink as follows.
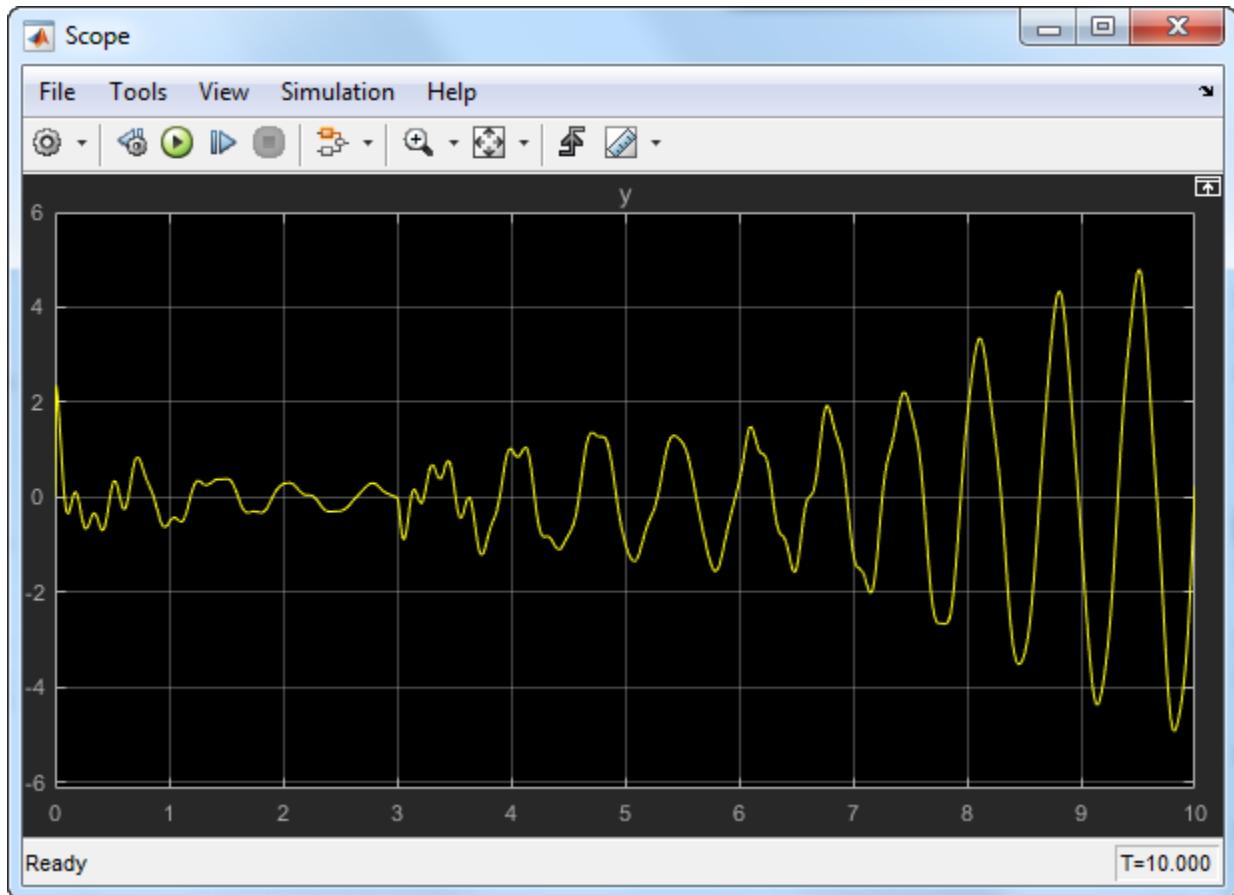
```
open_system('DelayedFeedback')
```

The communication delays are set to

```
T1 = 1;
T2 = 2;
```

Simulating this model shows that the communication delays destabilize the feedback loop.

Scattering Transformation

To mitigate the delay effects, you can use a simple linear transformation of the signals exchanged between the plant and controller over the network.
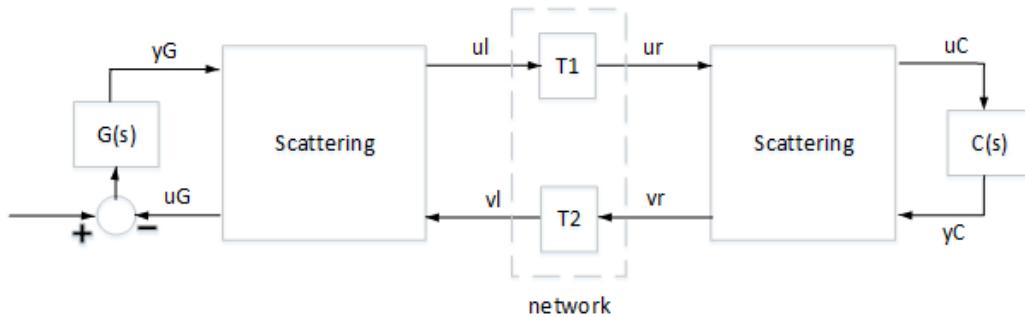
Figure 1: Networked Control System

This is called the "scattering transformation" and given by the formulas

$$\begin{pmatrix} u_l \\ v_l \end{pmatrix} = \begin{pmatrix} 1 & b \\ 1 & -b \end{pmatrix} \begin{pmatrix} u_G \\ y_G \end{pmatrix}, \quad \begin{pmatrix} u_r \\ v_r \end{pmatrix} = \begin{pmatrix} 1 & b \\ 1 & -b \end{pmatrix} \begin{pmatrix} y_C \\ u_C \end{pmatrix},$$

or equivalently

$$\begin{pmatrix} u_l \\ u_G \end{pmatrix} = S \begin{pmatrix} v_l \\ y_G \end{pmatrix}, \quad \begin{pmatrix} v_r \\ u_C \end{pmatrix} = S^{-1} \begin{pmatrix} u_r \\ y_C \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 2b \\ 1 & b \end{pmatrix}$$

with $b > 0$. Note that in the absence of delays, the two scattering transformations cancel each other and the block diagram in Figure 1 is equivalent to the negative feedback interconnection of G and C.

When delays are present, however, (u_l, v_l) is no longer equal to (u_r, v_r) and this scattering transformation alters the properties of the closed-loop system. In fact, observing that
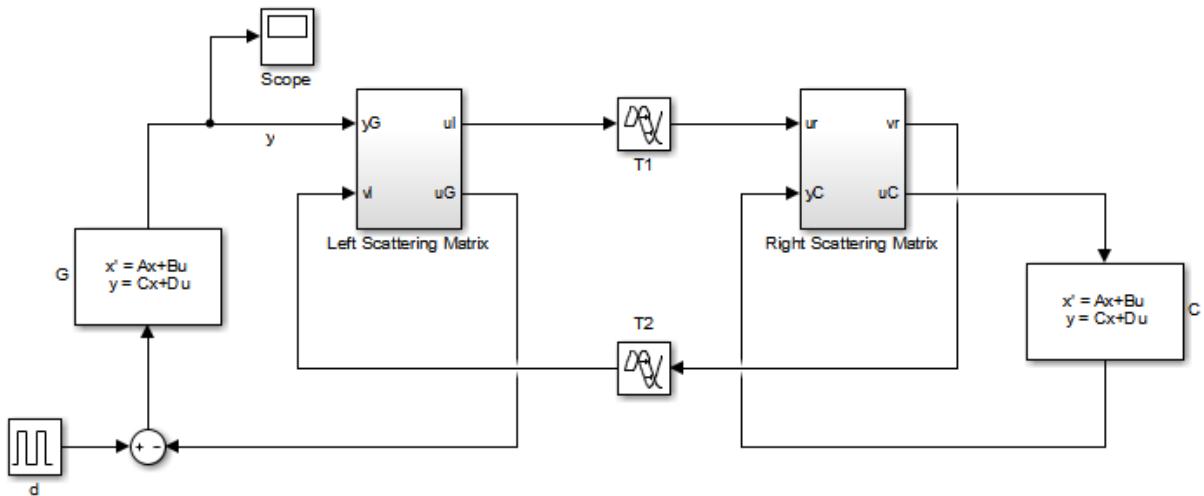
$$u_l = (1 - bC(s))/(1 + bC(s))v_l, \quad v_r = (G(s)/b - 1)/(G(s)/b + 1)u_r$$

and that bC and G/b strictly passive ensures that
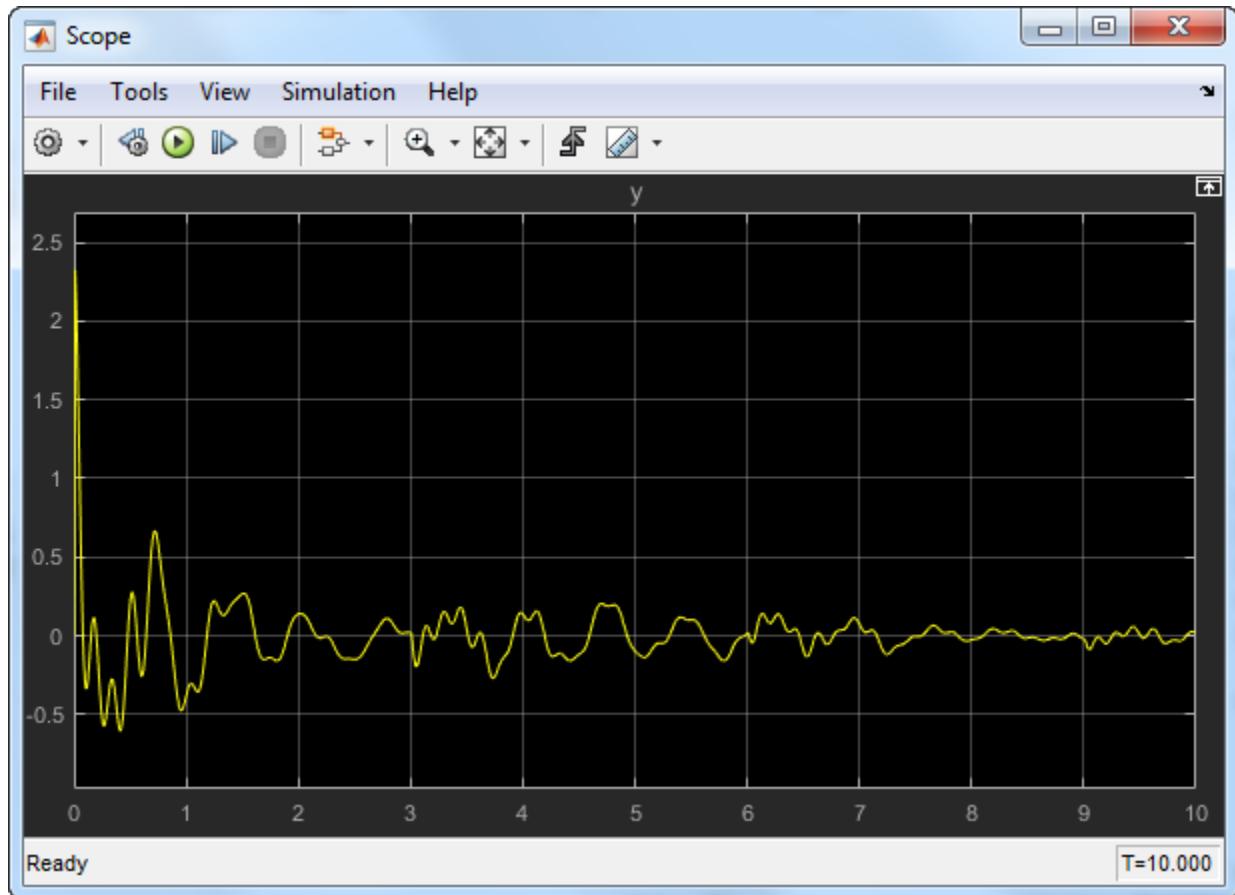
$$\|(1 - bC)/(1 + bC)\|_\infty < 1, \quad \|(G/b - 1)/(G/b + 1)\|_\infty < 1,$$

the Small Gain Theorem guarantees that the feedback interconnection of Figure 1 is always stable no matter how large the delays. Confirm this by building a Simulink model of the block diagram in Figure 1 for the value $b = 1$.

```
b = 1;
```

```
open_system('ScatteringTransformation')
```


Simulate the impulse response of the closed-loop system as done before. The response is now stable and similar to the delay-free response in spite of the large delays.

For more details on the scattering transformation, see T. Matiakis, S. Hirche, and M. Buss, "Independent-of-Delay Stability of Nonlinear Networked Control Systems by Scattering Transformation," Proceedings of the 2006 American Control Conference, 2006, pp. 2801-2806.

See Also

getPassiveIndex | isPassive

Related Examples

- "Vibration Control in Flexible Beam" on page 13-467

More About
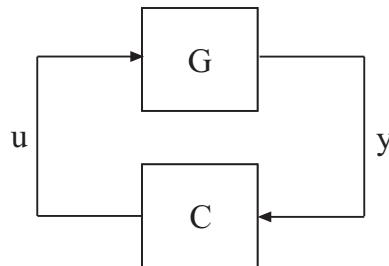
- "About Passivity and Passivity Indices" on page 10-2

Loop-Shaping Design
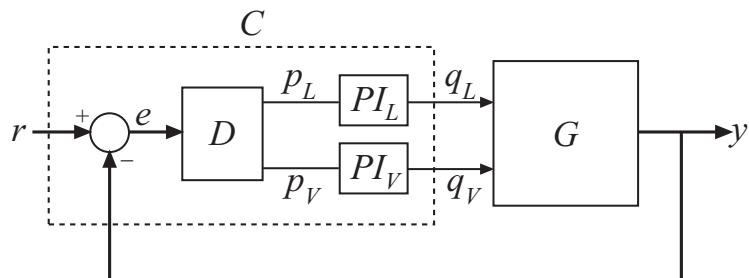
Structure of Control System for Tuning With looptune

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

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

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

The controller C has a fixed internal structure. C includes a gain matrix D , the PI controllers PI_L and PI_V, and a summing junction. The looptune command tunes free parameters of C such as the gains in D and the proportional and integral gains of PI_L and PI_V. You can also use looptune to co-tune free parameters in both C and G.

Set Up Your Control System for Tuning with looptune

Set Up Your Control System for looptunein MATLAB

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

1. Parameterize the tunable elements of your controller. You can use predefined structures such as `tunablePID`, `tunableGain`, and `tunableTF`. Or, you can create your own structure from elementary tunable parameters (`realp`).

2. Use model interconnection commands such as `series` and `connect` to build a tunable `genss` model representing the controller `CO`.

3. Create a "Numeric Linear Time Invariant (LTI) Models" on page 1-13 representing the plant `G`. For co-tuning the plant and controller, represent the plant as a tunable `genss` model.

Set Up Your Control System for looptune in Simulink

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

1. Use `slTuner` to create an interface to the Simulink model of your control system. When you create the interface, you specify which blocks to tune in your model.

2. Use `addPoint` to specify the control and measurement signals that define the boundaries between plant and controller. Use `addOpening` to mark optional loop-opening or signal injection sites for specifying and assessing open-loop requirements.

The `slTuner` interface automatically linearizes your Simulink model. The `slTuner` interface also automatically parametrizes the blocks that you specify as tunable blocks. For more information about this linearization, see the `slTuner` reference page and "How Tuned Simulink Blocks Are Parameterized" on page 13-36.

Related Examples

- "Tune MIMO Control System for Specified Bandwidth" on page 14-5
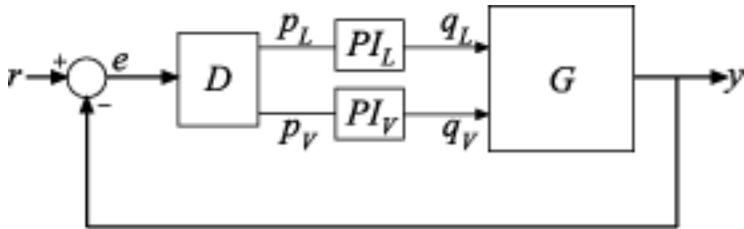
• "Tuning Feedback Loops with LOOPTUNE" on page 14-11

More About

• "Structure of Control System for Tuning With looptune" on page 14-2

Tune MIMO Control System for Specified Bandwidth

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

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

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

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

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

Create a model of the plant.

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

When you tune the control system, `looptune` uses the channel names `G.InputName` and `G.OutputName` to interconnect the plant and controller. Therefore, assign these channel names to match the illustration. When you set `G.OutputName = 'y'`, the `G.OutputName` is automatically expanded to `{'y(1)';'y(2)'}`. This expansion occurs because `G` is a two-output system.

Represent the components of the controller.

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

PI_L = tunablePID('PI_L','pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';

PI_V = tunablePID('PI_V','pi');
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';

sum1 = sumblk('e = r - y',2);
```
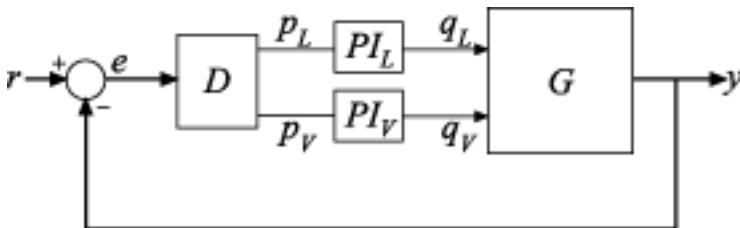
The control system includes several tunable control elements. `PI_L` and `PI_V` are tunable PI controllers. These elements represented by `tunablePID` models. The fixed control structure also includes a decoupling gain matrix D, represented by a tunable `tunableGain` model. When the control system is tuned, `D` ensures that each output of `G` tracks the corresponding reference signal `r` with minimal crosstalk.

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

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

Connect the controller components.

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

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

Tune the control system.

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

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

Final: Peak gain = 0.956, Iterations = 20
Achieved target gain value TargetGain=1.
```

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

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

Display the tuned controller parameters.

```
showTunable(C)

Decoupler =

  D =
            u1        u2
   y1     1.561   -0.9838
   y2    -1.128    0.9175

Name: Decoupler
Static gain.
----------------------------------
PI_L =

              1
  Kp + Ki * ---
              s
```

```
   with Kp = 2.17, Ki = 0.00986

Name: PI_L
Continuous-time PI controller in parallel form.
-----------------------------------
PI_V =

            1
  Kp + Ki * ---
            s

   with Kp = -2.68, Ki = -0.0376

Name: PI_V
Continuous-time PI controller in parallel form.
```
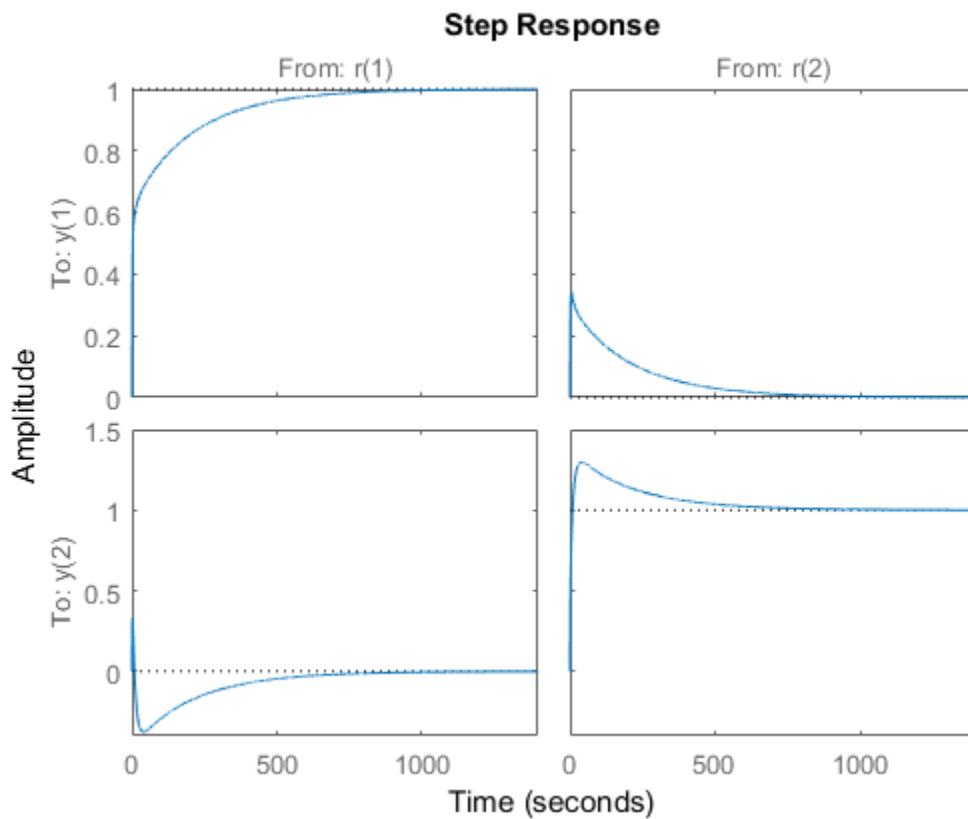
Check the time-domain response for the control system with the tuned coefficients. To produce a plot, construct a closed-loop model of the tuned control system. Plot the step response from reference to output.
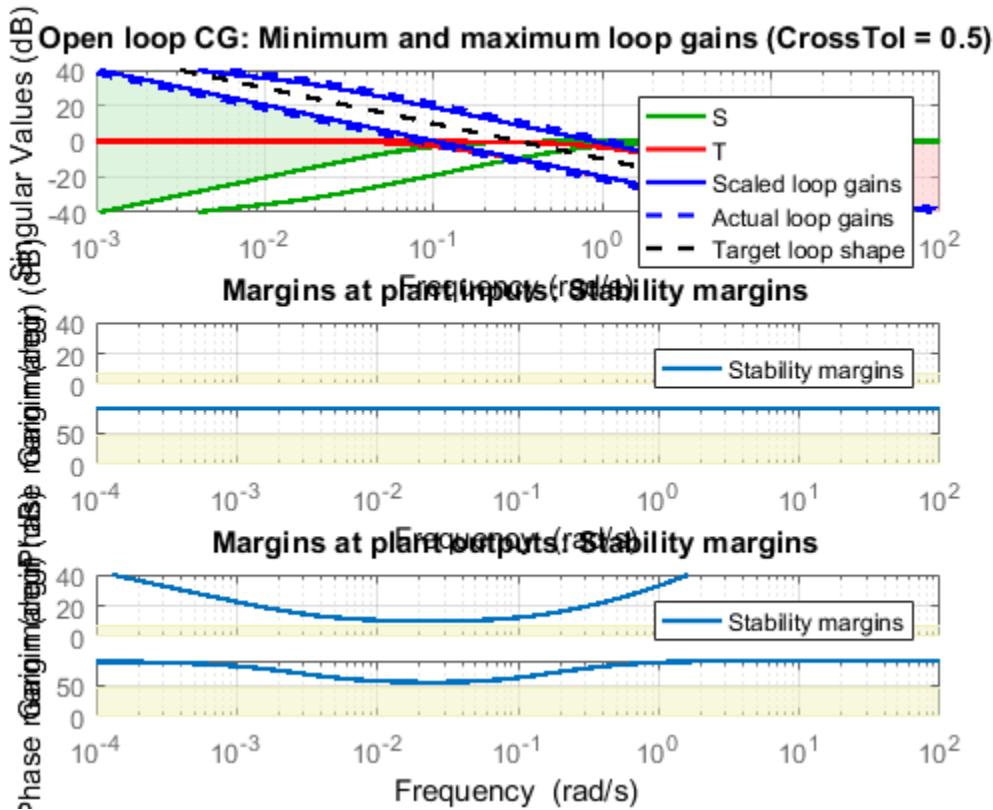
```
T = connect(G,C,'r','y');
step(T)
```

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

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

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

The first plot shows that the open-loop gain crossovers fall within the specified interval [0.1,1]. This plot also includes the maximum and tuned values of the sensitivity function $S = (I - GC)^{-1}$ and complementary sensitivity $T = I - S$. The second and third plots show that the MIMO stability margins of the tuned system (blue curve) do not exceed the upper limit (yellow curve).

Related Examples

- "Decoupling Controller for a Distillation Column" on page 14-17

More About

- "Structure of Control System for Tuning With looptune" on page 14-2

Tuning Feedback Loops with LOOPTUNE

This example shows the basic workflow of tuning feedback loops with the `looptune` command. `looptune` is similar to `systune` and meant to facilitate loop shaping design by automatically generating the tuning requirements.

Engine Speed Control

This example uses a simple engine speed control application as illustration. The control system consists of a single PID loop and the PID controller gains must be tuned to adequately respond to step changes in the desired speed. Specifically, we want the response to settle in less than 5 seconds with little or no overshoot.
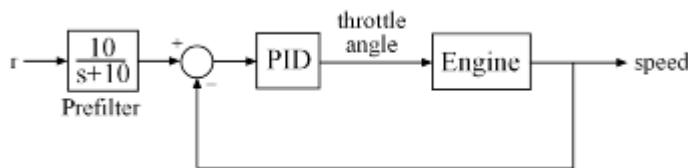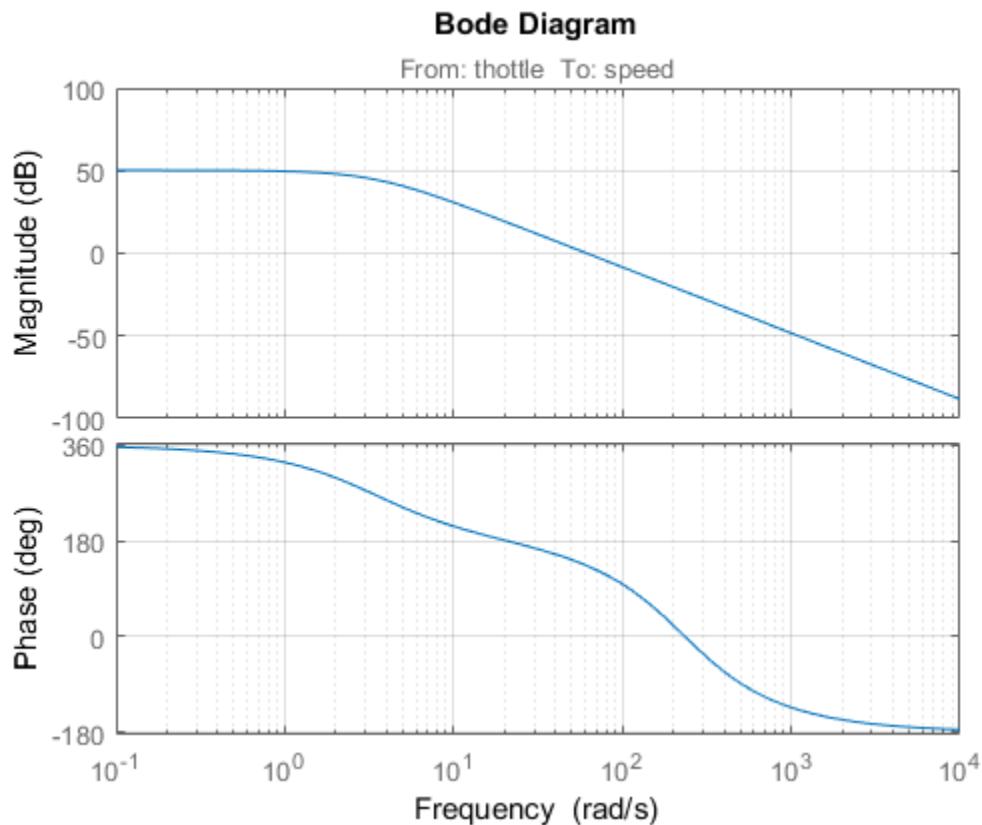
Figure 1: Engine Speed Control Loop

We use the following fourth-order model of the engine dynamics.

```
load rctExamples Engine
bode(Engine), grid
```

Bode Diagram

Specifying the Tunable Elements

We need to tune the four PID gains to achieve the desired performance. Use the `tunablePID` class to parameterize the PID controller.

```
PIDO = tunablePID('SpeedController','pid')


PIDO =

  Parametric continuous-time PID controller "SpeedController" with formula:

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

```
        s               Tf*s+1

  and tunable parameters Kp, Ki, Kd, Tf.

Type "pid(PIDO)" to see the current value and "get(PIDO)" to see all properties.
```

Building a Tunable Model of the Feedback Loop

`looptune` tunes the generic SISO or MIMO feedback loop of Figure 2. This feedback loop models the interaction between the plant and the controller. Note that this is a *positive* feedback interconnection.
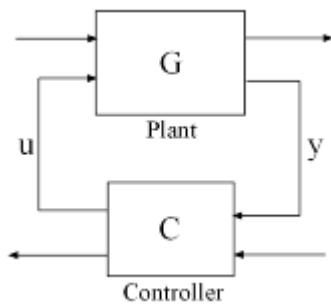
Figure 2: Generic Feedback Loop

For the speed control loop, the plant G is the engine model and the controller C consists of the PID and the prefilter F.
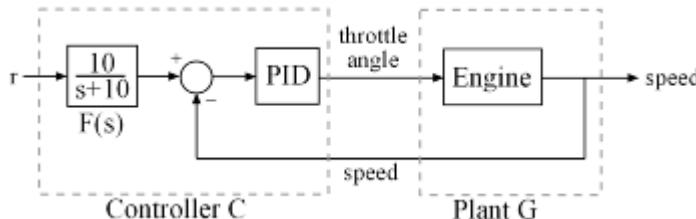
Figure 3: Feedback Loop for Engine Speed Control

To use `looptune`, create models for G and C in Figure 3. Assign names to the inputs and outputs of each model to specify the feedback paths between plant and controller. Note

that the controller C has two inputs: the speed reference "r" and the speed measurement "speed".

```
F = tf(10,[1 10]);   % prefilter

G = Engine;
G.InputName = 'throttle';
G.OutputName = 'speed';

CO = PIDO * [F , -1];
CO.InputName = {'r','speed'};
CO.OutputName = 'throttle';
```

Here `CO` is a generalized state-space model (`genss`) that depends on the tunable PID block `PIDO`.

Tuning the Controller Parameters

You can now use `looptune` to tune the PID gains subject to a simple control bandwidth requirement. To achieve the 5-second settling time, the gain crossover frequency of the open-loop response should be approximately 1 rad/s. Given this basic requirement, `looptune` automatically shapes the open-loop response to provide integral action, high-frequency roll-off, and adequate stability margins. Note that you could specify additional requirements to further constrain the design, see *"Decoupling Controller for a Distillation Column"* for an example.

```
wc = 1;  % target gain crossover frequency

[~,C,~,Info] = looptune(G,CO,wc);

Final: Peak gain = 0.92, Iterations = 3
Achieved target gain value TargetGain=1.
```

The final value is less than 1, indicating that the desired bandwidth was achieved with adequate roll-off and stability margins. `looptune` returns the tuned controller `C`. Use `getBlockValue` to retrieve the tuned value of the PID block.

```
PIDT = getBlockValue(C,'SpeedController')


PIDT =

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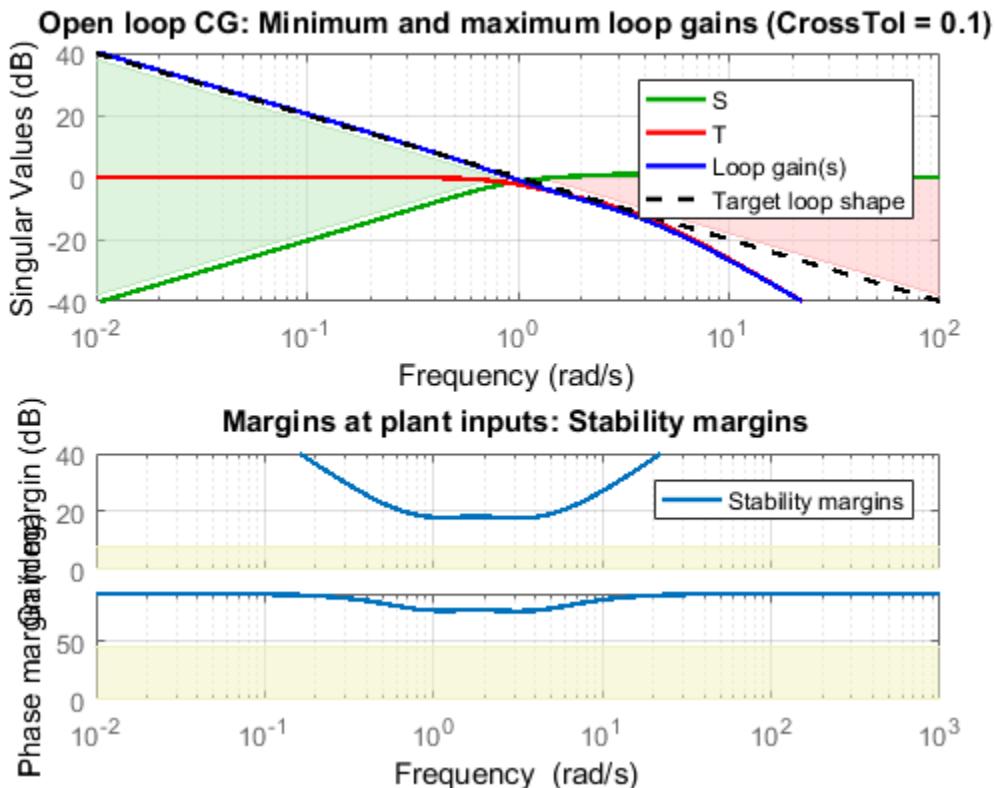
```
          s              Tf*s+1

  with Kp = 0.000484, Ki = 0.00324, Kd = 0.000835, Tf = 1

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

Validating Results
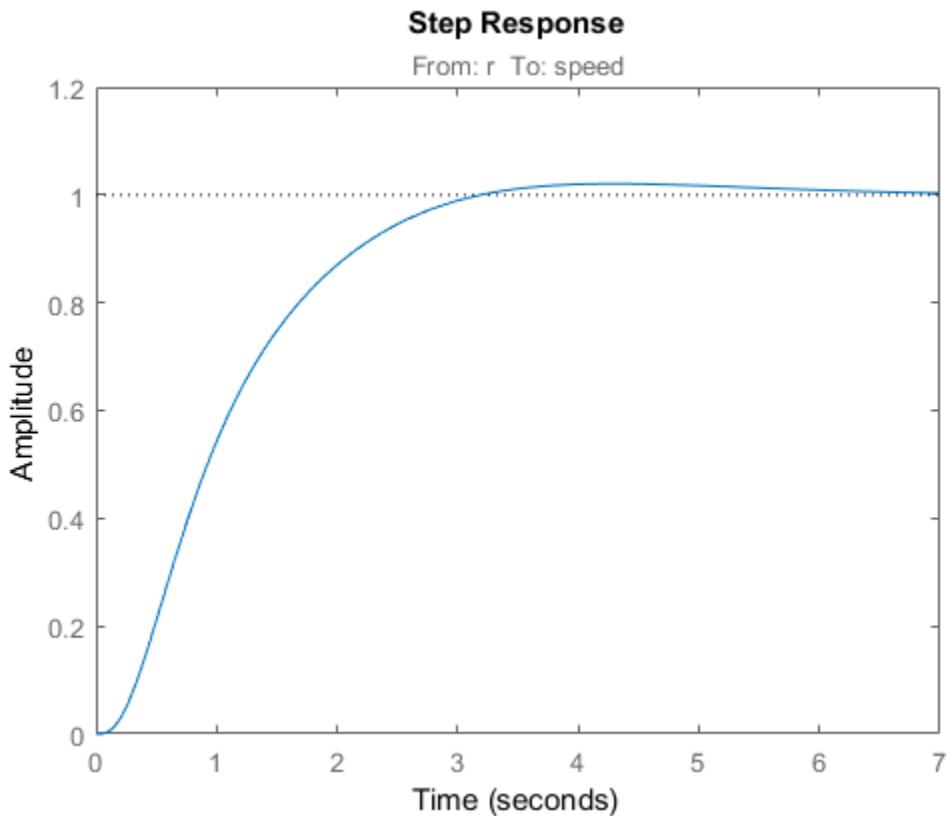
Use `loopview` to validate the design and visualize the loop shaping requirements implicitly enforced by `looptune`.

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

Next plot the closed-loop response to a step command in engine speed. The tuned response satisfies our requirements.

```
T = connect(G,C,'r','speed');  % closed-loop transfer from r to speed
clf, step(T)
```

Step Response

From: r To: speed

Decoupling Controller for a Distillation Column

This example shows how to use `looptune` to decouple the two main feedback loops in a distillation column.

Distillation Column Model

This example uses a simple model of the distillation column shown below.
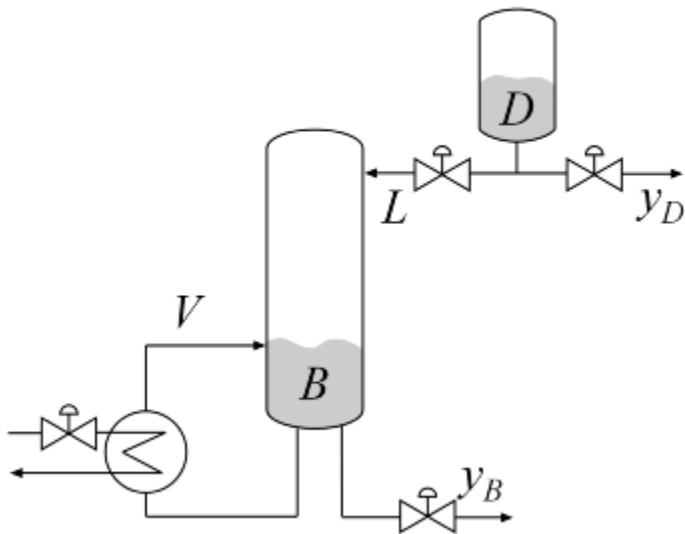
Figure 1: Distillation Column

In the so-called LV configuration, the controlled variables are the concentrations `yD` and `yB` of the chemicals D (tops) and B (bottoms), and the manipulated variables are the reflux `L` and boilup `V`. This process exhibits strong coupling and large variations in steady-state gain for some combinations of L and V. For more details, see Skogestad and Postlethwaite, *Multivariable Feedback Control*.

The plant is modeled as a first-order transfer function with inputs `L,V` and outputs `yD,yB`:

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

The unit of time is minutes (all plots are in minutes, not seconds).

```
s = tf('s');
G = [87.8 -86.4 ; 108.2 -109.6]/(75*s+1);
G.InputName = {'L','V'};
G.OutputName = {'yD','yB'};
```
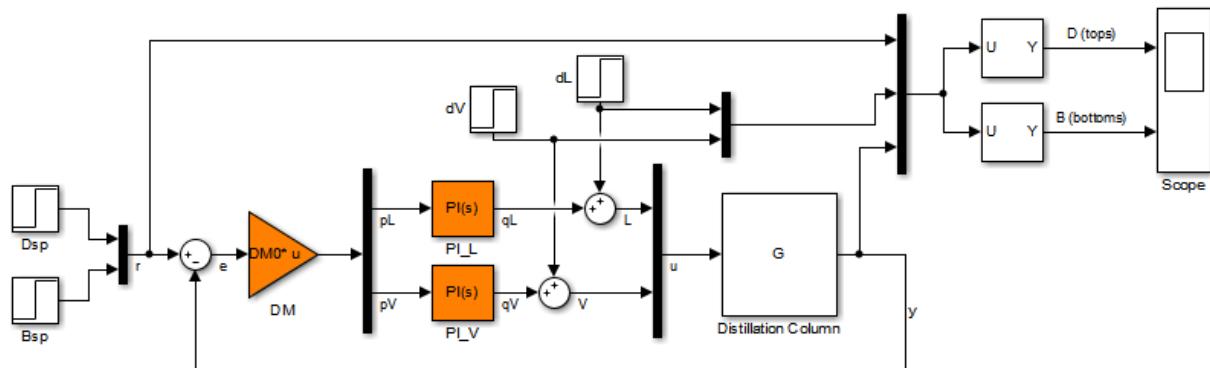
Control Architecture

The control objectives are as follows:

- Independent control of the tops and bottoms concentrations by ensuring that a change in the tops setpoint Dsp has little impact on the bottoms concentration B and vice versa
- Response time of about 4 minutes with less than 15% overshoot
- Fast rejection of input disturbances affecting the effective reflux L and boilup V

To achieve these objectives we use the control architecture shown below. This architecture consists of a static decoupling matrix DM in series with two PI controllers for the reflux L and boilup V.

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


Decoupling controller for a distillation column

Controller Tuning in Simulink with LOOPTUNE

The `looptune` command provides a quick way to tune MIMO feedback loops. When the control system is modeled in Simulink, you just specify the tuned blocks, the control and measurement signals, and the desired bandwidth, and `looptune` automatically sets up the problem and tunes the controller parameters. `looptune` shapes the open-loop response to provide integral action, roll-off, and adequate MIMO stability margins.

Use the `slTuner` interface to specify the tuned blocks, the controller I/Os, and signals of interest for closed-loop validation.

```
STO = slTuner('rct_distillation',{'PI_L','PI_V','DM'});

% Signals of interest
addPoint(STO,{'r','dL','dV','L','V','y'})
```

Set the control bandwidth by specifying the gain crossover frequency for the open-loop response. For a response time of 4 minutes, the crossover frequency should be approximately 2/4 = 0.5 rad/min.

```
wc = 0.5;
```

Use `TuningGoal` objects to specify the remaining control objectives. The response to a step command should have less than 15% overshoot. The response to a step disturbance at the plant input should be well damped, settle in less than 20 minutes, and not exceed 4 in amplitude.

```
OS = TuningGoal.Overshoot('r','y',15);

DR = TuningGoal.StepRejection({'dL','dV'},'y',4,20);
```

Next use `looptune` to tune the controller blocks `PI_L`, `PI_V`, and `DM` subject to the disturbance rejection requirement.
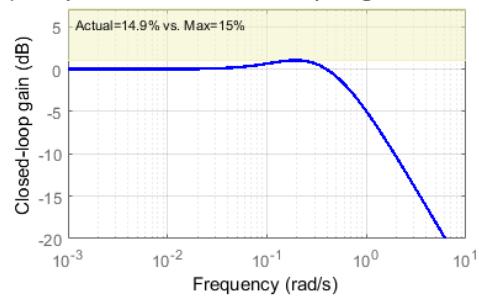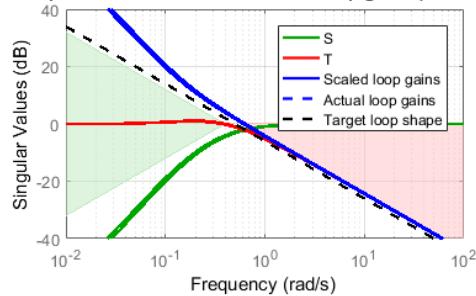
```
Controls = {'L','V'};
Measurements = 'y';
[ST,gam,Info] = looptune(STO,Controls,Measurements,wc,OS,DR);

Final: Peak gain = 0.997, Iterations = 48
Achieved target gain value TargetGain=1.
```
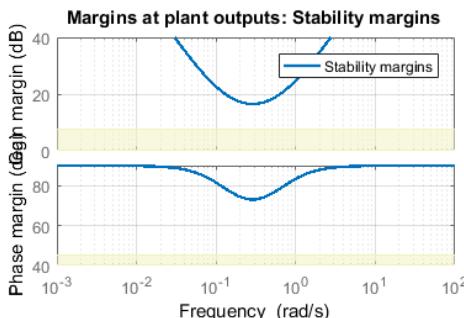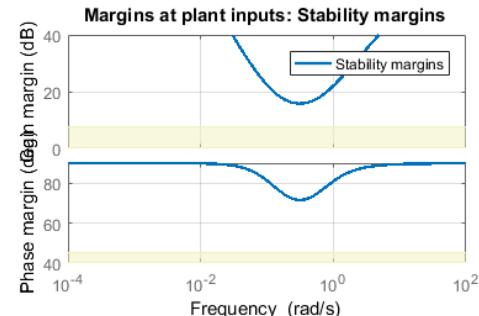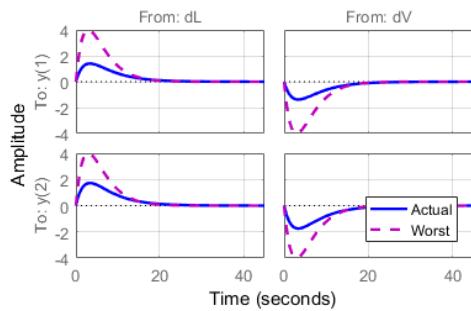
The final value is near 1 which indicates that all requirements were met. Use `loopview` to check the resulting design. The responses should stay outside the shaded areas.

```
figure('Position',[0,0,1000,1200])
loopview(ST,Info)
```
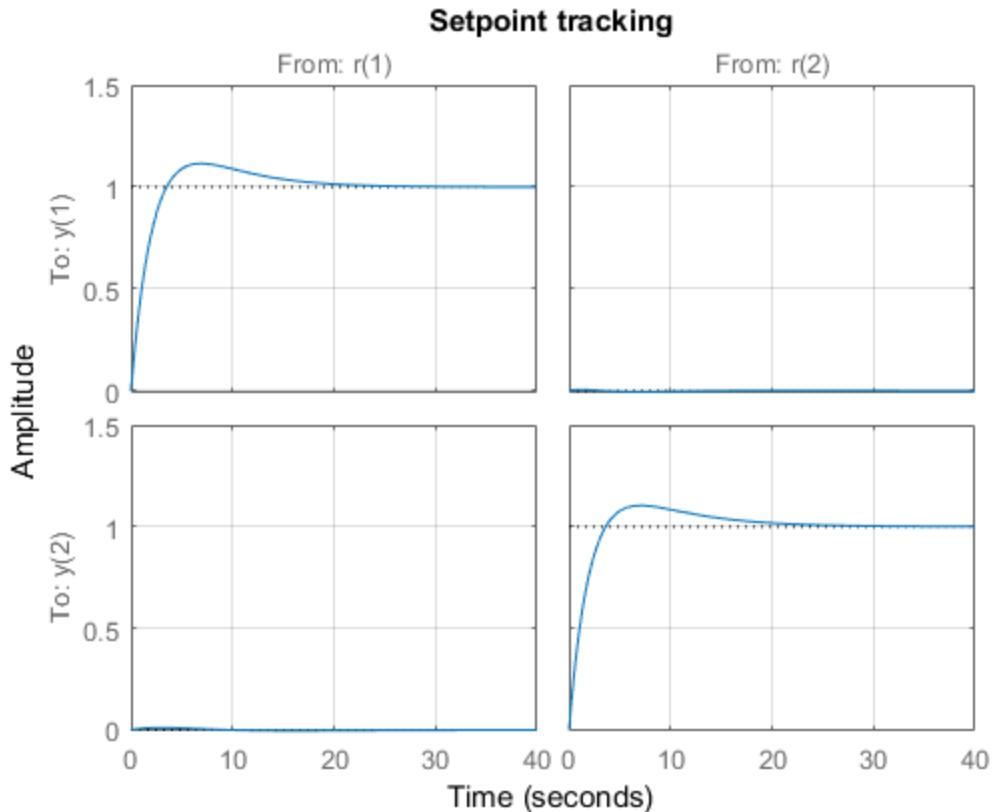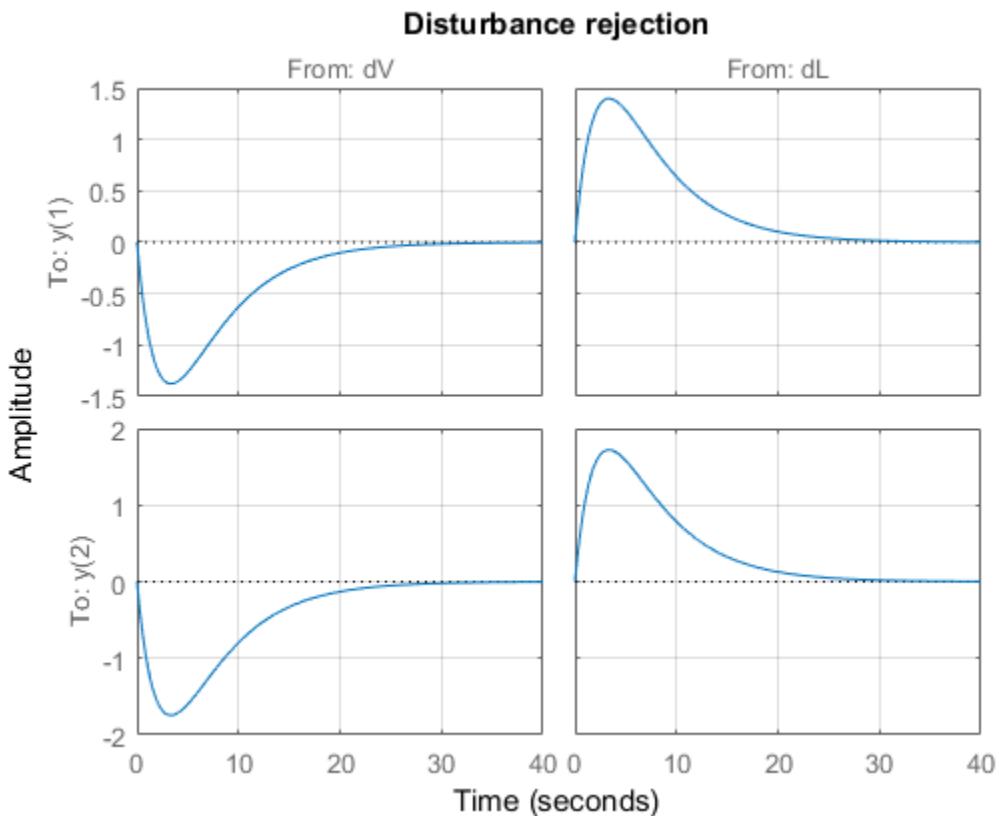
Use `getIOTransfer` to access and plot the closed-loop responses from reference and disturbance to the tops and bottoms concentrations. The tuned responses show a good compromise between tracking and disturbance rejection.

```
figure
Ttrack = getIOTransfer(ST,'r','y');
step(Ttrack,40), grid, title('Setpoint tracking')
```



```
Treject = getIOTransfer(ST,{'dV','dL'},'y');
step(Treject,40), grid, title('Disturbance rejection')
```
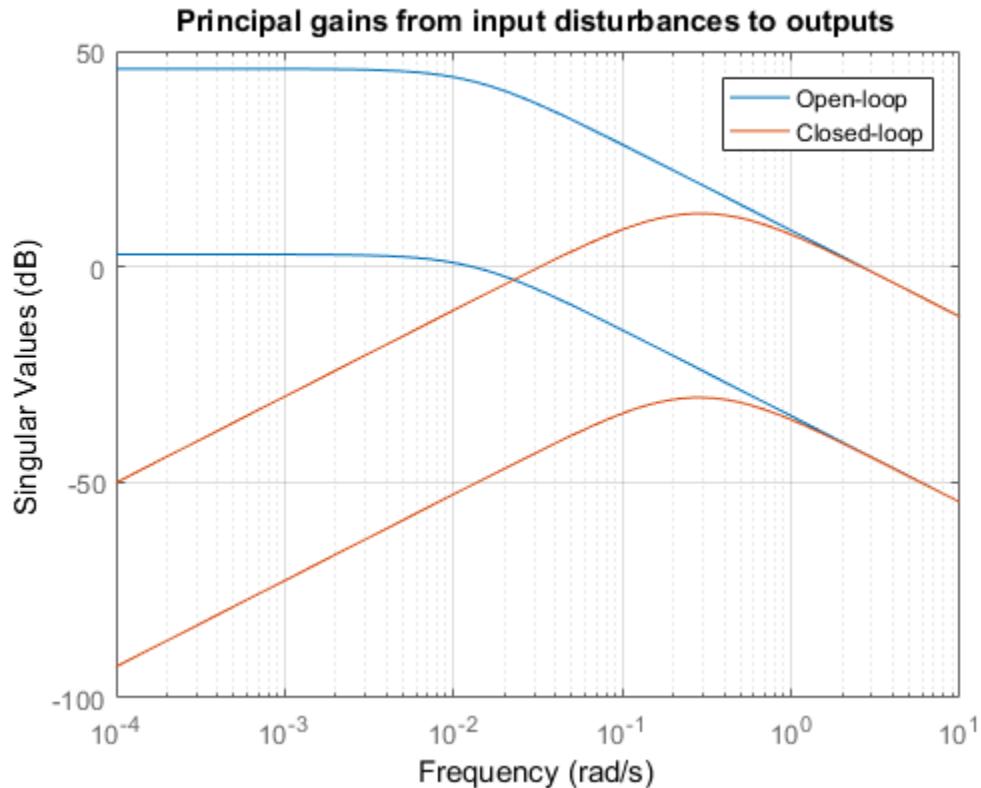
Disturbance rejection

Comparing the open- and closed-loop disturbance rejection characteristics in the frequency domain shows a clear improvement inside the control bandwidth.

```
clf, sigma(G,Treject), grid
title('Principal gains from input disturbances to outputs')
legend('Open-loop','Closed-loop')
```

Adding Constraints on the Tuned Variables

Inspection of the controller obtained above shows that the second PI controller has negative gains.

```
getBlockValue(ST,'PI_V')
```

```
ans =

            1
  Kp + Ki * ---
            s
```

```
   with Kp = -4.81, Ki = -0.651

Name: PI_V
Continuous-time PI controller in parallel form.
```

This is due to the negative signs in the second input channels of the plant G. In addition, the tunable elements are over-parameterized because multiplying DM by two and dividing the PI gains by two does not change the overall controller. To address these issues, fix the (1,1) entry of DM to 1 and the (2,2) entry to -1.
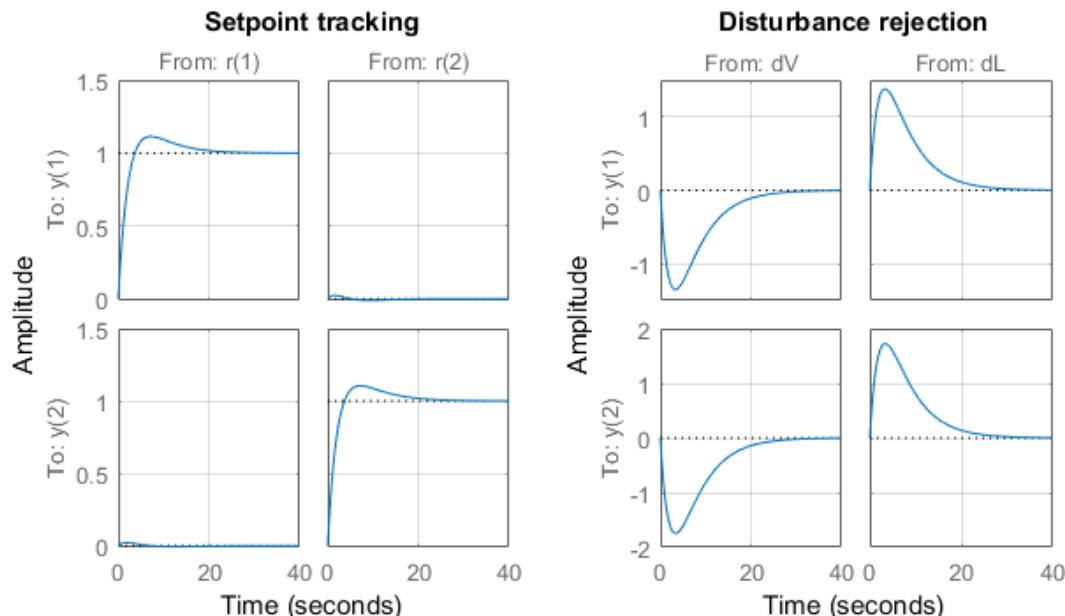
```
DM = getBlockParam(STO,'DM');
DM.Gain.Value = diag([1 -1]);
DM.Gain.Free = [false true;true false];
setBlockParam(STO,'DM',DM)
```

Re-tune the controller for the reduced set of tunable parameters.

```
[ST,gam,Info] = looptune(STO,Controls,Measurements,wc,OS,DR);

Final: Peak gain = 1, Iterations = 80
Achieved target gain value TargetGain=1.
```

The step responses look similar but the values of DM and the PI gains are more suitable for implementation.

```
figure('Position',[0,0,700,350])

subplot(121)
Ttrack = getIOTransfer(ST,'r','y');
step(Ttrack,40), grid, title('Setpoint tracking')

subplot(122)
Treject = getIOTransfer(ST,{'dV','dL'},'y');
step(Treject,40), grid, title('Disturbance rejection')
```

```
showTunable(ST)


Block 1: rct_distillation/PI_L =

            1
  Kp + Ki * ---
            s

  with Kp = 16.8, Ki = 2.23

Name: PI_L
Continuous-time PI controller in parallel form.

-----------------------------------

Block 2: rct_distillation/PI_V =

            1
  Kp + Ki * ---
            s
```

```
  with Kp = 13.2, Ki = 1.75

Name: PI_V
Continuous-time PI controller in parallel form.

---------------------------------

Block 3: rct_distillation/DM =

  D =
            u1        u2
    y1         1   -0.7708
    y2     1.256       -1

Name: DM
Static gain.
```

Equivalent Workflow in MATLAB

If you do not have a Simulink model of the control system, you can use LTI objects and Control Design blocks to create a MATLAB representation of the following block diagram.
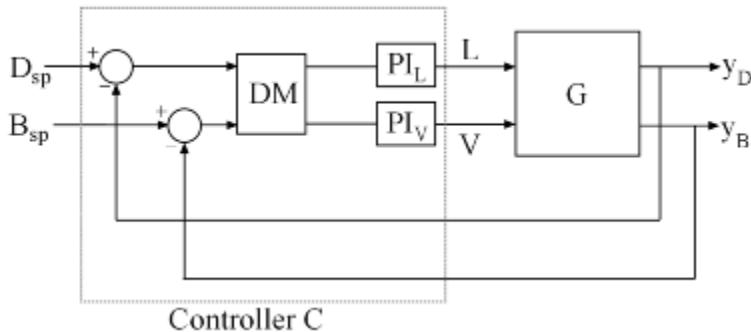
Figure 2: Block Diagram of Control System

First parameterize the tunable elements using Control Design blocks. Use the `tunableGain` object to parameterize DM and fix `DM(1,1)=1` and `DM(2,2)=-1`. This creates a 2x2 static gain with the off-diagonal entries as tunable parameters.

```
DM = tunableGain('Decoupler',diag([1 -1]));
```

```
DM.Gain.Free = [false true;true false];
```

Similarly, use the `tunablePID` object to parameterize the two PI controllers:

```
PI_L = tunablePID('PI_L','pi');
PI_V = tunablePID('PI_V','pi');
```

Next construct a model `CO` of the controller C in Figure 2.

```
CO = blkdiag(PI_L,PI_V) * DM * [eye(2) -eye(2)];

% Note: I/O names should be consistent with those of G
CO.InputName = {'Dsp','Bsp','yD','yB'};
CO.OutputName = {'L','V'};
```

Now tune the controller parameters with `looptune` as done previously.

```
% Crossover frequency
wc = 0.5;

% Overshoot and disturbance rejection requirements
OS = TuningGoal.Overshoot({'Dsp','Bsp'},{'yD','yB'},15);
DR = TuningGoal.StepRejection({'L','V'},{'yD','yB'},4,20);

% Tune controller gains
[~,C] = looptune(G,CO,wc,OS,DR);

Final: Peak gain = 0.997, Iterations = 76
Achieved target gain value TargetGain=1.
```
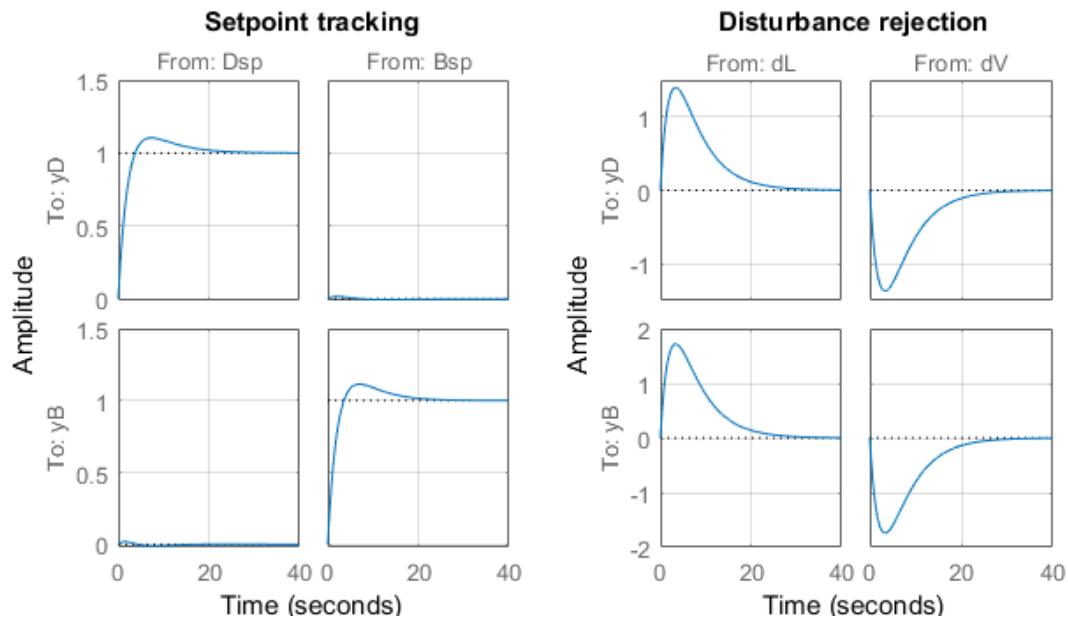
To validate the design, close the loop with the tuned compensator `C` and simulate the step responses for setpoint tracking and disturbance rejection.

```
Tcl = connect(G,C,{'Dsp','Bsp','L','V'},{'yD','yB'});

figure('Position',[0,0,700,350])

subplot(121)
Ttrack = Tcl(:,[1 2]);
step(Ttrack,40), grid, title('Setpoint tracking')

subplot(122)
Treject = Tcl(:,[3 4]);
Treject.InputName = {'dL','dV'};
step(Treject,40), grid, title('Disturbance rejection')
```

The results are similar to those obtained in Simulink.

Tuning of a Digital Motion Control System

This example shows how to use Control System Toolbox™ to tune a digital motion control system.

Motion Control System

The motion system under consideration is shown below.
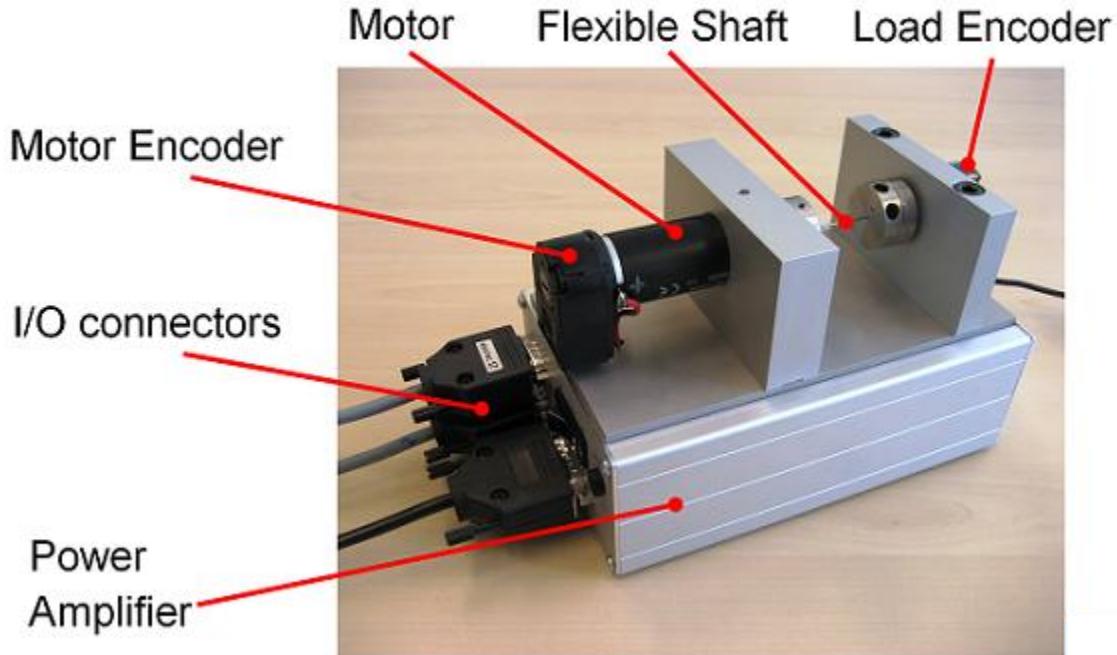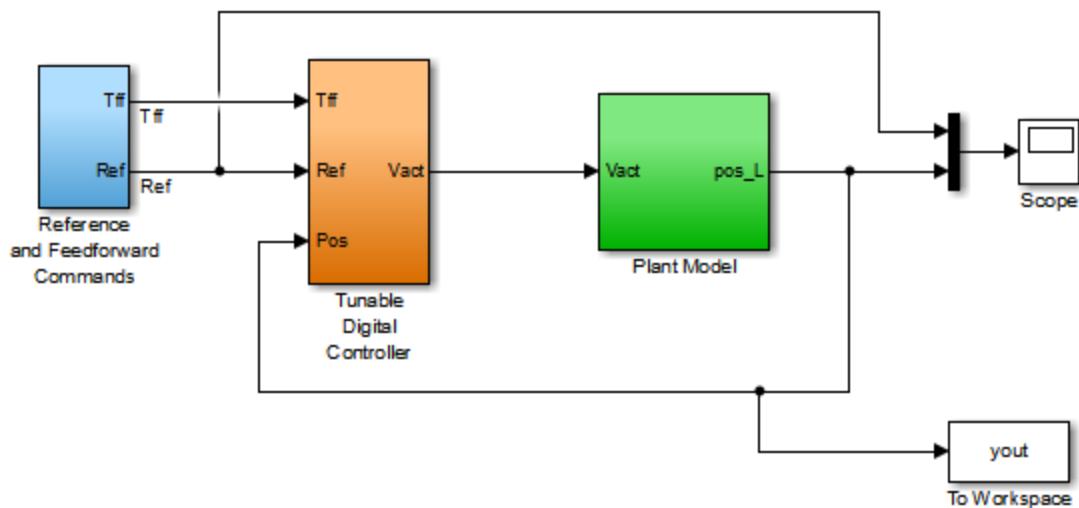
Figure 1: Digital motion control hardware

This device could be part of some production machine and is intended to move some load (a gripper, a tool, a nozzle, or anything else that you can imagine) from one angular position to another and back again. This task is part of the "production cycle" that has to be completed to create each product or batch of products.

The digital controller must be tuned to maximize the production speed of the machine without compromising accuracy and product quality. To do this, we first model the control system in Simulink using a 4th-order model of the inertia and flexible shaft:

```
open_system('rct_dmc')
```

Digital Motion Control System

The "Tunable Digital Controller" consists of a gain in series with a lead/lag controller.
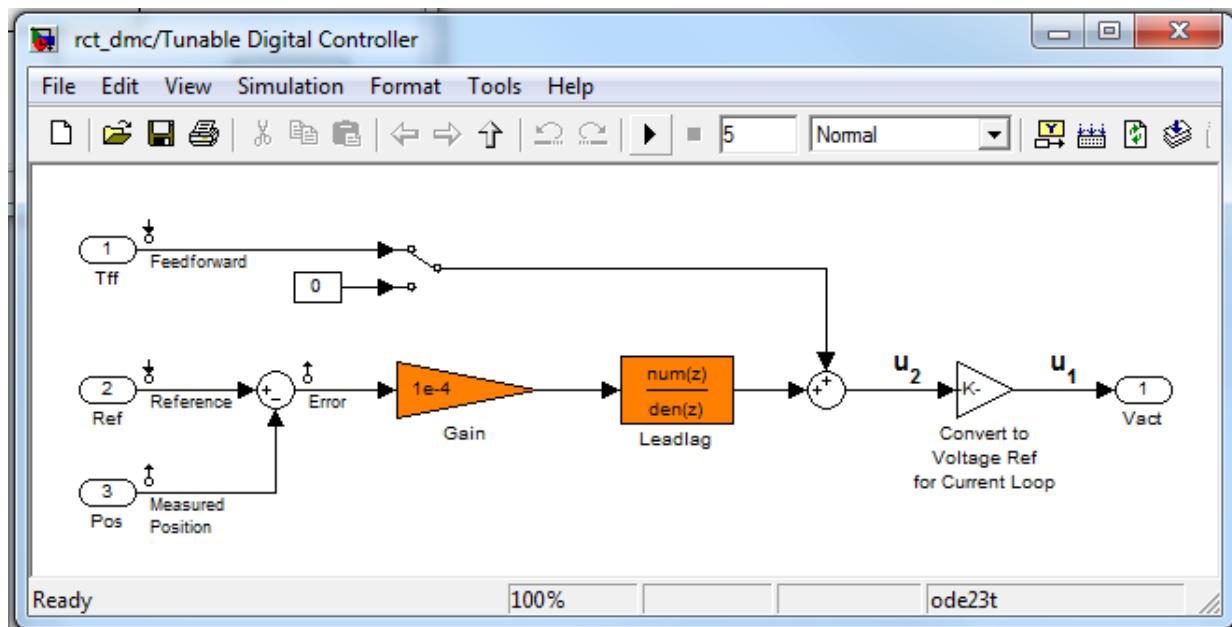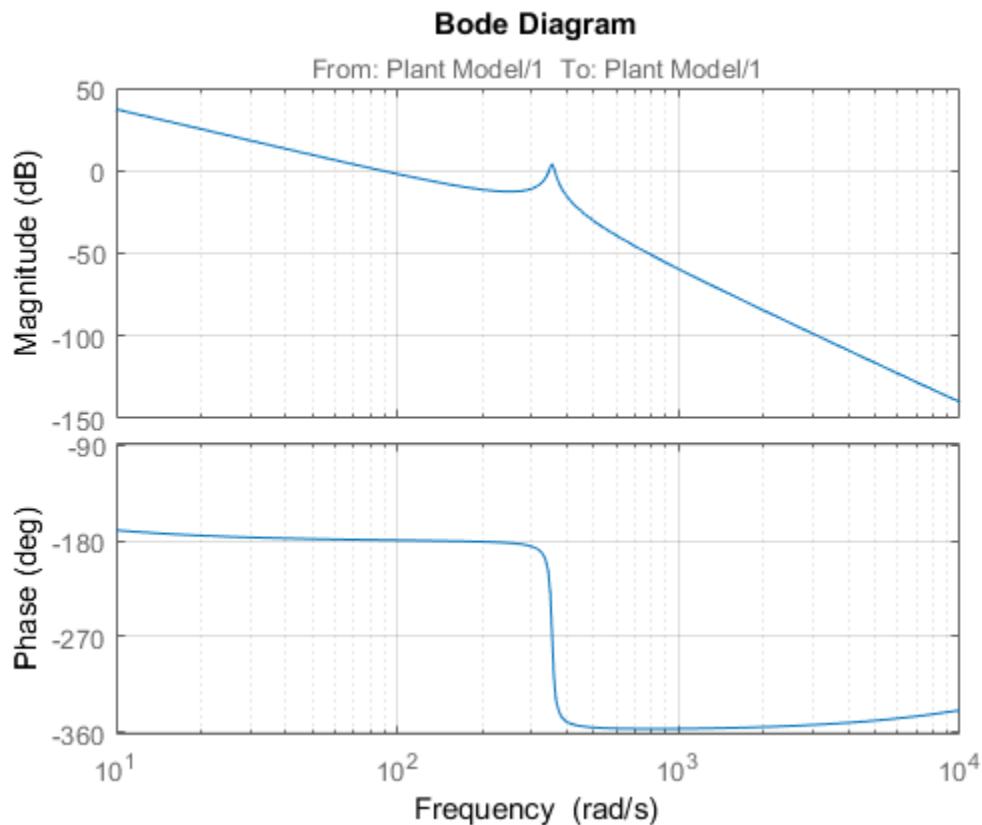
Figure 2: Digital controller

Tuning is complicated by the presence of a flexible mode near 350 rad/s in the plant:

```
G = linearize('rct_dmc','rct_dmc/Plant Model');
bode(G,{10,1e4}), grid
```

Bode Diagram

From: Plant Model/1 To: Plant Model/1

Compensator Tuning

We are seeking a 0.5 second response time to a step command in angular position with minimum overshoot. This corresponds to a target bandwidth of approximately 5 rad/s. The `looptune` command offers a convenient way to tune fixed-structure compensators like the one in this application. To use `looptune`, first instantiate the `slTuner` interface to automatically acquire the control structure from Simulink. Note that the signals of interest are already marked as Linear Analysis Points in the Simulink model.

```
ST0 = slTuner('rct_dmc',{'Gain','Leadlag'});
```

Next use `looptune` to tune the compensator parameters for the target gain crossover frequency of 5 rad/s:

```
Measurement = 'Measured Position';  % controller input
Control = 'Leadlag';                % controller output
ST1 = looptune(ST0,Control,Measurement,5);

Final: Peak gain = 0.975, Iterations = 21
Achieved target gain value TargetGain=1.
```

A final value below or near 1 indicates success. Inspect the tuned values of the gain and lead/lag filter:

```
showTunable(ST1)


Block 1: rct_dmc/Tunable Digital Controller/Gain =

  D =
              u1
   y1  2.753e-06

Name: Gain
Static gain.

----------------------------------

Block 2: rct_dmc/Tunable Digital Controller/Leadlag =

  30.54 s + 59.01
  --------------
     s + 18.94

Name: Leadlag
Continuous-time transfer function.
```
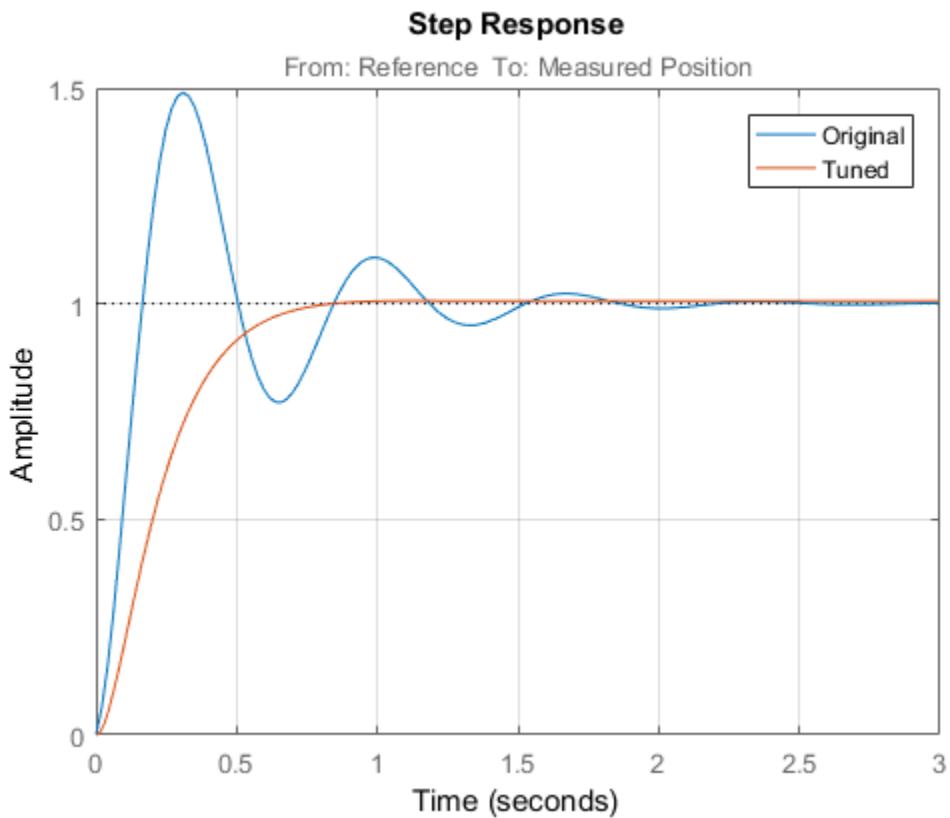
Design Validation

To validate the design, use the `slTuner` interface to quickly access the closed-loop transfer functions of interest and compare the responses before and after tuning.

```
T0 = getIOTransfer(ST0,'Reference','Measured Position');
T1 = getIOTransfer(ST1,'Reference','Measured Position');
step(T0,T1), grid
legend('Original','Tuned')
```

Step Response

From: Reference To: Measured Position

The tuned response has significantly less overshoot and satisfies the response time requirement. However these simulations are obtained using a continuous-time lead/lag compensator (`looptune` operates in continuous time) so we need to further validate the design in Simulink using a digital implementation of the lead/lag compensator. Use `writeBlockValue` to apply the tuned values to the Simulink model and automatically discretize the lead/lag compensator to the rate specified in Simulink.

```
writeBlockValue(ST1)
```
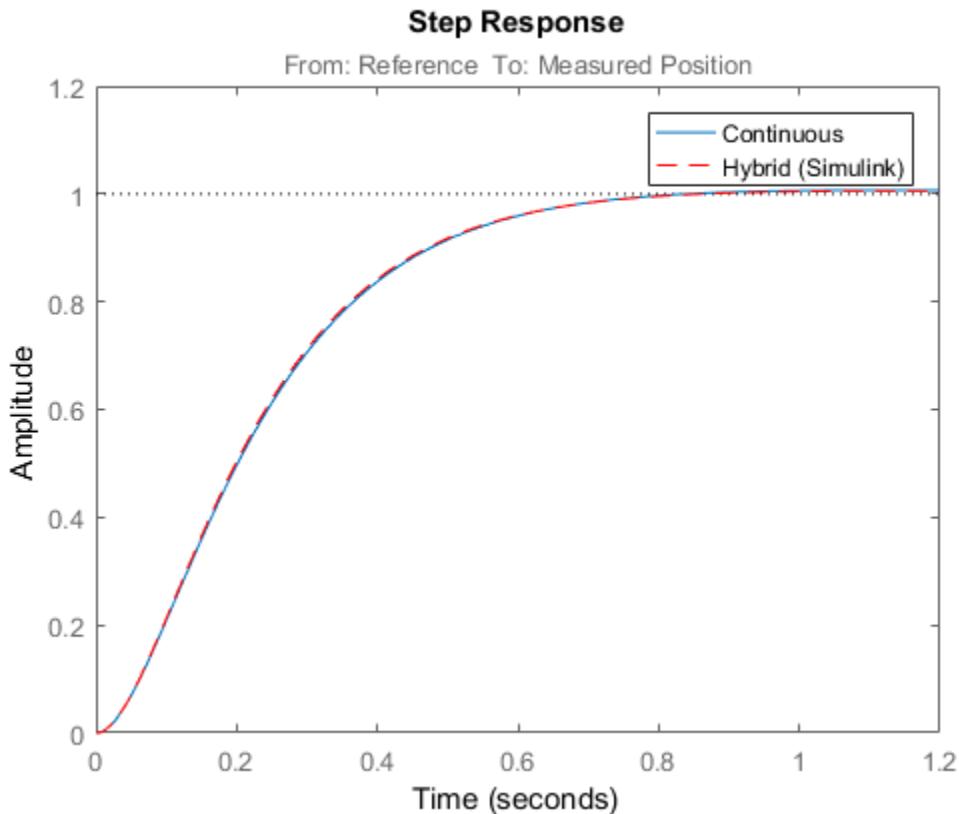
You can now simulate the response of the continuous-time plant with the digital controller:

```
sim('rct_dmc');   % angular position logged in "yout" variable
t = yout.time;
```

```
y = yout.signals.values;
step(T1), hold, plot(t,y,'r--')
legend('Continuous','Hybrid (Simulink)')
```

Current plot held

Step Response

From: Reference To: Measured Position

The simulations closely match and the coefficients of the digital lead/lag can be read from the "Leadlag" block in Simulink.

Tuning an Additional Notch Filter

Next try to increase the control bandwidth from 5 to 50 rad/s. Because of the plant resonance near 350 rad/s, the lead/lag compensator is no longer sufficient to get adequate

stability margins and small overshoot. One remedy is to add a notch filter as shown in Figure 3.
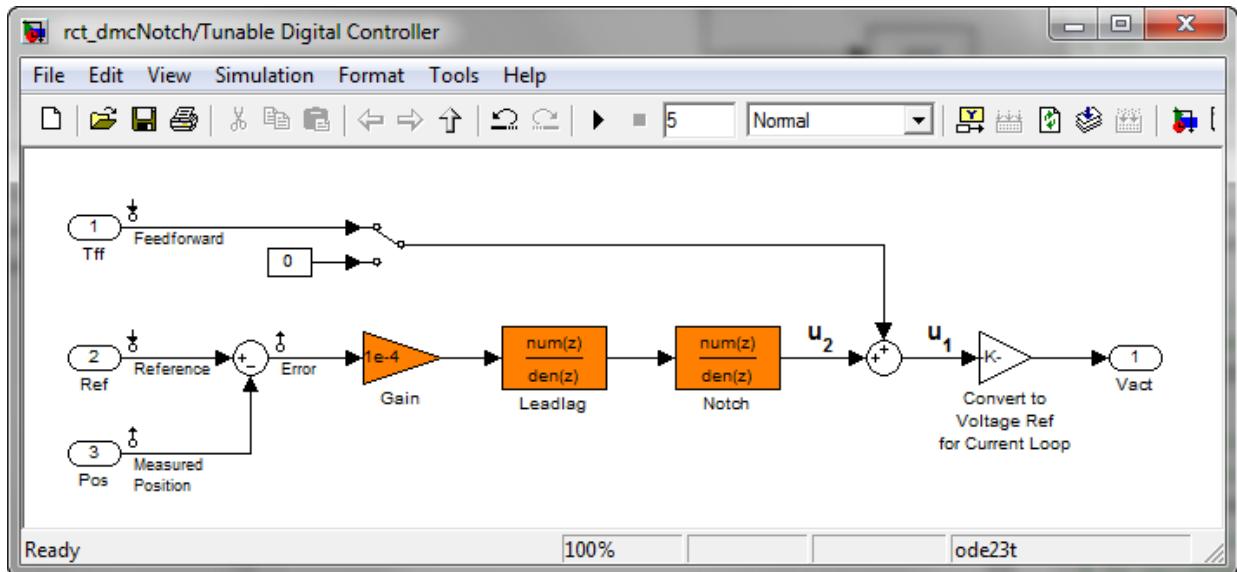
Figure 3: Digital Controller with Notch Filter

To tune this modified control architecture, create an `slTuner` instance with the three tunable blocks.

```
ST0 = slTuner('rct_dmcNotch',{'Gain','Leadlag','Notch'});
```

By default the "Notch" block is parameterized as any second-order transfer function. To retain the notch structure

$$N(s) = \frac{s^2 + 2\zeta_1 \omega_n s + \omega_n^2}{s^2 + 2\zeta_2 \omega_n s + \omega_n^2},$$

specify the coefficients $\omega_n, \zeta_1, \zeta_2$ as real parameters and create a parametric model N of the transfer function shown above:

```
wn = realp('wn',300);
```

```
zeta1 = realp('zeta1',1);
zeta2 = realp('zeta2',1);
zeta1.Minimum = 0;   zeta1.Maximum = 1;   % 0 <= zeta1 <= 1
zeta2.Minimum = 0;   zeta2.Maximum = 1;   % 0 <= zeta2 <= 1
N = tf([1 2*zeta1*wn wn^2],[1 2*zeta2*wn wn^2]);   % tunable notch filter
```

Then associate this parametric notch model with the "Notch" block in the Simulink model. Because the control system is tuned in the continuous time, you can use a continuous-time parameterization of the notch filter even though the "Notch" block itself is discrete.

```
setBlockParam(STO,'Notch',N);
```

Next use `looptune` to jointly tune the "Gain", "Leadlag", and "Notch" blocks with a 50 rad/s target crossover frequency. To eliminate residual oscillations from the plant resonance, specify a target loop shape with a -40 dB/decade roll-off past 50 rad/s.

```
% Specify target loop shape with a few frequency points
Freqs = [5 50 500];
Gains = [10 1 0.01];
TLS = TuningGoal.LoopShape('Notch',frd(Gains,Freqs));

Measurement = 'Measured Position';   % controller input
Control = 'Notch';                    % controller output
ST2 = looptune(STO,Control,Measurement,TLS);

Final: Peak gain = 1.05, Iterations = 65
```
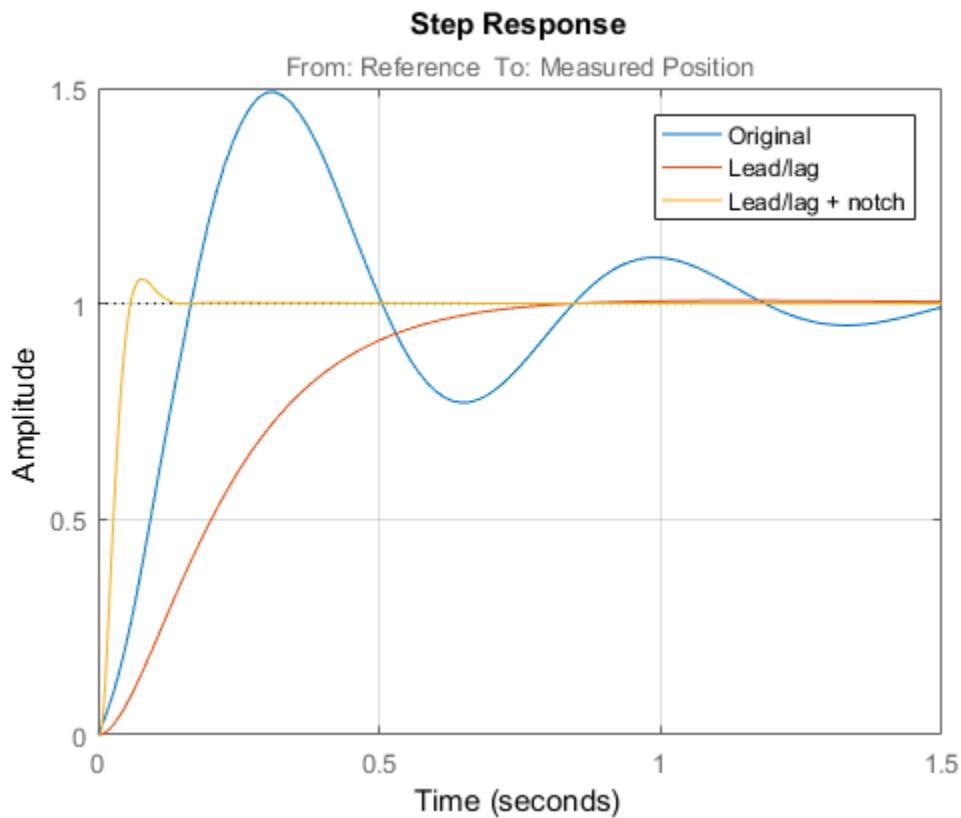
The final gain is close to 1, indicating that all requirements are met. Compare the closed-loop step response with the previous designs.

```
T2 = getIOTransfer(ST2,'Reference','Measured Position');
clf
step(T0,T1,T2,1.5), grid
legend('Original','Lead/lag','Lead/lag + notch')
```
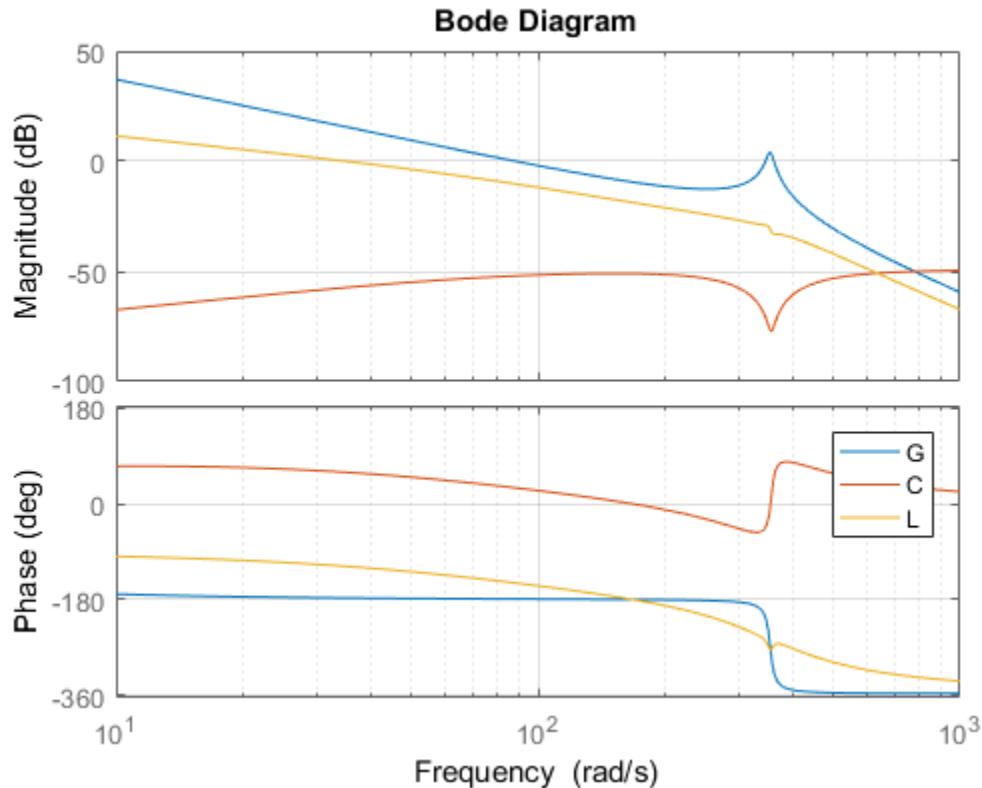
Step Response

To verify that the notch filter performs as expected, evaluate the total compensator `C` and the open-loop response `L` and compare the Bode responses of `G`, `C`, `L`:

```
% Get tuned block values (in the order blocks are listed in ST2.TunedBlocks)
[g,LL,N] = getBlockValue(ST2,'Gain','Leadlag','Notch');
C = N * LL * g;

L = getLoopTransfer(ST2,'Notch',-1);

bode(G,C,L,{1e1,1e3}), grid
legend('G','C','L')
```

This Bode plot confirms that the plant resonance has been correctly "notched out."
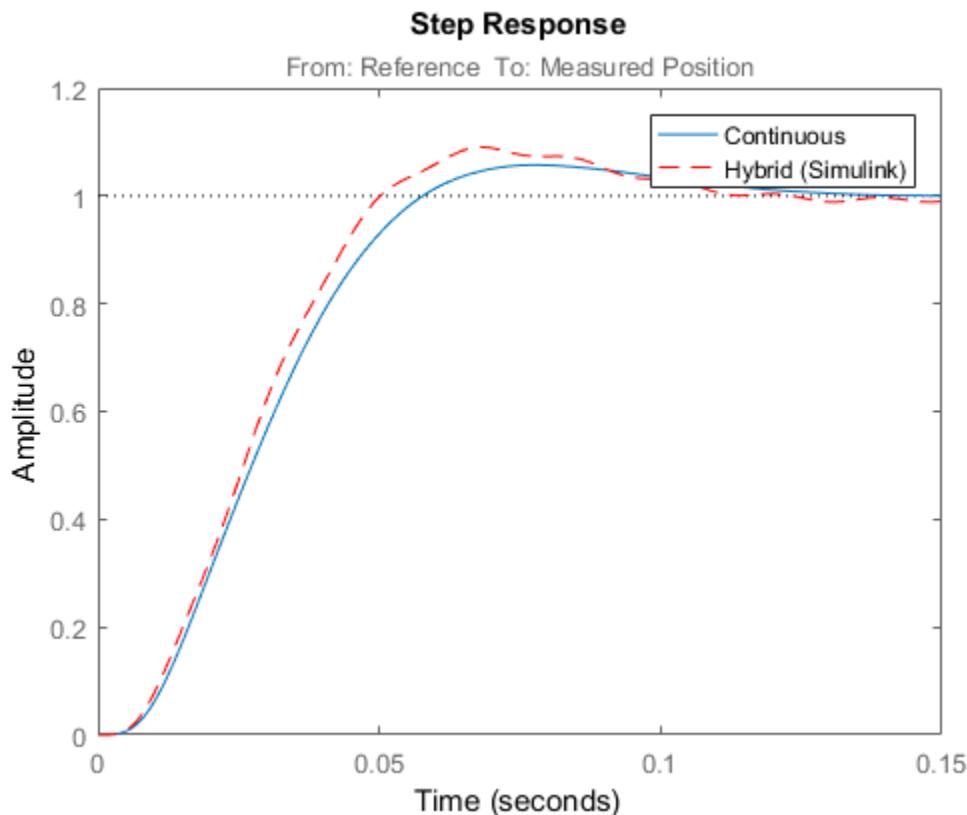
Discretizing the Notch Filter

Again use `writeBlockValue` to discretize the tuned lead/lag and notch filters and write their values back to Simulink. Compare the MATLAB and Simulink responses:

```
writeBlockValue(ST2)

sim('rct_dmcNotch');
t = yout.time;
y = yout.signals.values;
step(T2), hold, plot(t,y,'r--')
legend('Continuous','Hybrid (Simulink)')
```

Current plot held

Step Response
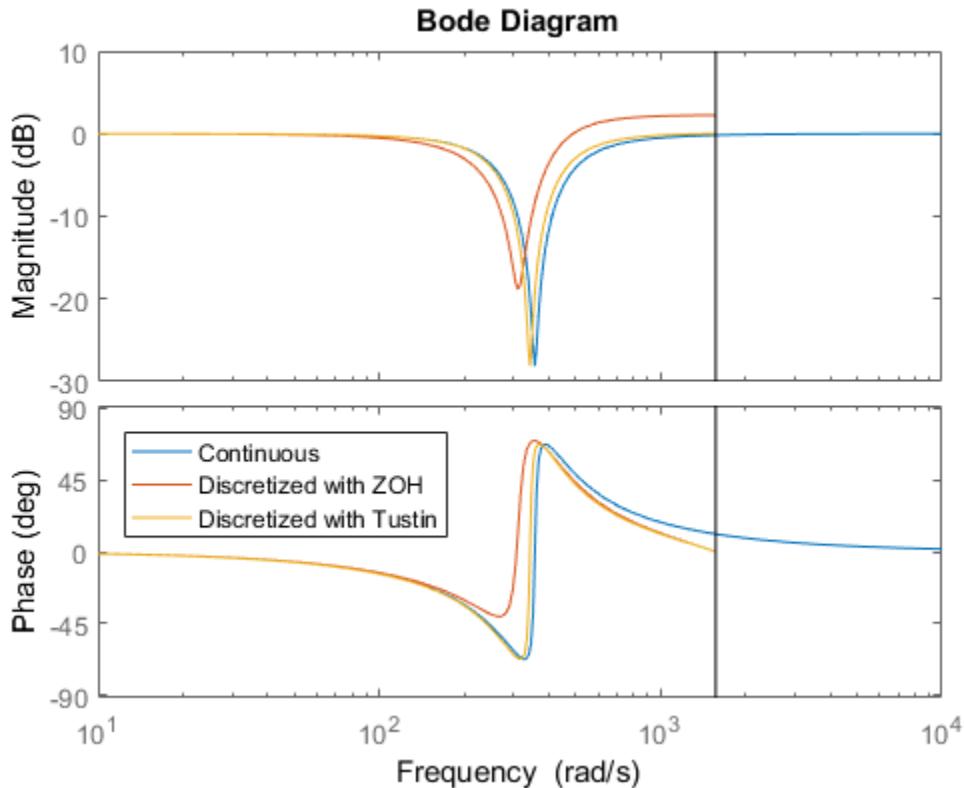
The Simulink response exhibits small residual oscillations. The notch filter discretization is the likely culprit because the notch frequency is close to the Nyquist frequency `pi/0.002=1570` rad/s. By default the notch is discretized using the ZOH method. Compare this with the Tustin method prewarped at the notch frequency:

```
wn = damp(N);   % natural frequency of the notch filter
Ts = 0.002;     % sample time of discrete notch filter

Nd1 = c2d(N,Ts,'zoh');
Nd2 = c2d(N,Ts,'tustin',c2dOptions('PrewarpFrequency',wn(1)));
```

```
clf, bode(N,Nd1,Nd2)
legend('Continuous','Discretized with ZOH','Discretized with Tustin',...
    'Location','NorthWest')
```
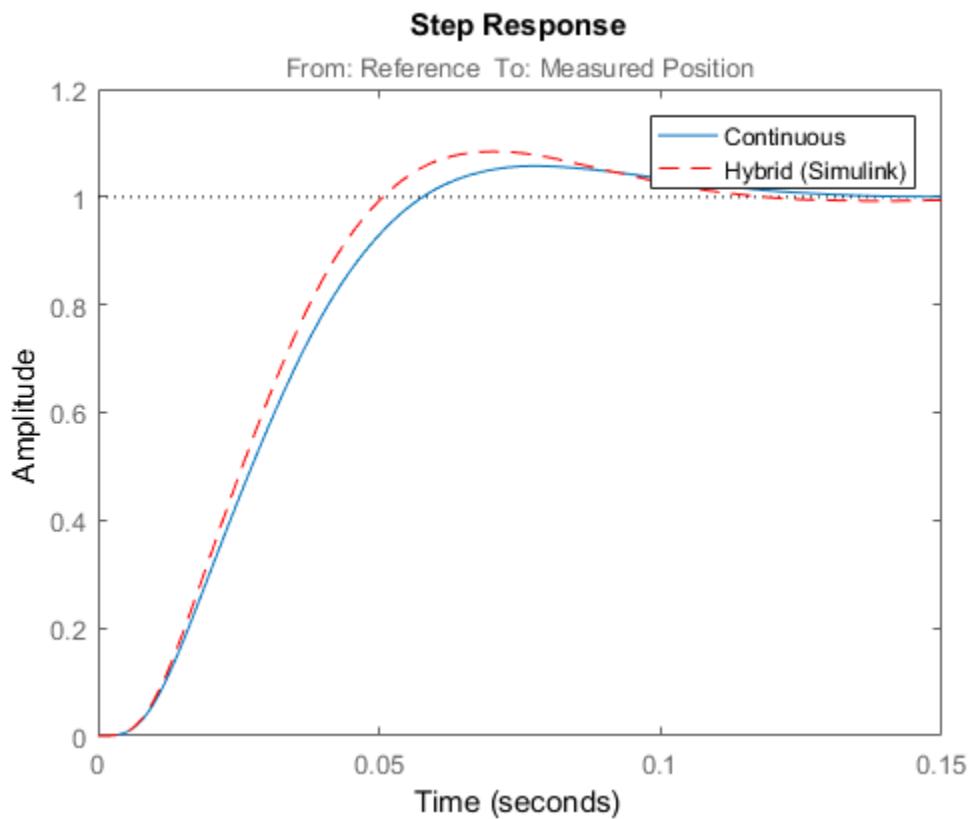

The ZOH method has significant distortion and prewarped Tustin should be used instead. To do this, specify the desired rate conversion method for the notch filter block:

```
setBlockRateConversion(ST2,'Notch','tustin',wn(1))
```

```
writeBlockValue(ST2)
```

writeBlockValue now uses Tustin prewarped at the notch frequency to discretize the notch filter and write it back to Simulink. Verify that this gets rid of the oscillations.

```
sim('rct_dmcNotch');
t = yout.time;
y = yout.signals.values;
step(T2), hold, plot(t,y,'r--')
legend('Continuous','Hybrid (Simulink)')
```

```
Current plot held
```


Direct Discrete-Time Tuning

Alternatively, you can tune the controller directly in discrete time to avoid discretization issues with the notch filter. To do this, specify that the Simulink model should be linearized and tuned at the controller sample time of 0.002 seconds:

```
ST0 = slTuner('rct_dmcNotch',{'Gain','Leadlag','Notch'});
ST0.Ts = 0.002;
```

To prevent high-gain control and saturations, add a requirement that limits the gain from reference to control signal (output of Notch block).
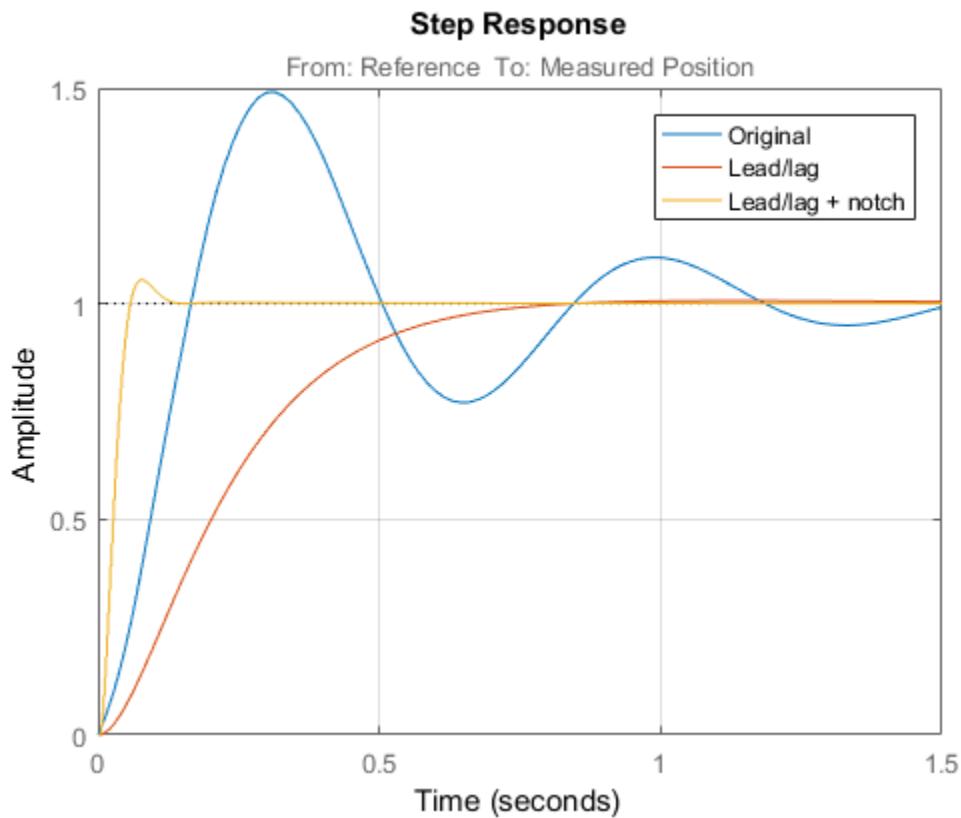
```
GL = TuningGoal.Gain('Reference','Notch',0.01);
```

Now retune the controller at the specified sampling rate and verify the tuned open- and closed-loop responses.
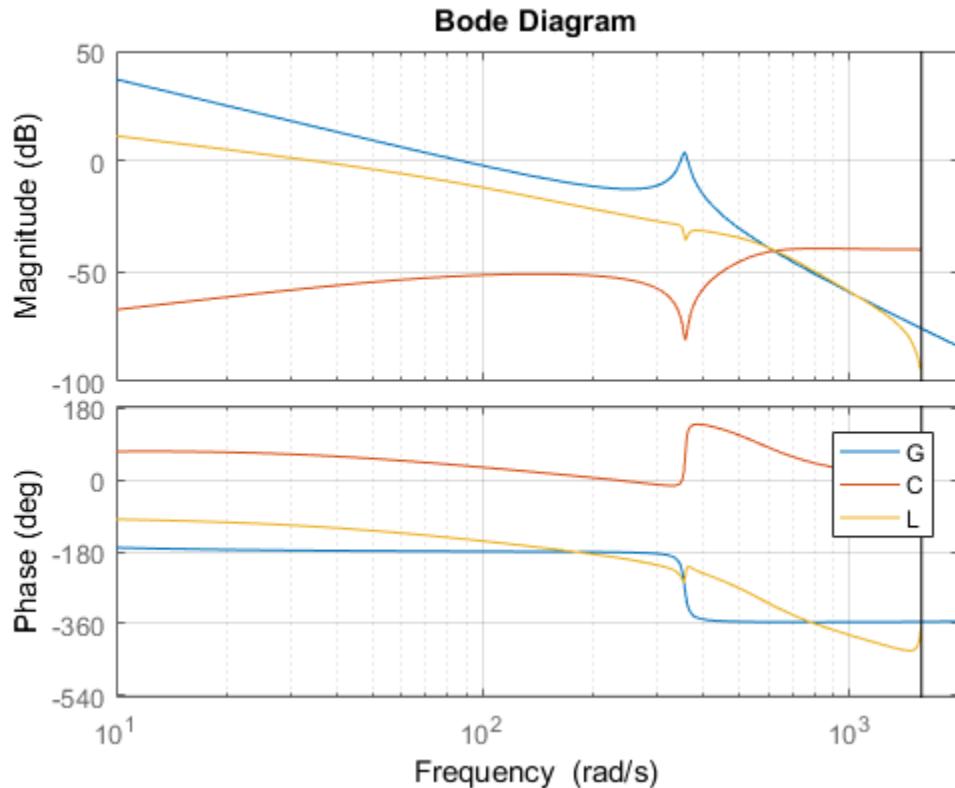
```
ST2 = looptune(ST0,Control,Measurement,TLS,GL);

% Closed-loop responses
T2 = getIOTransfer(ST2,'Reference','Measured Position');
clf
step(T0,T1,T2,1.5), grid
legend('Original','Lead/lag','Lead/lag + notch')

Final: Peak gain = 1.04, Iterations = 41
```

Step Response

From: Reference To: Measured Position


```
% Open-loop responses
[g,LL,N] = getBlockValue(ST2,'Gain','Leadlag','Notch');
C = N * LL * g;
L = getLoopTransfer(ST2,'Notch',-1);
bode(G,C,L,{1e1,2e3}), grid
legend('G','C','L')
```
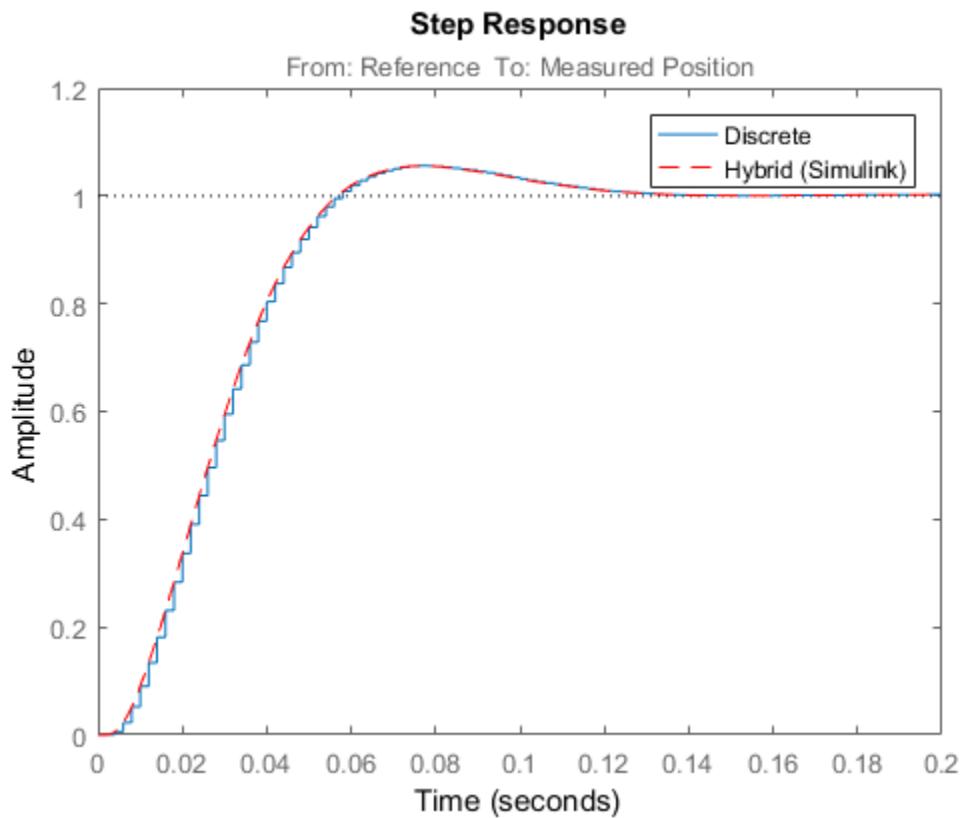
The results are similar to those obtained when tuning the controller in continuous time. Now validate the digital controller against the continuous-time plant in Simulink.

```
writeBlockValue(ST2)

sim('rct_dmcNotch');
t = yout.time;
y = yout.signals.values;
step(T2), hold, plot(t,y,'r--')
legend('Discrete','Hybrid (Simulink)')

Current plot held
```

Step Response

From: Reference To: Measured Position

This time, the hybrid response closely matches its discrete-time approximation and no further adjustment of the notch filter is required.

15

Gain-Scheduled Controllers

Gain-Scheduled Control Systems

A *gain-scheduled controller* is a controller whose gains are automatically adjusted as a function of time, operating condition, or plant parameters. Gain scheduling is a common strategy for controlling systems whose dynamics change with such variables. Such systems include linear parameter-varying (LPV) systems and large classes of nonlinear systems.

Gain scheduling is most suitable when the scheduling variables are external parameters that vary slowly compared to the control bandwidth, such as ambient temperature of a reaction or speed of a cruising aircraft. Gain scheduling is most challenging when the scheduling variables depend on fast-varying states of the system. Because local linear performance near operating points is no guarantee of global performance in nonlinear systems, extensive simulation-based validation is usually required. See [1] for an overview of gain scheduling and its challenges.

Typically, gain-scheduled controllers are fixed single-loop or multiloop control structures that use lookup tables to specify gain values as a function of the scheduling variables. For tuning purposes, it is convenient to replace lookup tables with parametric gain surfaces. A *parametric gain surface* is a basis-function expansion whose coefficients are tunable. For example, you can model a time-varying gain $k(t)$ as a cubic polynomial in t:

$$k(t) = k_0 + k_1 t + k_2 t^2 + k_3 t^3.$$

Here, $k_0,...,k_3$ are tunable coefficients. For applications where gains vary smoothly with the scheduling variables, this approach lets you tune a few coefficients rather than many individual lookup-table entries, drastically reducing the number of parameters. This approach also provides explicit formulas for the gains, and ensures smooth transitions between operating points. When you parameterize scheduled gains in this way, you can use `systune` to tune the gain surface coefficients automatically to meet your control objectives at a representative set of operating conditions. You can then use the resulting coefficients to interpolate gain values across the entire parameter range.

References

[1] Rugh, W.J., and J.S. Shamma, "Research on Gain Scheduling", *Automatica*, 36 (2000), pp. 1401-1425.

Related Examples

- "Tune Gain-Scheduled Controllers" on page 15-4

More About

Tune Gain-Scheduled Controllers

You can use `systune` to tune gain-scheduled controllers modeled with parametric gain surfaces. The general tuning workflow is as follows:

1 Select a set of design points that adequately covers the operating range of the variables on which the gain depends. The set can be a regular grid of values or a scattered set. Typically, you start with a few design points. If the performance that your tuned system achieves at the design points is not maintained between design points, add more design points and retune.

2 Build a collection of linear models describing the linearized plant dynamics at the selected design points. For example, you can:

   • Linearize a Simulink model at each operating condition represented in the grid of design points (requires Simulink Control Design). See "Set Up Simulink Models for Gain Scheduling" on page 15-17.

   • Sample an LPV model of the plant at the design points.

3 Model the variable gains with tunable gain surfaces. A gain surface is a parameterization of the gain in terms of the scheduling variables. For tuning with `systune`, you model the gain surface as a basis-function expansion whose coefficients are tunable. If σ is the vector of scheduling variables, such expansion is of the form:

$$K(\sigma) = K_0 + K_1 F_1\left(n(\sigma)\right) + \ldots + K_M F_M\left(n(\sigma)\right).$$

$n(\sigma)$ is a normalization function that brings the scheduling variables into the range [–1,1]. `systune` tunes the coefficients $K_0,\ldots,K_M$ over all the design points.

4 Incorporate the tunable gain surface into the model to be tuned.

   • In Simulink, associate the tunable gain surfaces with `Lookup Table` or `Interpolation` blocks used to mode the gain schedules. To do so, create an `slTuner` interface to the model, in which the `Lookup Table` or `Interpolation` blocks are tuned blocks. Then use `setBlockParam` to set the parameterization of these blocks to the tunable surface you created in step 3.

   • In MATLAB, interconnect the tunable surface with other LTI models to build an array of closed-loop generalized state-space (`genss`) models of the control system. The resulting model array covers all design points and depends on the tunable coefficients of the gain surfaces.

5 Use `systune` to tune the gain surface coefficients, subject to suitable requirements at each design point. From the `systune` perspective, doing so amounts to tuning one set of parameters, $K_0,...,K_M$, against many plant models (multimodel tuning).

Related Examples

More About

Plant Models for Gain-Scheduled Control

To tune a gain-scheduled control system, you need a collection of linear models describing the plant dynamics at the selected design points. This topic describes how to conceptualize this collection of models for two types of systems, linear parameter-varying plants and nonlinear plants.

In this section...
"Gain Scheduling for Linear Parameter-Varying Plants" on page 15-6
"Gain Scheduling for Nonlinear Plants" on page 15-7
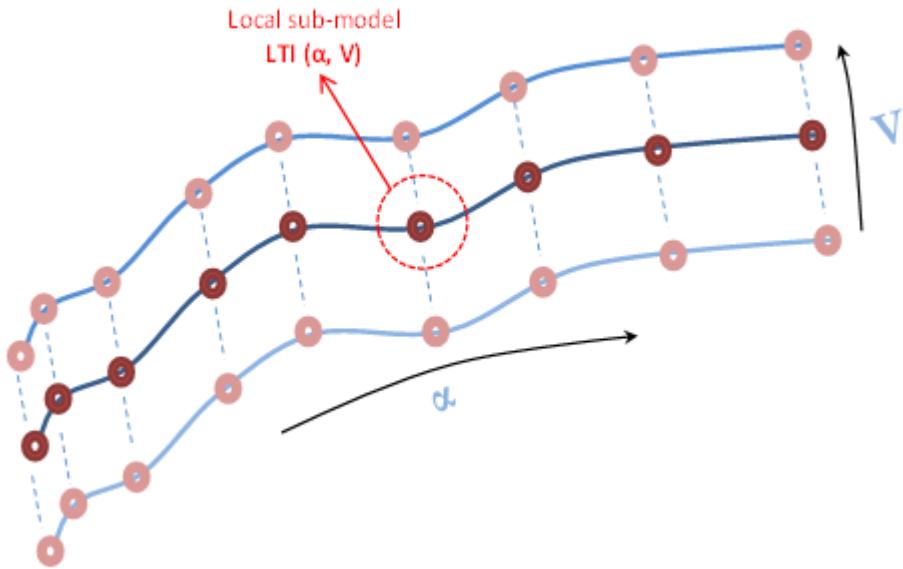
Gain Scheduling for Linear Parameter-Varying Plants

Gain-scheduled controllers are usually designed and tuned using a family of linear models that depend on the scheduling variables, σ:

$$
\frac{dx}{dt} = A(\sigma)x + B(\sigma)u
$$
$$
y = C(\sigma)x + D(\sigma)u.
$$

This family of models is called a *linear parameter-varying* (LPV) model. The LPV model describes how the (linearized) plant dynamics vary with time, operating condition, or any other scheduling variable. For example, the pitch axis dynamics of an aircraft can be approximated by an LPV model that depends on incidence angle, α, air speed, V, and altitude, h.

In practice, this continuum of plant models is often replaced by a finite set of linear models obtained for a suitable grid of σ values. This replacement amounts to sampling the LPV dynamics over the operating range and selecting a set of representative design points.

Gain-scheduled controllers yield best results when the plant dynamics vary smoothly between design points.

LPV Plants in MATLAB

In MATLAB, you can use an array of LTI plant models to represent an LPV system sampled at varying values of σ. Use the `SamplingGrid` property of the LTI model array to associate each linear model in the set with the underlying design points, σ. One way to obtain such an array is to create a parametric generalized state-space (`genss`) model of the system and sample the model with parameter variation to generate the array. For an example, see "Study Parameter Variation by Sampling Tunable Model" on page 2-111.

Gain Scheduling for Nonlinear Plants

In most applications, the plant is not given as an LPV model or a collection of linear models. Instead, the plant dynamics are described by nonlinear differential equations of the form:

$$\dot{x} = f(x, u, \sigma)$$
$$y = g(x, u, \sigma).$$

x is the state vector, u is the plant input, and y is the plant output. These equations can be specified explicitly. Or, they can be specified implicitly, such as by a Simulink model. For nonlinear plants, the linearized dynamics describe the local behavior of the plant around a family of operating points $(x(o), u(o))$ parameterized by the scheduling variables, o. Specifically, the deviations from nominal operating condition are defined as:

$$\delta x = x - x(\sigma), \quad \delta u = u - u(\sigma).$$

These deviations are governed, to first order, by the LPV dynamics:

$$\dot{\delta x} = A(\sigma)\delta x + B(\sigma)\delta u, \qquad \delta y = C(\sigma)\delta x + D(\sigma)\delta u,$$

$$A(\sigma) = \frac{\partial f}{\partial x}(x(\sigma), u(\sigma)) \qquad B(\sigma) = \frac{\partial f}{\partial u}(x(\sigma), u(\sigma))$$
$$C(\sigma) = \frac{\partial g}{\partial x}(x(\sigma), u(\sigma)) \qquad D(\sigma) = \frac{\partial g}{\partial u}(x(\sigma), u(\sigma)).$$

When repeated for a finite set of design points, σ, this local linearization produces the type of sampled LPV model described in "Gain Scheduling for Linear Parameter-Varying Plants" on page 15-6.

When your nonlinear plant is modeled in Simulink, use Simulink Control Design linearization tools to compute $A(o)$, $B(o)$, $C(o)$, and $D(o)$ for specific values of o. If the operating points are equilibrium points, you might need to first use `findop` to compute $(x(o), u(o))$ as a function of o. If you are controlling the system around a reference trajectory $(x(o), u(o))$, you can use snapshot linearization to acquire $A(o)$, $B(o)$, $C(o)$, and $D(o)$ at various points along the o trajectory. This second scenario includes time-varying systems where the scheduling variable is time. In either case, the result is a collection of linear models sampled at values of o. (See "Set Up Simulink Models for Gain Scheduling" on page 15-17.)

Typically, the main task of the gain-scheduled controller is to keep the closed-loop system close to equilibrium or close to the nominal trajectory, $(x(o), u(o))$. The controller provides

a corrective action, δu, which is added to the nominal command $u(o)$ to determine the total actuator command.

Related Examples

- "Set Up Simulink Models for Gain Scheduling" on page 15-17

More About

- "Parametric Gain Surfaces" on page 15-10

Parametric Gain Surfaces

A gain surface is a parameterization of a variable gain in terms of the scheduling variables. For tuning gain-scheduled control systems, use tunable, or parametric, gain surfaces to model the variable gains.

In this section...
"Basis Function Parameterization" on page 15-10
"Creating Tunable Surfaces" on page 15-12
"Tunable Surfaces in Simulink" on page 15-14
"Tunable Surfaces in MATLAB" on page 15-15

Basis Function Parameterization

In a gain-scheduled controller, the scheduled gains are functions of the scheduling variables, σ. For example, a gain-scheduled PI controller has the form:

$$C(s, \sigma) = K_p(\sigma) + \frac{K_i(\sigma)}{s}.$$
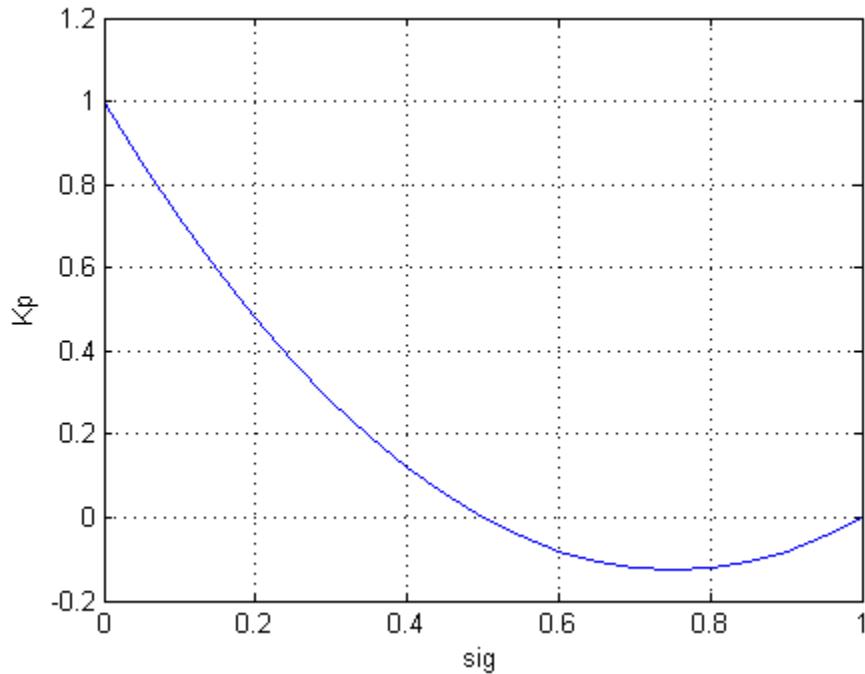
Tuning this controller requires determining the functional forms $K_p(\sigma)$ and $K_i(\sigma)$ that yield the best system performance over the operating range of σ values. However, tuning arbitrary functions is difficult. Therefore, it is necessary either to consider the function values at only a finite set of points, or restrict the generality of the functions themselves.

In the first approach, you choose a collection of design points, σ, and tune the gains K_p and K_i independently at each design point. The resulting set of gain values is stored in a lookup table driven by the scheduling variables, σ. A drawback of this approach is that tuning might yield substantially different values for neighboring design points, causing undesirable jumps when transitioning from one operating point to another.

Alternatively, you can model the gains as smooth functions of σ, but restrict the generality of such functions by using specific basis function expansions. For example, suppose σ is a scalar variable. You can model $K_p(\sigma)$ as a quadratic function of σ:

$$K_p(\sigma) = k_0 + k_1\sigma + k_2\sigma^2.$$
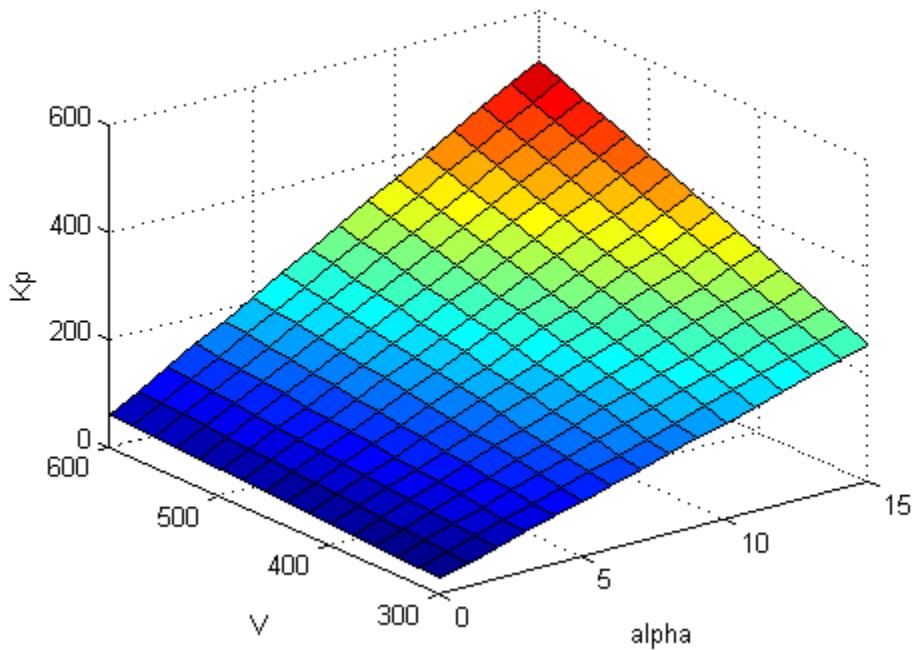
After tuning, this parametric gain might have a profile such as the following (the specific shape of the curve depends on the tuned coefficient values and range of o):

Or, suppose that o consists of two scheduling variables, a, and V. Then, you can model $K_p(o)$ as a bilinear function of a and V:

$$K_p(\alpha, V) = k_0 + k_1\alpha + k_2V + k_3\alpha V.$$

After tuning, this parametric gain might have a profile such as the following. Here too, the specific shape of the curve depends on the tuned coefficient values and ranges of o values:

For tuning gain schedules with `systune`, you use a *parametric gain surface* that is a particular expansion of the gain in basis functions of σ:

$$K(\sigma) = K_0 + K_1 F_1\left(n(\sigma)\right) + \ldots + K_M F_M\left(n(\sigma)\right).$$

The basis functions $F_1,\ldots,F_M$ are user-selected and fixed. These functions operate on $n(\sigma)$, normalized values of the scheduling variables in the interval [–1,1]. The coefficients of the expansion, $K_0,\ldots,K_M$, are the tunable parameters of the gain surface. $K_0,\ldots,K_M$ can be scalar or matrix-valued, depending on the I/O size of the gain $K(\sigma)$. The choice of basis function is problem-dependent, but it is generally good to try low-order polynomial expansions first.

Creating Tunable Surfaces

To tune a gain-scheduled controller with `systune`, construct a tunable model of a gain surface sampled over a grid of design points (σ values). To do so, use the

`tunableSurface` command. For example, consider the gain with bilinear dependence on two scheduling variables, α and V:

$$K_p(\alpha, V) = K_0 + K_1\alpha + K_2 V + K_3\alpha V.$$

Suppose that α is an angle of incidence that ranges from 0° to 15°, and V is a speed that ranges from 300 m/s to 700 m/s. Create a grid of design points that span these ranges. These design points should match the parameter values at which you sample your varying or nonlinear plant. Their values are stored with the tunable surface.

```
[alpha,V] = ndgrid(0:5:15,300:100:700);
domain = struct('alpha',alpha,'V',V);
```

Specify the basis functions for the parameterization of this surface, α, V, and αV. The `tunableSurface` command expects the basis functions to be arranged as a vector of functions of two input variables. You can use an anonymous function to express the basis functions.

```
shapefcn = @(alpha,V)[alpha,V,alpha*V];
```

Alternatively, use `polyBasis`, `fourierBasis`, or `ndBasis` to generate basis functions of as many scheduling variables as you need.

Create the tunable surface using the design points and basis functions.

```
Kp = tunableSurface('Kp',1,domain,shapefcn);
```

`Kp` is a tunable model of the gain surface. `tunableSurface` parameterizes the surface as:

$$K_p(\alpha, V) = \bar{K}_0 + \bar{K}_1\bar{\alpha} + \bar{K}_2\bar{V} + \bar{K}_3\bar{\alpha}\bar{V},$$

where

$$\bar{\alpha} = \frac{\alpha - 7.5}{7.5}, \quad \bar{V} = \frac{V - 500}{200}.$$

The surface is expressed in terms of the normalized variables, $\bar{\alpha}, \bar{V} \in [-1,1]^2$ rather than in terms of α and V. This normalization, performed automatically by `tunableSurface`, improves the conditioning of the optimization performed by the `systune`.

The second input argument to `tunableSurface` specifies the initial value of the constant coefficient, K_0. You can create array-valued tunable gains by providing an array for that input.
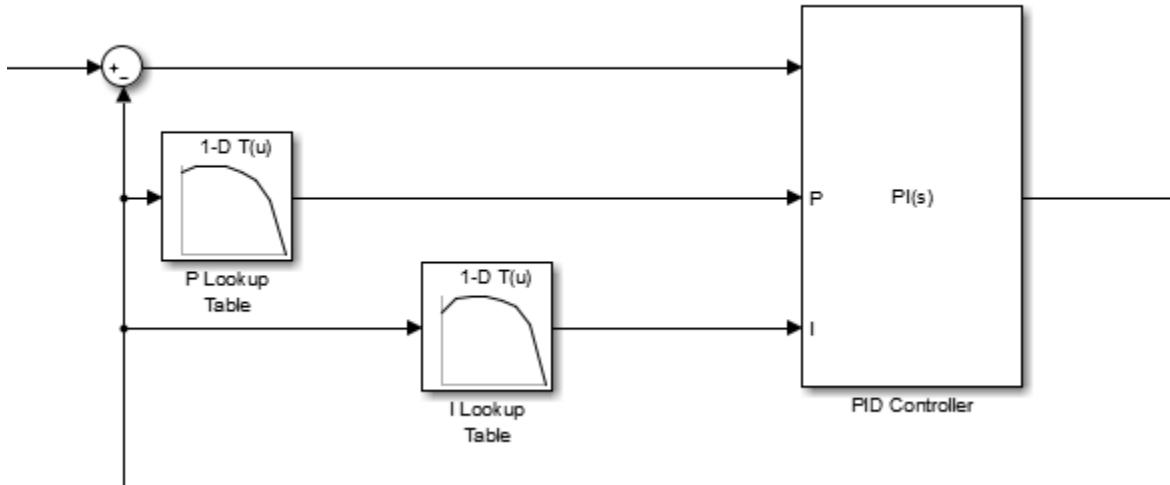
```
Karr = tunableSurface('Karr',ones(2),domain,shapefcn);
```

This command creates a two-by-two matrix in which each entry is a tunable surface with the same design points.

For more details about creating tunable surfaces, see the `tunableSurface` reference page.

Tunable Surfaces in Simulink

In your Simulink model, use `Lookup Table` blocks or `Interpolation` blocks to represent the scheduled gain. For example, this illustration shows a portion of a Simulink model in which `1-D Lookup Table` blocks feed the gains of a `PID Controller` block. In this model, each gain is a function of the same scheduling variable.

To tune these variable gains, create an `slTuner` interface to the model, in which the `Lookup Table` blocks are tuned blocks. Then use `setBlockParam` to set the

parameterization of each of these blocks to a tunable surface. For more details, see "Set Up Simulink Models for Gain Scheduling" on page 15-17.

For an example illustrating the entire tuning workflow in Simulink, see "Tuning of Gain-Scheduled Three-Loop Autopilot" on page 15-31.

Tunable Surfaces in MATLAB

For a control system modeled in MATLAB, use tunable surfaces to construct more complex gain-scheduled control elements, such as gain-scheduled PID controllers, filters, or state-space controllers. For example, suppose that you create two gain surfaces Kp and Ki using `tunableSurface`. The following command constructs a tunable gain-scheduled PI controller.

```
C0 = pid(Kp,Ki);
```

Similarly, suppose that you create four matrix-valued gain surfaces A, B, C, D. The following command constructs a tunable gain-scheduled state-space controller.

```
C1 = ss(A,B,C,D);
```

You then incorporate the gain-scheduled controller into a generalized model of your entire control system. For example, suppose G is an array of models of your plant sampled at the design points that are specified in Kp and Ki. Then, the following command builds a tunable model of a gain-scheduled single-loop PID control system.

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

When you interconnect a tunable surface with other LTI models, the resulting model is an array of tunable generalized `genss` models. The dimensions of the array are determined by the design points in the tunable surface. Thus, each entry in the array represents the system at the corresponding scheduling variable value. The `SamplingGrid` property of the array stores those design points.

```
T0 = feedback(G*Kp,1)

T0 =

  4x5 array of generalized continuous-time state-space models.
  Each model has 1 outputs, 1 inputs, 3 states, and the following blocks:
    Kp: Parametric 1x4 matrix, 1 occurrences.

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

```
"TO.Blocks" to interact with the blocks.
```

The resulting generalized model has tunable blocks corresponding to the gain surfaces used to create the model. In this example, the system has one gain surface, Kp, which has the four tunable coefficients corresponding to K_0, K_1, K_2, and K_3. Therefore, the tunable block is a vector-valued `realp` parameter with four entries.

When you tune the control system with `systune`, the software tunes the coefficients for each of the design points specified in the tunable surface.

For an example illustrating the entire workflow in MATLAB, see the section "Controller Tuning in MATLAB" in "Gain-Scheduled Control of a Chemical Reactor" on page 15-47.

See Also
tunableSurface

Related Examples

More About

Set Up Simulink Models for Gain Scheduling

To tune a gain-scheduled control system in Simulink, model the scheduled gains using Lookup Table or Interpolation blocks. Then, create an `slTuner` interface for the model that linearizes the system at a set of design points, or replaces the nonlinear plant model with a set of models linearized at these design points. These design points are the scheduling-variable values at which you tune the coefficients of the variable gain.
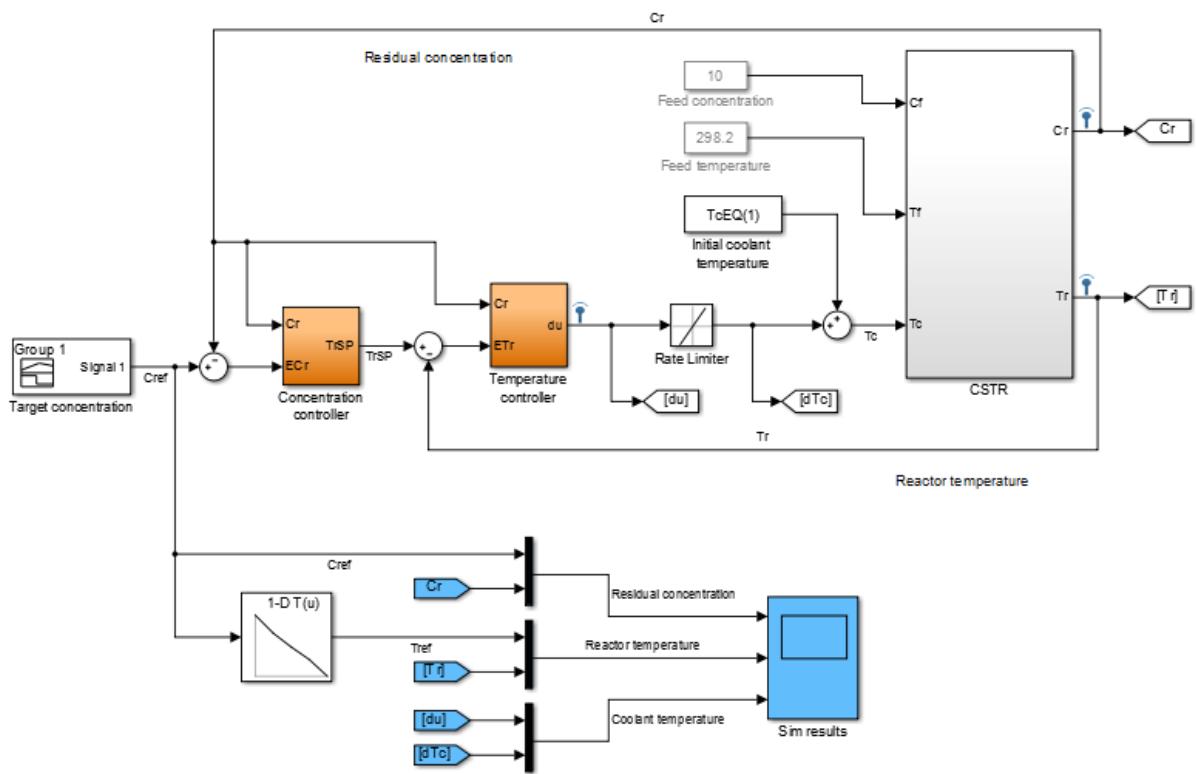
In this section...
"Model Scheduled Gains" on page 15-17
"Sample System at Design Points" on page 15-20
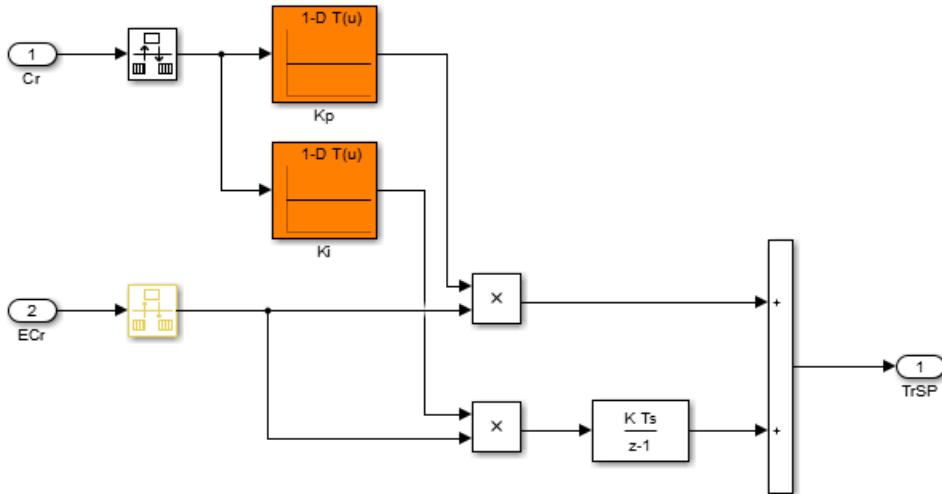
Model Scheduled Gains

The model `rct_CSTR` includes a PI controller and lead-lag controller in which the controller gains are implemented as lookup tables using `1-D Lookup Table` blocks. Open that model.

```
open_system('rct_CSTR')
```

Both the `Concentration controller` and `Temperature controller` blocks take the `CSTR` plant output, `Cr`, as an input. This value is both the controlled variable of the system and the scheduling variable on which the controller action depends. Double-click the `Concentration controller` block.

Gain-scheduled PID controller Kp + Ki * Ts/(z-1)

This block is a PI controller in which the proportional gain `Kp` and integrator gain `Ki` are determined by feeding the scheduling parameter `Cr` into a `1-D Lookup Table` block. Similarly, the `Temperature controller` block contains three gains implemented as lookup tables. To model gains that depend on more than one scheduling parameter, use `n-D Lookup Table` blocks.

Tuning a gain-scheduled control system amounts to identifying appropriate values for these lookup tables. When you configure an `slTuner` interface for gain-scheduled tuning, you parameterize the lookup tables as functions of the scheduling variables. For example, to tune the gains `Kp` and `Ki` of the `rct_CSTR` system, you create an `slTuner` interface for `rct_CSTR` in which the blocks `Kp` and `Ki` are designated as tuned blocks.

To set the parameterization of those blocks in the interface, create a tunable gain surface for each lookup table, as described in "Parametric Gain Surfaces" on page 15-10. Then, set the parameterizations of the blocks to the tunable gain surfaces using the `setBlockParam` command.

Sample System at Design Points
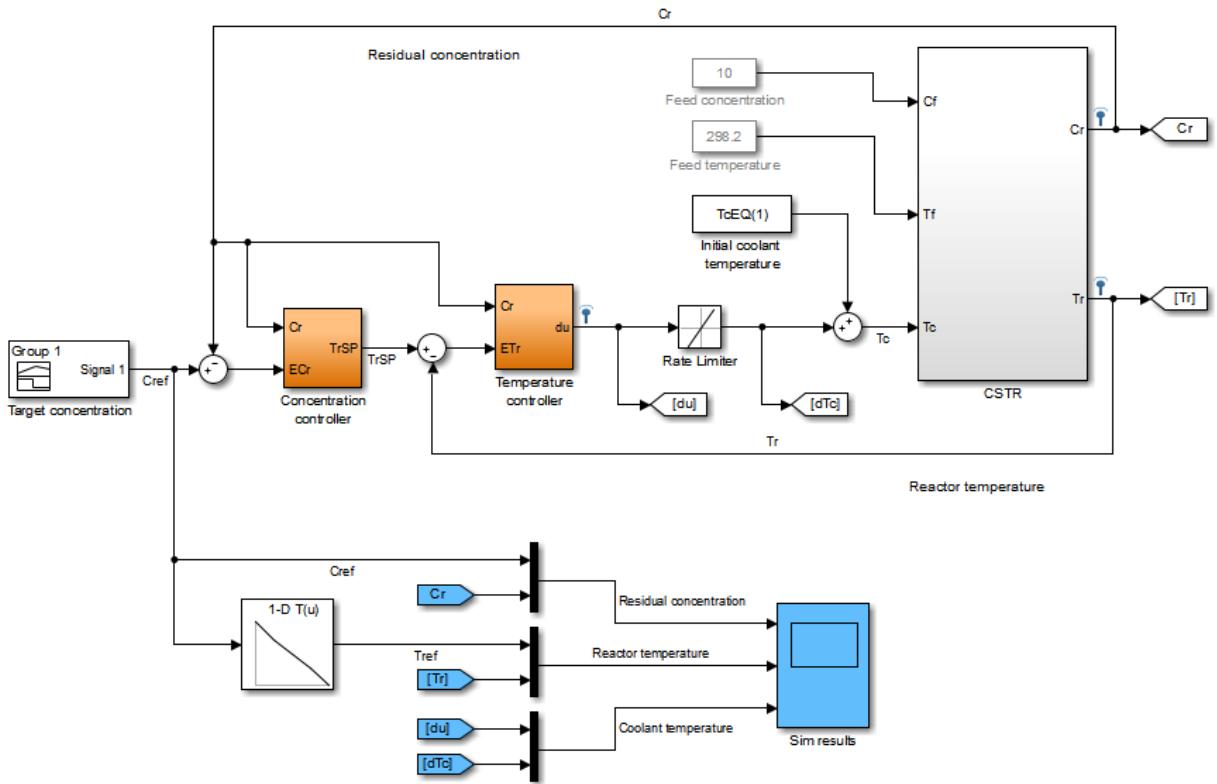
As described in "Plant Models for Gain-Scheduled Control" on page 15-6, gain-scheduled controllers are designed for a family of models that correspond to the system sampled under a range of operating conditions. When tuning a control system modeled in Simulink, configure the slTuner interface to linearize or sample the system over your grid of design points. There several ways to achieve this configuration.

Set Up for Gain Scheduling by Linearizing at Design Points

This example shows how to linearize a plant model at a set of design points for tuning of a gain-scheduled controller. The example then uses the resulting linearized models to configure an slTuner interface for tuning the gain schedule.
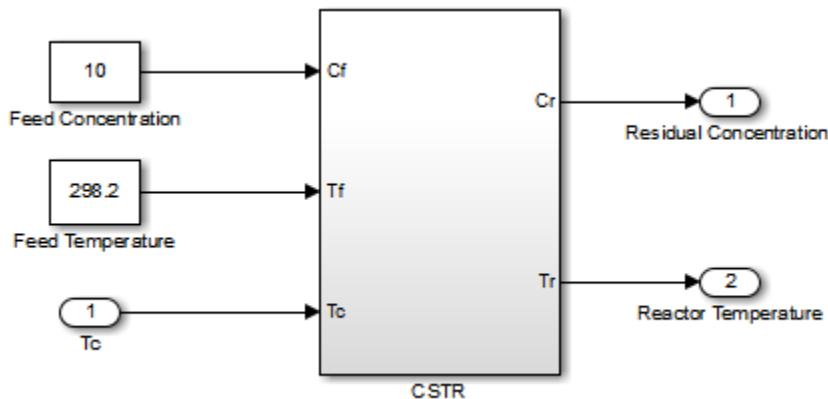
Consider the rct_CSTR model.

```
open_system('rct_CSTR')
```

To tune this gain-scheduled system, linearize the plant at a range of steady-state operating points that correspond to different values of the scheduling parameter Cr. Sometimes, it is convenient to use a model of the plant for trimming and linearization under various operating conditions. For example, in this case, the most straightforward way to obtain these linearizations is to use a separate open-loop model of the plant, rct_CSTR_OL.

```
open_system('rct_CSTR_OL')
```

Suppose that you want to control this plant at a range of `Cr` values from 4 to 8. Trim the model to find steady-state operating points for a set of values in this range.

```
Cr = (4:8)';          % concentrations
for k=1:length(Cr)
   opspec = operspec('rct_CSTR_OL');
   % Set desired residual concentration
   opspec.Outputs(1).y = Cr(k);
   opspec.Outputs(1).Known = true;
   % Compute equilibrium condition
   [op(k),report(k)] = findop('rct_CSTR_OL',opspec,findopOptions('DisplayReport','off')
end
```

`op` is an array of steady-state operating points. (For more information about steady-state operating points, see "About Operating Points" in the Simulink Control Design documentation.) Linearizing the plant block using `op` returns an array of plant models, each linearized at the corresponding operating condition.

```
G = linearize('rct_CSTR_OL','rct_CSTR_OL/CSTR',op);
```

To tune the control system `rct_CSTR`, create an `slTuner` interface that linearizes the system at those design points. Use block substitution to replace the plant in `rct_CSTR` with the linearized plant-model array `G`. (For more information about block substitution, see the slTuner reference page.)

```
blocksub.Name = 'rct_CSTR/CSTR';
blocksub.Value = G;
```

```
tunedblocks = {'Kp','Ki'};
STO = slTuner('rct_CSTR',tunedblocks,blocksub);
```

For this example, only the PI coefficients of `Concentration controller` are designated as tuned blocks. In general, however, `tunedblocks` lists all the blocks to tune.

Next, create tunable gain surfaces and set the parameterization of the tuned blocks. For example, parameterize `'Kp'` as a quadratic function of the scheduling variable. The design points in the tunable surface must match the design points in the `slTuner` interface.

```
designpoints = struct('Cr',Cr);
shapefcn = @(Cr)[Cr,Cr^2];
Kp = tunableSurface('Kp',0,designpoints,shapefcn);
setBlockParam(STO,'Kp',Kp);
```

After you parameterize all the scheduled gains, you can create your tuning goals and tune the system with `systune`.

Set Up by Sampling System at Varying Parameter Values

If the scheduling variable is a parameter in the Simulink model, use parameter variation to create an `slTuner` interface that samples the control system over a parameter grid. For example, suppose that you want to tune a model named `suspension_gs` that contains two parameters, `Ks` and `Bs`. These parameters each can vary over some known range, and a controller gain in the model varies as a function of both parameters.

To set up such a model for tuning, create a grid of parameter values. For this example, let `Ks` vary between 1 and 5, and let `Bs` vary between 0.6 and 0.9.

```
Ks = 1:5;
Bs = [0.6:0.1:0.9];
[Ksgrid,Bsgrid] = ndgrid(Ks,Bs);
```

These values are the design points at which to sample and tune the system. For example, create an `slTuner` interface to the model, assuming one tunable block, a `Lookup Table` block named `K` that models the parameter-dependent gain.

```
params(1) = struct('Name','Ks','Value',Ksgrid);
params(2) = struct('Name','Bs','Value',Bsgrid);
STO = slTuner('suspension_gs','K',params);
```

`slTuner` samples the model at all `(Ksgrid,Bsgrid)` values specified in `params`.

Next, use the same design points to create a tunable gain surface for parameterizing K.

```
design = struct('Ks',Ksgrid,'Bs',Bsgrid);
shapefcn = @(Ks,Bs)[Ks,Bs,Ks*Bs];
K = tunableSurface('K',1,design,shapefcn);
setBlockParam(STO,'K',K);
```

After you parameterize all the scheduled gains, you can create your tuning goals and tune the system with systune.

Set Up by Sampling System at Simulation Snapshots

If you are controlling the system around a reference trajectory $(x(\sigma),u(\sigma))$, use snapshot linearization to sample the system at various points along the σ trajectory. Use this approach for time-varying systems where the scheduling variable is time.

To linearize a system at a set of simulation snapshots, use a vector of positive scalars as the op input argument of linearize , slLinearizer, or slTuner. These scalars are the simulation times at which to linearize the model. Use the same set of time values as the design points in tunable surfaces for the system.

See Also
setBlockParam | slTuner | tunableSurface

Related Examples
- "Tuning of Gain-Scheduled Three-Loop Autopilot" on page 15-31
- "Gain-Scheduled Control of a Chemical Reactor" on page 15-47

More About
- "Plant Models for Gain-Scheduled Control" on page 15-6
- "Parametric Gain Surfaces" on page 15-10

Validating Gain-Scheduled Controllers

Gain-scheduled controllers require careful validation. The tuning process guarantees suitable performance only near each design point. In addition, the tuning ignores dynamic couplings between the plant state variables and the scheduling variables (see Section 4.3, "Hidden Coupling", in [1]). Best practices for validation include:

- Examine the tuned gains as a function of the scheduling variables to make sure that they are smooth and well-behaved over the operating range. Visualize tuned gain surfaces using the `viewSurf` command.

- Check linear performance on a denser grid of σ values than you used for design. If adequate linear performance is not maintained between design points, you can add more design points and retune.

- Perform nonlinear simulations that drive the closed-loop system through its entire operating range. Pay special attention to maneuvers that cause rapid variations of the scheduling variables.

References

[1] Rugh, W.J., and J.S. Shamma, "Research on Gain Scheduling", *Automatica*, 36 (2000), pp. 1401-1425.

See Also

`viewSurf`

Related Examples

- "Tuning of Gain-Scheduled Three-Loop Autopilot" on page 15-31
- "Gain-Scheduled Control of a Chemical Reactor" on page 15-47

More About

- "Changing Requirements with Operating Condition" on page 15-26

Changing Requirements with Operating Condition

In gain-scheduled control systems, it is not uncommon for the control objectives to vary with the operating condition. In this case, the control objectives also depend on the scheduling variables. For example, your system might require a reduced control bandwidth if transmission delays increase or temperature drops. When you use `systune` to tune such a system, there are two approaches to specifying such scheduled tuning goals.

In this section...
"Create Separate Requirement for Each Design Point" on page 15-26
"Build Variation into the Model" on page 15-26

Create Separate Requirement for Each Design Point

One way to enforce a requirement that varies with design point is to create a separate instance of the requirement for each design point. For example, suppose that you want to enforce a 1/*s* loop shape with a crossover frequency that depends on the scheduling variables. Suppose also that you have created a table, `wc`, that contains the target bandwidth for each design point, *σ*. Then you can construct one `TuningGoal.LoopShape` requirement for each design point. Associate each `TuningGoal.LoopShape` requirement with the corresponding design point using the `Models` property of the requirement.

```
for ct=1:num_design_points
   R(ct) = TuningGoal.LoopShape('u',wc(ct));
   R(ct).Model = ct;
end
```

Build Variation into the Model

Instead of creating varying requirements, you can incorporate the varying portion of the requirement into the closed-loop model of the control system. This approach is a form of goal normalization that makes it possible to cover all design points with a single uniform goal.

For example, suppose that you want to limit the gain from `d` to `y` to a quantity that depends on the scheduling variables. Suppose that `T0` is an array of models of the closed-

loop system at each design point. Suppose further that you have created a table, gmax, of the maximum gain values for each design point, *o*. Then you can add another output ys = y/gmax to the closed-loop model, as follows.

```
% Create array of scalar gains 1/gmax
yScaling = reshape(1./gmax,[1 1 size(gmax)]);
yScaling = ss(yScaling,'InputName','y','OutputName','ys');

% Connect these gains in series to y output of TO
TO = connect(TO,yScaling,TO.InputName,[TO.OutputName ; {'ys'}]);
```

The maximum gain changes at each design point according to the table gmax. You can then use a single requirement that limits to 1 the gain from d to the scaled output ys.

```
R = TuningGoal.Gain('d','ys',1);
```

Such effective normalization of requirements moves the requirement variability from the requirement object, R, to the closed-loop model, TO.

In Simulink, you can use a similar approach by feeding the relevant model inputs and outputs through a gain block. Then, when you linearize the model, change the gain value of the block with the operating condition. For example, set the gain to a MATLAB variable, and use the Parameters property in slLinearizer to change the variable value with each linearization condition.

More About

· "Tune Gain-Scheduled Controllers" on page 15-4
· "Plant Models for Gain-Scheduled Control" on page 15-6

Tunable Gain With Two Independent Scheduling Variables

This example shows how to model a scalar gain K with a bilinear dependence on two scheduling variables, α and V, as follows:

$$K(\alpha, V) = K_0 + K_1 x + K_2 y + K_3 xy.$$

Here, x and y are the normalized scheduling variables. Suppose that for this example, α is an angle of incidence that ranges from 0 to 15 degrees, and V is a speed that ranges from 300 to 600 m/s. Then, x and y are given by:

$$x = \frac{\alpha - 7.5}{7.5}, \quad y = \frac{V - 450}{150}.$$

The coefficients $K_0, ..., K_3$ are the tunable parameters of this variable gain.

Create a grid of design points, (α, V), that are linearly spaced in α and V. These design points are the scheduling-variable values used for tuning the gain-surface coefficients. They must correspond to parameter values at which you have sampled the plant.

```
[alpha,V] = ndgrid(0:3:15,300:50:600);
```

These arrays, `alpha` and `V`, represent the independent variation of the two scheduling variables, each across its full range. Put them into a structure to define the design points for the tunable surface.

```
domain = struct('alpha',alpha,'V',V);
```

Create the basis functions that describe the bilinear expansion.

```
shapefcn = @(x,y) [x,y,x*y];  % or use polyBasis('canonical',2,1)
```

In the array returned by `shapefcn`, the basis functions are:
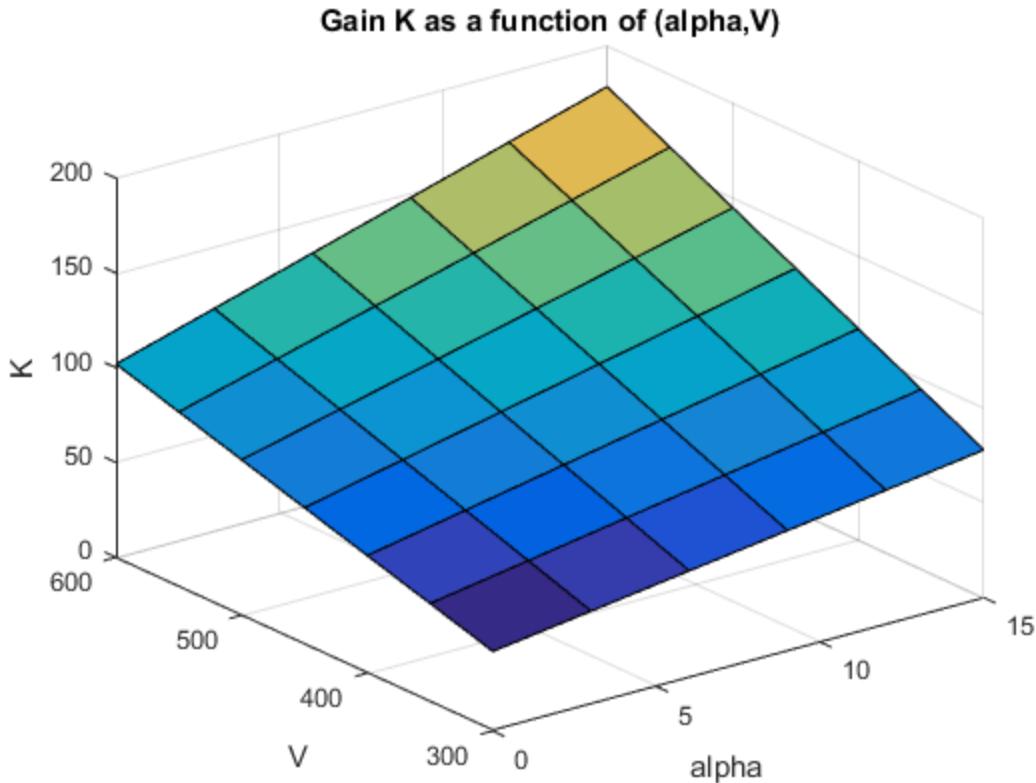
$$F_1(x, y) = x$$
$$F_2(x, y) = y$$
$$F_3(x, y) = xy.$$

Create the tunable gain surface.

```
K = tunableSurface('K',1,domain,shapefcn);
```

You can use the tunable surface as the parameterization for a Lookup Table block in a Simulink model. Or, use model interconnection commands to incorporate it as a tunable element in a control system modeled in MATLAB. After you tune the coefficients, you can examine the resulting gain surface using the `viewSurf` command. For this example, instead of tuning, manually set the coefficients to non-zero values and view the resulting gain.

```
Ktuned = setData(K,[100,28,40,10]);
viewSurf(Ktuned)
```


Gain K as a function of (alpha,V)

`viewSurf` displays the gain surface as a function of the scheduling variables, for the ranges of values specified by `domain` and stored in the `SamplingGrid` property of the gain surface.

See Also
`setBlockParam` | `slTuner` | `tunableSurface`

Related Examples
- "Tuning of Gain-Scheduled Three-Loop Autopilot" on page 15-31

More About
- "Parametric Gain Surfaces" on page 15-10
- "Plant Models for Gain-Scheduled Control" on page 15-6

Tuning of Gain-Scheduled Three-Loop Autopilot

This example uses `systune` to generate smooth gain schedules for a three-loop autopilot.

Airframe Model and Three-Loop Autopilot

This example uses a three-degree-of-freedom model of the pitch axis dynamics of an airframe. The states are the Earth coordinates (X_e, Z_e), the body coordinates (u, w), the pitch angle θ, and the pitch rate $q = \dot{\theta}$. Figure 1 summarizes the relationship between the inertial and body frames, the flight path angle γ, the incidence angle α, and the pitch angle θ.
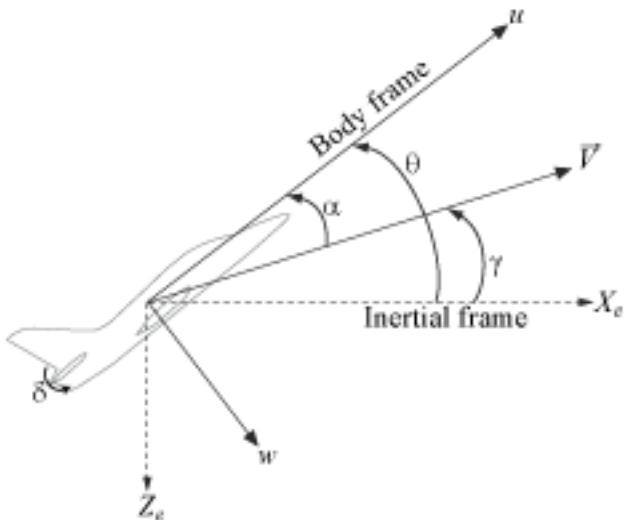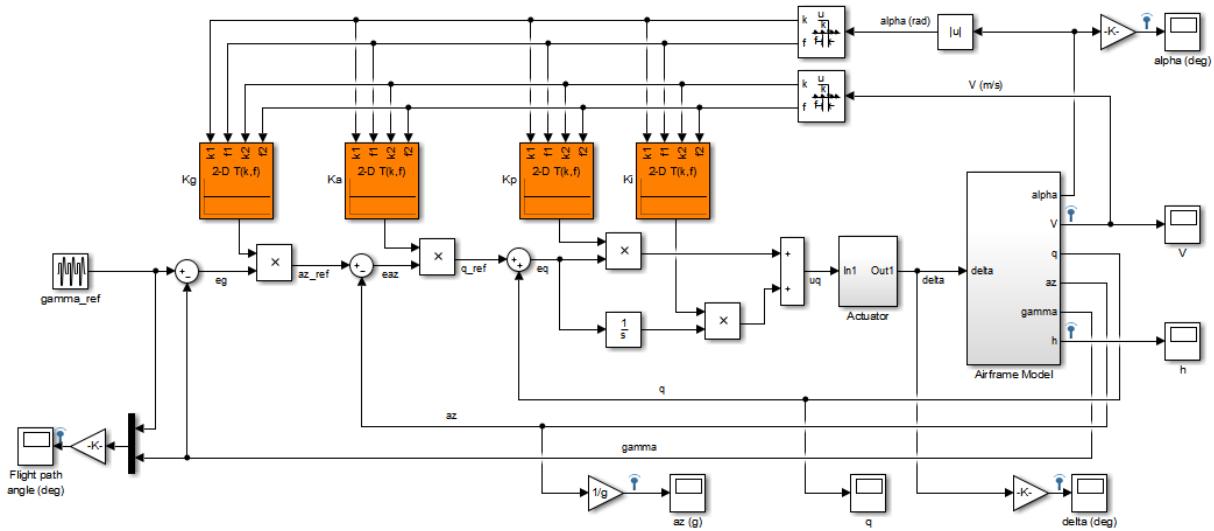
Figure 1: Airframe dynamics.

We use a classic three-loop autopilot structure to control the flight path angle γ. This autopilot adjusts the flight path by delivering adequate bursts of normal acceleration a_z (acceleration along w). In turn, normal acceleration is produced by adjusting the elevator deflection δ to cause pitching and vary the amount of lift. The autopilot uses

Proportional-Integral (PI) control in the pitch rate loop q and proportional control in the a_z and γ loops. The closed-loop system (airframe and autopilot) are modeled in Simulink.

```
open_system('rct_airframeGS')
```


Autopilot Gain Scheduling

The airframe dynamics are nonlinear and the aerodynamic forces and moments depend on speed V and incidence α. To obtain suitable performance throughout the (α, V) flight envelope, the autopilot gains must be adjusted as a function of α and V to compensate for changes in plant dynamics. This adjustment process is called "gain scheduling" and α, V are called the scheduling variables. In the Simulink model, gain schedules are implemented as look-up tables driven by measurements of α and V.

Gain scheduling is a linear technique for controlling nonlinear or time-varying plants. The idea is to compute linear approximations of the plant at various operating conditions, tune the controller gains at each operating condition, and swap gains as a function of operating condition during operation. Conventional gain scheduling involves three major steps:

1 Trim and linearize the plant at each operating condition

2 Tune the controller gains for the linearized dynamics at each operating condition

3 Reconcile the gain values to provide smooth transition between operating conditions.

In this example, we combine Steps 2. and 3. by parameterizing the autopilot gains as first-order polynomials in α, V and directly tuning the polynomial coefficients for the entire flight envelope. This approach eliminates Step 3. and guarantees smooth gain variations as a function of α and V. Moreover, the gain schedule coefficients can be automatically tuned with `systune`.
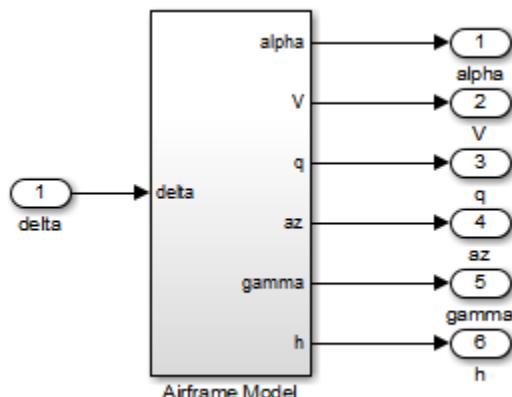
Trimming and Linearization

Assume that the incidence α varies between -20 and 20 degrees and that the speed V varies between 700 and 1400 m/s. When neglecting gravity, the airframe dynamics are symmetric in α so consider only positive values of α. Use a 5-by-9 grid of linearly spaced (α, V) pairs to cover the flight envelope:

```
nA = 5;  % number of alpha values
nV = 9;  % number of V values
[alpha,V] = ndgrid(linspace(0,20,nA)*pi/180,linspace(700,1400,nV));
```

For each flight condition (α, V), linearize the airframe dynamics at trim (zero normal acceleration and pitching moment). This requires computing the elevator deflection δ and pitch rate q that result in steady w and q. To do this, first isolate the airframe model in a separate Simulink model.

```
open_system('rct_airframeTRIM')
```


Airframe Model

Use `operspec` to specify the trim condition, use `findop` to compute the trim values of δ and q, and linearize the airframe dynamics for the resulting operating point. See the "Trimming and Linearizing an Airframe" example in Simulink Control Design for details. Repeat these steps for the 45 flight conditions (α, V).

```
% Compute trim condition for each (alpha,V) pair
clear op
for ct=1:nA*nV
   alpha_ini = alpha(ct);      % Incidence [rad]
   v_ini = V(ct);              % Speed [m/s]

   % Specify trim condition
   opspec = operspec('rct_airframeTRIM');
   % Xe,Ze: known, not steady
   opspec.States(1).Known = [1;1];
   opspec.States(1).SteadyState = [0;0];
   % u,w: known, w steady
   opspec.States(3).Known = [1 1];
   opspec.States(3).SteadyState = [0 1];
   % theta: known, not steady
   opspec.States(2).Known = 1;
   opspec.States(2).SteadyState = 0;
   % q: unknown, steady
   opspec.States(4).Known = 0;
   opspec.States(4).SteadyState = 1;

   % TRIM
   Options = findopOptions('DisplayReport','off');
   op(ct) = findop('rct_airframeTRIM',opspec,Options);
end

% Linearize at trim conditions
G = linearize('rct_airframeTRIM',op);
G = reshape(G,[nA nV]);
G.u = 'delta';
G.y = {'alpha' 'V' 'q' 'az' 'gamma' 'h'};
```
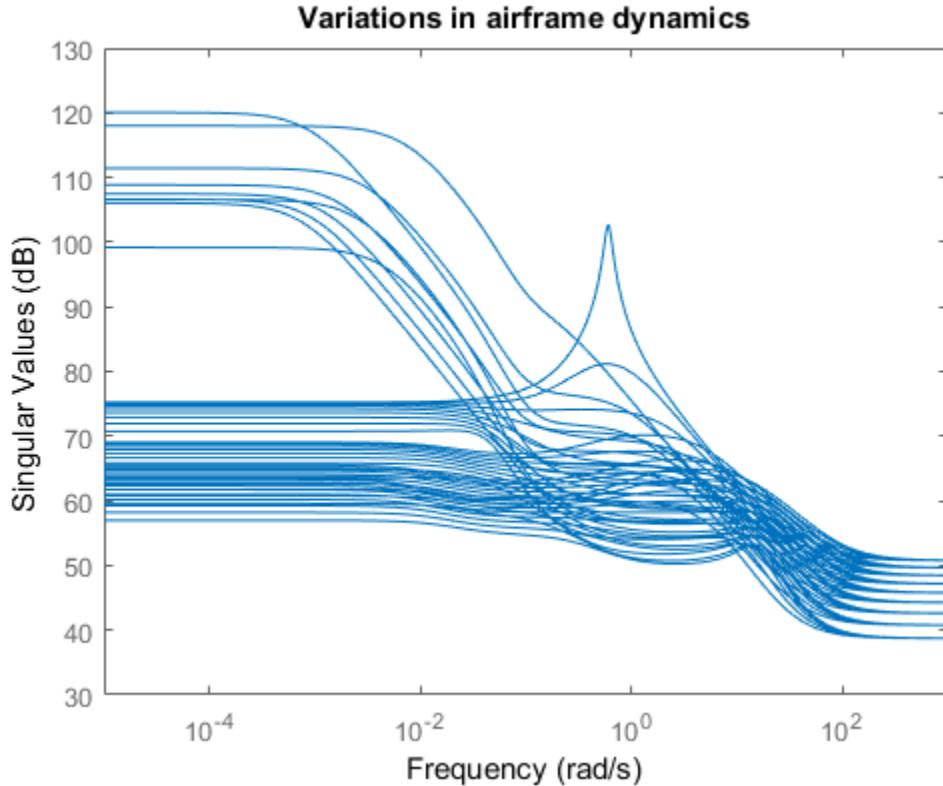
This produces a 5-by-9 array of linearized plant models at the 45 flight conditions (α, V). The plant dynamics vary substantially across the flight envelope.

```
sigma(G), title('Variations in airframe dynamics')
```

Variations in airframe dynamics

Tunable Gain Surface

The autopilot consists of four gains K_p, K_i, K_a, K_g to be "scheduled" (adjusted) as a function of α and V. Practically, this means tuning 88 values in each of the corresponding four look-up tables. Rather than tuning each table entry separately, parameterize the gains as a two-dimensional gain surfaces, for example, surfaces with a simple multilinear dependence on α and V:

$$K(\alpha, V) = K_0 + K_1\alpha + K_2V + K_3\alpha V.$$

This cuts the number of variables from 88 down to 4 for each lookup table. Use the `tunableSurface` object to parameterize each gain surface. Note that:

- `TuningGrid` specifies the "tuning grid" (design points). This grid should match the one used for linearization but needs not match the loop-up table breakpoints

- `ShapeFcn` specifies the basis functions for the surface parameterization (α, V, and αV)

Each surface is initialized to a constant gain using the tuning results for α = 10 deg and V = 1050 m/s (mid-range design).

```
TuningGrid = struct('alpha',alpha,'V',V);
ShapeFcn = @(alpha,V) [alpha,V,alpha*V];

Kp = tunableSurface('Kp', 0.1, TuningGrid, ShapeFcn);
Ki = tunableSurface('Ki', 2, TuningGrid, ShapeFcn);
Ka = tunableSurface('Ka', 0.001, TuningGrid, ShapeFcn);
Kg = tunableSurface('Kg', -1000, TuningGrid, ShapeFcn);
```

Next create an `slTuner` interface for tuning the gain surfaces. Use block substitution to replace the nonlinear plant model by the linearized models over the tuning grid. Use `setBlockParam` to associate the tunable gain surfaces `Kp`, `Ki`, `Ka`, `Kg` with the Interpolation blocks of the same name.

```
BlockSubs = struct('Name','rct_airframeGS/Airframe Model','Value',G);
ST0 = slTuner('rct_airframeGS',{'Kp','Ki','Ka','Kg'},BlockSubs);

% Register points of interest
ST0.addPoint({'az_ref','az','gamma_ref','gamma','delta'})

% Parameterize look-up table blocks
ST0.setBlockParam('Kp',Kp);
ST0.setBlockParam('Ki',Ki);
ST0.setBlockParam('Ka',Ka);
ST0.setBlockParam('Kg',Kg);
```
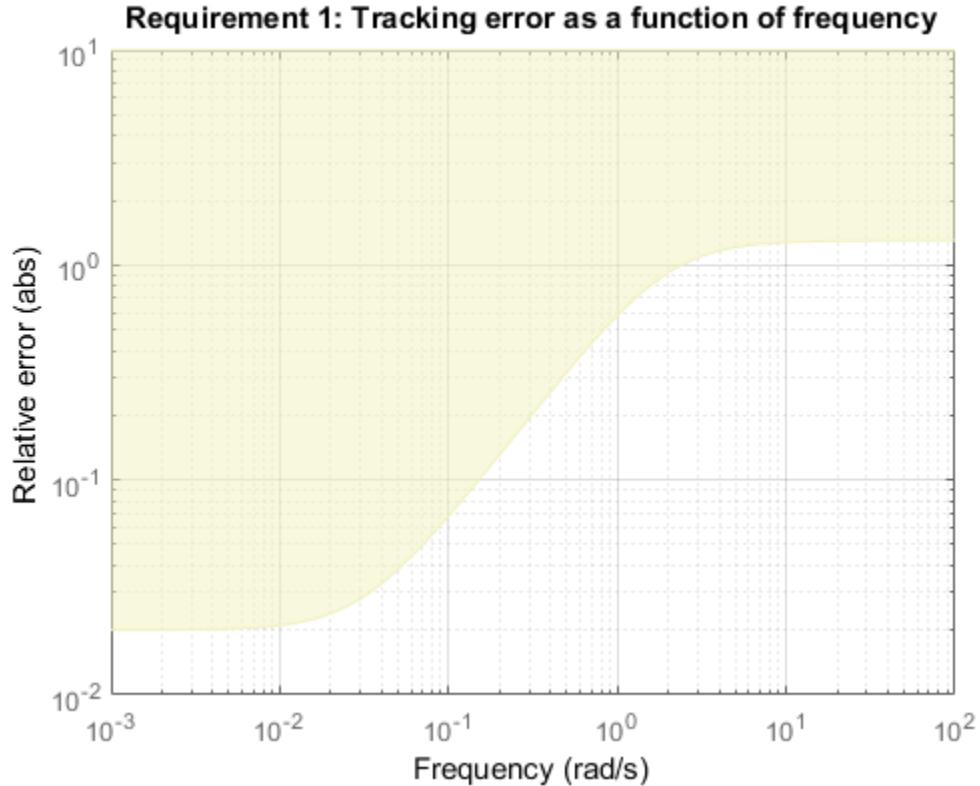
Autopilot Tuning

`systune` can automatically tune the gain surface coefficients for the entire flight envelope. Use `TuningGoal` objects to specify the performance objectives:
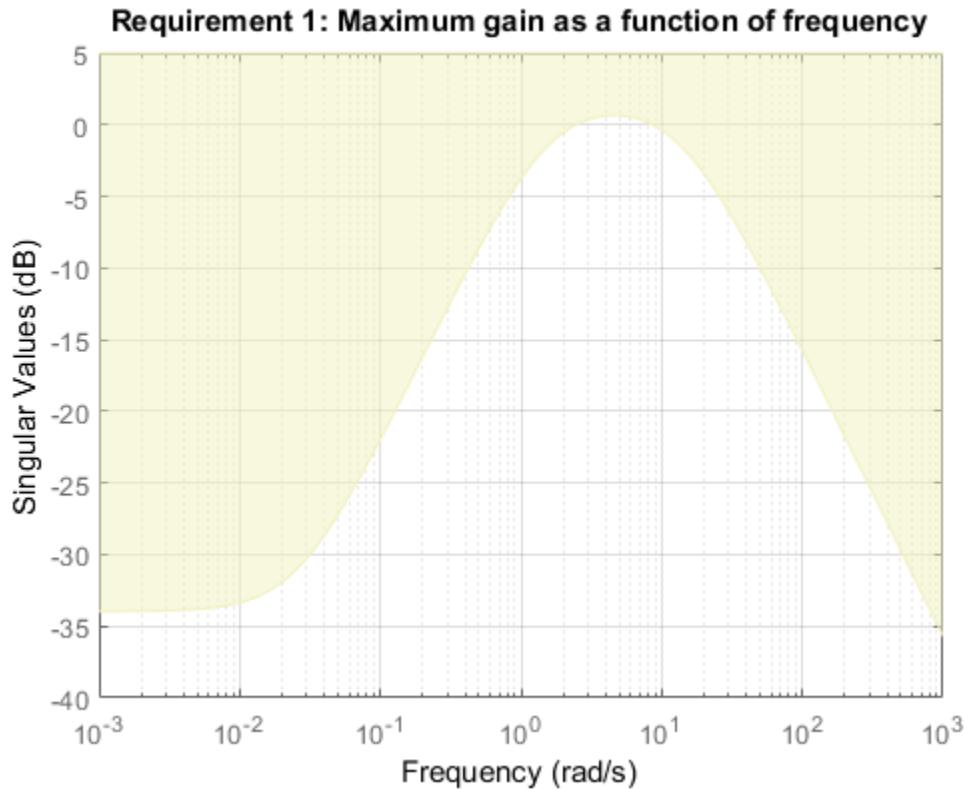
- γ loop: Track the setpoint with a 1 second response time, less than 2% steady-state error, and less than 30% peak error.

```
Req1 = TuningGoal.Tracking('gamma_ref','gamma',1,0.02,1.3);
viewSpec(Req1)
```

Requirement 1: Tracking error as a function of frequency
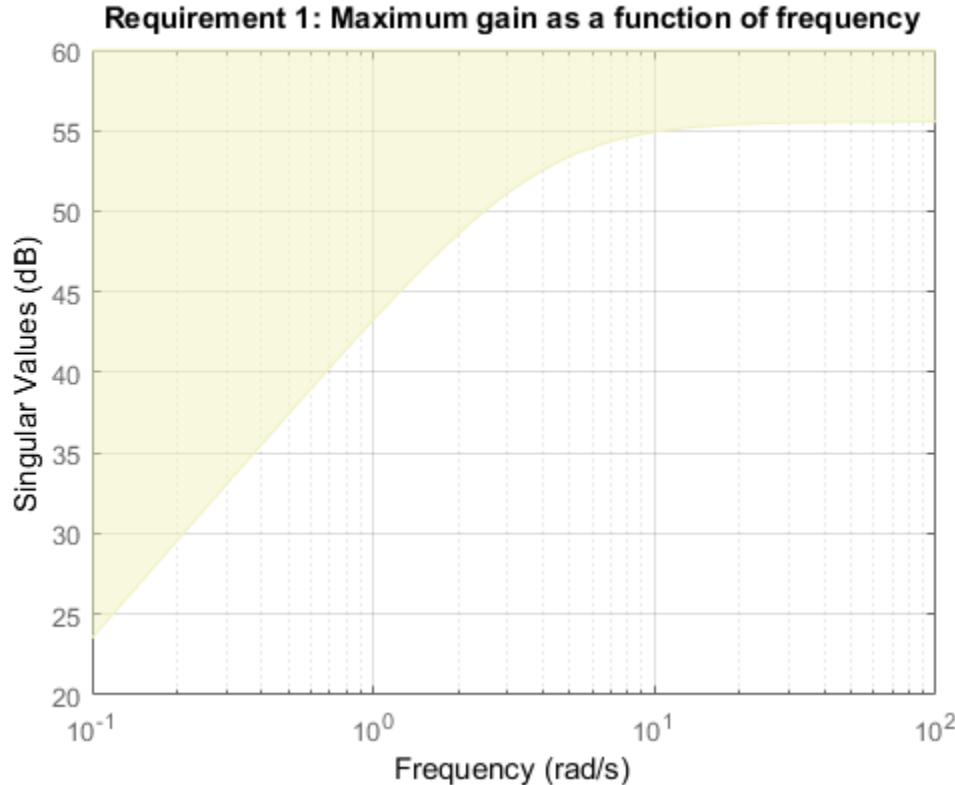
- a_z loop: Ensure good disturbance rejection at low frequency (to track acceleration demands) and past 10 rad/s (to be insensitive to measurement noise).

```
% Note: The disturbance is injected at the az_ref location
RejectionProfile = frd([0.02 0.02 1.2 1.2 0.1],[0 0.02 2 15 150]);
Req2 = TuningGoal.Gain('az_ref','az',RejectionProfile);
viewSpec(Req2)
```

Requirement 1: Maximum gain as a function of frequency

- q loop: Ensure good disturbance rejection up to 10 rad/s. The disturbance is injected at the plant input `delta`.

```
Req3 = TuningGoal.Gain('delta','az',600*tf([0.25 0],[0.25 1]));
viewSpec(Req3)
```

Requirement 1: Maximum gain as a function of frequency

- Transients: Ensure a minimum damping ratio of 0.35 for oscillation-free transients
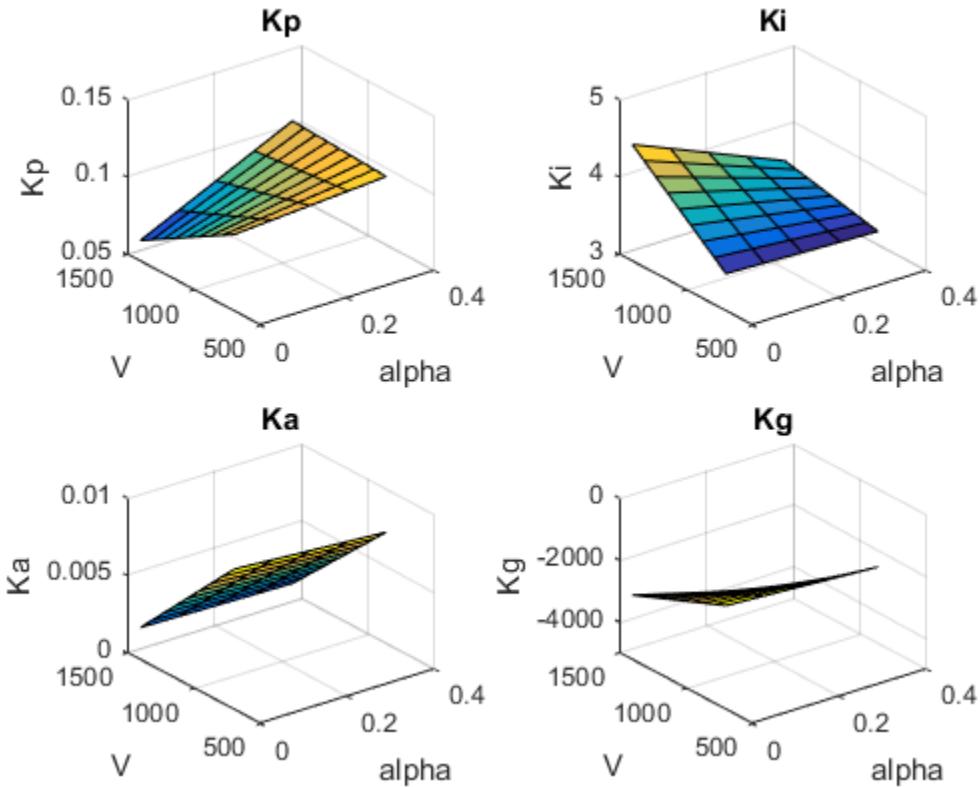
```
MinDamping = 0.35;
Req4 = TuningGoal.Poles(0,MinDamping);
```

Using `systune`, tune the 16 gain surface coefficients to best meet these performance requirements at all 45 flight conditions.

```
ST = systune(ST0,[Req1 Req2 Req3 Req4]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 78
```

The final value of the combined objective is close to 1, indicating that all requirements are nearly met. Visualize the resulting gain surfaces.
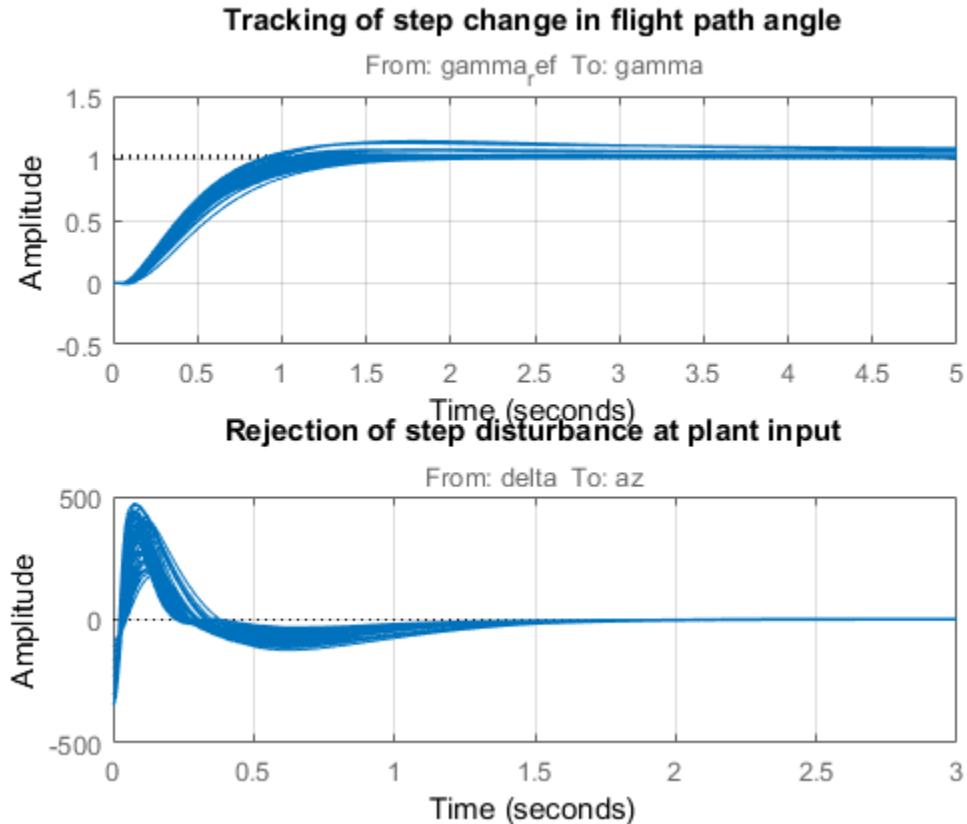
```
% Get tuned gain surfaces
TGS = getBlockParam(ST);

% Plot gain surfaces
clf
subplot(221), viewSurf(TGS.Kp), title('Kp')
subplot(222), viewSurf(TGS.Ki), title('Ki')
subplot(223), viewSurf(TGS.Ka), title('Ka')
subplot(224), viewSurf(TGS.Kg), title('Kg')
```

Validation

First validate the tuned autopilot at the 45 flight conditions considered above. Plot the response to a step change in flight path angle and the response to a step disturbance in elevator deflection.

```
clf
subplot(211), step(getIOTransfer(ST,'gamma_ref','gamma'),5), grid
title('Tracking of step change in flight path angle')
subplot(212), step(getIOTransfer(ST,'delta','az'),3), grid
title('Rejection of step disturbance at plant input')
```


The responses are satisfactory at all flight conditions. Next validate the autopilot against the nonlinear airframe model. First use `writeBlockValue` to apply the tuning results

to the Simulink model. This evaluates each gain surface formula at the breakpoints specified in the two Prelookup blocks and writes the result in the corresponding Interpolation block.

```
writeBlockValue(ST)
```
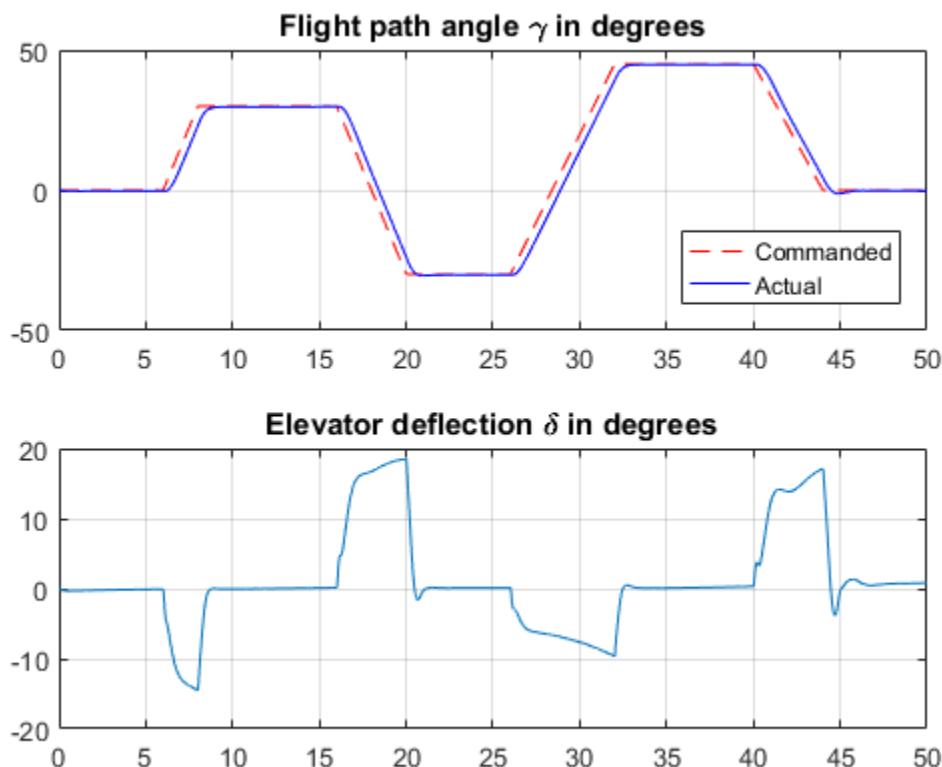
Now simulate the autopilot performance for a maneuver that takes the airframe through a large portion of its flight envelope. The code below is equivalent to pressing the Play button in the Simulink model and inspecting the responses in the Scope blocks.

```matlab
% Initial conditions
h_ini = 1000;
alpha_ini = 0;
v_ini = 700;

% Simulate
SimOut = sim('rct_airframeGS', 'ReturnWorkspaceOutputs', 'on');

% Extract simulation data
SimData = get(SimOut,'sigsOut');
Sim_gamma = getElement(SimData,'gamma');
Sim_alpha = getElement(SimData,'alpha');
Sim_V = getElement(SimData,'V');
Sim_delta = getElement(SimData,'delta');
Sim_h = getElement(SimData,'h');
Sim_az = getElement(SimData,'az');
t = Sim_gamma.Values.Time;

% Plot the main flight variables
clf
subplot(211)
plot(t,Sim_gamma.Values.Data(:,1),'r--',t,Sim_gamma.Values.Data(:,2),'b'), grid
legend('Commanded','Actual','location','SouthEast')
title('Flight path angle \gamma in degrees')
subplot(212)
plot(t,Sim_delta.Values.Data), grid
title('Elevator deflection \delta in degrees')
```
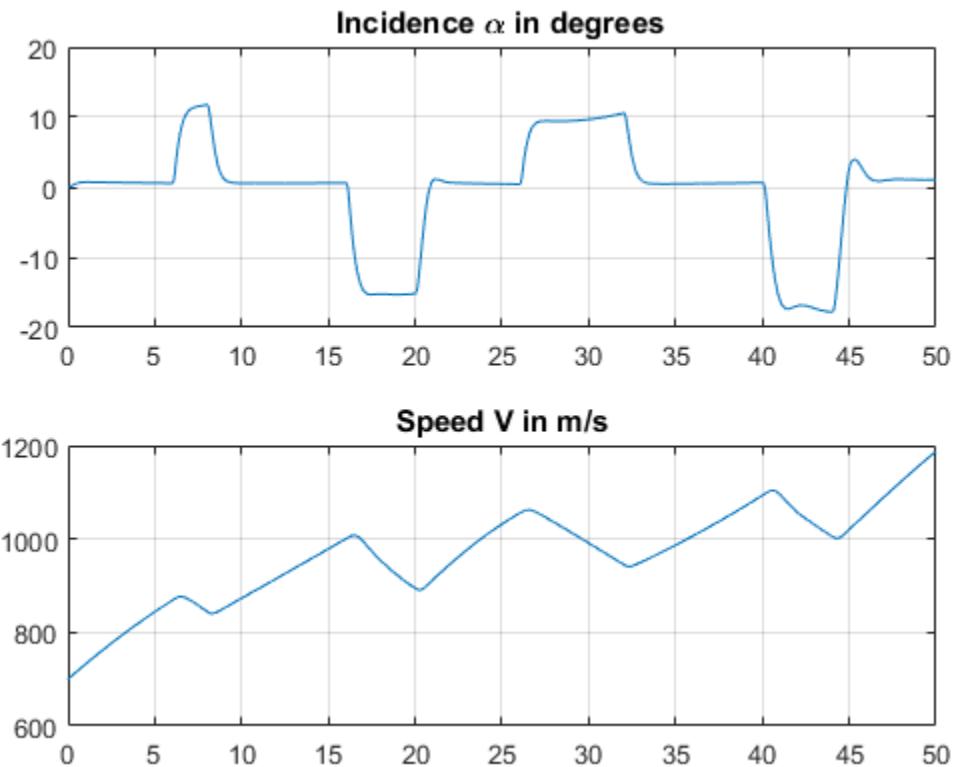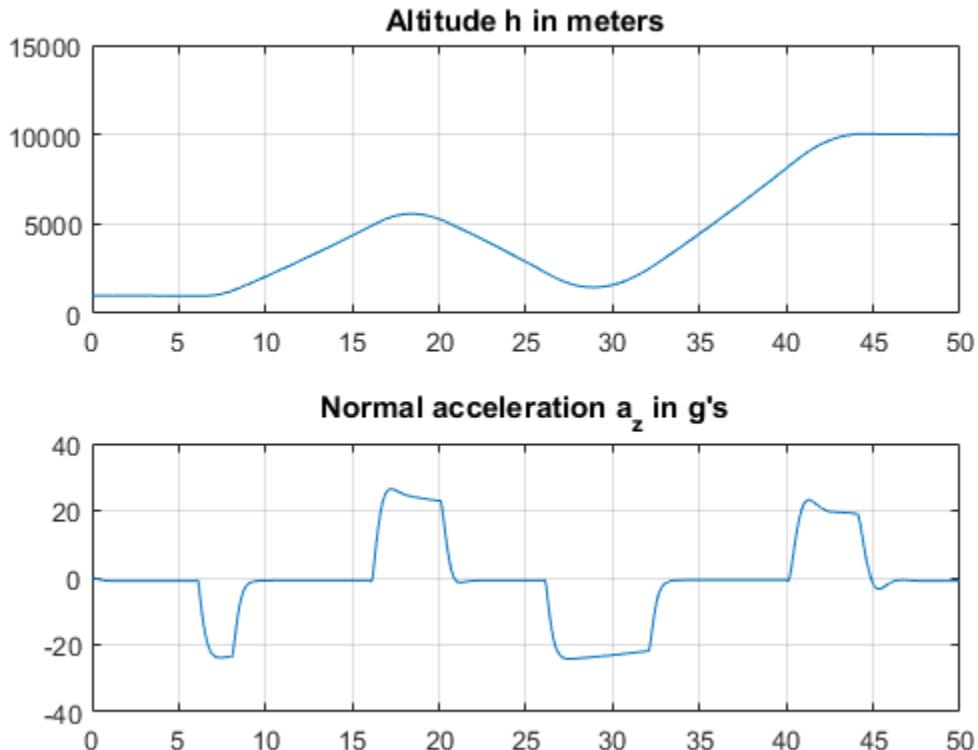
Flight path angle γ in degrees

Elevator deflection δ in degrees

```
subplot(211)
plot(t,Sim_alpha.Values.Data), grid
title('Incidence \alpha in degrees')
subplot(212)
plot(t,Sim_V.Values.Data), grid
title('Speed V in m/s')
```

Incidence α in degrees

Speed V in m/s


```
subplot(211)
plot(t,Sim_h.Values.Data), grid
title('Altitude h in meters')
subplot(212)
plot(t,Sim_az.Values.Data), grid
title('Normal acceleration a_z in g''s')
```

Altitude h in meters

Normal acceleration a_z in g's

Tracking of the flight path angle profile remains good throughout the maneuver. Note that the variations in incidence α and speed V cover most of the flight envelope considered here ([-20,20] degrees for α and [700,1400] for V). And while the autopilot was tuned for a nominal altitude of 3000 m, it fares well for altitude changing from 1,000 to 10,000 m.

The nonlinear simulation results confirm that the gain-scheduled autopilot delivers consistently high performance throughout the flight envelope. The "gain surface tuning" procedure provides simple explicit formulas for the gain dependence on the scheduling

variables. Instead of using look-up tables, you can use these formulas directly for an more memory-efficient hardware implementation.

See Also
`setBlockParam` | `slTuner` | `tunableSurface`

Related Examples
- "Set Up Simulink Models for Gain Scheduling" on page 15-17
- "Gain-Scheduled Control of a Chemical Reactor" on page 15-47

More About
- "Gain-Scheduled Control Systems" on page 15-2
- "Parametric Gain Surfaces" on page 15-10
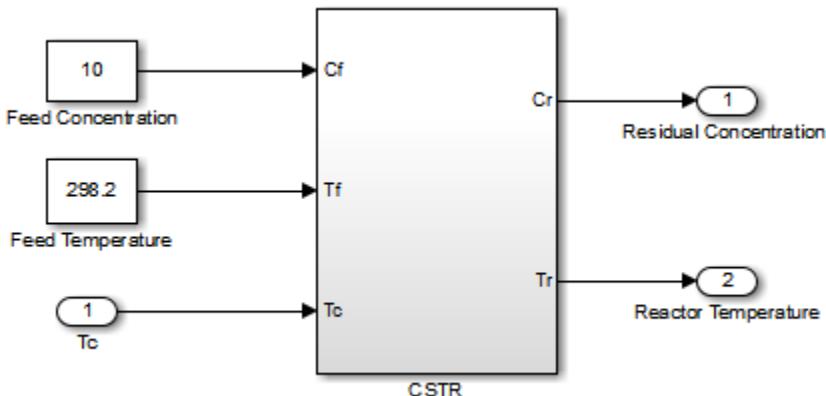
Gain-Scheduled Control of a Chemical Reactor

This example shows how to design and tune a gain-scheduled controller for a chemical reactor transitioning from low to high conversion rate. For background, see Seborg, D.E. et al., "Process Dynamics and Control", 2nd Ed., 2004, Wiley, pp. 34-36.

Continuous Stirred Tank Reactor

The process considered here is a continuous stirred tank reactor (CSTR) during transition from low to high conversion rate (high to low residual concentration). Because the chemical reaction is exothermic (produces heat), the reactor temperature must be controlled to prevent a thermal runaway. The control task is complicated by the fact that the process dynamics are nonlinear and transition from stable to unstable and back to stable as the conversion rate increases. The reactor dynamics are modeled in Simulink. The controlled variables are the residual concentration Cr and the reactor temperature Tr, and the manipulated variable is the temperature Tc of the coolant circulating in the reactor's cooling jacket.

```
open_system('rct_CSTR_OL')
```
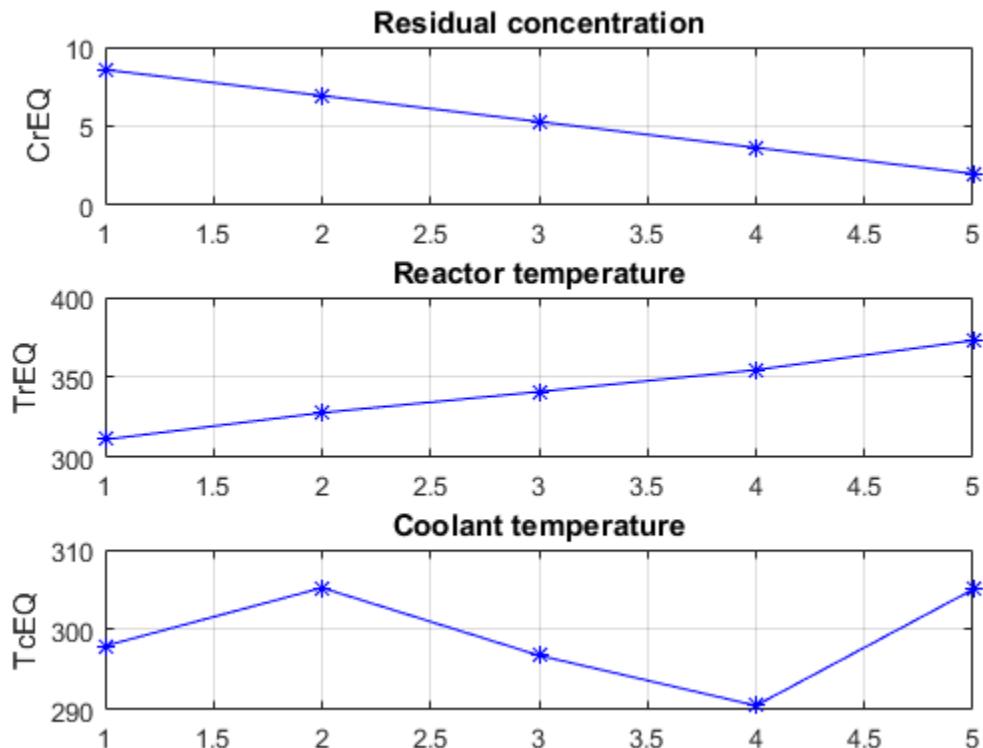

We want to transition from a residual concentration of 8.57 kmol/m^3 initially down to 2 kmol/m^3. To understand how the process dynamics evolve with the residual concentration Cr, find the equilibrium conditions for five values of Cr between 8.57 and 2 and linearize the process dynamics around each equilibrium. Log the reactor and coolant temperatures at each equilibrium point.

```
clear G
CrEQ = linspace(8.57,2,5)';  % concentrations
TrEQ = zeros(5,1);           % reactor temperatures
TcEQ = zeros(5,1);           % coolant temperatures
for k=1:5
   opspec = operspec('rct_CSTR_OL');
   % Set desired residual concentration
   opspec.Outputs(1).y = CrEQ(k);
   opspec.Outputs(1).Known = true;
   % Compute equilibrium condition
   [op,report] = findop('rct_CSTR_OL',opspec,...
                       findopOptions('DisplayReport','off'));
   % Log temperatures
   TrEQ(k) = report.Outputs(2).y;
   TcEQ(k) = op.Inputs.u;
   % Linearize process dynamics
   G(:,:,k) = linearize('rct_CSTR_OL', 'rct_CSTR_OL/CSTR', op);
end
G.InputName = {'Cf','Tf','Tc'};
G.OutputName = {'Cr','Tr'};
```

Plot the reactor and coolant temperatures at equilibrium as a function of concentration.

```
subplot(311), plot(CrEQ,'b-*'), grid, title('Residual concentration'), ylabel('CrEQ')
subplot(312), plot(TrEQ,'b-*'), grid, title('Reactor temperature'), ylabel('TrEQ')
subplot(313), plot(TcEQ,'b-*'), grid, title('Coolant temperature'), ylabel('TcEQ')
```

An open-loop control strategy consists of following the coolant temperature profile above to smoothly transition between the `Cr=8.57` and `Cr=2` equilibria. However, this strategy is doomed by the fact that the reaction is unstable in the mid range and must be properly cooled to avoid thermal runaway. This is confirmed by inspecting the poles of the linearized models for the five equilibrium points considered above (three out of the five models are unstable).

```
pole(G)
```

```
ans(:,:,1) =

  -0.5225 + 0.0000i
  -0.8952 + 0.0000i
```

```
ans(:,:,2) =

   0.1733 + 0.0000i
  -0.8866 + 0.0000i


ans(:,:,3) =

   0.5114 + 0.0000i
  -0.8229 + 0.0000i


ans(:,:,4) =

   0.0453 + 0.0000i
  -0.4991 + 0.0000i


ans(:,:,5) =

  -1.1077 + 1.0901i
  -1.1077 - 1.0901i
```
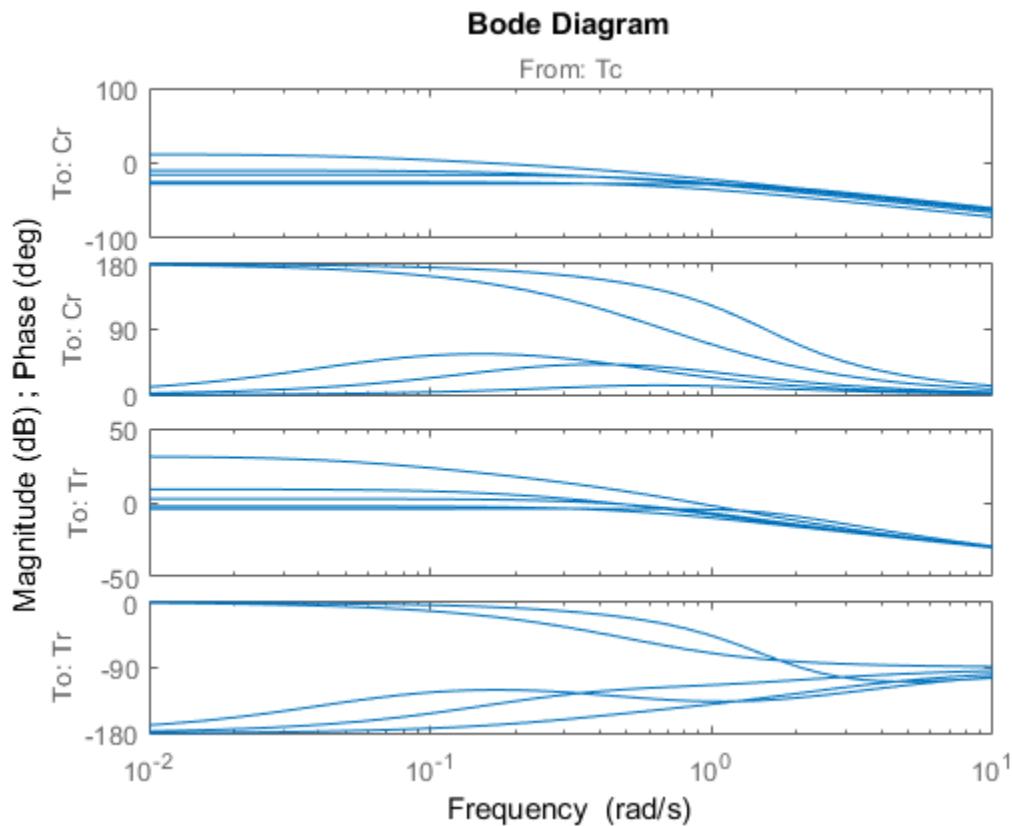
The Bode plot further highlights the significant variations in plant dynamics while transitioning from Cr=8.57 to Cr=2.

```
clf, bode(G(:,'Tc'),{0.01,10})
```

Bode Diagram
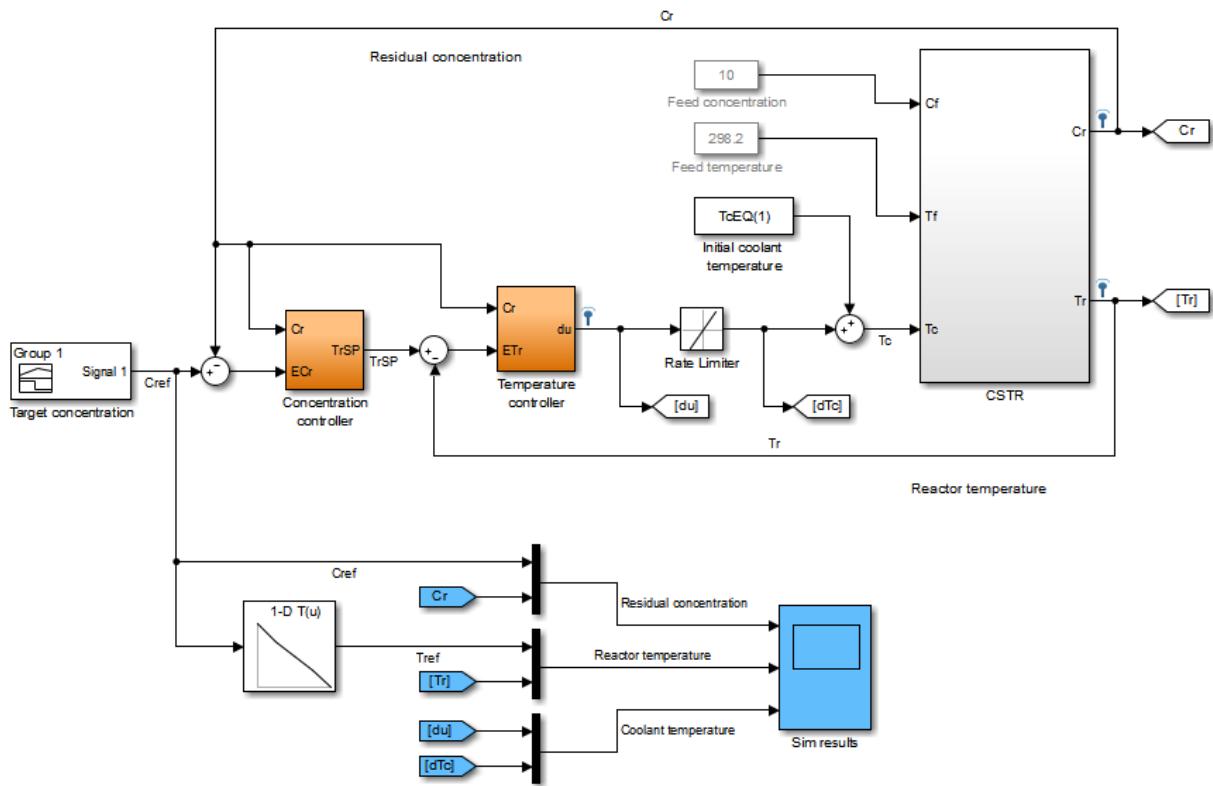
Feedback Control Strategy

To prevent thermal runaway while ramping down the residual concentration, use feedback control to adjust the coolant temperature Tc based on measurements of the residual concentration Cr and reactor temperature Tr. For this application, we use a cascade control architecture where the inner loop regulates the reactor temperature and the outer loop tracks the concentration setpoint. Both feedback loops are digital with a sampling period of 0.5 minutes.

```
open_system('rct_CSTR')
```
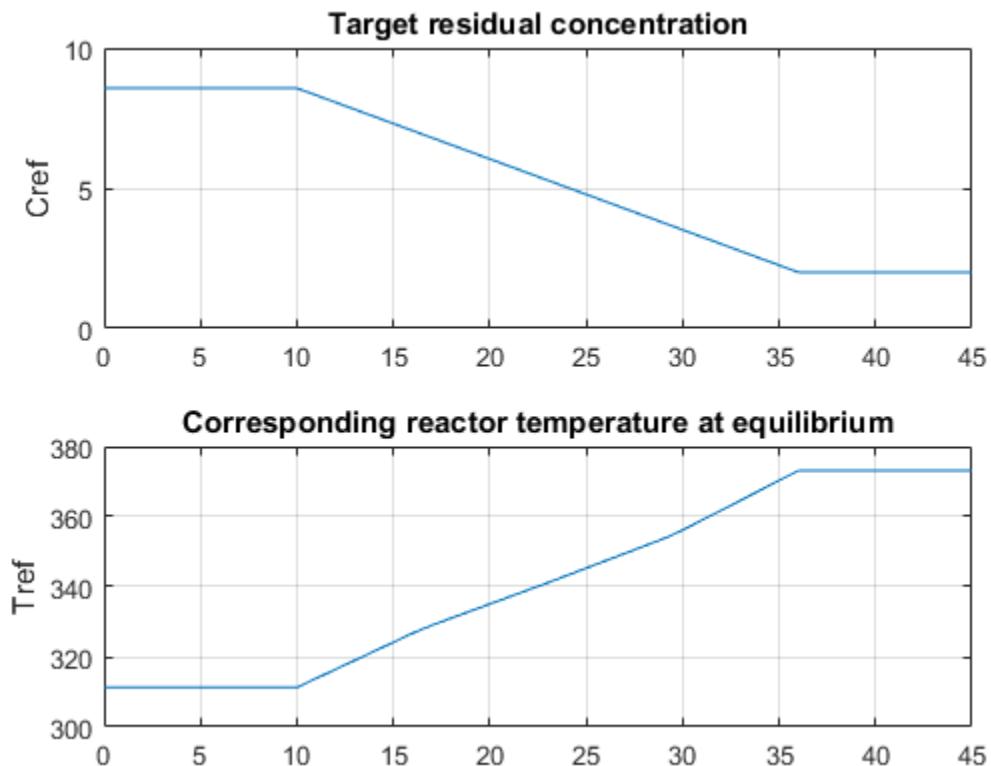
The target concentration `Cref` ramps down from 8.57 kmol/m^3 at t=10 to 2 kmol/m^3 at t=36 (the transition lasts 26 minutes). The corresponding profile `Tref` for the reactor temperature is obtained by interpolating the equilibrium values `TrEQ` from trim analysis. The controller computes the coolant temperature adjustment `dTc` relative to the initial equilibrium value `TcEQ(1)=297.98` for `Cr=8.57`. Note that the model is set up so that initially, the output `TrSP` of the "Concentration controller" block matches the reactor temperature, the adjustment `dTc` is zero, and the coolant temperature `Tc` is at its equilibrium value `TcEQ(1)`.

```
clf
t = [0 10:36 45];
C = interp1([0 10 36 45],[8.57 8.57 2 2],t);
subplot(211), plot(t,C), grid, set(gca,'ylim',[0 10])
title('Target residual concentration'), ylabel('Cref')
```

```
subplot(212), plot(t,interp1(CrEQ,TrEQ,C))
title('Corresponding reactor temperature at equilibrium'), ylabel('Tref'), grid
```


Control Objectives

Use `TuningGoal` objects to capture the design requirements. First, `Cr` should follow setpoints `Cref` with a response time of about 5 minutes.

```
R1 = TuningGoal.Tracking('Cref','Cr',5);
```

The inner loop (temperature) should stabilize the reaction dynamics with sufficient damping and fast enough decay.

```
MinDecay = 0.2;
MinDamping = 0.5;
% Constrain closed-loop poles of inner loop with the outer loop open
R2 = TuningGoal.Poles('Tc',MinDecay,MinDamping);
R2.Openings = 'TrSP';
```

The Rate Limit block at the controller output specifies that the coolant temperature `Tc` cannot vary faster than 10 degrees per minute. This is a severe limitation on the controller authority which, when ignored, can lead to poor performance or instability. To take this rate limit into account, observe that `Cref` varies at a rate of 0.25 kmol/m^3/min. To ensure that `Tc` does not vary faster than 10 degrees/min, the gain from `Cref` to `Tc` should be less than 10/0.25=40.

```
R3 = TuningGoal.Gain('Cref','Tc',40);
```

Finally, require at least 7 dB of gain margin and 45 degrees of phase margin at the plant input `Tc`.

```
R4 = TuningGoal.Margins('Tc',7,45);
```

Gain-Scheduled Controller

To achieve these requirements, we use a PI controller in the outer loop and a lead compensator in the inner loop. Due to the slow sampling rate, the lead compensator is needed to adequately stabilize the chemical reaction at the mid-range concentration `Cr = 5.28` kmol/m^3/min. Because the reaction dynamics vary substantially with concentration, we further schedule the controller gains as a function of concentration. This is modeled in Simulink using Lookup Table blocks as shown in Figures 1 and 2.
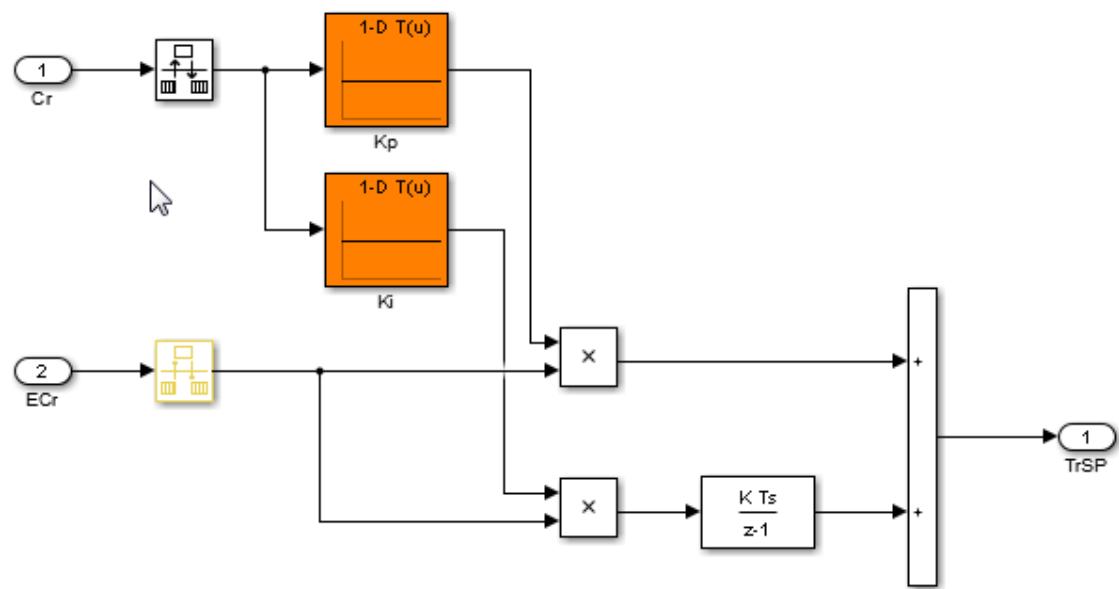
Figure 1: Gain-scheduled PI controller for concentration loop.
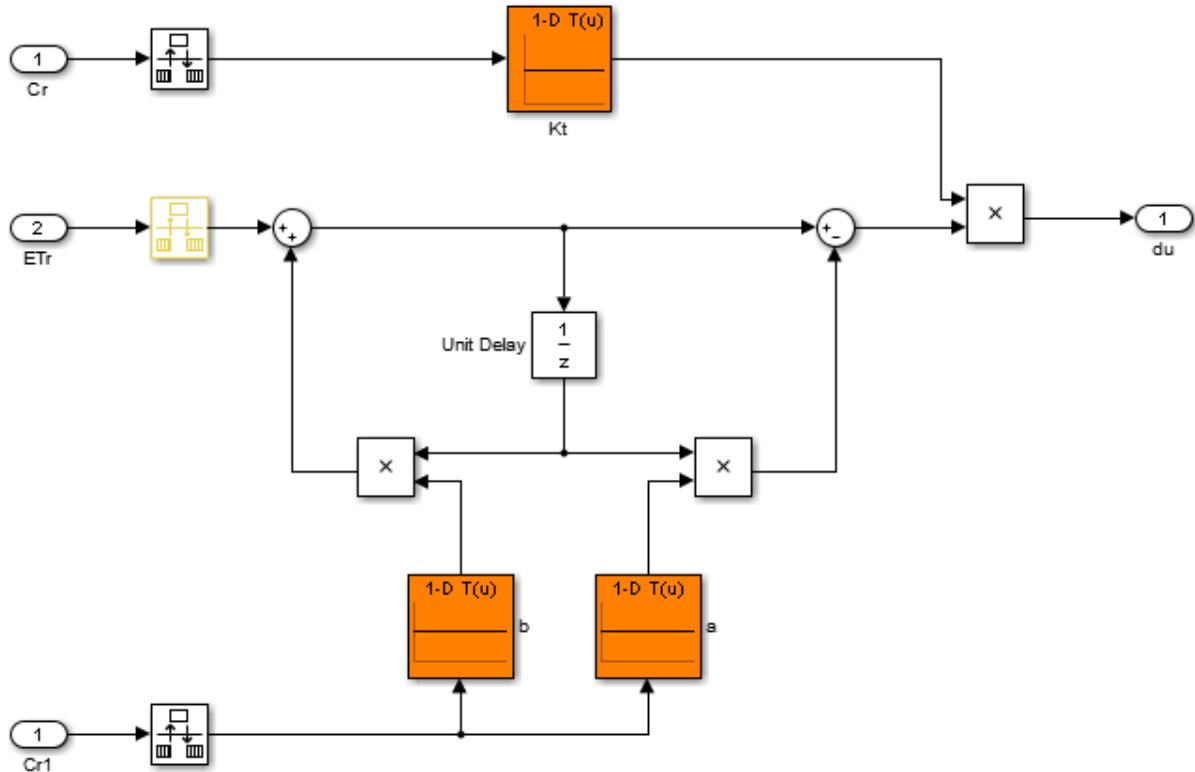
Figure 2: Gain-scheduled lead compensator for temperature loop.

Tuning this gain-scheduled controller amounts to tuning the look-up table data over a range of concentration values. Rather than tuning individual look-up table entries, parameterize the controller gains Kp,Ki,Kt,a,b as quadratic polynomials in Cr, for example,

$$K_p(C_r) = K_{p0} + K_{p1}C_r + K_{p2}C_r^2.$$

Besides reducing the number of variables to tune, this approach ensures smooth gain transitions as Cr varies. Using systune, you can automatically tune the coefficients $K_{p0}, K_{p1}, K_{p2}, K_{i0}, \cdots$ to meet the requirements R1-R4 at the five equilibrium points

computed above. This amounts to tuning the gain-scheduled controller at five design points along the Cref trajectory. Use the tunableSurface object to parameterize each gain as a quadratic function of Cr. The "tuning grid" is set to the five concentrations

CrEQ and the basis functions for the quadratic parameterization are C_r, C_r^2. Most gains are initialized to be identically zero.

```
TuningGrid = struct('Cr',CrEQ);
ShapeFcn = @(Cr) [Cr , Cr^2];

Kp = tunableSurface('Kp', 0, TuningGrid, ShapeFcn);
Ki = tunableSurface('Ki', -2, TuningGrid, ShapeFcn);
Kt = tunableSurface('Kt', 0, TuningGrid, ShapeFcn);
a = tunableSurface('a', 0, TuningGrid, ShapeFcn);
b = tunableSurface('b', 0, TuningGrid, ShapeFcn);
```

Controller Tuning

Because the target bandwidth is within a decade of the Nyquist frequency, it is easier to tune the controller directly in the discrete domain. Discretize the linearized process dynamics with sample time of 0.5 minutes. Use the ZOH method to reflect how the digital controller interacts with the continuous-time plant.

```
Ts = 0.5;
Gd = c2d(G,Ts);
```

Create an slTuner interface for tuning the quadratic gain schedules introduced above. Use block substitution to replace the nonlinear plant model by the five discretized linear models Gd obtained at the design points CrEQ. Use setBlockParam to associate the tunable gain functions Kp, Ki, Kt, a, b with the Lookup Table blocks of the same name.

```
BlockSubs = struct('Name','rct_CSTR/CSTR','Value',Gd);
STO = slTuner('rct_CSTR',{'Kp','Ki','Kt','a','b'},BlockSubs);
STO.Ts = Ts;  % sample time for tuning

% Register points of interest
STO.addPoint({'Cref','Cr','Tr','TrSP','Tc'})

% Parameterize look-up table blocks
STO.setBlockParam('Kp',Kp);
STO.setBlockParam('Ki',Ki);
STO.setBlockParam('Kt',Kt);
STO.setBlockParam('a',a);
STO.setBlockParam('b',b);
```
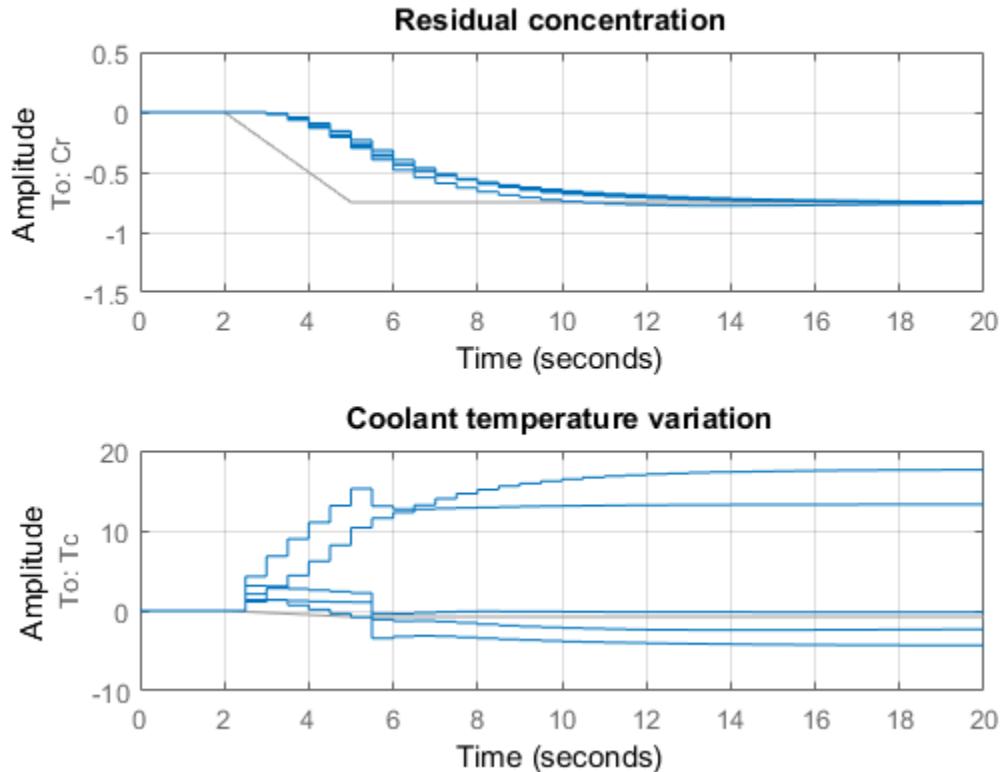
You can now use `systune` to tune the controller coefficients against the requirements `R1`-`R4`. Make the stability margin requirement a hard constraints and optimize the remaining requirements.

```
ST = systune(ST0,[R1 R2 R3],R4);
```

```
Final: Soft = 1.21, Hard = 1, Iterations = 238
```

The resulting design satisfies the hard constraint (`Hard<1`) and nearly satisfies the remaining requirements (`Soft` close to 1). To validate this design, simulate the responses to a ramp in concentration with the same slope as `Cref`. Each plot shows the linear responses at the five design points `CrEQ`.

```
t = 0:Ts:20;
uC = interp1([0 2 5 20],(-0.25)*[0 0 3 3],t);
subplot(211), lsim(getIOTransfer(ST,'Cref','Cr'),uC)
grid, set(gca,'ylim',[-1.5 0.5]), title('Residual concentration')
subplot(212), lsim(getIOTransfer(ST,'Cref','Tc'),uC)
grid, title('Coolant temperature variation')
```

Note that rate of change of the coolant temperature remains within the physical limits (10 degrees per minute or 5 degrees per sample period).
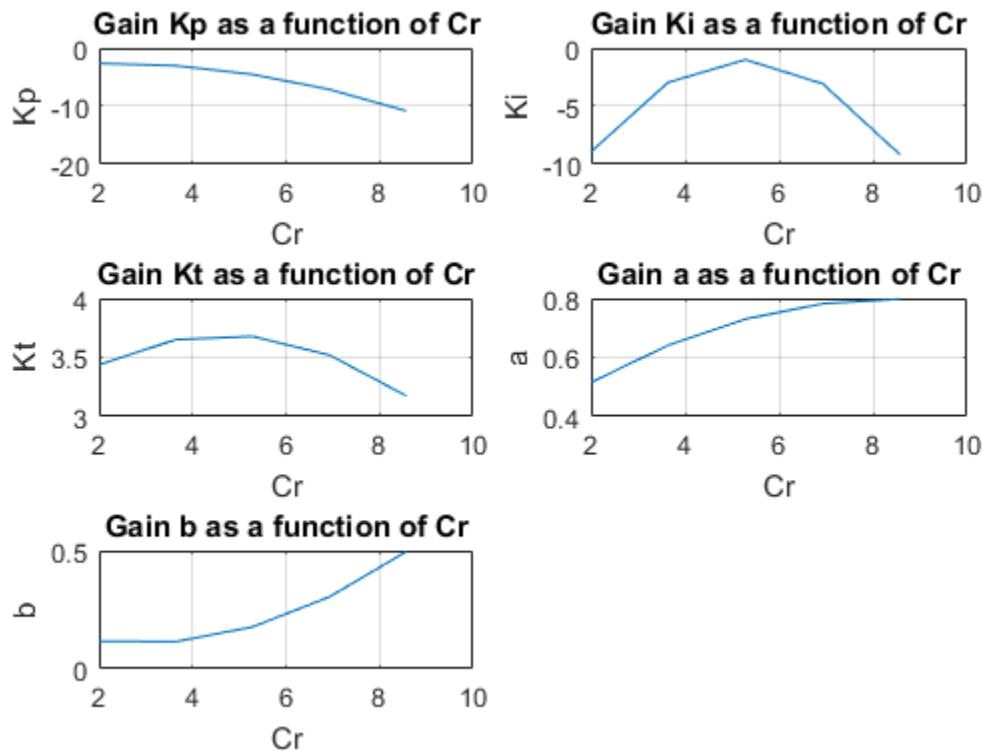
Controller Validation

Inspect how each gain varies with `Cr` during the transition.

```
% Access tuned gain schedules
TGS = getBlockParam(ST);

% Plot gain profiles
clf
subplot(321), viewSurf(TGS.Kp), ylabel('Kp')
subplot(322), viewSurf(TGS.Ki), ylabel('Ki')
```

```
subplot(323), viewSurf(TGS.Kt), ylabel('Kt')
subplot(324), viewSurf(TGS.a), ylabel('a')
subplot(325), viewSurf(TGS.b), ylabel('b')
```


To validate the gain-scheduled controller in Simulink, first use `writeBlockValue` to apply the tuning results to the Simulink model. For each Lookup Table block, this evaluates the corresponding quadratic gain formula at the table breakpoints and updates the table data accordingly.

```
writeBlockValue(ST)
```

Next push the Play button to simulate the response with the tuned gain schedules. The simulation results appear in Figure 3. The gain-scheduled controller successfully drives the reaction through the transition with adequate response time and no saturation

of the rate limits (controller output `du` matches effective temperature variation `dTc`). The reactor temperature stays close to its equilibrium value `Tref`, indicating that the controller keeps the reaction near equilibrium while preventing thermal runaway.
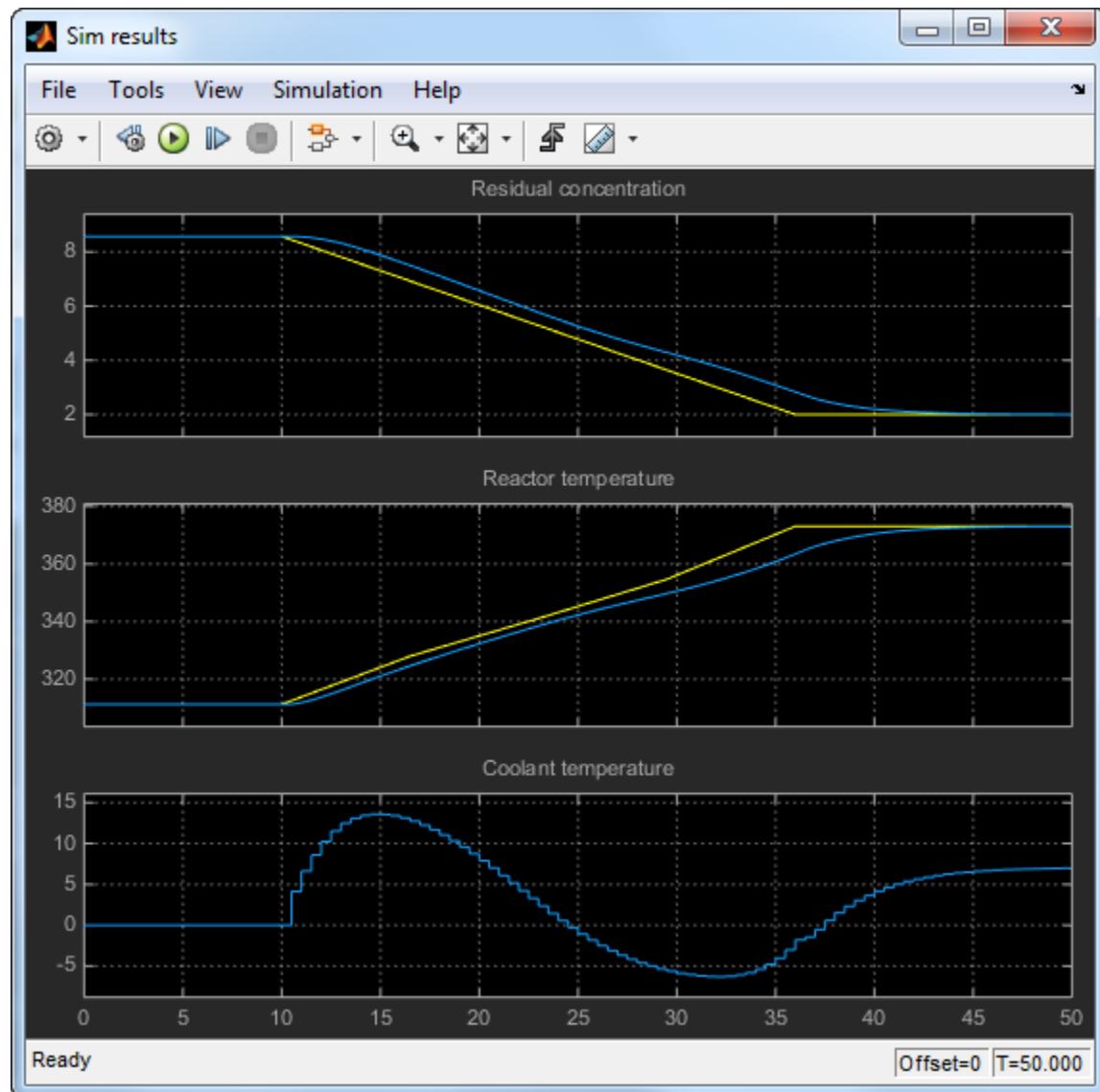
Figure 3: Transition with gain-scheduled cascade controller.

Controller Tuning in MATLAB

Alternatively, you can tune the gain schedules directly in MATLAB without using the `slTuner` interface. First parameterize the gains as quadratic functions of `Cr` as done above.

```
TuningGrid = struct('Cr',CrEQ);
ShapeFcn = @(Cr) [Cr , Cr^2];

Kp = tunableSurface('Kp', 0, TuningGrid, ShapeFcn);
Ki = tunableSurface('Ki', -2, TuningGrid, ShapeFcn);
Kt = tunableSurface('Kt', 0, TuningGrid, ShapeFcn);
a = tunableSurface('a', 0, TuningGrid, ShapeFcn);
b = tunableSurface('b', 0, TuningGrid, ShapeFcn);
```

Use these gains to build the PI and lead controllers.

```
PI = pid(Kp,Ki,'Ts',Ts,'TimeUnit','min');
PI.u = 'ECr';   PI.y = 'TrSP';

LEAD = Kt * tf([1 -a],[1 -b],Ts,'TimeUnit','min');
LEAD.u = 'ETr';   LEAD.y = 'Tc';
```

Use `connect` to build a closed-loop model of the overall control system at the five design points. Mark the controller outputs `TrSP` and `Tc` as "analysis points" so that loops can be opened and stability margins evaluated at these locations. The closed-loop model `T0` is a 5-by-1 array of linear models depending on the tunable coefficients of `Kp,Ki,Kt,a,b`. Each model is discrete and sampled every half minute.

```
Gd.TimeUnit = 'min';
S1 = sumblk('ECr = Cref - Cr');
S2 = sumblk('ETr = TrSP - Tr');
T0 = connect(Gd(:,'Tc'),LEAD,PI,S1,S2,'Cref','Cr',{'TrSP','Tc'});
```

Finally, use `systune` to tune the gain schedule coefficients.

```
T = systune(T0,[R1 R2 R3],R4);

Final: Soft = 1.21, Hard = 0.9995, Iterations = 242
```
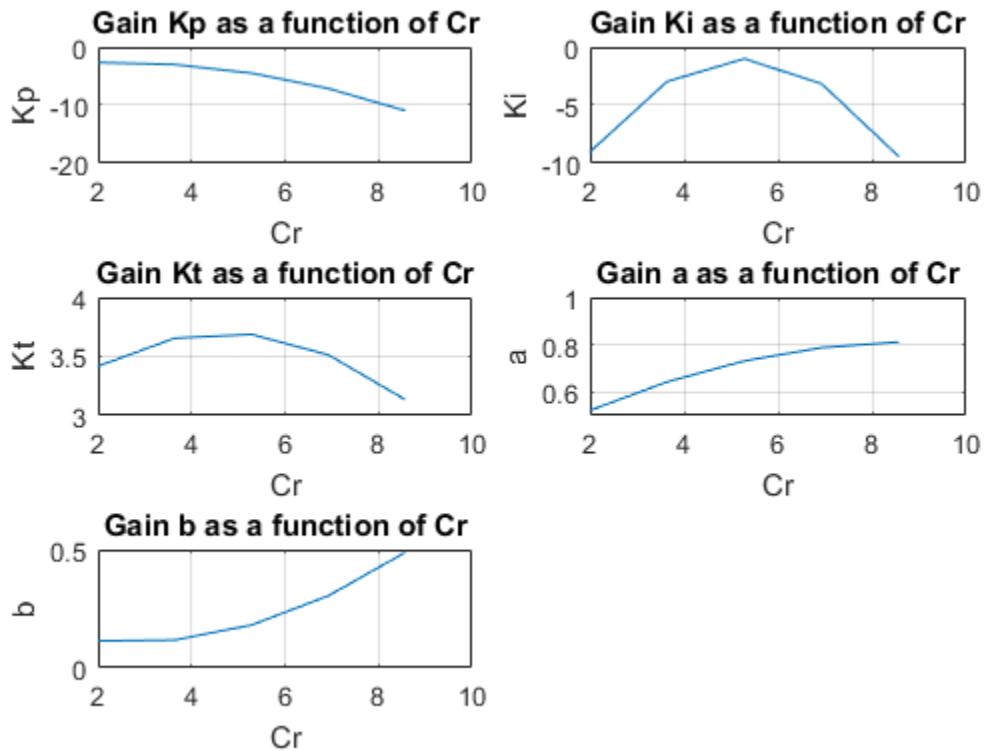
The result is similar to the one obtained above. Confirm by plotting the gains as a function of `Cr` using the tuned coefficients in `T`.

```
clf
subplot(321), viewSurf(setBlockValue(Kp,T)), ylabel('Kp')
subplot(322), viewSurf(setBlockValue(Ki,T)), ylabel('Ki')
subplot(323), viewSurf(setBlockValue(Kt,T)), ylabel('Kt')
subplot(324), viewSurf(setBlockValue(a,T)), ylabel('a')
subplot(325), viewSurf(setBlockValue(b,T)), ylabel('b')
```


You can further validate the design by simulating the linear responses at each design point. However, you need to return to Simulink to simulate the nonlinear response of the gain-scheduled controller.

See Also

setBlockParam | slTuner | tunableSurface

Related Examples

More About

Customization

Preliminaries

Terminology

You can use the Control System Toolbox editors to set properties and preferences in the SISO Design Tool, the Linear System Analyzer, and in any response plots that you create from the MATLAB prompt.

Properties refer to settings that are specific to an individual response plot. This includes the following:

- Axes labels, and limits
- Data units and scales
- Plot styles, such as grids, fonts, and axes foreground colors
- Plot characteristics, such as rise time, peak response, and gain and phase margins

Preferences refers to properties that persist either

- Within a single session for a specific instance of a Linear System Analyzer or a SISO Design Tool
- Across Control System Toolbox sessions

The former are called *tool preferences*, the latter *toolbox preferences*.
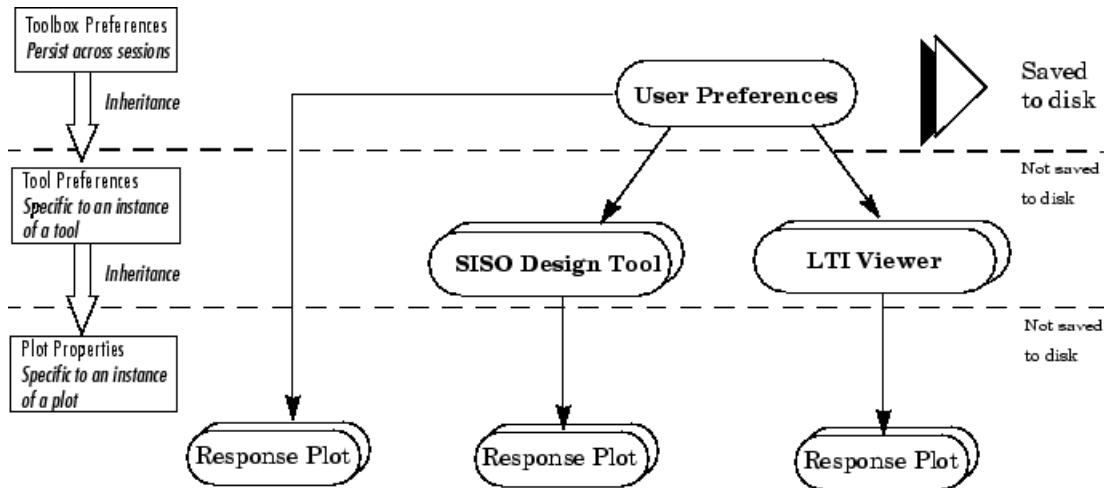
Property and Preferences Hierarchy

You can use three graphical user interfaces (GUIs) to control over the visualization of time and frequency plots generated by the toolbox:

- Toolbox Preferences
- Tool Preferences
- Plot Properties

Preferences refer to global options that you can save from session to session or to any Linear System Analyzer or SISO Design Tool that you open during a single session. *Properties* are options that apply only to the current window. This section gives an overview of the three GUIs.

Although you can set plot properties in any response plot, you can use the Toolbox Preferences Editor to set properties for any response plot that you generate. This figure shows the inheritance hierarchy from toolbox preference to plot properties.

Preference and Property Inheritance Hierarchy

You can activate preference and plot editors by doing the following:

- Toolbox preferences — Select **Toolbox Preferences** under **File** in either the Linear System Analyzer or the SISO Design Tool.

- Tool preferences — Select **SISO Tool Preferences** under **Edit** for the SISO Design Tool and **Viewer Preferences** under **Edit** in the Linear System Analyzer.

- Plot properties — Double-click any Control System Toolbox response plot or select **Properties** from the right-click menus.

Ways to Customize Plots

You can customize your plots by changing plot properties. For example, you can change the plot units. The following table describes ways that you can customize plots.

To change plot properties of	For more information, see
A single plot, directly from the plot	• "Customizing Response Plots Using the Response Plots Property Editor" on page 19-3 and "Customizing Response Plots Using Plot Tools" on page 19-20 for response plots • "Linear System Analyzer Preferences Editor" on page 18-2 for Linear System Analyzer plots
A single plot or many plots, programmatically from the command line	"Customizing Response Plots from the Command Line" on page 19-24
All Control System Toolbox plots (changes apply globally to all plot types and persist from session to session)	"Toolbox Preferences Editor" on page 17-2

Setting Toolbox Preferences

Toolbox Preferences Editor

In this section...
"Overview of the Toolbox Preferences Editor" on page 17-2
"Opening the Toolbox Preferences Editor" on page 17-2

Overview of the Toolbox Preferences Editor
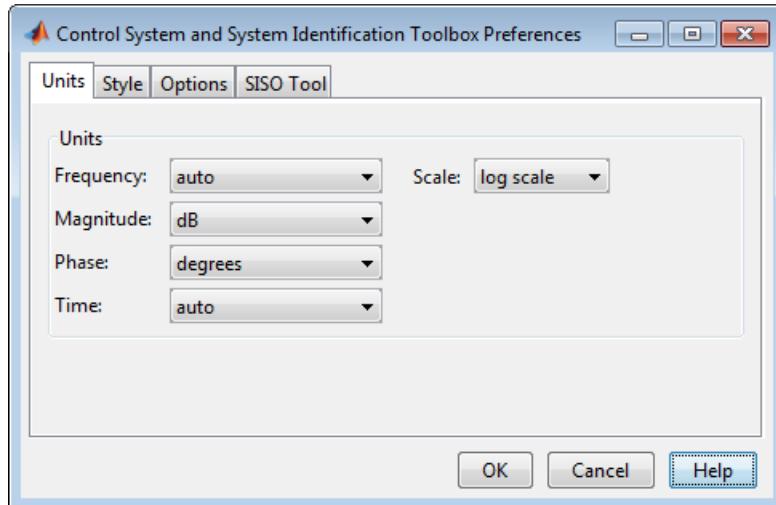
The Toolbox Preferences editor allows you to set plot preferences that will persist from session to session.

Opening the Toolbox Preferences Editor

To open the Toolbox Preferences editor, select **Toolbox Preferences** from the **File** menu of the Linear System Analyzer or the SISO Design Tool. Alternatively, you can type

```
ctrlpref
```
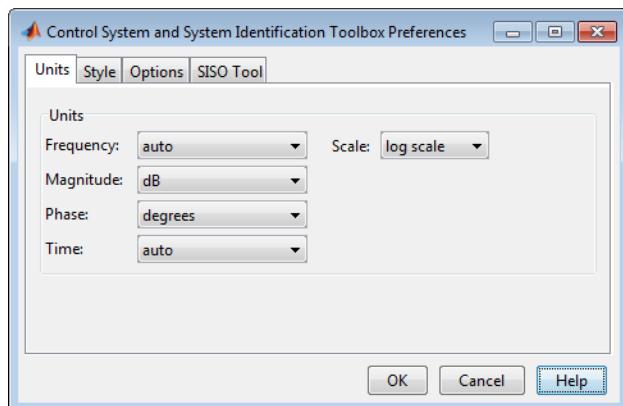
at the MATLAB prompt.

Control System Toolbox Preferences Editor

- "Units Pane" on page 17-4
- "Style Pane" on page 17-7
- "Options Pane" on page 17-8
- "Control System DesignerPane" on page 17-9

Units Pane

Use the **Units** pane to set preferences for the following:

- **Frequency**

  The default auto option uses rad/TimeUnit as the frequency units relative to the system time units, where TimeUnit is the system time units specified in the TimeUnit property of the system on frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  For the frequency axis, you can select logarithmic or linear scales.

  Other Frequency Units Options

  - 'Hz'
  - 'rad/s'
  - 'rpm'
  - 'kHz'
  - 'MHz'
  - 'GHz'
  - 'rad/nanosecond'
  - 'rad/microsecond'
  - 'rad/millisecond'

- `'rad/minute'`
- `'rad/hour'`
- `'rad/day'`
- `'rad/week'`
- `'rad/month'`
- `'rad/year'`
- `'cycles/nanosecond'`
- `'cycles/microsecond'`
- `'cycles/millisecond'`
- `'cycles/hour'`
- `'cycles/day'`
- `'cycles/week'`
- `'cycles/month'`
- `'cycles/year'`

- **Magnitude** — Decibels (dB) or absolute value (abs)
- **Phase** — Degrees or radians
- **Time**

  The default `auto` option uses the time units specified in the `TimeUnit` property of the system on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  ### Other Time Units Options

  - `'nanoseconds'`
  - `'microseconds'`
  - `'milliseconds'`
  - `'seconds'`
  - `'minutes'`
  - `'hours'`
  - `'days'`
  - `'weeks'`

- `'months'`
- `'years'`

Style Pane

Use the **Style** pane to toggle grid visibility and set font preferences and axes foreground colors for all plots you create. This figure shows the Style pane.
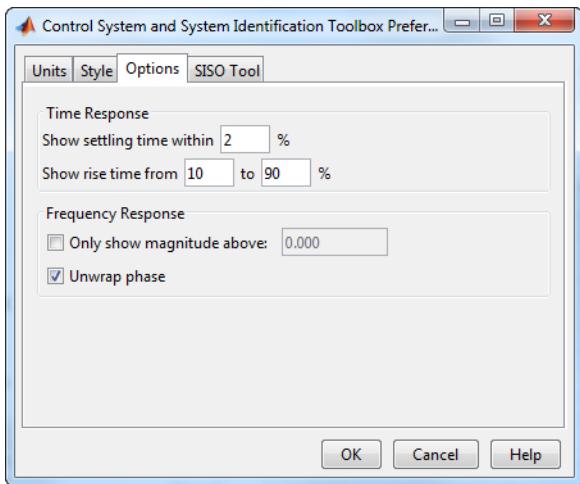
You have the following choices:

- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic). Select font sizes from the menus or type any font-size values in the fields.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

  If you do not want to specify RGB values numerically, click the **Select** button to open the **Select Colors** dialog box.

Options Pane

The Options pane has selections for time responses and frequency responses. This figure shows the Options pane with default settings.
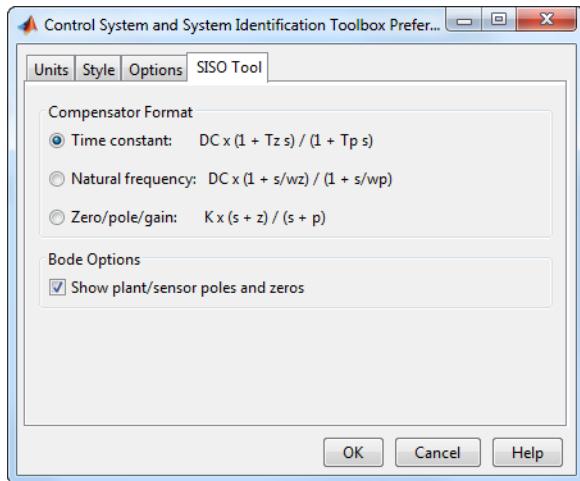
The following are the available options for the Options pane:

- **Time Response**:

  - Show settling time within *xx*%— You can set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.

  - Specify rise time from *xx*% to *yy*%— The standard definition of rise time is the time it takes the signal to go from 10% to 90% of the final value. You can choose any percentages you like (from 0% to 100%), provided that the first value is smaller than the second.

- **Frequency Response:**

  - Only show magnitude above *xx*—Specify a lower limit for magnitude values in response plots so that you can focus on a region of interest.

  - Unwrap phase—By default, the phase is unwrapped. Wrap the phrase by clearing this box. If the phase is wrapped, all phase values are shifted such that their equivalent value displays in the range [0°, 360°).

Control System DesignerPane

The Control System Designer pane has settings for the **Control System Designer**. This figure shows the Control System Designer pane with default settings.

You can make the following selections:

- **Compensator Format** — Select the time constant, natural frequency, or zero/pole/gain format. The time constant format is a factorization of the compensator transfer function of the form

$$K \times \frac{\left(1 + T_{z_1} s\right)\left(1 + T_{z_2} s\right)}{\left(1 + T_{p_1} s\right)\left(1 + T_{p_2} s\right)} \cdots$$

where K is compensator DC gain, Tz_1, Tz_2, ..., are the zero time constants, and Tp_1, Tp_2, ..., are the pole time constants.

The natural frequency format is

$$K \times \frac{\left(1 + s/\omega_{z_1}\right)\left(1 + s/\omega_{z_2}\right)}{\left(1 + s/\omega_{p_1}\right)\left(1 + s/\omega_{p_2}\right)} \cdots$$

where K is compensator DC gain, ω_{z1}, and ω_{z2}, ... and ω_{p1}, ω_{p2}, ..., are the natural frequencies of the zeros and poles, respectively.

The zero/pole/gain format is

$$K \times \frac{(s + z_1)(s + z_2)}{(s + p_1)(s + p_2)}$$

where K is the overall compensator gain, and z_1, z_2, ... and p_1, p_2, ..., are the zero and pole locations, respectively.

- **Bode Options** — By default, the **Control System Designer** shows the plant and sensor poles and zeros as blue x's and o's, respectively. Clear this box to eliminate the plant's poles and zeros from the Bode plot. Note that the compensator poles and zeros (in red) will still appear.

Setting Tool Preferences
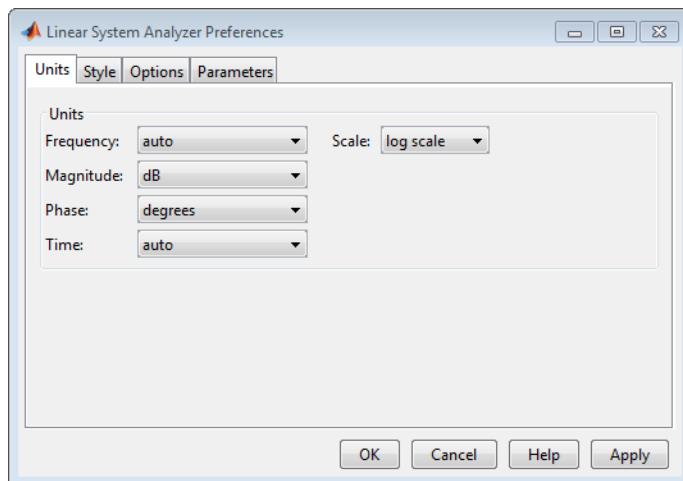
Linear System Analyzer Preferences Editor

In this section...

Opening the Linear System Analyzer Preference Editor

In the Linear System Analyzer, select **Edit** > **Linear System Analyzer Preferences**. The Linear System Analyzer Preferences dialog box let you customize various Linear System Analyzer properties, including units, fonts, and various other characteristics. This figure shows the editor open to its first pane.

Units Pane

You can select the following on the **Units** pane:

- **Frequency**

  The default `auto` option uses `rad/TimeUnit` as the frequency units relative to the system time units, where `TimeUnit` is the system time units specified in the `TimeUnit` property of the system on frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  For the frequency axis, you can select logarithmic or linear scales.

  ### Other Frequency Units Options

  - `'Hz'`
  - `'rad/s'`
  - `'rpm'`
  - `'kHz'`
  - `'MHz'`
  - `'GHz'`
  - `'rad/nanosecond'`

- `'rad/microsecond'`
- `'rad/millisecond'`
- `'rad/minute'`
- `'rad/hour'`
- `'rad/day'`
- `'rad/week'`
- `'rad/month'`
- `'rad/year'`
- `'cycles/nanosecond'`
- `'cycles/microsecond'`
- `'cycles/millisecond'`
- `'cycles/hour'`
- `'cycles/day'`
- `'cycles/week'`
- `'cycles/month'`
- `'cycles/year'`

- **Magnitude** — Decibels (dB) or absolute value (abs)
- **Phase** — Degrees or radians
- **Time**

  The default `auto` option uses the time units specified in the `TimeUnit` property of the system on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

Other Time Units Options

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`

- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Style Pane

Use the **Style** pane to toggle grid visibility and set font preferences and axes foreground colors for all plots in the Linear System Analyzer. This figure shows the Style pane.
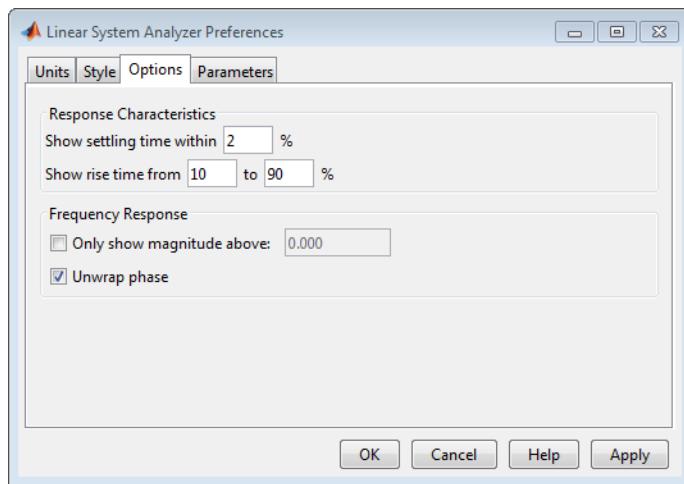
You have the following choices:

- **Grid** — Activate grids for all plots in the Linear System Analyzer
- **Fonts** — Set the font size, weight (bold), and angle (italic). Select font sizes from the menus or type any font-size values in the fields.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.
- If you do not want to specify the RGB values numerically, press the **Select** button to open the **Select Colors** window.

Options Pane

The Options pane has selections for time responses and frequency responses.
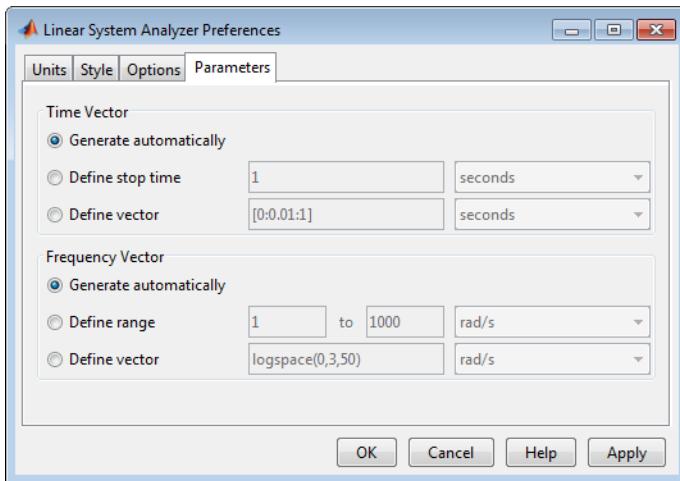
The following choices are available:

- **Time Response**:

  - Show settling time within *xx*% — You can set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.

  - Specify rise time from *xx*% to *yy*%— The standard definition of rise time is the time it takes the signal to go from 10% to 90% of the final value. You can choose any percentages you like (from 0% to 100%), provided that the first value is smaller than the second.

- **Frequency Response:**

  - Only show magnitude above *xx* — Specify a lower limit for magnitude values in response plots so that you can focus on a region of interest.

  - Unwrap phase—By default, the phase is unwrapped. Wrap the phrase by clearing this box. If the phase is wrapped, all phase values are shifted such that their equivalent value displays in the range [0°, 360°).

Parameters Pane

Use the **Parameters** pane, shown below, to specify input vectors for time and frequency simulation.

The defaults are to generate time and frequency vectors for your plots automatically. You can, however, override the defaults as follows:

- **Time Vector**:

  - Define stop time — Specify the final time value for your simulation
  - Define vector — Specify the time vector manually using equal-sized time steps

- **Frequency Vector**:

  - Define range — Specify the bandwidth of your response. Whether it's in rad/sec or Hz depends on the selection you made in the Units pane.

  - Define vector — Specify the vector for your frequency values. Any real, positive, strictly monotonically increasing vector is valid.

See Also

linearSystemAnalyzer

More About

- "Linear System Analyzer Overview" on page 22-2

19

Customizing Response Plot Properties

Introduction

The lowest level of the "Property and Preferences Hierarchy" on page 16-3 is setting response plot properties. This means that any property you set for a given plot will only affect that plot.

Customizing Response Plots Using the Response Plots Property Editor

In this section...
"Opening the Property Editor" on page 19-3
"Overview of Response Plots Property Editor" on page 19-4
"Labels Pane" on page 19-6
"Limits Pane" on page 19-6
"Units Pane" on page 19-7
"Style Pane" on page 19-14
"Options Pane" on page 19-15
"Editing Subplots Using the Property Editor" on page 19-19

Opening the Property Editor
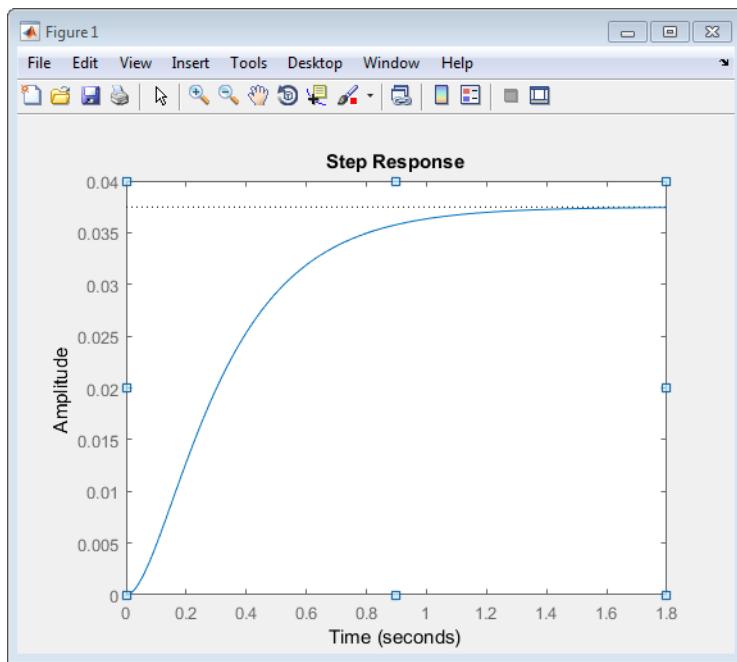
After you create a response plot, there are two ways to open the Property Editor:

- Double-click in the plot region
- Select **Properties** from the right-click menu

Before looking at the Property Editor, open a step response plot using these commands.
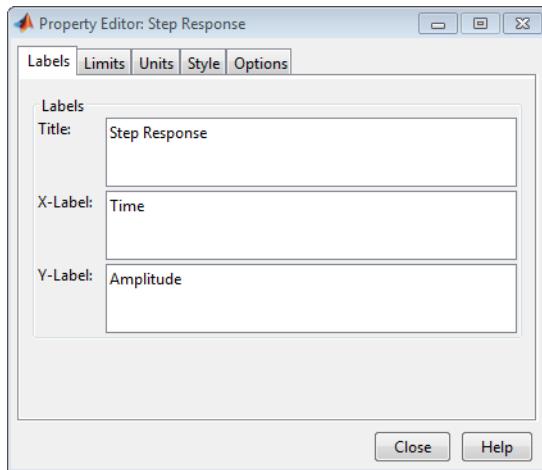
```
load ltiexamples
step(sys_dc)
```

This creates a step plot. Select **Properties** from the right-click menu. Note that when you open the **Property Editor**, squares appear around the step response plot.

Overview of Response Plots Property Editor

This figure shows the **Property Editor** dialog box for a step response.

The Property Editor for Step Response

In general, you can change the following properties of response plots. Note that only the **Labels** and **Limits** panes are available when using the **Property Editor** with Simulink Design Optimization software.

- Titles and X- and Y-labels in the **Labels** pane.
- Numerical ranges of the X and Y axes in the **Limits** pane.
- Units where applicable (e.g., rad/s to Hertz) in the **Units** pane.

  If you cannot customize units, the Property Editor will display that no units are available for the selected plot.

- Styles in the **Styles** pane.

  You can show a grid, adjust font properties, such as font size, bold and italics, and change the axes foreground color

- Change options where applicable in the **Options** pane.
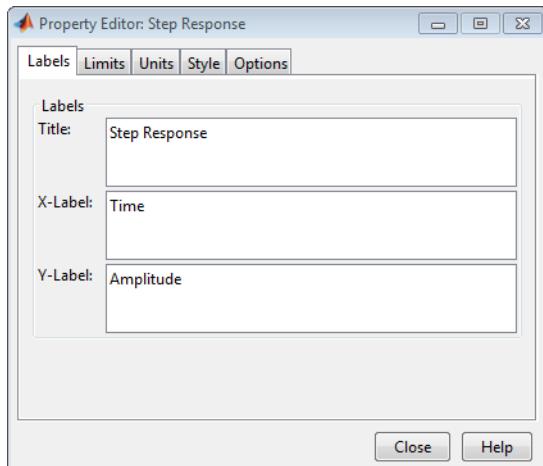
  These include peak response, settling time, phase and gain margins, etc. Plot options change with each plot response type. The Property Editor displays only the options that make sense for the selected response plot. For example, phase and gain margins are not available for step responses.

As you make changes in the Property Editor, they display immediately in the response plot. Conversely, if you make changes in a plot using right-click menus, the Property

Editor for that plot automatically updates. The Property Editor and its associated plot are dynamically linked.

Labels Pane
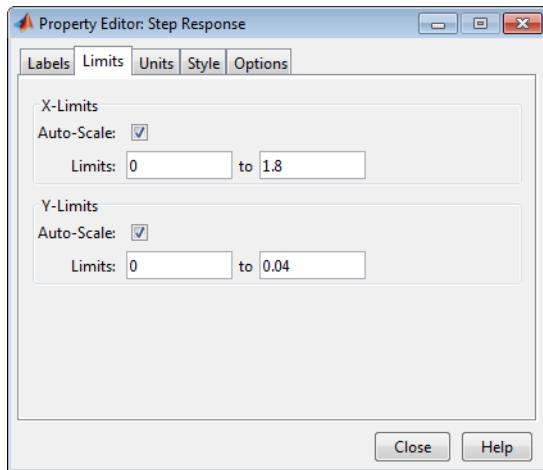
To specify new text for plot titles and axis labels, type the new string in the field next to the label you want to change. Note that the label changes immediately as you type, so you can see how the new text looks as you are typing.

Limits Pane
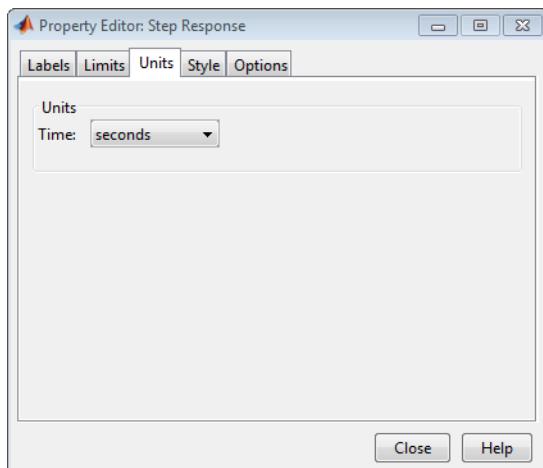
Default values for the axes limits make sure that the maximum and minimum x and y values are displayed. If you want to override the default settings, change the values in the Limits fields. The **Auto-Scale** box automatically clears if you click a different field. The new limits appear immediately in the response plot.

To reestablish the default values, select the **Auto-Scale** box again.

Units Pane

You can use the **Units** pane to change units in your response plot. The contents of this pane depend on the response plot associated with the editor. Use the menus to toggle between units.

Optional Unit Conversions for Response Plots

Response Plot	Unit Conversions
Bode and Bode Magnitude	• Frequency By default, shows rad/TimeUnit where TimeUnit is the system time units specified in the TimeUnit property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'`

Response Plot	Unit Conversions
	• Frequency scale is logarithmic or linear. • Magnitude in decibels (dB) or the absolute value • Phase in degrees or radians
Impulse	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`
Nichols Chart	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'`

Response Plot	Unit Conversions
	• `'GHz'`
	• `'rad/nanosecond'`
	• `'rad/microsecond'`
	• `'rad/millisecond'`
	• `'rad/minute'`
	• `'rad/hour'`
	• `'rad/day'`
	• `'rad/week'`
	• `'rad/month'`
	• `'rad/year'`
	• `'cycles/nanosecond'`
	• `'cycles/microsecond'`
	• `'cycles/millisecond'`
	• `'cycles/hour'`
	• `'cycles/day'`
	• `'cycles/week'`
	• `'cycles/month'`
	• `'cycles/year'`
	• Phase in degrees or radians
Nyquist Diagram	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'`

Response Plot	Unit Conversions
	• `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'`
Pole/Zero Map	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'`
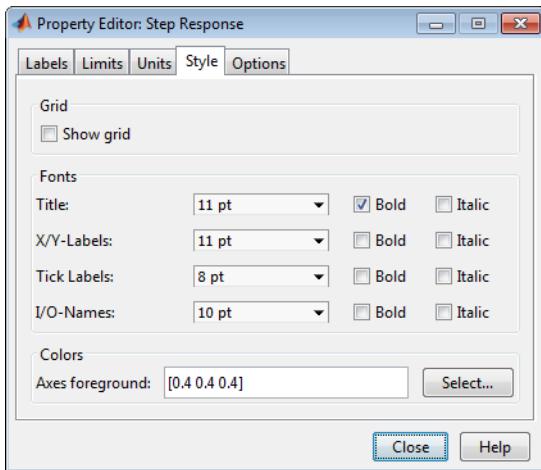
Response Plot	Unit Conversions
	`'hours'` `'days'` `'weeks'` `'months'` `'years'` - Frequency By default, shows rad/TimeUnit where TimeUnit is the system time units specified in the TimeUnit property of the input system. **Frequency Units Options** `'Hz'` `'rad/s'` `'rpm'` `'kHz'` `'MHz'` `'GHz'` `'rad/nanosecond'` `'rad/microsecond'` `'rad/millisecond'` `'rad/minute'` `'rad/hour'` `'rad/day'` `'rad/week'` `'rad/month'` `'rad/year'` `'cycles/nanosecond'` `'cycles/microsecond'` `'cycles/millisecond'`

Response Plot	Unit Conversions
	• `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'`
Singular Values	• Frequency By default, shows rad/TimeUnit where TimeUnit is the system time units specified in the TimeUnit property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'`

Response Plot	Unit Conversions
	• `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` • Frequency scale is logarithmic or linear. • Magnitude in decibels or the absolute value using logarithmic or linear scale
Step	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`

Style Pane

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for response plots.
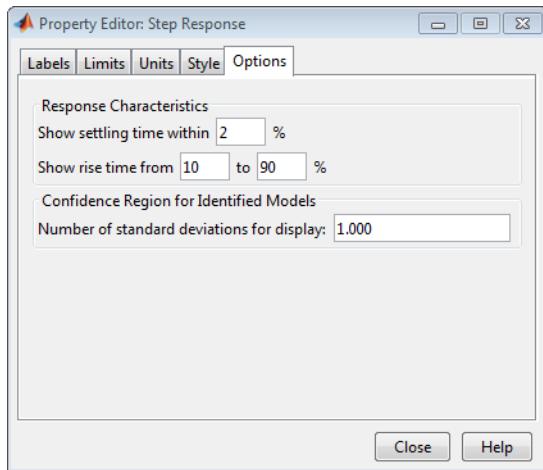
You have the following choices:

- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic) for fonts used in response plot titles, X/Y-labels, tick labels, and I/O names. Select font sizes from the menus or type any font-size values in the fields.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

  If you do not want to specify RGB values numerically, click the **Select** button to open the **Select Color** dialog box.

Options Pane

The **Options** pane enables you to customize response characteristics for plots. Each response plot has its own set of characteristics and optional settings. When you change the value in a field, press **Enter** on your keyboard to update the response plot.

Response Characteristic Options for Response Plots

Plot	Customizable Feature
Bode Diagram and Bode Magnitude	• **Magnitude Response** Select lower magnitude limit. • **Phase Response** Adjust phase offsets to keep phase close to a particular value, within a range of 0º–360º, at a given frequency. Unwrap phase (default is unwrapped). • **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for plotting the response confidence region. To see the confidence region, right-click the plot, and select **Characteristics** > **Confidence Region**.

Plot	Customizable Feature
Impulse	• **Response Characteristics** Show settling time within *xx*% (specify the percentage). • **Confidence Region for Identified Models** These options are available with System Identification Toolbox. **Display using zero mean interval**: For an identified model with impulse response y and standard deviation Δy, plot the uncertainty ±Δy as a function of time (default). If cleared, y±Δy as a function of time is plotted. **Number of standard deviations for display**: Specify number of standard deviations for plotting the uncertainty. To see the confidence interval, right-click the plot, and select **Characteristics** > **Confidence Region**.
Nichols Chart	• **Magnitude Response** Select lower magnitude limit. • **Phase Response** Adjust phase offsets to keep phase close to a particular value, within a range of 0º–360º, at a given frequency. Unwrap phase (default is unwrapped).
Nyquist Diagram	• **Confidence Region for Identified Models** These options are available with System Identification Toolbox.

Plot	Customizable Feature
	Number of standard deviations for display: Specify number of standard deviations for plotting the confidence ellipses. **Display spacing**: Specify the frequency spacing of confidence ellipses. The default is 5, which means that the confidence ellipses are shown at every fifth frequency sample. To see the confidence ellipses, right-click the plot, and select **Characteristics** > **Confidence Region**.
Pole/Zero Map	• **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for displaying the confidence region characteristic. To see the confidence region, right-click the plot, and select **Characteristics** > **Confidence Region**.
Sigma	None
Step	• **Response Characteristics** Show settling time within *xx*% (specify the percentage). Show rise time from *xx* to *yy*% (specify the percentages) • **Confidence Region for Identified Models** This option is available with System Identification Toolbox.

Plot	Customizable Feature
	Specify number of standard deviations for plotting the response confidence region. To see the confidence region, right-click the plot, and select **Characteristics** > **Confidence Region**.

Editing Subplots Using the Property Editor

If you create more than one plot in a single figure window, you can edit each plot individually. For example, the following code creates a figure with two plots, a step and an impulse response with two randomly selected systems:

```
subplot(2,1,1)
step(rss(2,1))
subplot(2,1,2)
impulse(rss(1,1))
```

After the figure window appears, double-click in the upper (step response) plot to activate the **Property Editor**. You will see a set of small squares appear around the step response, indicating that it is the active plot for the editor. To switch to the lower (impulse response) plot, click once in the impulse response plot region. The set of squares switches to the impulse response, and the **Property Editor** updates as well.

Customizing Response Plots Using Plot Tools

In this section...
"Properties You Can Customize Using Plot Tools" on page 19-20
"Opening and Working with Plot Tools" on page 19-21
"Example of Changing Line Color Using Plot Tools" on page 19-21

Properties You Can Customize Using Plot Tools

The following table shows the plot properties you can customize using plot tools.

For...	You can customize the following properties:
Responses	• System name • Line color • Line style • Line width • Marker type For SISO systems, these changes apply to a single plot line or an array of plot lines representing the system on one axis. For MIMO systems, these changes apply to all of the plotted lines representing the system on multiple axes.
Plot axes	• Title • X-label • Y-label
Figures	• Figure name • Colormap • Figure color

Note: To make other changes to response plots, see "Customizing Response Plots Using the Response Plots Property Editor" on page 19-3 and "Customizing Response Plots from the Command Line" on page 19-24.

Opening and Working with Plot Tools

See the following documentation for information about how to open and work with Plot Tools and the Plot Tools Property Editor:

- "Customize Graph Using Plot Tools" in the MATLAB documentation.
- "Customize Objects in Graph" in the MATLAB documentation.

Example of Changing Line Color Using Plot Tools

To change the line color of a MIMO system plot:

1 Create a step response plot of a MIMO system by typing
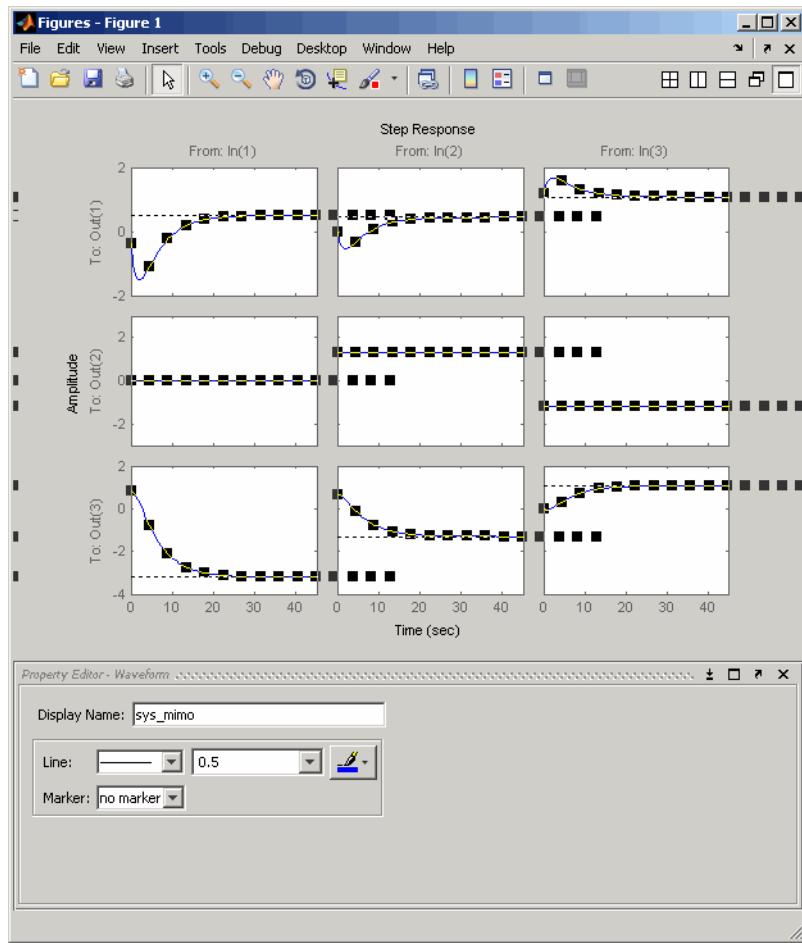
```
sys_mimo=rss(3,3,3);
stepplot(sys_mimo)
```

2 In the figure window, select **View** > **Property Editor**.

This action opens the Plot Tools Property Editor.

3 Click the plot line in any of the nine axis.

This action selects the response for the sys_mimo system and updates the Plot Tools Property Editor to show the available editable properties for the response.

Note: The Plot Tools Property Editor applies changes to the response of the MIMO system. Any change you make applies to all of the plotted lines in the figure.

Tip You can also change the properties of the response using the right-click menu while in plot edit mode.

4 In the **Property Editor – Waveform** pane, select the color red.

This action changes the color of the response that represents the MIMO system to red.

Customizing Response Plots from the Command Line

In this section...
"Overview of Customizing Plots from the Command Line" on page 19-24
"Obtaining Plot Handles" on page 19-27
"Obtaining Plot Options Handles" on page 19-28
"Examples of Customizing Plots from the Command Line" on page 19-30
"Properties and Values Reference" on page 19-34
"Property Organization Reference" on page 19-45

Overview of Customizing Plots from the Command Line

- "When to Customize Plots from the Command Line" on page 19-24
- "How to Customize Plots from the Command Line" on page 19-24
- "Change Bode Plot Units from the Command Line" on page 19-25

When to Customize Plots from the Command Line

You can customize any response plot from the command line. The command line is the most efficient way to customize a large number of plots. For example, if you have a batch job that produces many plots, you can change the *x*-axis units automatically for all the plot with just a few lines of code.

How to Customize Plots from the Command Line

You can use the Control System Toolbox application program interface (API) to customize plotting options for response plots from the command line.

Note This section assumes some very basic familiarity with Handle Graphics® and object-oriented concepts, namely, classes, objects, and Handle Graphics handles. See "Role of Classes in MATLAB" and "Graphics Objects" in the MATLAB online documentation for more information.

To customize plots from the command line:

1 Obtain the *plot handle*, which is an identifier for the plot, using the API's plotting syntax.

For example,

```
h=stepplot(sys)
```

returns the plot handle h for the step plot.

For more information on obtaining plot handles, see "Obtaining Plot Handles" on page 19-27.

2 Obtain the *plot options handle*, which is an identifier for all settable plot options. To get a plot options handle for a given plot, type

```
p=getoptions(h);
```

p is the plot options handle for plot handle h.

For more information on obtaining plot options handles, see "Obtaining Plot Options Handles" on page 19-28.

3 Use `setoptions`, along with the plot handle and the plot options handle, to access and modify many plot options.
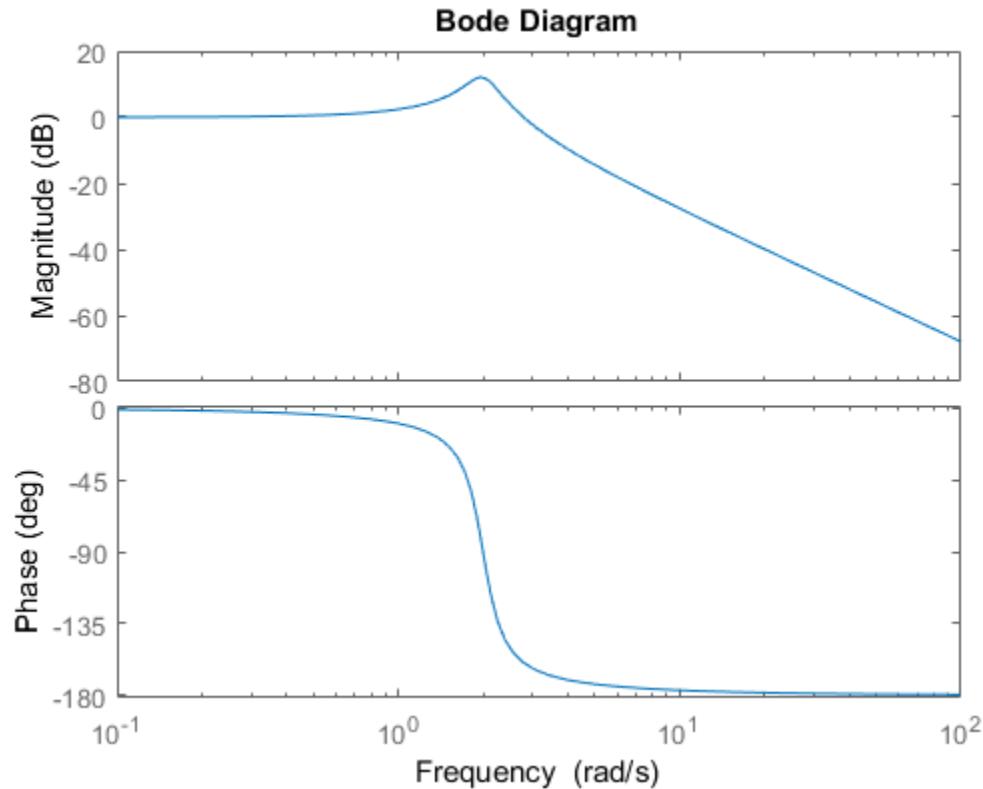
Note: You can also use `setoptions` to customize plots using property/value pairs instead of the plot options handle. Using property/value pairs shortens the procedure to one line of code.

Change Bode Plot Units from the Command Line

This example shows how to change the units of a Bode plot from rad/s to Hz.

Create a system and generate a Bode Plot of the system's response. The plot uses the default units, rad/s.
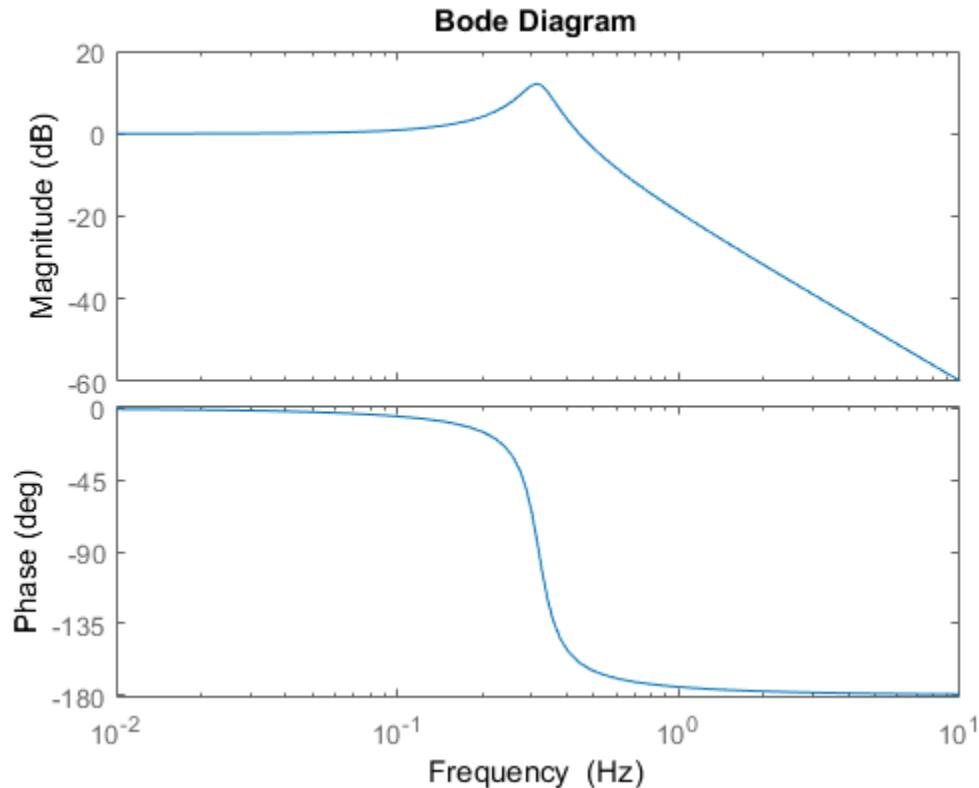
```
sys = tf(4,[1 0.5 4]);
h = bodeplot(sys);
```

The `bodeplot` command returns a plot handle that you can use to change properties of the plot.

Change the units to Hz.

```
p = getoptions(h);
p.FreqUnits = 'Hz';
setoptions(h,p)
```

The x-axis label updates to reflect the change of unit.

For more examples of customizing plots from the command line, see "Examples of Customizing Plots from the Command Line".

Obtaining Plot Handles

To programmatically interact with response plot, you need the *plot handle*. This handle is an identifier to the response plot object. Because the Control System Toolbox plotting commands, bode, rlocus, etc., all use the plot handle internally, this API provides a set of commands that explicitly return the handle to your response plot. These functions all end with "plot," which makes them easy to identify. This table lists the functions.

Functions That Return the Plot Handle

Function	Plot
`bodeplot`	Bode magnitude and phase
`hsvplot`	Hankel singular values
`impulseplot`	Impulse response
`initialplot`	Initial condition
`iopzplot`	Pole/zero maps for input/output pairs
`lsimplot`	Time response to arbitrary inputs
`nicholsplot`	Nichols chart
`nyquistplot`	Nyquist
`pzplot`	Pole/zero
`rlocusplot`	Root locus
`sigmaplot`	Singular values of the frequency response
`stepplot`	Step response

To get a plot handle for any response plot, use the functions from the table. For example,

```
h = bodeplot(sys)
```

returns plot handle `h` (it also renders the Bode plot). Once you have this handle, you can modify the plot properties using the `setoptions` and `getoptions` methods of the plot object, in this case, a Bode plot handle.

Obtaining Plot Options Handles

- "Overview of Plot Options Handles" on page 19-28
- "Retrieving a Handle" on page 19-29
- "Creating a Handle" on page 19-29
- "Which Properties Can You Modify?" on page 19-30

Overview of Plot Options Handles

Once you have the plot handle, you need the *plot options handle*, which is an identifier for all the settable plot properties for a given response plot. There are two ways to create a plot options handle:

- Retrieving a Handle — Use `getoptions` to get the handle.
- Creating a Handle — Use `<responseplot>options` to instantiate a handle. See Functions for Creating Plot Options Handles for a complete list.

Retrieving a Handle

The `getoptions` function retrieves a plot options handle from a plot handle.

```
p=getoptions(h) % Returns plot options handle p for plot handle h.
```

If you specify a property name as an input argument, `getoptions` returns the property value associated with the property name.

```
property_value=getoptions(h,PropertyName) % Returns a property
                                          % value.
```

Creating a Handle

You can create a default plot options handle by using functions in the form of

```
<responseplot>options
```

For example,

```
p=bodeoptions;
```

instantiates a handle for Bode plots. See "Properties and Values Reference" on page 19-34 for a list of default values.

If you want to set the default values to the Control System Toolbox default values, pass `cstprefs` to the function. For example,

```
p = bodeoptions('cstprefs');
```

set the Bode plot property/value pairs to the Control System Toolbox default values.

This table lists the functions that create a plot options handle.

Functions for Creating Plot Options Handles

Function	Type of Plot Options Handle Created
bodeoptions	Bode phase and magnitude

Function	Type of Plot Options Handle Created
`hsvoptions`	Hankel singular values
`nicholsoptions`	Nichols plot
`nyquistoptions`	Nyquist plot
`pzoptions`	Pole/zero plot
`sigmaoptions`	Sigma (singular values) plot
`timeoptions`	Time response (impulse, step, etc.)

Which Properties Can You Modify?

Use

```
help <responseplot>options
```

to see a list of available property value pairs that you can modify. For example,

```
help bodeoptions
```

You can modify any of these parameters using `setoptions`. The next topic provides examples of modifying various response plots.

See "Properties and Values Reference" on page 19-34 for a complete list of property/value pairs for response plots.

Examples of Customizing Plots from the Command Line

- "Manipulating Plot Options Handles" on page 19-30
- "Changing Plot Units" on page 19-31
- "Create Plots Using Existing Plot Options Handle" on page 19-32
- "Creating a Default Plot Options Handle" on page 19-32
- "Using Dot Notation Like a Structure" on page 19-33
- "Setting Property Pairs in setoptions" on page 19-33

Manipulating Plot Options Handles

There are two fundamental ways to manipulate plot option handles:

- Dot notation — Treat the handle like a MATLAB structure.
- Property value pairs — Specify property/value pairs explicitly as input arguments to `setoptions`.

For some examples, both dot notation and property/value pairs approaches are shown. For all examples, use

```
sys=tf(1,[1 1])
```

for the system.
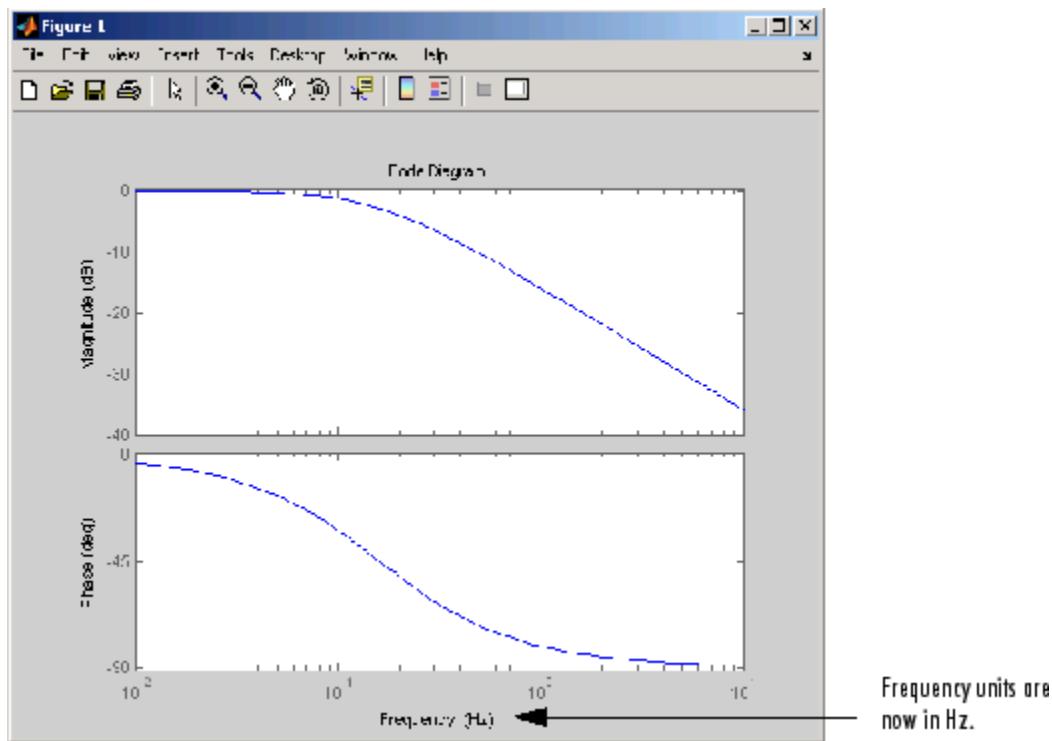
Changing Plot Units

Change the frequency units of a Bode plot from rad/s to Hz.

```
h = bodeplot(sys);
p = getoptions(h);
p.FreqUnits = 'Hz'
setoptions(h,p)
```

or, for the last three lines, substitute

```
setoptions(h,'FreqUnits','Hz')
```

Frequency units are now in Hz.

Create Plots Using Existing Plot Options Handle

You can use an existing plot options handle to customize a second plot:

```
h1 = bodeplot(sys);
p1 = getoptions(h1);
h2 = bodeplot(sys,p1);
```

or

```
h1 = bodeplot(sys);
h2 = bodeplot(sys2);
setoptions(h2,getoptions(h1))
```

Creating a Default Plot Options Handle

Instantiate a plot options handle with this code.

```
p = bodeoptions;
```
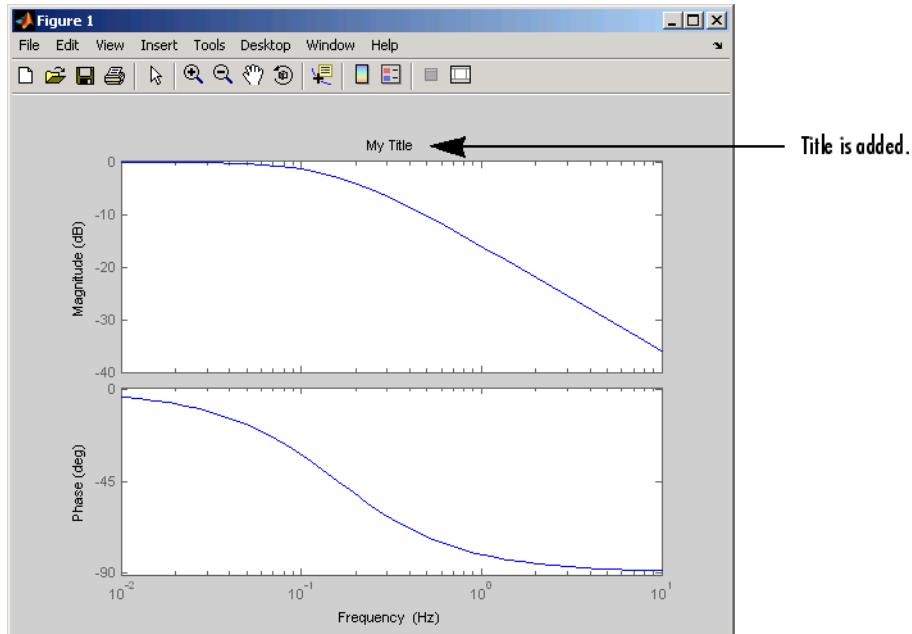
Change the frequency units and apply the changes to `sys`.

```
p.FreqUnits ='Hz';
h = bodeplot(sys,p);
```

Using Dot Notation Like a Structure

You can always use dot notation to assign values to properties.

```
h1 = bodeplot(sys)
p1 = getoptions(h1)
p1.FreqUnits = Hz'
p1.Title.String  =  'My Title';
setoptions(h1,p1)
```


Setting Property Pairs in setoptions

Instead of using dot notation, specify frequency units as property/value pairs in `setoptions`.

```
h1 = bodeplot(sys)
setoptions(h1,'FreqUnits','Hz')
```

Verify that the units have changed from rad/s to Hz.

```
getoptions(h1,'FreqUnits') % Returns frequency units for h1.

ans =

Hz
```

Properties and Values Reference

Property/Value Pairs Common to All Response Plots

The following tables discuss property/value pairs common to all response plots.

Title

Property	Default Value	Description
Title.String	none	String
Title.FontSize	8	Double
Title.FontWeight	normal	[light | normal | demi]
Title.FontAngle	normal	[normal | italic | oblique]
Title.Color	[0 0 0]	1-by-3 RGB vector

X Label

Property	Default Value	Description
XLabel.String	none	String
Xlabel.FontSize	8	Double
Xlabel.FontWeight	normal	[light | normal | demi]

Property	Default Value	Description
XLabel.FontAngle	normal	`[normal	italic	oblique]`
Xlabel.Color	[0 0 0]	`1-by-3 RGB vector`

Y Label

Property	Default Value	Description
YLabel.String	none	`String`
Ylabel.FontSize	8	`Double`
Ylabel.FontWeight	normal	`[light	normal	demi]`
YLabel.FontAngle	normal	`[normal	italic	oblique]`
Ylabel.Color	[0 0 0]	`1-by-3 RGB vector`

Tick Label

Property	Default Value	Description
TickLabel.FontSize	8	`Double`
TickLabel.FontWeight	normal	`[light	normal	demi]`
TickLabel.FontAngle	normal	`[normal	italic	oblique]`
Ticklabel.Color	[0 0 0]	`1-by-3 RGB vector`

Grid and Axis Limits

Property	Default Value	Description
grid	off	`[on	off]`
Xlim	{[]}	A cell array of 1-by-2 doubles that specifies the *x*-axis limits when `XLimMode` is set to manual. When `XLim` is scalar, scalar expansion is applied; otherwise the length of the cell array must equal the number of columns (i.e., number of system inputs) for the plot. The 1-by-2 doubles must be a strictly increasing pair `[xmin, xmax]`.
XLimMode	{auto}	A cell array of strings `[auto	manual]` that specifies the *x*-axis limits mode. When `XLimMode` is set to manual the limits are set to the values specified in `XLim`. When `XLimMode` is scalar, scalar expansion is applied; otherwise

Property	Default Value	Description
		the length of the cell array must equal the number of columns (i.e., number of system inputs) for the plot.
YLim	{[]}	A cell array of 1-by-2 doubles specifies the *y*-axis limits when YLimMode is set to manual. When YLim is scalar, scalar expansion is applied; otherwise the length of the cell array must equal the number of rows (i.e., number of system outputs) for the plot. The 1-by-2 doubles must be a strictly increasing pair [ymin, ymax].
YLimMode	{auto}	A cell array of strings [auto | manual] that specifies the *y*-axis limits mode. When YLimMode is set to manual the limits are set to the values specified in YLim. When YLimMode is scalar, scalar expansion is applied; otherwise the length of the cell array must equal the number of rows (i.e., number of system outputs) for the plot.

I/O Grouping

Property	Default Value	Description
IOGrouping	none	[none | inputs | outputs | all] Specifies input/output groupings for responses.

Input Labels

Property	Default Value	Description
InputLabels.FontSize	8	Double
InputLabels.FontWeight	normal	[light | normal | demi]
InputLabels.FontAngle	normal	[normal | italic | oblique]
InputLabels.Color	[0 0 0]	1-by-3 RGB vector

Output Labels

Property	Default Value	Description
OutputLabel.FontSize	8	Double
OutputLabels.FontWeight	normal	[light | normal | demi]

Property	Default Value	Description
OutputLabels.FontAngle	normal	[normal | italic | oblique]
OutputLabels.Color	[0 0 0]	1-by-3 RGB vector

Input/Output Visible

Property	Default Value	Description
InputVisible	{on}	[on | off] A cell array that specifies the visibility of each input channel. If the value is a scalar, scalar expansion is applied.
OutputVisible	{on}	[on | off] A cell array that specifies the visibility of each output channel. If the value is a scalar, scalar expansion is applied.

Bode Plots

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day'

Property	Default Value	Description
		• `'rad/week'`
		• `'rad/month'`
		• `'rad/year'`
		• `'cycles/nanosecond'`
		• `'cycles/microsecond'`
		• `'cycles/millisecond'`
		• `'cycles/hour'`
		• `'cycles/day'`
		• `'cycles/week'`
		• `'cycles/month'`
		• `'cycles/year'`
FreqScale	log	[linear | log]
MagUnits	dB	[db | abs]
MagScale	linear	[linear | log]
PhaseUnits	deg	[rad | deg]
PhaseWrapping	off	[on | off]
MagVisible	on	[on | off]
PhaseVisible	on	[on | off]
MagLowerLimMode	auto	[auto | manual] Enables a manual lower magnitude limit specification by `MagLowerLim`.
MagLowerLim	0	Double Specifies the lower magnitude limit when `MagLowerLimMode` is set to `manual`.
PhaseMatching	off	[on | off] Enables adjusting phase effects for phase response.
PhaseMatchingFreq	0	Double

Property	Default Value	Description
PhaseMatchingValue	0	Double

Hankel Singular Values

Property	Default Value	Description
Yscale	linear	[linear | log]
AbsTol	0	Double See hsvd and stabsep for details.
RelTol	1*e-08	Double See hsvd and stabsep for details.
Offset	1*e-08	Double See hsvd and stabsep for details.

Nichols Plots

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week'

Property	Default Value	Description
		• `'rad/month'`
		• `'rad/year'`
		• `'cycles/nanosecond'`
		• `'cycles/microsecond'`
		• `'cycles/millisecond'`
		• `'cycles/hour'`
		• `'cycles/day'`
		• `'cycles/week'`
		• `'cycles/month'`
		• `'cycles/year'`
`MagUnits`	`dB`	[dB | abs]
`PhaseUnits`	`deg`	[rad | deg]
`MagLowerLimMode`	`auto`	[auto | manual]
`MagLowerLim`	`0`	double
`PhaseMatching`	off	[on | off]
`PhaseMatchingFreq`	0	Double
`PhaseMatchingValue`	0	Double

Nyquist Charts

Property	Default Value	Description
`FreqUnits`	`rad/s`	**Available Options**
		• `'Hz'`
		• `'rad/s'`
		• `'rpm'`
		• `'kHz'`
		• `'MHz'`
		• `'GHz'`
		• `'rad/nanosecond'`

Property	Default Value	Description
		• 'rad/microsecond'
		• 'rad/millisecond'
		• 'rad/minute'
		• 'rad/hour'
		• 'rad/day'
		• 'rad/week'
		• 'rad/month'
		• 'rad/year'
		• 'cycles/nanosecond'
		• 'cycles/microsecond'
		• 'cycles/millisecond'
		• 'cycles/hour'
		• 'cycles/day'
		• 'cycles/week'
		• 'cycles/month'
		• 'cycles/year'
MagUnits	dB	[dB | abs]
PhaseUnits	deg	[rad | deg]
ShowFullContour	on	[on | off]

Pole/Zero Maps

Property	Default Value	Description
FreqUnits	rad/s	**Available Options**
		• 'Hz'
		• 'rad/s'
		• 'rpm'
		• 'kHz'
		• 'MHz'
		• 'GHz'

Property	Default Value	Description
		• `'rad/nanosecond'`
		• `'rad/microsecond'`
		• `'rad/millisecond'`
		• `'rad/minute'`
		• `'rad/hour'`
		• `'rad/day'`
		• `'rad/week'`
		• `'rad/month'`
		• `'rad/year'`
		• `'cycles/nanosecond'`
		• `'cycles/microsecond'`
		• `'cycles/millisecond'`
		• `'cycles/hour'`
		• `'cycles/day'`
		• `'cycles/week'`
		• `'cycles/month'`
		• `'cycles/year'`
`TimeUnits`	seconds	**Available Options**
		• `'nanoseconds'`
		• `'microseconds'`
		• `'milliseconds'`
		• `'seconds'`
		• `'minutes'`
		• `'hours'`
		• `'days'`
		• `'weeks'`
		• `'months'`
		• `'years'`
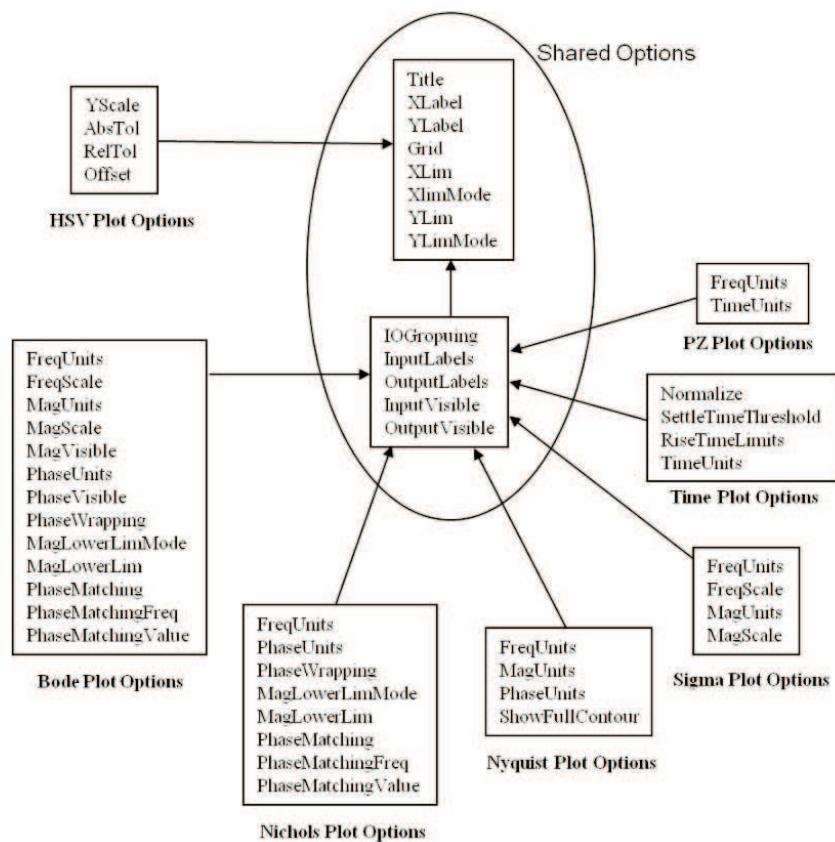
Sigma Plots

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'`
FreqScale	log	[linear | log]
MagUnits	dB	[dB | abs]
MagScale	linear	[linear | log]

Time Response Plots

| Property	Default Value	Description
`Normalize`	`off`	`[on	off]` Normalize the *y*-scale of all responses in the plot.
`SettleTimeThreshold`	`0.02`	`Double` Specifies the settling time threshold. `0.02 = 2%`.
`RiseTimeLimits`	`[0.1, 0.9]`	`1-by-2 double` Specifies the limits used to define the rise time. `[0.1, 0.9]` is 10% to 90%.
`TimeUnits`	`seconds`	**Available Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`

Property Organization Reference

HSV Plot Options
- YScale
- AbsTol
- RelTol
- Offset

Shared Options
- Title
- XLabel
- YLabel
- Grid
- XLim
- XlimMode
- YLim
- YLimMode

- IOGropuing
- InputLabels
- OutputLabels
- InputVisible
- OutputVisible

PZ Plot Options
- FreqUnits
- TimeUnits

Time Plot Options
- Normalize
- SettleTimeThreshold
- RiseTimeLimits
- TimeUnits

Bode Plot Options
- FreqUnits
- FreqScale
- MagUnits
- MagScale
- MagVisible
- PhaseUnits
- PhaseVisible
- PhaseWrapping
- MagLowerLimMode
- MagLowerLim
- PhaseMatching
- PhaseMatchingFreq
- PhaseMatchingValue

Nichols Plot Options
- FreqUnits
- PhaseUnits
- PhaseWrapping
- MagLowerLimMode
- MagLowerLim
- PhaseMatching
- PhaseMatchingFreq
- PhaseMatchingValue

Nyquist Plot Options
- FreqUnits
- MagUnits
- PhaseUnits
- ShowFullContour

Sigma Plot Options
- FreqUnits
- FreqScale
- MagUnits
- MagScale

Customizing Plots Inside the SISO Design Tool

In this section...
"Overview of Customizing SISO Design Tool Plots" on page 19-46
"Root Locus Property Editor" on page 19-46
"Open-Loop Bode Property Editor" on page 19-50
"Open-Loop Nichols Property Editor" on page 19-53
"Prefilter Bode Property Editor" on page 19-54

Overview of Customizing SISO Design Tool Plots

Customizing plots inside the SISO Design Tool is similar to how you customize any response plot. The following property editors are specific to the SISO Design Tool:

- Root Locus Property Editor
- Open-Loop Bode Property Editor
- Open-Loop Nichols Property Editor
- Prefilter Bode Property Editor
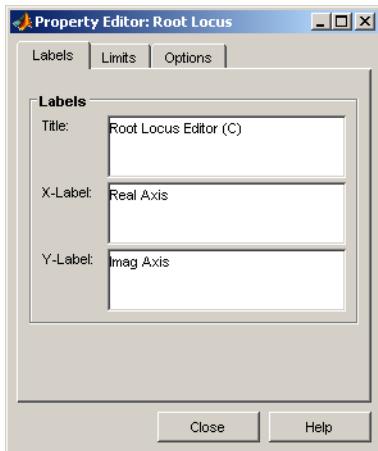
You can use each of these property editors to create the customized plots within the SISO Design tool.

Root Locus Property Editor

There are three ways to open the Property Editor for root locus plots:

- Double-click in the root locus away from the curve
- Select **Properties** from the right-click menu
- Select **Root Locus** and then **Properties** from Edit in the menu bar
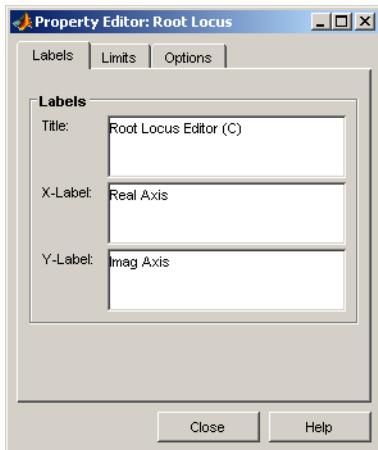
This figure shows the **Property Editor: Root Locus** window.

- "Labels Pane" on page 19-47
- "Limits Pane" on page 19-48
- "Options Pane" on page 19-49

Labels Pane

You can use the **Label** pane to specify plot titles and axis labels. To specify a new label, type the string in the appropriate field. The root locus plot automatically updates.

Limits Pane

The SISO Design Tool specifies default values for the real and imaginary axes ranges to make sure that all the poles and zeros in your model appear in the root locus plot. Use the Limits pane, shown below, to override the default settings.

To change the limits, specify the new limits in the real and imaginary axes **Limits** fields. The **Auto-Scale** check box automatically clears once you click in a different field. Your root locus diagram updates immediately. If you want to reapply the default limits, select the **Auto-Scale** check boxes again.

The Limit Stack pane provides support for storing and retrieving custom limit specifications. There are four buttons available:

 — Add the current limits to the stack

 — Retrieve the previous stack entry

 — Retrieve the next stack entry
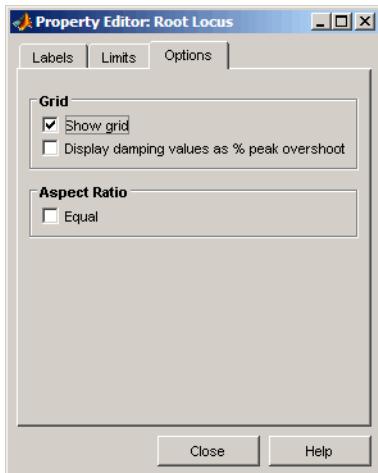
 — Remove the current limits from the stack

Using these buttons, you can store and retrieve any number of saved custom axes limits.

Options Pane
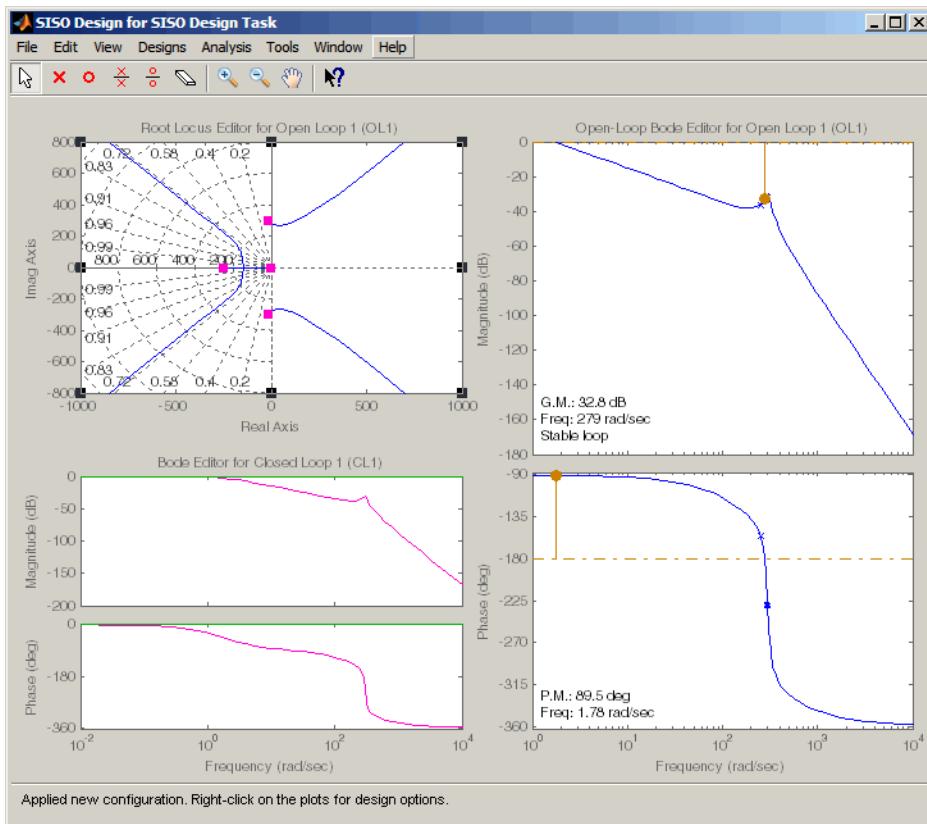
The Options pane contains settings for adding a grid and changing the plot's aspect ratio.

Select **Show grid** to display a grid on the root locus. If you have damping ratio constraints on your root locus, selecting **Display damping ratios as % peak overshoot** displays the damping ratio values along the grid lines. This figure shows both options activated for an imported model, Gservo. If you want to verify these settings, type

```
load ltiexamples
```

at the MATLAB prompt and import Gservo from the workspace into your SISO Design Tool.

The numbers displayed on the root locus gridlines are the damping ratios as a percentage of the overshoot values.

If you select the **Equal** check box in the **Aspect Ratio** pane, the *x* and *y*-axes are set to equal limit values.
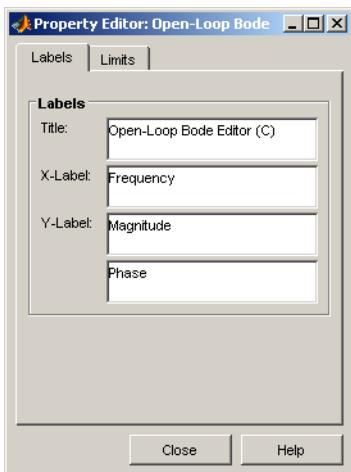
Open-Loop Bode Property Editor

As is the case with the root locus Property Editor, there are three ways to open the Bode diagram property editor:

·   Double-click in the Bode magnitude or phase plot away from the curve.

·   Select **Properties** from the right-click menu.

- Select **Open-Loop Bode** and then **Properties** from **Edit** in the menu bar.
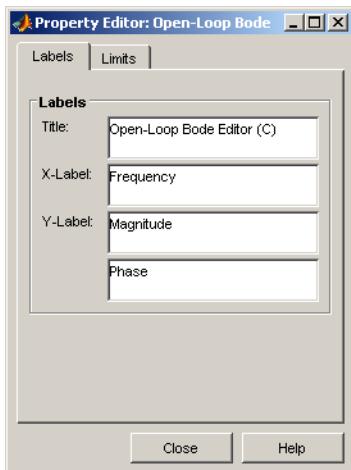
This figure shows the **Property Editor: Open-Loop Bode** editor.

- "Labels Pane" on page 19-51
- "Limits Pane" on page 19-52
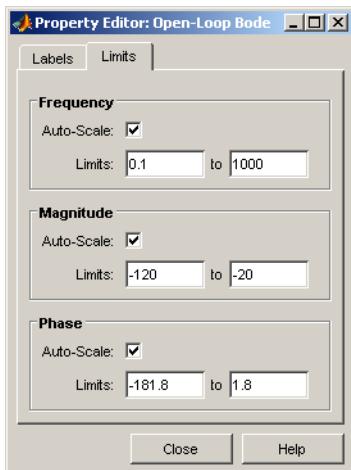
Labels Pane

You can use the **Label** pane to specify plot titles and axis labels. To specify a new label, type the string in the appropriate field. The Bode diagram automatically updates.

Limits Pane

You can use the Limits pane to override the default limits for the frequency, magnitude, and phase scales for your plots.

To change the limits, specify the new values in the **Limits** fields for frequency, magnitude, and phase. The **Auto-Scale** check box automatically deactivates once you click in a different field. The Bode diagram updates immediately.
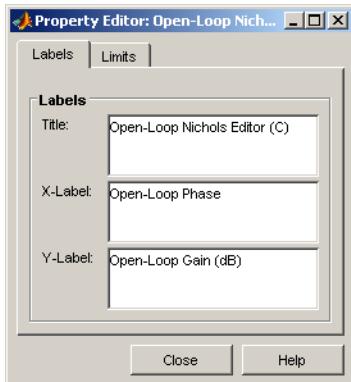
To restore the default settings, select the **Auto-Scale** boxes again.

Open-Loop Nichols Property Editor

As is the case with the root locus Property Editor, there are three ways to open the Nichols plot property editor:

- Double-click in the Nichols plot away from the curve.
- Select **Properties** from the right-click menu.
- Select **Open-Loop Nichols** and then **Properties** from **Edit** in the menu bar.
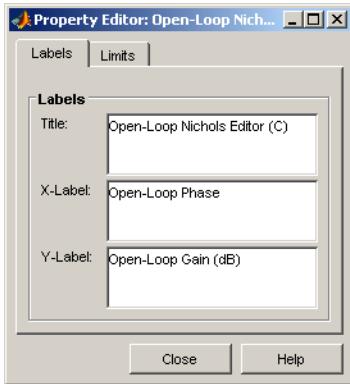
This figure shows the **Property Editor: Open-Loop Nichols** editor.

- "Labels Pane" on page 19-53
- "Limits Pane" on page 19-54
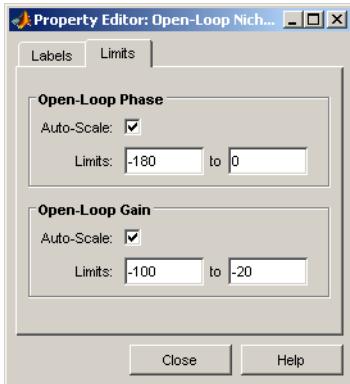
Labels Pane

You can use the Label pane to specify plot titles and axis labels. To specify a new label, type the string in the appropriate field. The Nichols plot automatically updates.

Limits Pane

You can use the Limits pane to override the default limits for the frequency, magnitude, and phase scales for your plots.

To change the limits, specify the new values in the **Limits** fields for open-loop phase and/ or gain. The **Auto-Scale** check box automatically deactivates once you click in a different field. The Nichols plot updates immediately.

To restore the default settings, select the **Auto-Scale** boxes again.

Prefilter Bode Property Editor

The **Prefilter Bode Property** editor is identical to the Open-Loop Bode diagram property editor. There are three ways to open the prefilter editor:

- Double-click in the prefilter Bode magnitude or phase plot away from the curve.
- Select **Properties** from the right-click menu.
- Select **Prefilter Bode** and then **Properties** from **Edit** in the menu bar.

See "Open-Loop Bode Property Editor" on page 19-50 for a description of the features of this editor.

Build GUI With Interactive Response-Plot Updates

This example shows how to create a GUI to display a Control System Toolbox response plot that changes in response to interactive input.

The GUI in this example displays the step response of a second-order dynamic system of fixed natural frequency. The GUI includes a slider that sets the system's damping ratio. To cause the response plot to reflect the slider setting, you must define a callback for the slider. This callback uses the `updateSystem` command to update the plot with new system data in response to changes in the slider setting.

Set the initial values of the second-order dynamic system and create the system model.
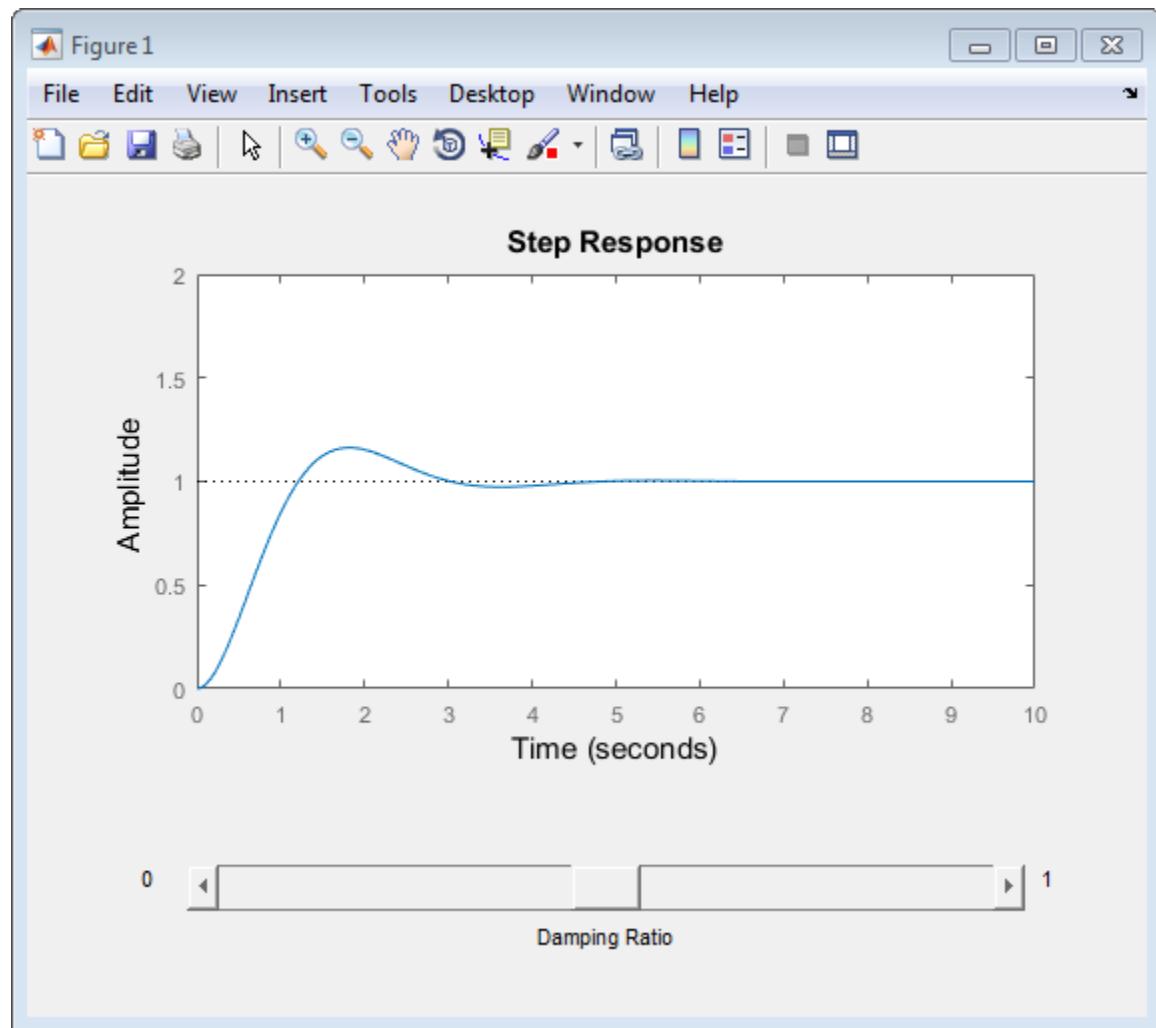
```
zeta = .5;                              % Damping Ratio
wn = 2;                                 % Natural Frequency
sys = tf(wn^2,[1,2*zeta*wn,wn^2]);
```

Create a figure for the GUI and configure the axes for displaying the step response.

```
f = figure;
ax = axes('Parent',f,'position',[0.13 0.39  0.77 0.54]);
h = stepplot(ax,sys);
setoptions(h,'XLim',[0,10],'YLim',[0,2]);
```

Add the slider and slider label text to the figure.

```
b = uicontrol('Parent',f,'Style','slider','Position',[81,54,419,23],...
                'value',zeta, 'min',0, 'max',1);
bgcolor = f.Color;
bl1 = uicontrol('Parent',f,'Style','text','Position',[50,54,23,23],...
                'String','0','BackgroundColor',bgcolor);
bl2 = uicontrol('Parent',f,'Style','text','Position',[500,54,23,23],...
                'String','1','BackgroundColor',bgcolor);
bl3 = uicontrol('Parent',f,'Style','text','Position',[240,25,100,23],...
                'String','Damping Ratio','BackgroundColor',bgcolor);
```

Set the callback that updates the step response plot as the damping ratio slider is moved.
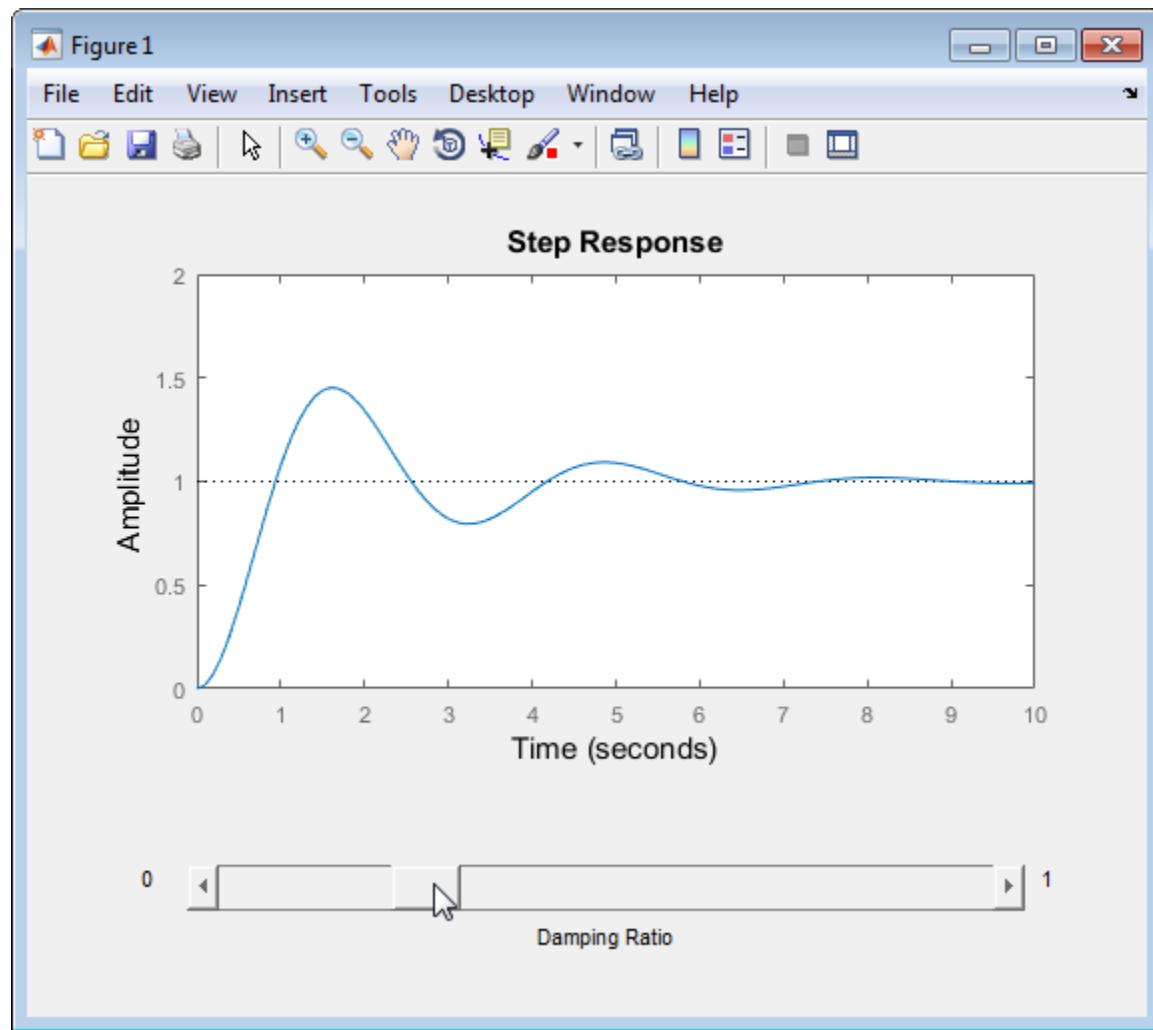
```
b.Callback = @(es,ed) updateSystem(h,tf(wn^2,[1,2*(es.Value)*wn,wn^2]));
```

This code sets the callback for the slider (identified as b) to an anonymous function. The input arguments to this anonymous function, es and ed, are automatically passed to the callback when the slider is used. es is the handle of the uicontrol that represents the

slider, and `ed` is the event data structure which the slider automatically passes to the callback. You do not need to define these variables in the workspace or set their values. (For more information about UI callbacks, see "Callback Definition".)

The callback is a call to the `updateSystem` function, which replaces the plotted response data with a response derived from a new transfer function. The callback uses the slider data `es.Value` to define a second-order system whose damping ratio is the current value of the slider.

Now that you have set the callback, move the slider. The displayed step response changes as expected.

See Also

uicontrol | updateSystem

Related Examples

- "Callback Definition"

- "Write Callbacks for Apps Created Programmatically"

Design Case Studies

Design Yaw Damper for Jet Transport

In this section...

Overview of this Case Study

This case study demonstrates the tools for classical control design by stepping through the design of a yaw damper for a $747^{®}$ jet transport aircraft.

Creating the Jet Model

The jet model during cruise flight at MACH = 0.8 and H = 40,000 ft. is

```
A=[-.0558 -.9968 .0802 .0415;
    .598 -.115 -.0318 0;
   -3.05 .388 -.4650 0;
      0 0.0805 1 0];

B=[ .00729  0;
   -0.475   0.00775;
    0.153   0.143;
      0      0];

C=[0 1 0 0;
   0 0 0 1];

D=[0 0;
   0 0];

sys = ss(A,B,C,D);
```

The following commands specify this state-space model as an LTI object and attach names to the states, inputs, and outputs.

```
states = {'beta' 'yaw' 'roll' 'phi'};
inputs = {'rudder' 'aileron'};
outputs = {'yaw' 'bank angle'};

sys = ss(A,B,C,D,'statename',states,...
                 'inputname',inputs,...
                 'outputname',outputs);
```

You can display the LTI model `sys` by typing `sys`. This command produces the following result.

```
a =
            beta      yaw     roll      phi
   beta   -0.0558  -0.9968   0.0802   0.0415
   yaw      0.598   -0.115  -0.0318        0
   roll     -3.05    0.388   -0.465        0
   phi          0   0.0805        1        0

b =
         rudder  aileron
   beta  0.00729        0
   yaw    -0.475  0.00775
   roll    0.153    0.143
   phi         0        0

c =
             beta  yaw  roll   phi
   yaw          0    1     0     0
   bank angle   0    0     0     1

d =
             rudder  aileron
   yaw             0        0
   bank angle      0        0
```

Continuous-time model.

The model has two inputs and two outputs. The units are radians for `beta` (sideslip angle) and `phi` (bank angle) and radians/sec for `yaw` (yaw rate) and `roll` (roll rate). The rudder and aileron deflections are in radians as well.

Computing Open-Loop Poles
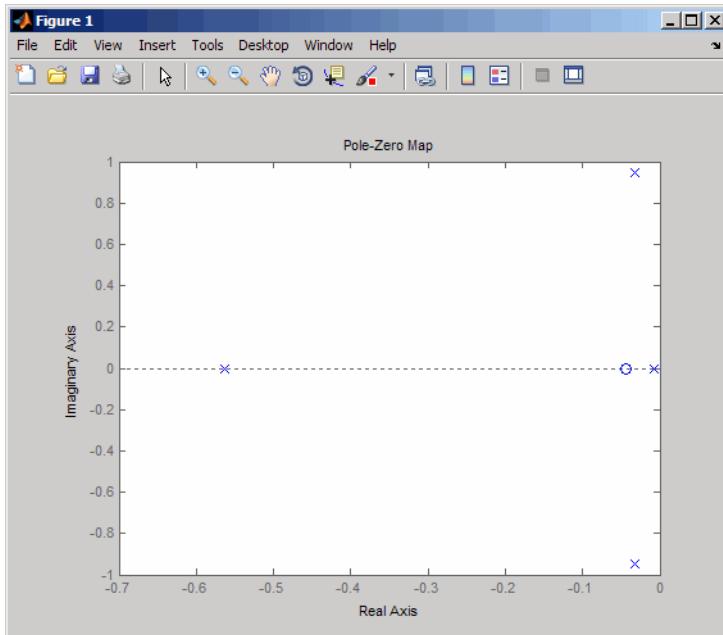
Compute the open-loop poles and plot them in the *s*-plane.

```
>> damp(sys)

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

```
-7.28e-03                    1.00e+00      7.28e-03      1.37e+02
-5.63e-01                    1.00e+00      5.63e-01      1.78e+00
-3.29e-02 + 9.47e-01i        3.48e-02      9.47e-01      3.04e+01
-3.29e-02 - 9.47e-01i        3.48e-02      9.47e-01      3.04e+01
```

```
pzmap(sys)
```
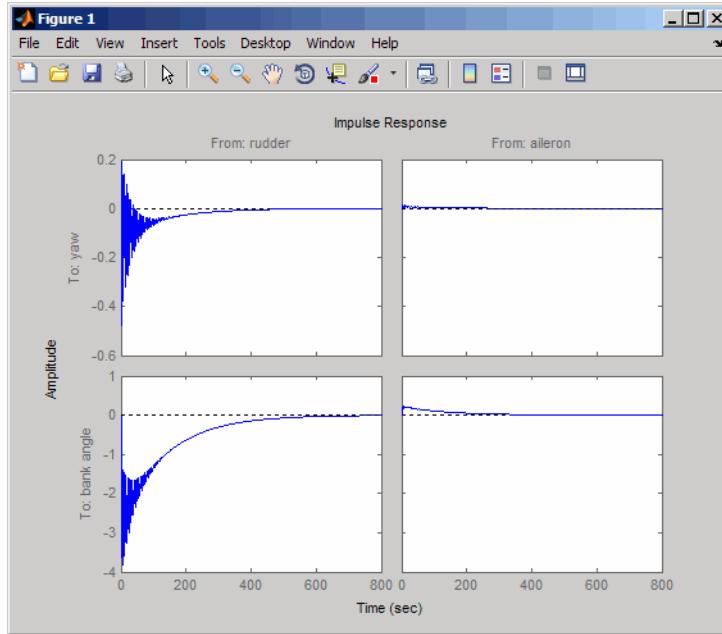

This model has one pair of lightly damped poles. They correspond to what is called the "Dutch roll mode."

Suppose you want to design a compensator that increases the damping of these poles, so that the resulting complex poles have a damping ratio $\zeta > 0.35$ with natural frequency $\omega_n < 1$ rad/sec. You can do this using the Control System Toolbox analysis tools.

Open-Loop Analysis
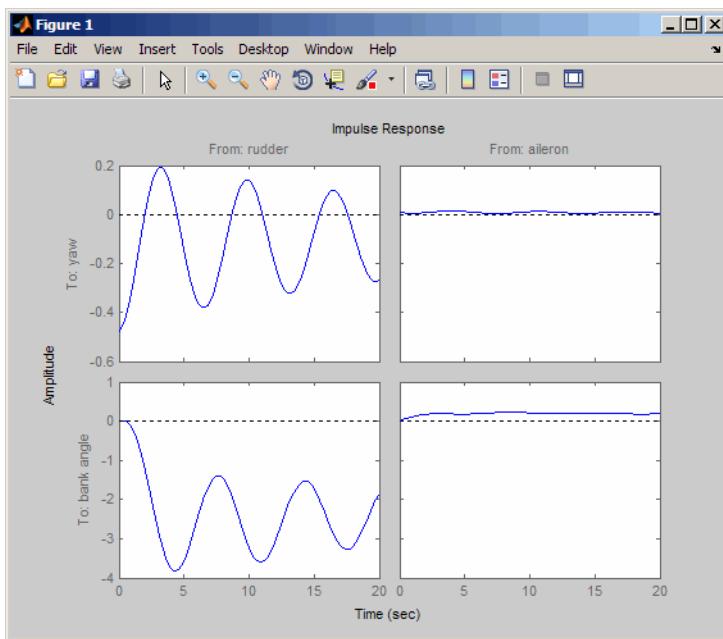
First, perform some open-loop analysis to determine possible control strategies. Start with the time response (you could use `step` or `impulse` here).

```
impulse(sys)
```
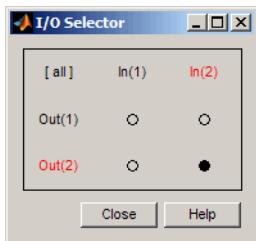

The impulse response confirms that the system is lightly damped. But the time frame is much too long because the passengers and the pilot are more concerned about the behavior during the first few seconds rather than the first few minutes. Next look at the response over a smaller time frame of 20 seconds.
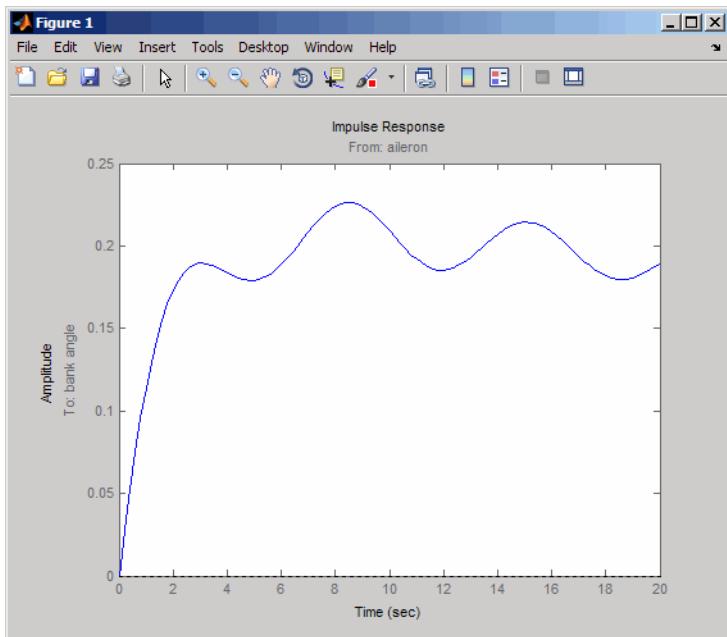
```
impulse(sys,20)
```

Look at the plot from aileron (input 2) to bank angle (output 2). To show only this plot, right-click and choose **I/O Selector**, then click on the (2,2) entry. The I/O Selector should look like this.
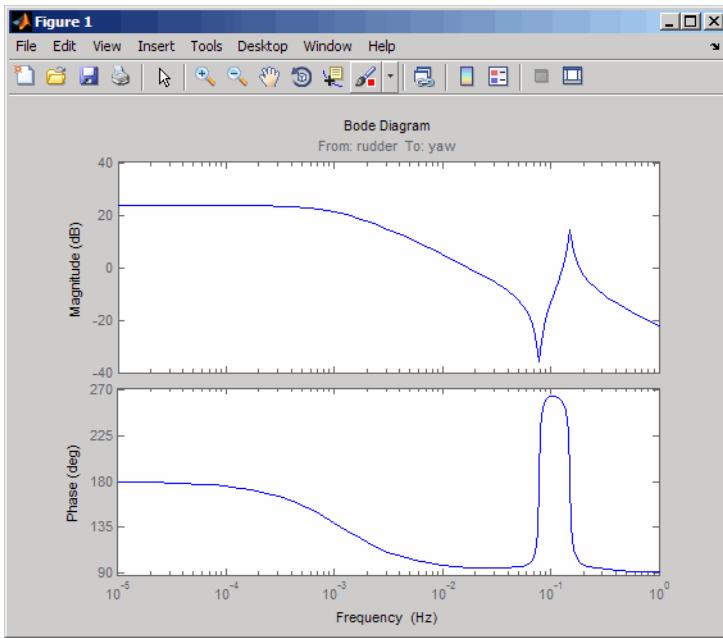
The new figure is shown below.

The aircraft is oscillating around a nonzero bank angle. Thus, the aircraft is turning in response to an aileron impulse. This behavior will prove important later in this case study.

Typically, yaw dampers are designed using the yaw rate as sensed output and the rudder as control input. Look at the corresponding frequency response.

```
sys11=sys('yaw','rudder') % Select I/O pair.
bode(sys11)
```
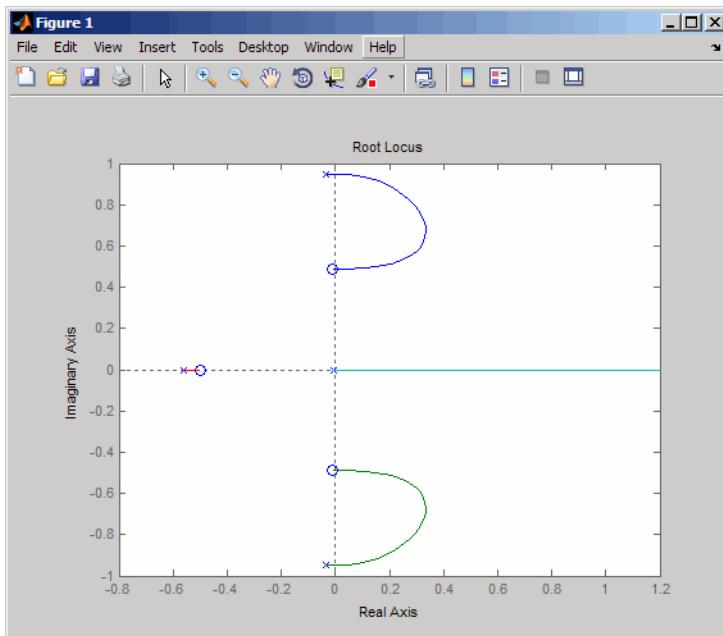
From this Bode diagram, you can see that the rudder has significant effect around the lightly damped Dutch roll mode (that is, near $\omega = 1$ rad/sec).

Root Locus Design
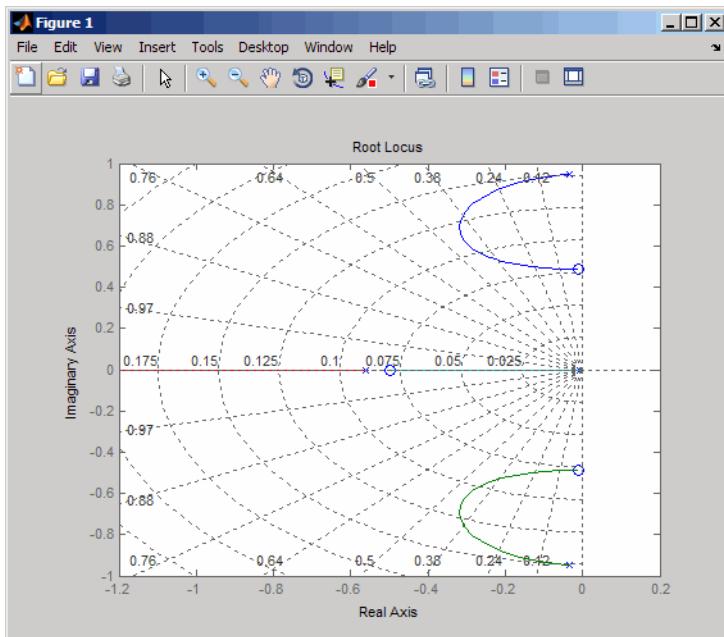
A reasonable design objective is to provide a damping ration $\zeta > 0.35$ with a natural frequency $\omega_n < 1.0$ rad/sec. Since the simplest compensator is a static gain, first try to determine appropriate gain values using the root locus technique.

```
% Plot the root locus for the rudder to yaw channel
rlocus(sys11)
```
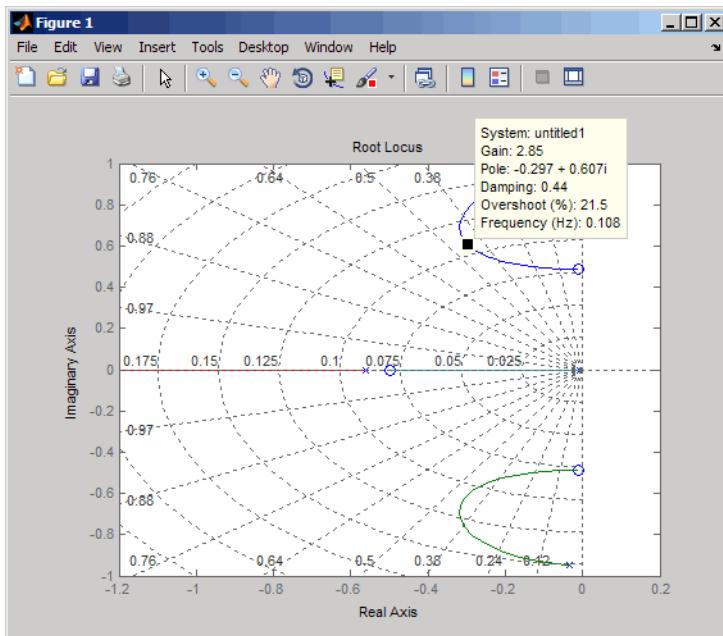
This is the root locus for negative feedback and shows that the system goes unstable almost immediately. If, instead, you use positive feedback, you may be able to keep the system stable.

```
rlocus(-sys11)
sgrid
```

This looks better. By using simple feedback, you can achieve a damping ratio of ζ > 0.45. Click on the blue curve and move the data marker to track the gain and damping values. To achieve a 0.45 damping ratio, the gain should be about 2.85. This figure shows the data marker with similar values.
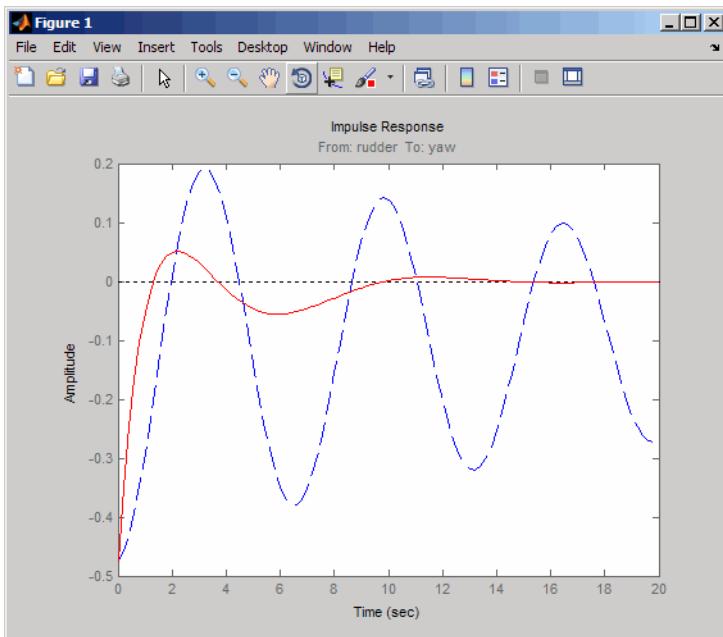
Next, close the SISO feedback loop.

```
K = 2.85;
cl11 = feedback(sys11,-K);  % Note: feedback assumes negative
                            % feedback by default
```

Plot the closed-loop impulse response for a duration of 20 seconds, and compare it to the open-loop impulse response.

```
impulse(sys11,'b--',cl11,'r',20)
```

The closed-loop response settles quickly and does not oscillate much, particularly when compared to the open-loop response.
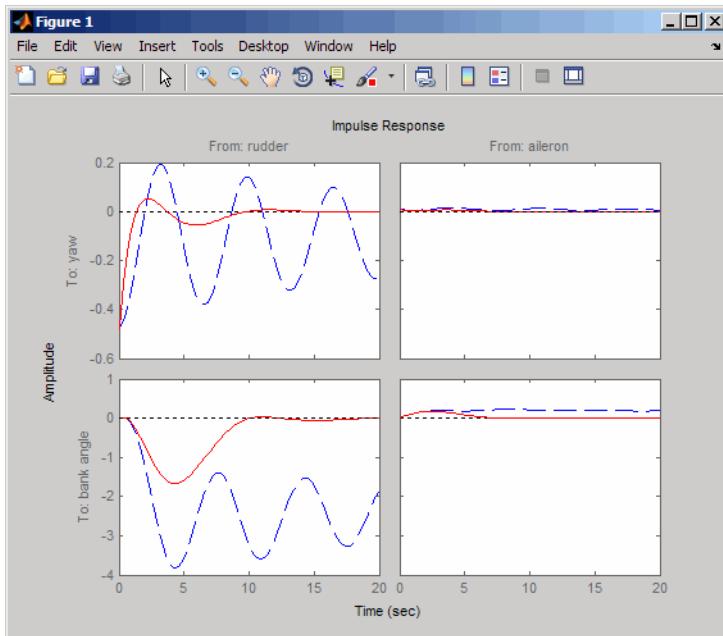
Now close the loop on the full MIMO model and see how the response from the aileron looks. The feedback loop involves input 1 and output 1 of the plant (use `feedback` with index vectors selecting this input/output pair). At the MATLAB prompt, type

```
cloop = feedback(sys,-K,1,1);
damp(cloop)    % closed-loop poles
```

Pole	Damping	Frequency (rad/seconds)	Time Constant (seconds)
-3.42e-01	1.00e+00	3.42e-01	2.92e+00
-2.97e-01 + 6.06e-01i	4.40e-01	6.75e-01	3.36e+00
-2.97e-01 - 6.06e-01i	4.40e-01	6.75e-01	3.36e+00
-1.05e+00	1.00e+00	1.05e+00	9.50e-01

Plot the MIMO impulse response.

```
impulse(sys,'b--',cloop,'r',20)
```

The yaw rate response is now well damped, but look at the plot from aileron (input 2) to bank angle (output 2). When you move the aileron, the system no longer continues to bank like a normal aircraft. You have over-stabilized the spiral mode. The spiral mode is typically a very slow mode and allows the aircraft to bank and turn without constant aileron input. Pilots are used to this behavior and will not like your design if it does not allow them to fly normally. This design has moved the spiral mode so that it has a faster frequency.

Washout Filter Design

What you need to do is make sure the spiral mode does not move further into the left-half plane when you close the loop. One way flight control designers have addressed this problem is to use a washout filter $kH(s)$ where

$$H(s) = \frac{s}{s + \alpha}$$

The washout filter places a zero at the origin, which constrains the spiral mode pole to remain near the origin. We choose α = 0.2 for a time constant of five seconds and use the

root locus technique to select the filter gain H. First specify the fixed part $s/(s + a)$ of the washout by
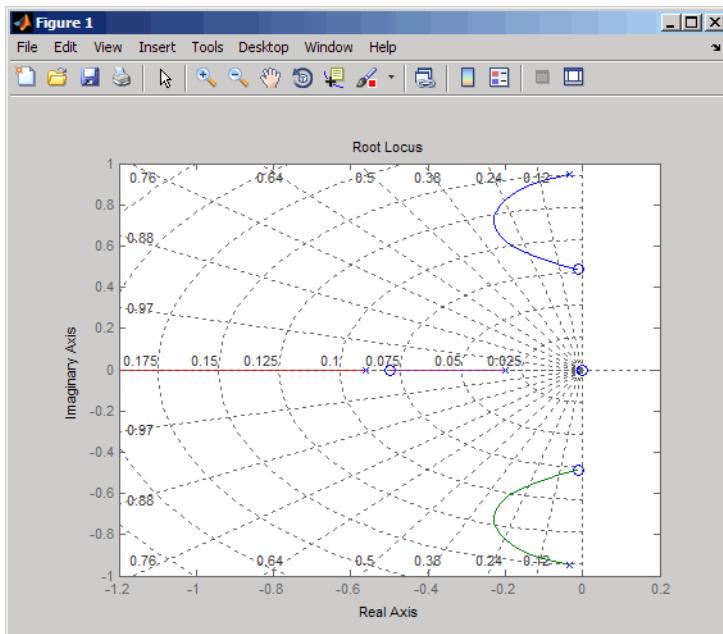
```
H = zpk(0,-0.2,1);
```

Connect the washout in series with the design model sys11 (relation between input 1 and output 1) to obtain the open-loop model

```
oloop = H * sys11;
```
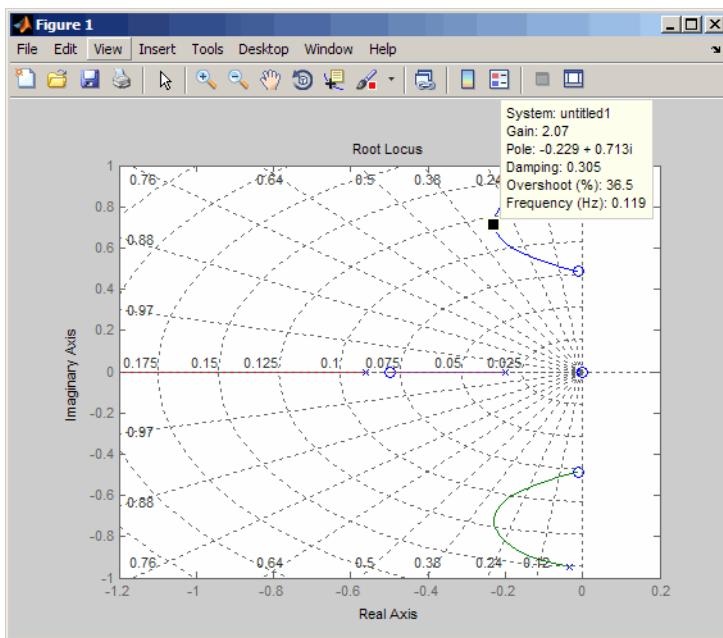
and draw another root locus for this open-loop model.

```
rlocus(-oloop)
sgrid
```
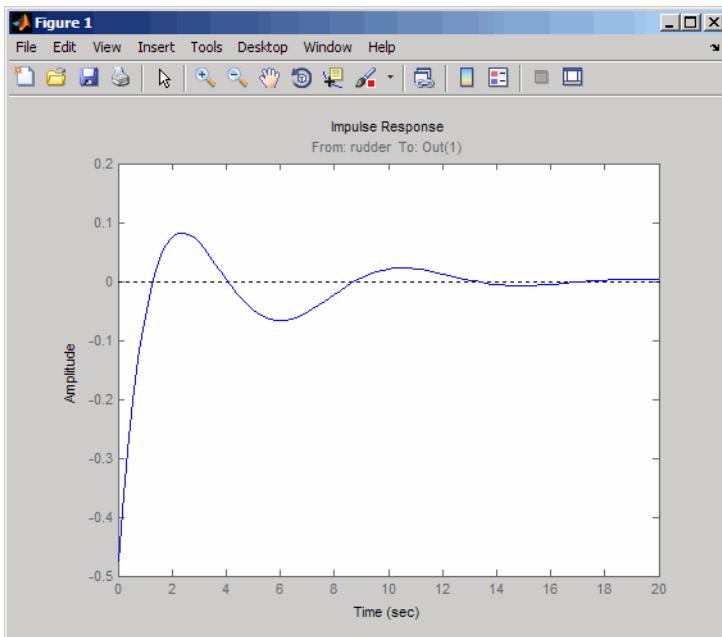

Create and drag a data marker around the upper curve to locate the maximum damping, which is about $\zeta = 0.3$.

This figure shows a data marker at the maximum damping ratio; the gain is approximately 2.07.

Look at the closed-loop response from rudder to yaw rate.

```
K = 2.07;
cl11 = feedback(oloop,-K);
impulse(cl11,20)
```
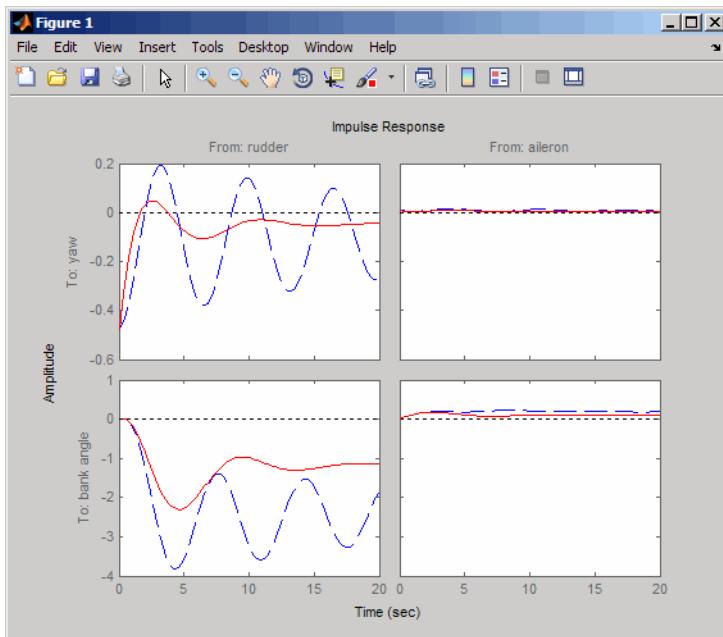
The response settles nicely but has less damping than your previous design. Finally, you can verify that the washout filter has fixed the spiral mode problem. First form the complete washout filter $kH(s)$ (washout + gain).
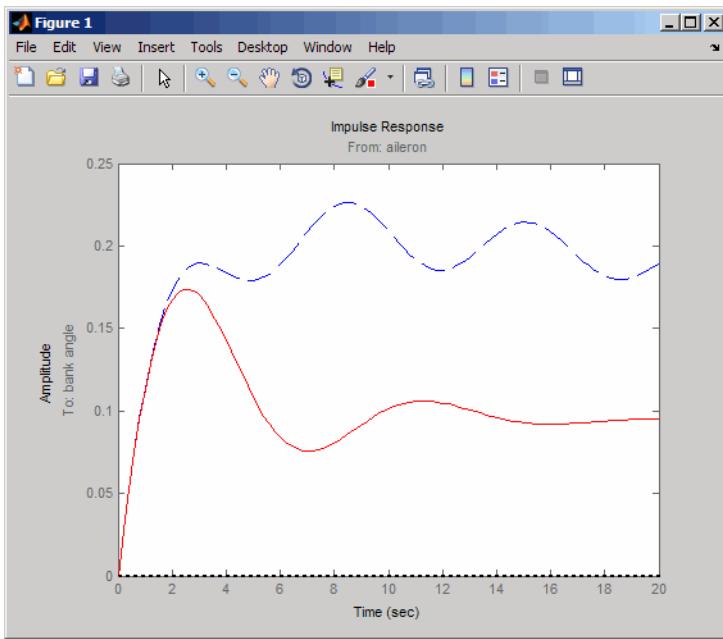
```
WOF = -K * H;
```

Then close the loop around the first I/O pair of the MIMO model sys and simulate the impulse response.

```
cloop = feedback(sys,WOF,1,1);

% Final closed-loop impulse response
impulse(sys,'b--',cloop,'r',20)
```

The bank angle response (output 2) due to an aileron impulse (input 2) now has the desired nearly constant behavior over this short time frame. To inspect the response more closely, use the I/O Selector in the right-click menu to select the (2,2) I/O pair.

Although you did not quite meet the damping specification, your design has increased the damping of the system substantially and now allows the pilot to fly the aircraft normally.

LQG Regulation: Rolling Mill Case Study

Overview of this Case Study
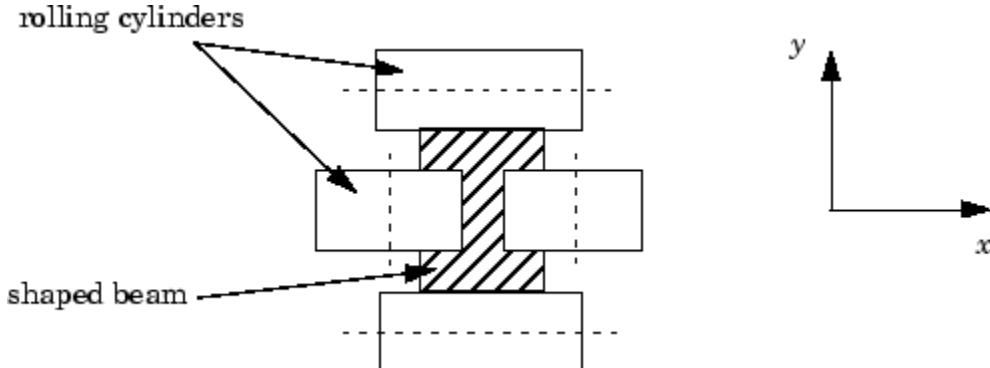
This case study demonstrates the use of the LQG design tools in a process control application. The goal is to regulate the horizontal and vertical thickness of the beam produced by a hot steel rolling mill. This example is adapted from [1]. The full plant model is MIMO and the example shows the advantage of direct MIMO LQG design over separate SISO designs for each axis. Type

```
milldemo
```
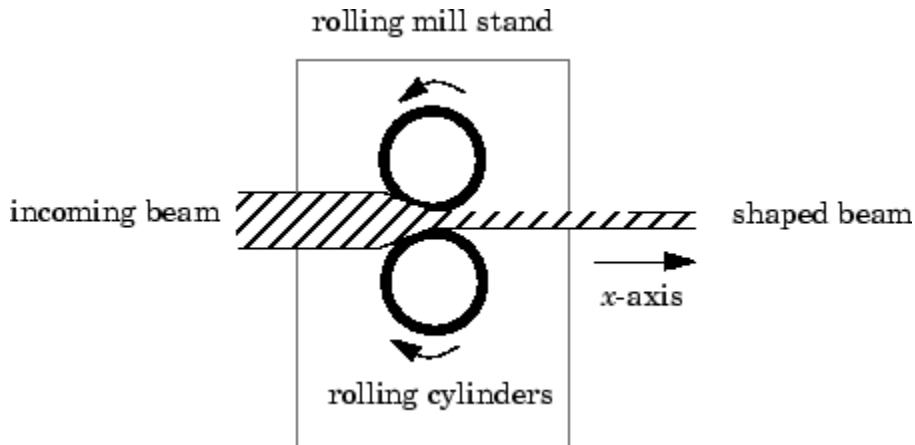
at the command line to run this demonstration interactively.

Process and Disturbance Models

The rolling mill is used to shape rectangular beams of hot metal. The desired outgoing shape is sketched below.

This shape is impressed by two pairs of rolling cylinders (one per axis) positioned by hydraulic actuators. The gap between the two cylinders is called the *roll gap*.

rolling mill stand

incoming beam   shaped beam

x-axis

rolling cylinders

The objective is to maintain the beam thickness along the *x*- and *y*-axes within the quality assurance tolerances. Variations in output thickness can arise from the following:
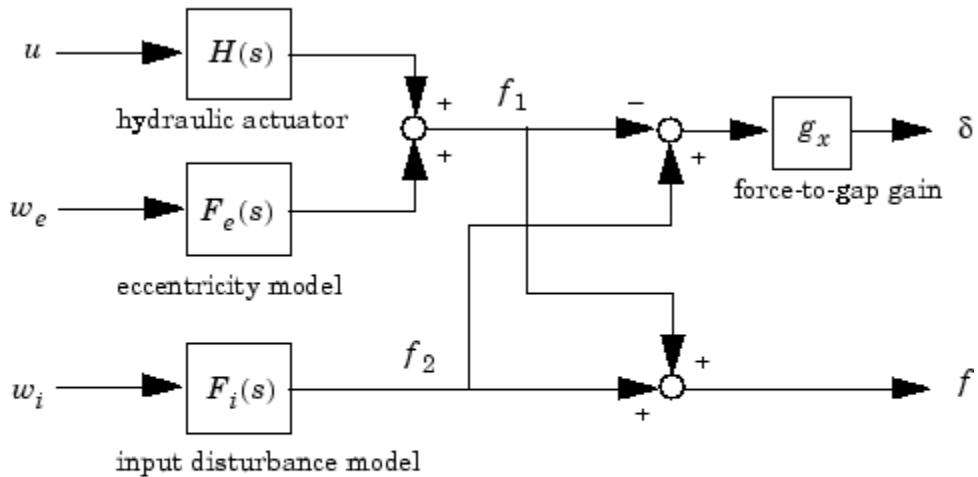
· Variations in the thickness/hardness of the incoming beam

· Eccentricity in the rolling cylinders

Feedback control is necessary to reduce the effect of these disturbances. Because the roll gap cannot be measured close to the mill stand, the rolling force is used instead for feedback.

The input thickness disturbance is modeled as a low pass filter driven by white noise. The eccentricity disturbance is approximately periodic and its frequency is a function of the rolling speed. A reasonable model for this disturbance is a second-order bandpass filter driven by white noise.

This leads to the following generic model for each axis of the rolling process.

Open-loop Model for x- or y-axis

u	command
δ	thickness gap (in mm)
f	incremental rolling force
w_i, w_e	driving white noise for disturbance models

The measured rolling force variation f is a combination of the incremental force delivered by the hydraulic actuator and of the disturbance forces due to eccentricity and input thickness variation. Note that:

- The outputs of $H(s)$, $F_e(s)$, and $F_i(s)$ are the incremental forces delivered by each component.

- An increase in hydraulic or eccentricity force *reduces* the output thickness gap δ.

- An increase in input thickness *increases* this gap.

The model data for each axis is summarized below.

Model Data for the x-Axis

$$H_x(s) = \frac{2.4 \times 10^8}{s^2 + 72s + 90^2}$$

$$F_{ix}(s) = \frac{10^4}{s + 0.05}$$

$$F_{ex}(s) = \frac{3 \times 10^4 s}{s^2 + 0.125s + 6^2}$$

$$g_x = 10^{-6}$$

Model Data for the y-Axis

$$H_y(s) = \frac{7.8 \times 10^8}{s^2 + 71s + 88^2}$$

$$F_{iy}(s) = \frac{2 \times 10^4}{s + 0.05}$$

$$F_{ey}(s) = \frac{10^5 s}{s^2 + 0.19s + 9.4^2}$$

$$g_y = 0.5 \times 10^{-6}$$

LQG Design for the x-Axis

As a first approximation, ignore the cross-coupling between the x- and y-axes and treat each axis independently. That is, design one SISO LQG regulator for each axis. The design objective is to reduce the thickness variations δ_x and δ_y due to eccentricity and input thickness disturbances.

Start with the x-axis. First specify the model components as transfer function objects.

```
% Hydraulic actuator (with input "u-x")
Hx = tf(2.4e8,[1 72 90^2],'inputname','u-x')

% Input thickness/hardness disturbance model
Fix = tf(1e4,[1 0.05],'inputn','w-ix')
```

```
% Rolling eccentricity model
Fex = tf([3e4 0],[1 0.125 6^2],'inputn','w-ex')

% Gain from force to thickness gap
gx = 1e-6;
```

Next build the open-loop model shown in "Process and Disturbance Models" on page 20-19. You could use the function `connect` for this purpose, but it is easier to build this model by elementary `append` and `series` connections.

```
% I/O map from inputs to forces f1 and f2
Px = append([ss(Hx) Fex],Fix)

% Add static gain from f1,f2 to outputs "x-gap" and "x-force"
Px = [-gx gx;1 1] * Px

% Give names to the outputs:
set(Px,'outputn',{'x-gap' 'x-force'})
```

Note To obtain minimal state-space realizations, always convert transfer function models to state space *before* connecting them. Combining transfer functions and then converting to state space may produce nonminimal state-space models.

The variable `Px` now contains an open-loop state-space model complete with input and output names.

```
Px.inputname

ans =
    'u-x'
    'w-ex'
    'w-ix'

Px.outputname

ans =
    'x-gap'
    'x-force'
```

The second output `'x-force'` is the rolling force measurement. The LQG regulator will use this measurement to drive the hydraulic actuator and reduce disturbance-induced thickness variations δ_x.

The LQG design involves two steps:

1   Design a full-state-feedback gain that minimizes an LQ performance measure of the form

$$J(u_x) = \int_0^\infty \left\{ q\delta_x^2 + ru_x^2 \right\} dt$$

2   Design a Kalman filter that estimates the state vector given the force measurements `'x-force'`.

The performance criterion $J(u_x)$ penalizes low and high frequencies equally. Because low-frequency variations are of primary concern, eliminate the high-frequency content of δ_x with the low-pass filter $30/(s + 30)$ and use the filtered value in the LQ performance criterion.

```
lpf = tf(30,[1 30])

% Connect low-pass filter to first output of Px
Pxdes = append(lpf,1) * Px
set(Pxdes,'outputn',{'x-gap*' 'x-force'})

% Design the state-feedback gain using LQRY and q=1, r=1e-4
kx = lqry(Pxdes(1,1),1,1e-4)
```

Note `lqry` expects all inputs to be commands and all outputs to be measurements. Here the command `'u-x'` and the measurement `'x-gap*'` (filtered gap) are the first input and first output of `Pxdes`. Hence, use the syntax `Pxdes(1,1)` to specify just the I/O relation between `'u-x'` and `'x-gap*'`.

Next, design the Kalman estimator with the function `kalman`. The process noise

$$w_x = \begin{bmatrix} w_{ex} \\ w_{ix} \end{bmatrix}$$

has unit covariance by construction. Set the measurement noise covariance to 1000 to limit the high frequency gain, and keep only the measured output `'x-force'` for estimator design.

```
estx = kalman(Pxdes(2,:),eye(2),1000)
```
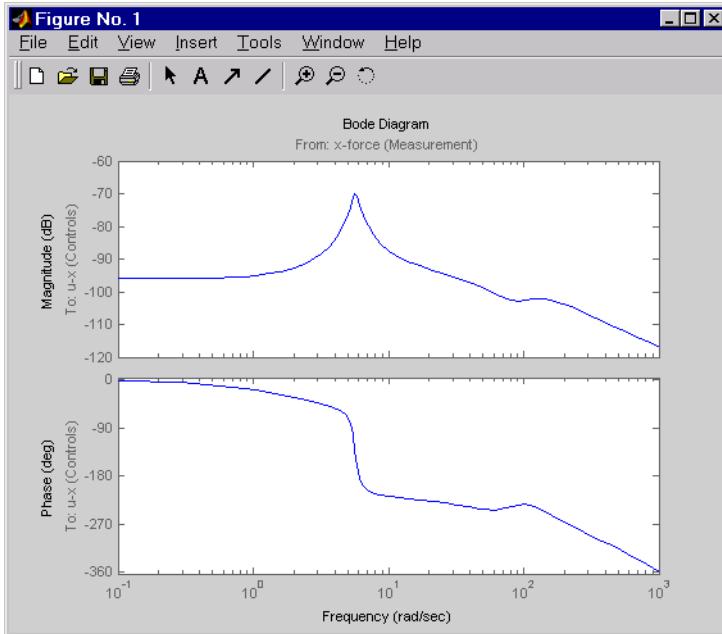
Finally, connect the state-feedback gain `kx` and state estimator `estx` to form the LQG regulator.

```
Regx = lqgreg(estx,kx)
```

This completes the LQG design for the x–axis.

Let's look at the regulator Bode response between 0.1 and 1000 rad/sec.

```
h = bodeplot(Regx,{0.1 1000})
setoptions(h,'PhaseMatching','on')
```


The phase response has an interesting physical interpretation. First, consider an increase in input thickness. This low-frequency disturbance boosts both output thickness and rolling force. Because the regulator phase is approximately 0° at low frequencies, the feedback loop then adequately reacts by increasing the hydraulic force to offset the thickness increase. Now consider the effect of eccentricity. Eccentricity causes fluctuations in the roll gap (gap between the rolling cylinders). When the roll gap is minimal, the rolling force increases and the beam thickness diminishes. The hydraulic force must then be reduced (negative force feedback) to restore the desired thickness.

This is exactly what the LQG regulator does as its phase drops to -180° near the natural frequency of the eccentricity disturbance (6 rad/sec).

Next, compare the open- and closed-loop responses from disturbance to thickness gap. Use `feedback` to close the loop. To help specify the feedback connection, look at the I/O names of the plant `Px` and regulator `Regx`.

```
Px.inputname
ans =
     'u-x'
     'w-ex'
     'w-ix'

Regx.outputname
ans =
     'u-x'

Px.outputname
ans =
     'x-gap'
     'x-force'

Regx.inputname
ans =
     'x-force'
```
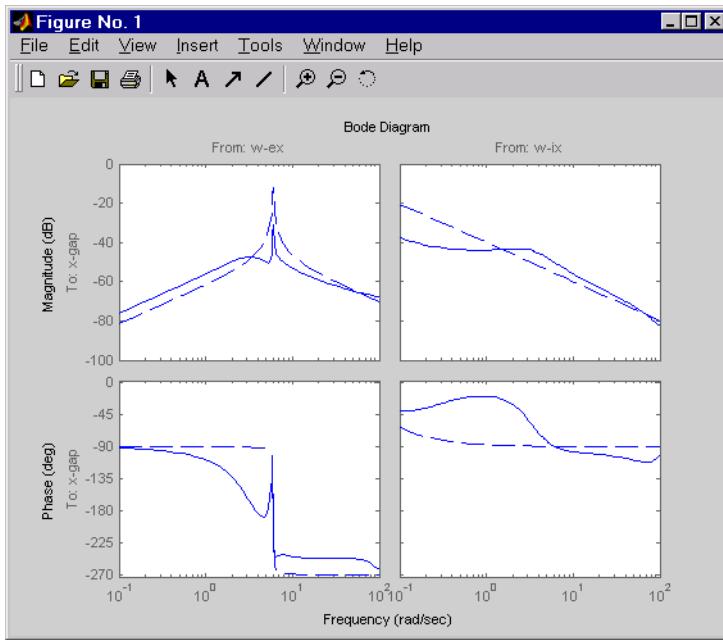
This indicates that you must connect the first input and second output of `Px` to the regulator.

```
clx = feedback(Px,Regx,1,2,+1)      % Note: +1 for positive feedback
```

You are now ready to compare the open- and closed-loop Bode responses from disturbance to thickness gap.

```
h = bodeplot(Px(1,2:3),'--',clx(1,2:3),'-',{0.1 100})
setoptions(h,'PhaseMatching','on')
```

The dashed lines show the open-loop response. Note that the peak gain of the eccentricity-to-gap response and the low-frequency gain of the input-thickness-to-gap response have been reduced by about 20 dB.
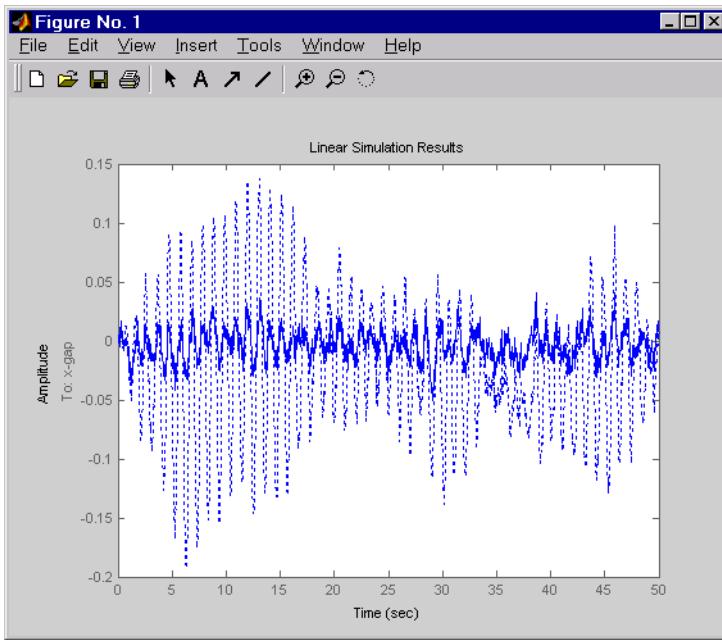
Finally, use `lsim` to simulate the open- and closed-loop time responses to the white noise inputs w_{ex} and w_{ix}. Choose `dt=0.01` as sample time for the simulation, and derive equivalent discrete white noise inputs for this sampling rate.

```
dt = 0.01
t = 0:dt:50    % time samples

% Generate unit-covariance driving noise wx = [w-ex;w-ix].
% Equivalent discrete covariance is 1/dt
wx = sqrt(1/dt) * randn(2,length(t))

lsim(Px(1,2:3),':',clx(1,2:3),'-',wx,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.

The dotted lines correspond to the open-loop response. In this simulation, the LQG regulation reduces the peak thickness variation by a factor 4.

LQG Design for the y-Axis

The LQG design for the *y*-axis (regulation of the *y* thickness) follows the exact same steps as for the *x*-axis.

```
% Specify model components
Hy = tf(7.8e8,[1 71 88^2],'inputn','u-y')
Fiy = tf(2e4,[1 0.05],'inputn','w-iy')
Fey = tf([1e5 0],[1 0.19 9.4^2],'inputn','w-ey')
gy = 0.5e-6 % force-to-gap gain

% Build open-loop model
Py = append([ss(Hy) Fey],Fiy)
Py = [-gy gy;1 1] * Py
set(Py,'outputn',{'y-gap' 'y-force'})

% State-feedback gain design
```

```
Pydes = append(lpf,1) * Py % Add low-freq. weigthing
set(Pydes,'outputn',{'y-gap*' 'y-force'})
ky = lqry(Pydes(1,1),1,1e-4)

% Kalman estimator design
esty = kalman(Pydes(2,:),eye(2),1e3)

% Form SISO LQG regulator for y-axis and close the loop
Regy = lqgreg(esty,ky)
cly = feedback(Py,Regy,1,2,+1)
```

Compare the open- and closed-loop response to the white noise input disturbances.
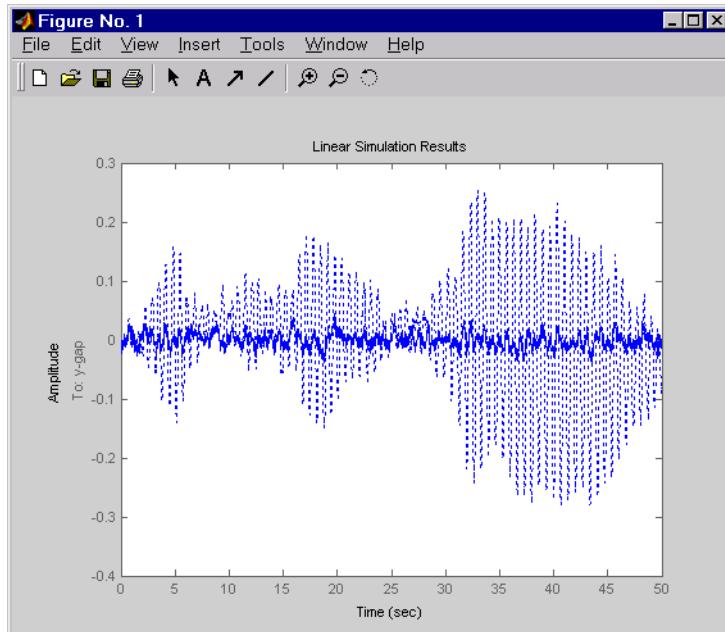
```
dt = 0.01
t = 0:dt:50
wy = sqrt(1/dt) * randn(2,length(t))

lsim(Py(1,2:3),':',cly(1,2:3),'-',wy,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.
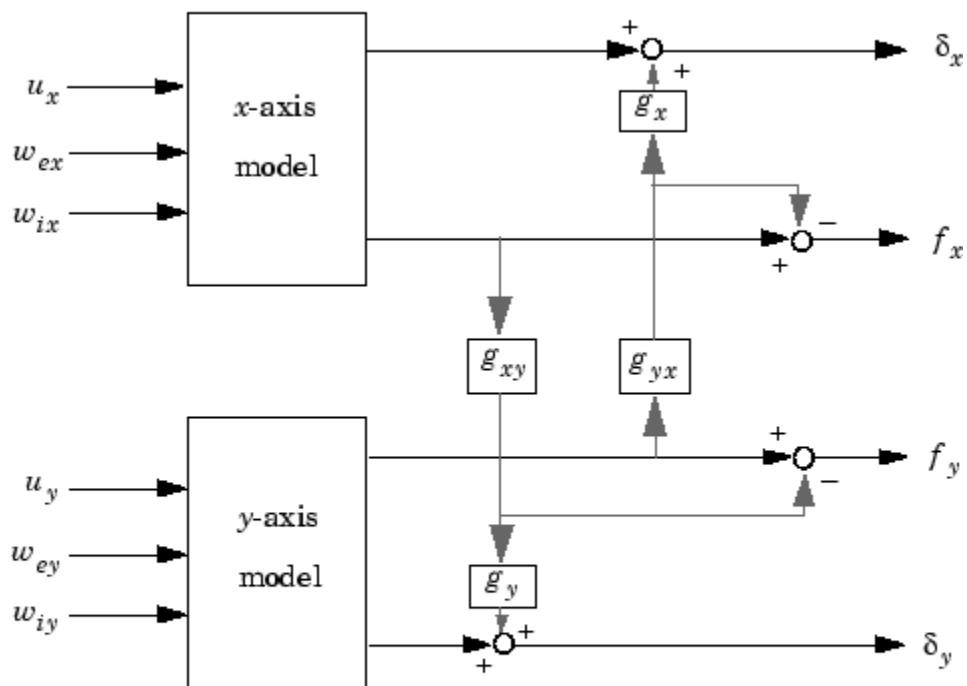
The dotted lines correspond to the open-loop response. The simulation results are comparable to those for the *x*-axis.

Cross-Coupling Between Axes

The *x/y* thickness regulation, is a MIMO problem. So far you have treated each axis separately and closed one SISO loop at a time. This design is valid as long as the two axes are fairly decoupled. Unfortunately, this rolling mill process exhibits some degree of cross-coupling between axes. Physically, an increase in hydraulic force along the *x*-axis compresses the material, which in turn boosts the repelling force on the *y*-axis cylinders. The result is an increase in *y*-thickness and an equivalent (relative) decrease in hydraulic force along the *y*-axis.

The figure below shows the coupling.

Coupling Between the x- and y-axes

$$g_{xy} = 0.1$$
$$g_{yx} = 0.4$$

Accordingly, the thickness gaps and rolling forces are related to the outputs $\bar{\delta}_x, \bar{f}_x, \ldots$ of the x- and y-axis models by

$$
\begin{bmatrix} \delta_x \\ \delta_y \\ f_x \\ f_x \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & g_{yx}g_x \\ 0 & 1 & g_{xy}g_y & 0 \\ 0 & 0 & 1 & -g_{yx} \\ 0 & 0 & -g_{xy} & 1 \end{bmatrix}}_{\text{cross-coupling matrix}} \begin{bmatrix} \bar{\delta}_x \\ \bar{\delta}_y \\ \bar{f}_x \\ \bar{f}_y \end{bmatrix}
$$

Let's see how the previous "decoupled" LQG design fares when cross-coupling is taken into account. To build the two-axes model, shown above, append the models Px and Py for the *x*- and *y*-axes.

```
P = append(Px,Py)
```

For convenience, reorder the inputs and outputs so that the commands and thickness gaps appear first.

```
P = P([1 3 2 4],[1 4 2 3 5 6])
P.outputname

ans =
    'x-gap'
    'y-gap'
    'x-force'
    'y-force'
```

Finally, place the cross-coupling matrix in series with the outputs.

```
gxy = 0.1; gyx = 0.4;
CCmat = [eye(2) [0 gyx*gx;gxy*gy 0] ; zeros(2) [1 -gyx;-gxy 1]]
Pc = CCmat * P
Pc.outputname = P.outputname
```
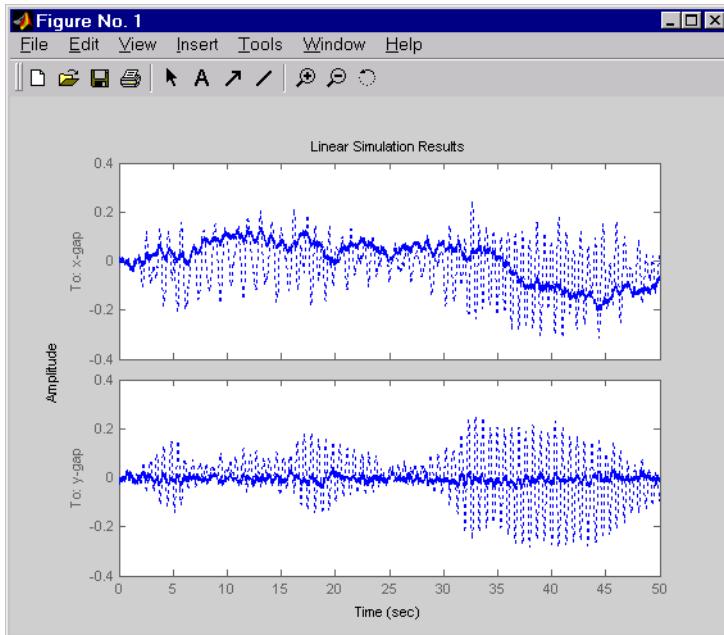
To simulate the closed-loop response, also form the closed-loop model by

```
feedin = 1:2 % first two inputs of Pc are the commands
feedout = 3:4 % last two outputs of Pc are the measurements
cl = feedback(Pc,append(Regx,Regy),feedin,feedout,+1)
```

You are now ready to simulate the open- and closed-loop responses to the driving white noises wx (for the *x*-axis) and wy (for the *y*-axis).

```
wxy = [wx ; wy]
lsim(Pc(1:2,3:6),':',cl(1:2,3:6),'-',wxy,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.

The response reveals a severe deterioration in regulation performance along the *x*-axis (the peak thickness variation is about four times larger than in the simulation without cross-coupling). Hence, designing for one loop at a time is inadequate for this level of cross-coupling, and you must perform a joint-axis MIMO design to correctly handle coupling effects.

MIMO LQG Design

Start with the complete two-axis state-panespace model `Pc` derived in "Cross-Coupling Between Axes" on page 20-30. The model inputs and outputs are

```
Pc.inputname

ans =
    'u-x'
    'u-y'
    'w-ex'
```

```
    'w-ix'
    'w_ey'
    'w_iy'

P.outputname

ans =
    'x-gap'
    'y-gap'
    'x-force'
    'y-force'
```

As earlier, add low-pass filters in series with the `'x-gap'` and `'y-gap'` outputs to penalize only low-frequency thickness variations.

```
Pdes = append(lpf,lpf,eye(2)) * Pc
Pdes.outputn = Pc.outputn
```

Next, design the LQ gain and state estimator as before (there are now two commands and two measurements).

```
k = lqry(Pdes(1:2,1:2),eye(2),1e-4*eye(2))     % LQ gain
est = kalman(Pdes(3:4,:),eye(4),1e3*eye(2))    % Kalman estimator

RegMIMO = lqgreg(est,k)      % form MIMO LQG regulator
```

The resulting LQG regulator `RegMIMO` has two inputs and two outputs.
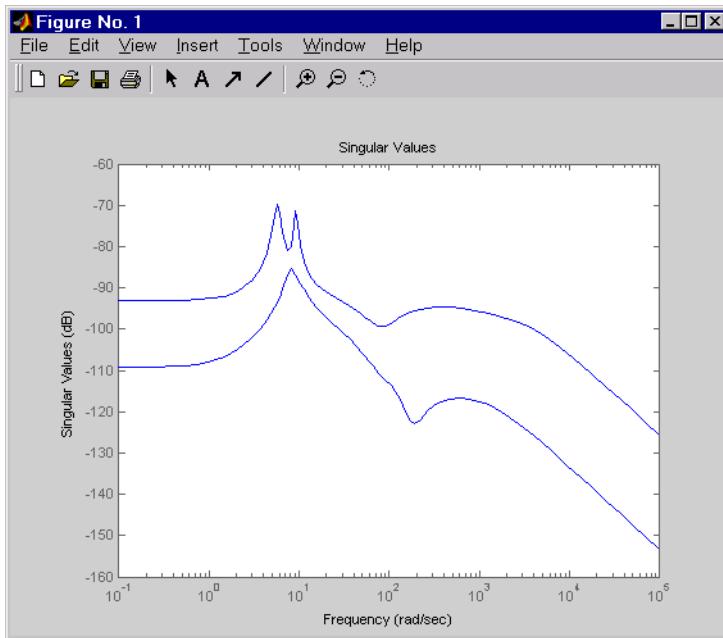
```
RegMIMO.inputname

ans =
    'x-force'
    'y-force'

RegMIMO.outputname

ans =
    'u-x'
    'u-y'
```

Plot its singular value response (principal gains).

```
sigma(RegMIMO)
```
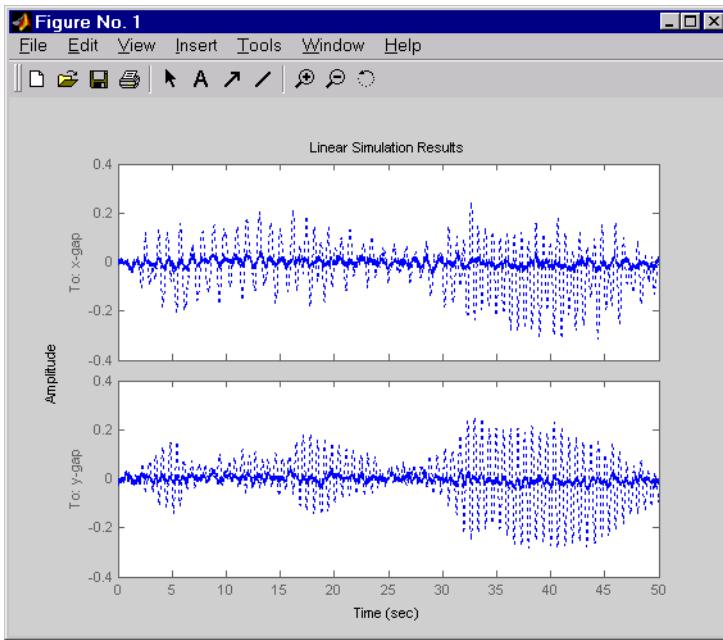
Next, plot the open- and closed-loop time responses to the white noise inputs (using the MIMO LQG regulator for feedback).

```
% Form the closed-loop model
cl = feedback(Pc,RegMIMO,1:2,3:4,+1);

% Simulate with lsim using same noise inputs
lsim(Pc(1:2,3:6),':',cl(1:2,3:6),'-',wxy,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.

The MIMO design is a clear improvement over the separate SISO designs for each axis. In particular, the level of *x/y* thickness variation is now comparable to that obtained in the decoupled case. This example illustrates the benefits of direct MIMO design for multivariable systems.

References

[1] Grimble, M.J., *Robust Industrial Control: Optimal Design Approach for Polynomial Systems*, Prentice Hall, 1994, p. 261 and pp. 443-456.

Kalman Filtering

This case study illustrates Kalman filter design and simulation. Both steady-state and time-varying Kalman filters are considered.

Overview of the Case Study

This case study illustrates Kalman filter design and simulation. Both steady-state and time-varying Kalman filters are considered.

Consider a discrete plant with additive Gaussian noise w_n on the input $u[n]$:

$$x[n+1] = Ax[n] + B(u[n] + w[n])$$
$$y[n] = Cx[n].$$

The following matrices represent the dynamics of this plant.

```
A = [1.1269    -0.4940      0.1129;
     1.0000          0          0;
          0     1.0000          0];

B = [-0.3832;
      0.5919;
      0.5191];

C = [1 0 0];
```

Discrete Kalman Filter

The equations of the steady-state Kalman filter for this problem are given as follows.

- Measurement update:

  $$\hat{x}[n|n] = \hat{x}[n|n-1] + M(y_v[n] - C\hat{x}[n|n-1])$$

- Time update:

  $$\hat{x}[n+1|n] = A\hat{x}[n|n] + Bu[n]$$

In these equations:

- $\hat{x}[n|n-1]$ is the estimate of $x[n]$, given past measurements up to $y_v[n-1]$.

- $\hat{x}\,[n|n]$ is the updated estimate based on the last measurement $y_v\,[n]$.

Given the current estimate $\hat{x}\,[n|n]$, the time update predicts the state value at the next sample $n + 1$ (one-step-ahead predictor). The measurement update then adjusts this prediction based on the new measurement $y_v\,[n + 1]$. The correction term is a function of the innovation, that is, the discrepancy between the measured and predicted values of $y\,[n + 1]$. This discrepancy is given by:

$$y_v\,[n + 1] - C\hat{x}\,[n + 1\,|n]$$

The innovation gain M is chosen to minimize the steady-state covariance of the estimation error, given the noise covariances:

$$E\left(w\,[n]\,w[n]^T\right) = Q \qquad E\left(v\,[n]\,v[n]^T\right) = R \qquad N = E\left(w\,[n]\,v[n]^T\right) = 0$$

You can combine the time and measurement update equations into one state-space model, the Kalman filter:

$$\hat{x}\,[n + 1\,|n] = A\,(I - MC)\,\hat{x}\,[n\,|n - 1] + \begin{bmatrix} B & AM \end{bmatrix} \begin{bmatrix} u\,[n] \\ y_v\,[n] \end{bmatrix}$$
$$\hat{y}\,[n|\,n] = C\,(I - MC)\,\hat{x}\,[n\,|n - 1] + CMy_v\,[n]\,.$$

This filter generates an optimal estimate $\hat{y}\,[n|n]$ of y_n. Note that the filter state is $\hat{x}\,[n|n - 1]$.

Steady-State Design

You can design the steady-state Kalman filter described above with the function `kalman`. First specify the plant model with the process noise:

$$x\,[n + 1] = Ax\,[n] + Bu\,[n] + Bw\,[n]$$
$$y\,[n] = Cx\,[n]$$

Here, the first expression is the state equation, and the second is the measurement equation.

The following command specifies this plant model. The sample time is set to -1, to mark the model as discrete without specifying a sample time.

```
Plant = ss(A,[B B],C,0,-1,'inputname',{'u' 'w'},'outputname','y');
```

Assuming that $Q = R = 1$, design the discrete Kalman filter.

```
Q = 1;
R = 1;
[kalmf,L,P,M] = kalman(Plant,Q,R);
```
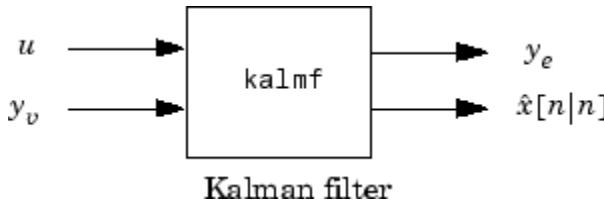
This command returns a state-space model `kalmf` of the filter, as well as the innovation gain `M`.

```
M
```

```
M =

    0.3798
    0.0817
   -0.2570
```

The inputs of `kalmf` are u and y_v, and. The outputs are the plant output and the state estimates, $y_e = \hat{y}\,[n|n]$ and $\hat{x}\,[n|n]$.


Kalman filter
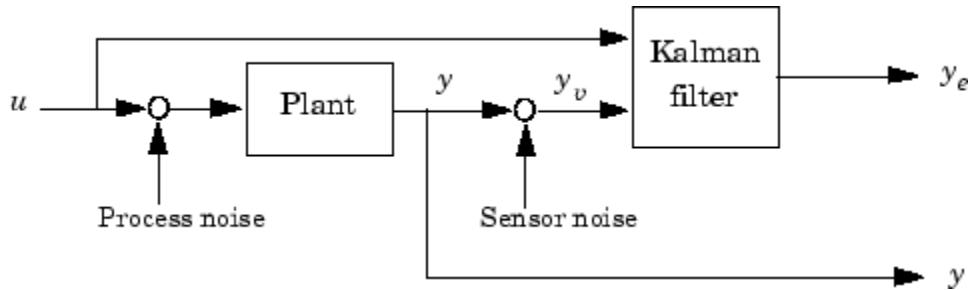
Because you are interested in the output estimate y_e, select the first output of `kalmf` and discard the rest.

```
kalmf = kalmf(1,:);
```

To see how the filter works, generate some input data and random noise and compare the filtered response y_e with the true response y. You can either generate each response separately, or generate both together. To simulate each response separately, use `lsim` with the plant alone first, and then with the plant and filter hooked up together. The joint simulation alternative is detailed next.

The block diagram below shows how to generate both true and filtered outputs.
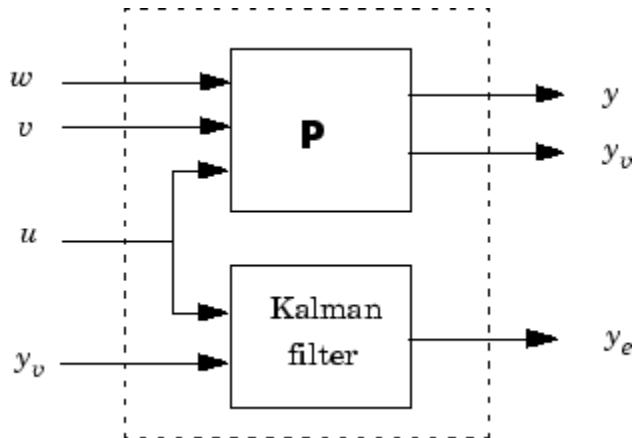
You can construct a state-space model of this block diagram with the functions `parallel` and `feedback`. First build a complete plant model with *u*, *w*, *v* as inputs, and *y* and y_v (measurements) as outputs.

```
a = A;
b = [B B 0*B];
c = [C;C];
d = [0 0 0;0 0 1];
P = ss(a,b,c,d,-1,'inputname',{'u' 'w' 'v'},'outputname',{'y' 'yv'});
```

Then use `parallel` to form the parallel connection of the following illustration.


```
sys = parallel(P,kalmf,1,1,[],[]);
```

Finally, close the sensor loop by connecting the plant output y_v to filter input y_v with positive feedback.

```
SimModel = feedback(sys,1,4,2,1);   % Close loop around input #4 and output #2
SimModel = SimModel([1 3],[1 2 3]); % Delete yv from I/O list
```

The resulting simulation model has w, v, u as inputs, and y and y_e as outputs. View the
`InputName` and `OutputName` properties to verify.

```
SimModel.InputName
```

```
ans =

    'w'
    'v'
    'u'
```

```
SimModel.OutputName
```

```
ans =

    'y'
    'y_e'
```

You are now ready to simulate the filter behavior. Generate a sinusoidal input u and
process and measurement noise vectors w and v.

```
t = [0:100]';
u = sin(t/5);

n = length(t);
rng default
w = sqrt(Q)*randn(n,1);
v = sqrt(R)*randn(n,1);
```
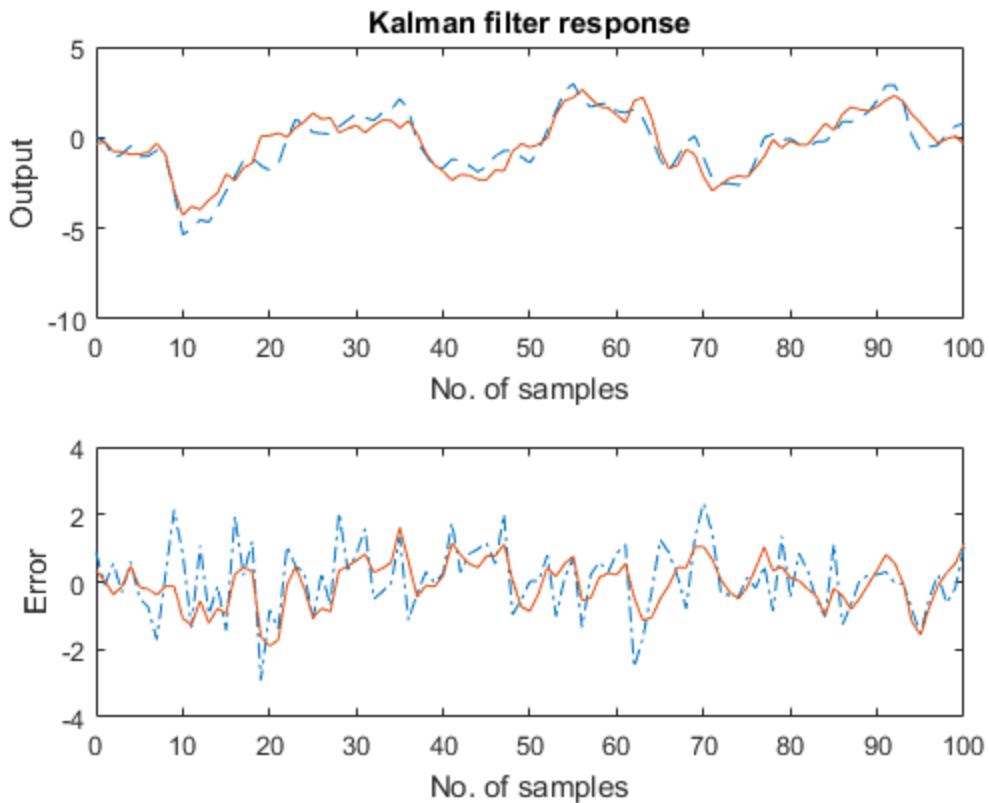
Simulate the responses.

```
[out,x] = lsim(SimModel,[w,v,u]);

y = out(:,1);    % true response
ye = out(:,2);   % filtered response
yv = y + v;      % measured response
```

Compare the true and filtered responses graphically.

```
subplot(211), plot(t,y,'--',t,ye,'-'),
xlabel('No. of samples'), ylabel('Output')
title('Kalman filter response')
subplot(212), plot(t,y-yv,'-.',t,y-ye,'-'),
xlabel('No. of samples'), ylabel('Error')
```


The first plot shows the true response y (dashed line) and the filtered output y_e (solid line). The second plot compares the measurement error (dash-dot) with the estimation error (solid). This plot shows that the noise level has been significantly reduced. This is confirmed by calculating covariance errors. The error covariance before filtering (measurement error) is:

```
MeasErr = y-yv;
```

```
MeasErrCov = sum(MeasErr.*MeasErr)/length(MeasErr)


MeasErrCov =

    0.9992
```

The error covariance after filtering (estimation error) is reduced:

```
EstErr = y-ye;
EstErrCov = sum(EstErr.*EstErr)/length(EstErr)


EstErrCov =

    0.4944
```

Time-Varying Kalman Filter

The time-varying Kalman filter is a generalization of the steady-state filter for time-varying systems or LTI systems with nonstationary noise covariance.

Consider the following plant state and measurement equations.

$$\begin{array}{rcl} x\,[n+1] & = & Ax\,[n] + Bu\,[n] + Gw\,[n] \\ y_v\,[n] & = & Cx\,[n] + v\,[n]\,. \end{array}$$

The time-varying Kalman filter is given by the following recursions:

• Measurement update:

$$\begin{array}{rcl} \hat{x}\,[n|n] & = & \hat{x}\,[n|n-1] + M\,[n]\,(y_v\,[n] - C\hat{x}\,[n|n-1]) \\ M\,[n] & = & P\,[n|n-1]\,C^T(R\,[n] + CP\,[n|n-1]\,C^T)^{-1} \\ P\,[n|n] & = & (I - M\,[n]\,C)P\,[n|n-1]\,. \end{array}$$

• Time update:

$$\begin{array}{rcl} \hat{x}\,[n+1|n] & = & A\hat{x}\,[n|n] + Bu\,[n] \\ P\,[n+1|n] & = & AP\,[n|n]\,A^T + GQ\,[n]\,G^T\,. \end{array}$$

Here, $\hat{x}\,[n|n-1]$ and $\hat{x}\,[n|n]$ are as described previously. Additionally:

$$Q\,[n] = E(w\,[n]\,w[n]^T)$$
$$R\,[n] = E(v\,[n]\,v[n]^T)$$
$$P\,[n|n] = E(\{x\,[n] - x\,[n|n]\}\,\{x\,[n] - x\,[n|n]\}^T)$$
$$P\,[n|n-1] = E(\{x\,[n] - x\,[n|n-1]\}\,\{x\,[n] - x\,[n|n-1]\}^T).$$

For simplicity, the subscripts indicating the time dependence of the state-space matrices have been dropped.

Given initial conditions $x\,[1|0]$ and $P\,[1|0]$, you can iterate these equations to perform the filtering. You must update both the state estimates $x\,[n|.]$ and error covariance matrices $P\,[n|.]$ at each time sample.

Time-Varying Design

To implement these filter recursions, first genereate noisy output measurements. Use the process noise w and measurement noise v generated previously.

```
sys = ss(A,B,C,0,-1);
y = lsim(sys,u+w);
yv = y + v;
```

Assume the following initial conditions:

$$x\,[1|0] = 0, \quad P\,[1|0]\ = BQB^T$$

Implement the time-varying filter with a `for` loop.

```
P = B*Q*B';           % Initial error covariance
x = zeros(3,1);       % Initial condition on the state
ye = zeros(length(t),1);
ycov = zeros(length(t),1);

for i = 1:length(t)
  % Measurement update
  Mn = P*C'/(C*P*C'+R);
  x = x + Mn*(yv(i)-C*x);    % x[n|n]
  P = (eye(3)-Mn*C)*P;       % P[n|n]

  ye(i) = C*x;
```

```
   errcov(i) = C*P*C';

   % Time update
   x = A*x + B*u(i);        % x[n+1|n]
   P = A*P*A' + B*Q*B';     % P[n+1|n]
end
```
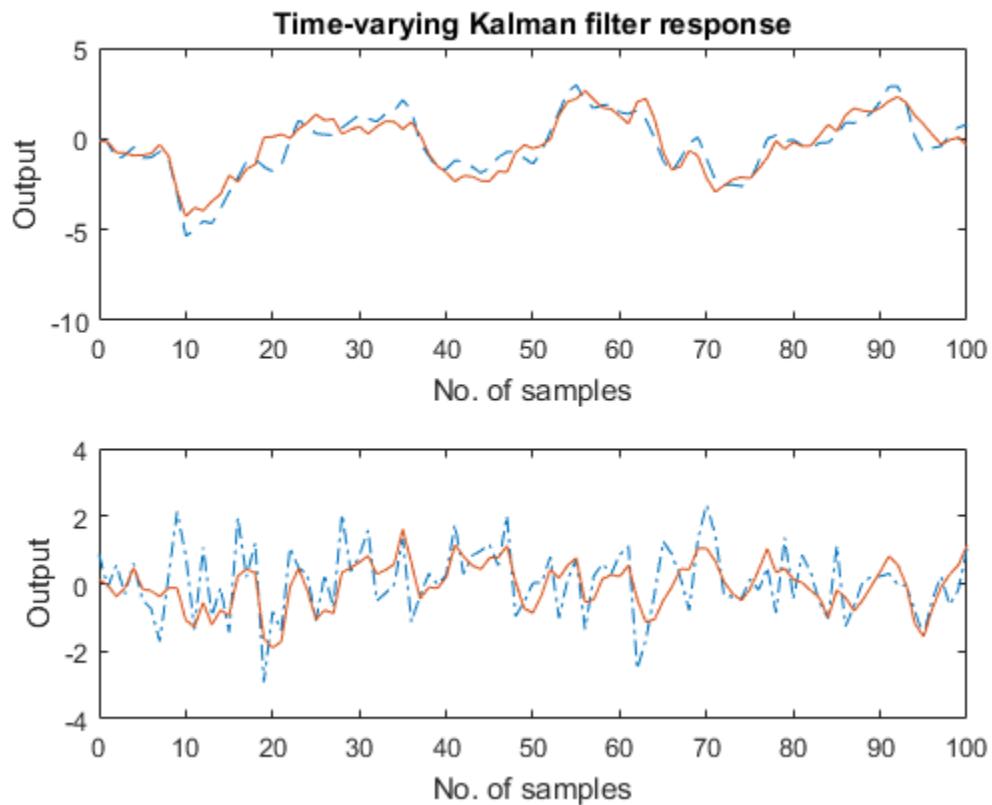
Compare the true and estimated output graphically.

```
subplot(211), plot(t,y,'--',t,ye,'-')
title('Time-varying Kalman filter response')
xlabel('No. of samples'), ylabel('Output')
subplot(212), plot(t,y-yv,'-.',t,y-ye,'-')
xlabel('No. of samples'), ylabel('Output')
```
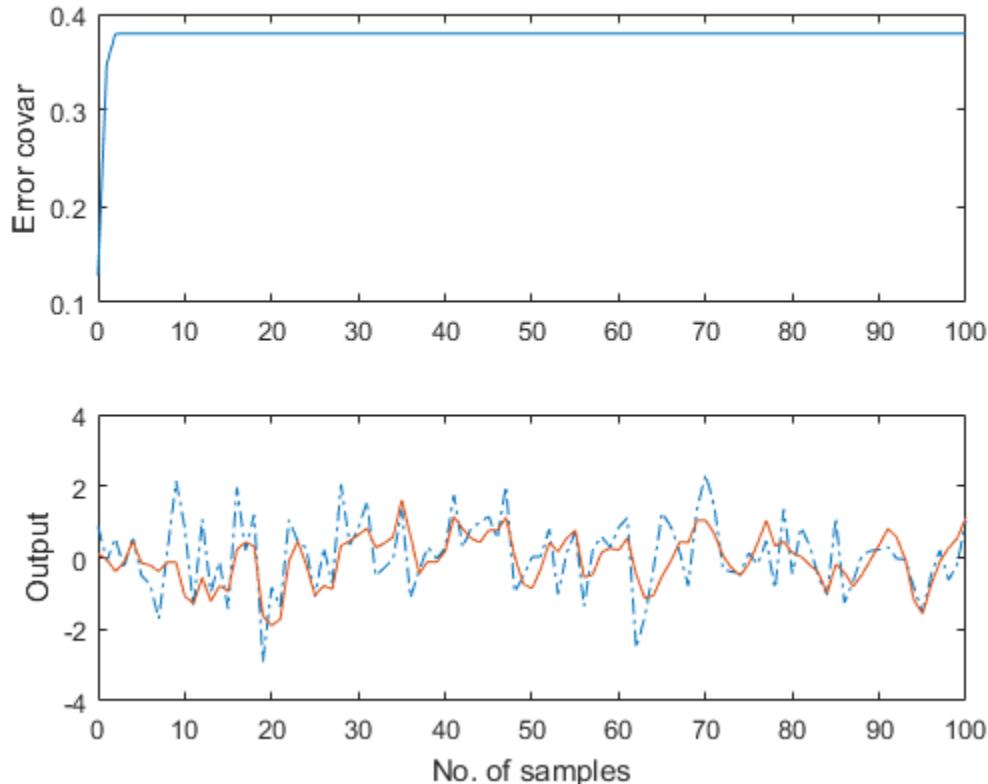
The first plot shows the true response y (dashed line) and the filtered response y_e (solid line). The second plot compares the measurement error (dash-dot) with the estimation error (solid).

The time-varying filter also estimates the covariance `errcov` of the estimation error $y - y_e$ at each sample. Plot it to see if your filter reached steady state (as you expect with stationary input noise).

```
subplot(211)
plot(t,errcov), ylabel('Error covar')
```


From this covariance plot, you can see that the output covariance did indeed reach a steady state in about five samples. From then on, your time-varying filter has the same performance as the steady-state version.

Compare with the estimation error covariance derived from the experimental data:

```
EstErr = y - ye;
EstErrCov = sum(EstErr.*EstErr)/length(EstErr)


EstErrCov =

    0.4934
```

This value is smaller than the theoretical value `errcov` and close to the value obtained for the steady-state design.

Finally, note that the final value $M[n]$ and the steady-state value M of the innovation gain matrix coincide.

```
Mn


Mn =

    0.3798
    0.0817
   -0.2570


M


M =

    0.3798
    0.0817
   -0.2570
```

Bibliography

[1] Grimble, M.J., *Robust Industrial Control: Optimal Design Approach for Polynomial Systems*, Prentice Hall, 1994, p. 261 and pp. 443-456.

Reliable Computations

Scaling State-Space Models

In this section...

Why Scaling Is Important

When working with state-space models, proper scaling is important for accurate computations. A state-space model is well scaled when the following conditions exist:

- The entries of the A, B, and C matrices are homogenous in magnitude.
- The model characteristics are insensitive to small perturbations in A, B, and C (in comparison to their norms).

Working with poorly scaled models can cause your model a severe loss of accuracy and puzzling results. An example of a poorly scaled model is a dynamic system with two states in the state vector that have units of light years and millimeters. Such disparate units may introduce both very large and very small entries into the A matrix. Over the course of computations, this mix of small and large entries in the matrix could destroy important characteristics of the model and lead to incorrect results.

For more information on the harmful affects of a poorly scaled model, see Scaling Models to Maximize Accuracy.

When to Scale Your Model

You can avoid scaling issues altogether by carefully selecting units to reduce the spread between small and large coefficients.

In general, you do not have to perform your own scaling when using the Control System Toolbox software. The algorithms automatically scale your model to prevent loss of accuracy. The automated scaling chooses a frequency range to maximize accuracy based on the dominant dynamics of the model.

In most cases, automated scaling provides high accuracy without your intervention. For some models with dynamics spanning a wide frequency range, however, it is

impossible to achieve good accuracy at *all* frequencies and some tradeoff of accuracy in different frequency bands is necessary. In such cases, a warning alerts you of potential inaccuracies. If you receive this warning, evaluate the tradeoffs and consider manually adjusting the frequency interval where you most need high accuracy. For information on how to manually scale your model, see "Manually Scaling Your Model" on page 21-3.

Note: For models with satisfactory scaling, you can bypass automated scaling in the Control System Toolbox software. To do so, set the `Scaled` property of your state-space model to `1` (true). For information on how to set this property, see the `set` reference page.

Manually Scaling Your Model

If automatic scaling produces a warning, you can use the `prescale` command to manually scale your model and adjust the frequency interval where you most need high accuracy.

The `prescale` command includes a Scaling Tool GUI, which you can use to visualize accuracy tradeoffs and to adjust the frequency interval where this accuracy is maximized.

To scale your model using the Scaling Tool GUI, you perform the following steps:

- "Opening the Scaling Tool GUI" on page 21-3
- "Specifying the Frequency Axis Limits in the Scaling Tool GUI" on page 21-5
- "Specifying the Frequency Band for Maximum Accuracy in the Scaling Tool GUI" on page 21-5
- "Saving the Scaling in the Scaling Tool GUI" on page 21-6
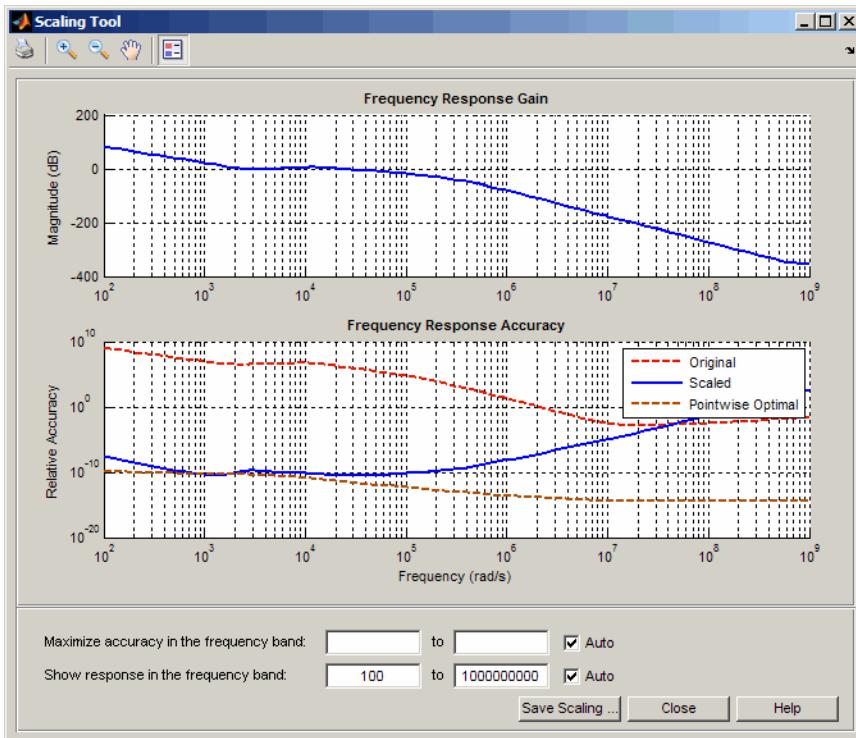
For an example of using the Scaling Tool GUI on a real model, see Scaling Models to Maximize Accuracy.

For more information about scaling models from the command line, see the `prescale` reference page.

Opening the Scaling Tool GUI

To open the Scaling Tool GUI for a state-space model named `sys`, type

```
prescale(sys)
```

The Scaling Tool GUI resembles one shown in the following figure.

The Scaling Tool GUI contains the following plots:

- The **Frequency Response Gain** plot helps you determine the frequency band over which you want to maximize scaling.

  For SISO systems, this plot shows the gain of your model. For MIMO systems, the plot shows the principle gain (largest singular value) of your model.

- The **Frequency Response Accuracy** plot allows you to view the accuracy tradeoffs for your model when maximizing accuracy in a particular frequency bands.

  This plot shows the following information:

  - Relative accuracy of the response of the original unscaled model in red
  - Relative accuracy of the response of the scaled model in blue
  - Best achievable accuracy when using independent scaling at each frequency in brown

When you compute some model characteristics, such as the frequency response or the system zeros, the software produces the exact answer for some perturbation of the model you specified. The *relative accuracy* is a measure of the worst-case relative gap between the frequency response of the original and perturbed models. The perturbation accounts for rounding errors during calculation. Any relative accuracy value greater than 1 implies poor accuracy.

Tip If the blue Scaled curve is close to the brown Pointwise Optimal curve in a particular frequency band, you already have the best possible accuracy in that frequency band.

Specifying the Frequency Axis Limits in the Scaling Tool GUI

You can change the limits of the plot axis to view a particular frequency band of interest in the Scaling Tool GUI. To view a particular frequency band, specify the band in the **Show response in the frequency band** fields.

This action updates the frequency axis of the Scaling tool to show the specified frequency band.

Tip To return to the default display, select the **Auto** check box.

Specifying the Frequency Band for Maximum Accuracy in the Scaling Tool GUI

To adjust the frequency band where you want maximum accuracy, set a new frequency band in the **Maximize accuracy in the frequency band** fields. You can visualize accuracy tradeoffs by trying out different frequency bands and viewing the resulting relative accuracy across the frequency band of interest.

Note: You can use the **Frequency Response Gain** plot, which plots the gain of your model, to view the dynamics in your model to help determine the frequency band to maximize accuracy.

Each time you specify a new frequency band, the **Frequency Response Accuracy** plot updates with the result of the new scaling. Compare the Scaled curve (blue) to the

Pointwise Optimal curve (brown) to determine where the new scaling is nearly optimal and where you need more accuracy.
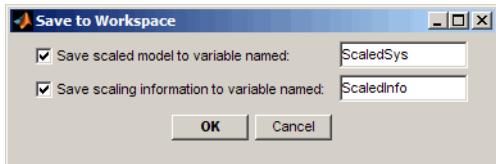

Tip To return to the default scaling, select the **Auto** check box.

Saving the Scaling in the Scaling Tool GUI

When you find a good scaling for your model, save the scaled model as follows:

1 Click **Save Scaling**.

This action opens the **Save to Workspace** dialog box.

2 In the **Save to Workspace** dialog box, verify that any of the following items you want to save are selected, and specify variable names for these items.

- Scaled model
- Scaling information, including:

  - Scaling factors
  - Frequencies used to test accuracy
  - Relative accuracy at each test frequency

  For details about the scaling information, see the `prescale` reference page.

3 Click **OK**.

This action sets the State-Space (@ss) object `Scaled` property of your model to true. When you set this property to `True`, the Control System Toolbox algorithms skip the automated scaling of the model.

Linear System Analyzer

Linear System Analyzer Overview

The Linear System Analyzer app simplifies the analysis of linear, time-invariant systems. Use Linear System Analyzer to view and compare the response plots of SISO and MIMO systems, or of several linear models at the same time. You can generate time and frequency response plots to inspect key response parameters, such as rise time, maximum overshoot, and stability margins.
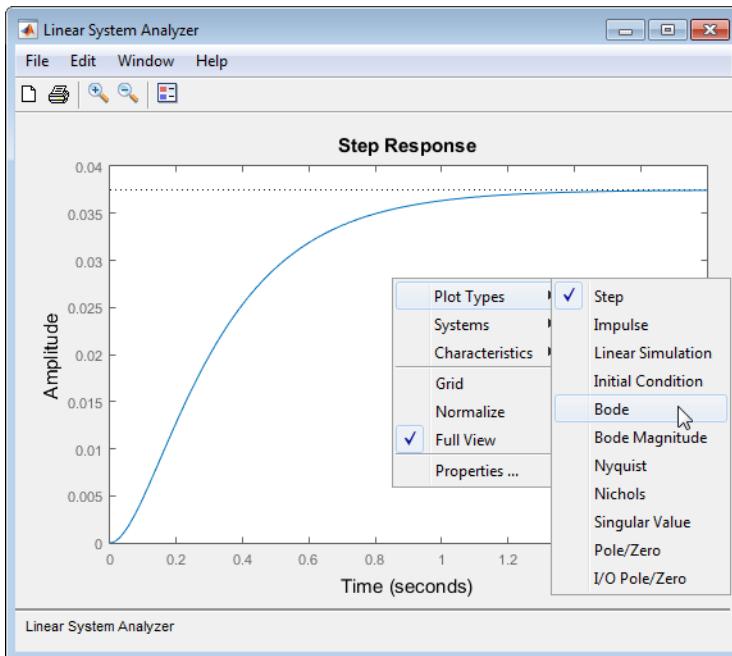
You can launch the Linear System Analyzer in two ways:

- Use the `linearSystemAnalyzer` command.
- In MATLAB, on the **Apps** tab under **Control System Design and Analysis**, click the app icon.

The easiest way to work with the Linear System Analyzer is to use the right-click menus. For example, type

```
load ltiexamples
linearSystemAnalyzer(sys_dc)
```

at the MATLAB prompt. The default plot is a step response.

The Linear System Analyzer can display up to seven different plot types simultaneously, including step, impulse, Bode (magnitude and phase or magnitude only), Nyquist, Nichols, sigma, pole/zero, and I/O pole/zero.

For examples of how to use the Linear System Analyzer, see "Linear Analysis Using the Linear System Analyzer". For more detailed information about Linear System Analyzer menus and options, see:

- "Using the Right-Click Menu in the Linear System Analyzer" on page 22-4
- "Importing, Exporting, and Deleting Models in the Linear System Analyzer" on page 22-9
- "Selecting Response Types" on page 22-12
- "Analyzing MIMO Models" on page 22-17
- "Customizing the Linear System Analyzer" on page 22-23

Using the Right-Click Menu in the Linear System Analyzer

In this section...
"Overview of the Right-Click Menu" on page 22-4
"Setting Characteristics of Response Plots" on page 22-4

Overview of the Right-Click Menu

The quickest way to manipulate views in the Linear System Analyzer is use the right-click menu. You can access several Linear System Analyzer controls and options, including:

- **Plot Type** — Changes the plot type
- **Systems** — Selects or deselects any of the models loaded in the Linear System Analyzer
- **Characteristics** — Displays key response characteristics and parameters
- **Grid** — Adds grids to your plot
- **Properties** — Opens the **Property Editor**, where you can customize plot attributes

In addition to right-click menus, all response plots include data markers. These allow you to scan the plot data, identify key data, and determine the source system for a given plot.
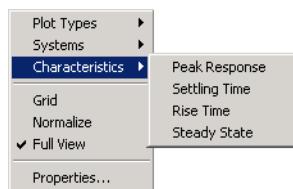
Setting Characteristics of Response Plots

The **Characteristics** menu changes for each plot response type. Characteristics refers to response plot information, such as peak response, or, in some cases, rise time and settling time.

The next sections describe the menu items for each of the eight plot types.

Step Response

**Step** plots the model's response to a step input.
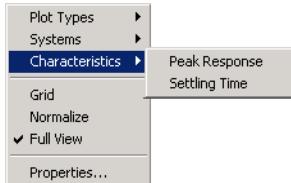
You can display the following information in the step response:

- **Peak Response** — The largest deviation from the steady-state value of the step response
- **Settling Time** — The time required for the step response to decline and stay at 5% of its final value
- **Rise Time** — The time require for the step response to rise from 10% to 90% of its final value
- **Steady-State** — The final value for the step response

Note You can change the definitions of settling time and rise time using the **Characteristics** pane of the ???, the "Linear System Analyzer Preferences Editor" on page 18-2, or the ???.

Impulse Response

**Impulse Response** plots the model's response to an impulse.
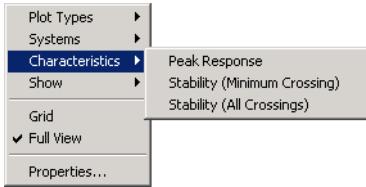
The Linear System Analyzer can display the following information in the impulse response:

- **Peak Response** — The maximum positive deviation from the steady-state value of the impulse response
- **Settling Time** — The time required for the step response to decline and stay at 5% of its final value

Bode Diagram
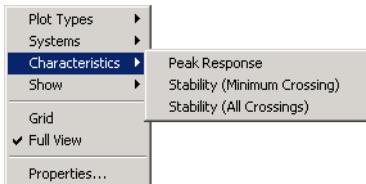
Bode plots the open-loop Bode phase and magnitude diagrams for the model.

The Linear System Analyzer can display the following information in the Bode diagram:

- **Peak Response** — The maximum value of the Bode magnitude plot over the specified region

- **Stability Margins (Minimum Crossing)** — The minimum phase and gain margins. The gain margin is defined to the gain (in dB) when the phase first crosses -180°. The phase margin is the distance, in degrees, of the phase from -180° when the gain magnitude is 0 dB.

- **Stability Margins (All Crossings)** — Display all stability margins

Bode Magnitude
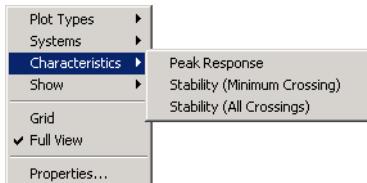
Bode Magnitude plots the Bode magnitude diagram for the model.

The Linear System Analyzer can display the following information in the Bode magnitude diagram:

- **Peak Response**, which is the maximum value of the Bode magnitude in decibels (dB), over the specified range of the diagram.

- **Stability (Minimum Crossing)** — The minimum gain margins. The gain margin is defined to the gain (in dB) when the phase first crosses -180°.

- **Stability (All Crossings)** — Display all gain stability margins

Nyquist Diagrams
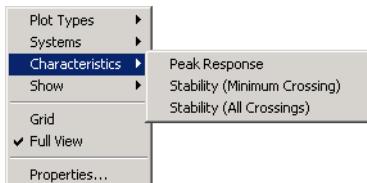
Nyquist plots the Nyquist diagram for the model.

The Linear System Analyzer can display the following types of information in the Nyquist diagram:

• **Peak Response** — The maximum value of the Nyquist diagram over the specified region

• **Stability (Minimum Crossing)** — The minimum gain and phase margins for the Nyquist diagram. The gain margin is the distance from the origin to the phase crossover of the Nyquist curve. The phase crossover is where the curve meets the real axis. The phase margin is the angle subtended by the real axis and the gain crossover on the circle of radius 1.

• **Stability (All Crossings)** — Display all gain stability margins

Nichols Charts

Nichols plots the Nichols Chart for the model.
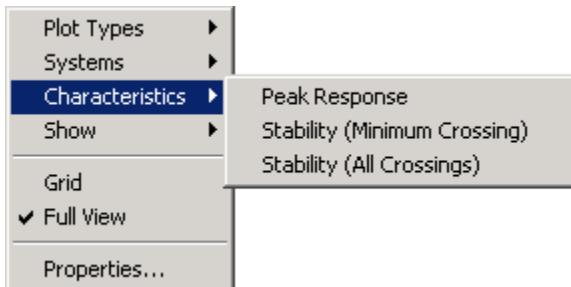
The Linear System Analyzer can display the following types of information in the Nichols chart:

• **Peak Response** — The maximum value of the Nichols chart in the plotted region.

• **Stability (Minimum Crossing)** — The minimum gain and phase margins for the Nichols chart.

- **Stability (All Crossings)** — Display all gain stability margins

Singular Values

Singular Values plots the singular values for the model.

The Linear System Analyzer can display the **Peak Response**, which is the largest magnitude of the Singular Values curve over the plotted region.

Pole/Zero and I/O Pole/Zero

Pole/Zero plots the poles and zeros of the model with `x' for poles and `o' for zeros. I/O Pole/Zero plots the poles and zeros of I/O pairs.

There are no **Characteristics** available for pole-zero plots.

See Also
linearSystemAnalyzer

Related Examples
- "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About
- "Linear System Analyzer Overview" on page 22-2
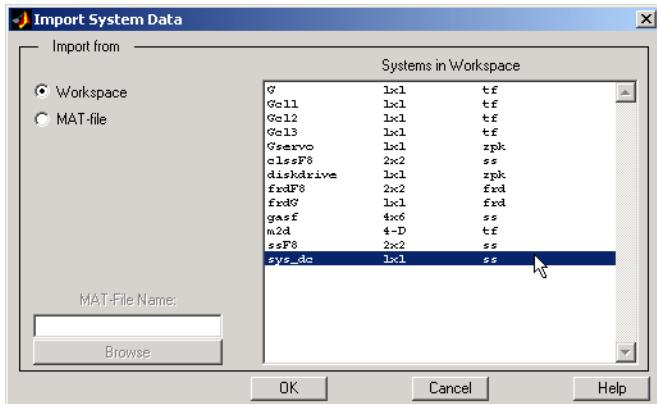
Importing, Exporting, and Deleting Models in the Linear System Analyzer

In this section...
"Importing Models" on page 22-9
"Exporting Models" on page 22-10
"Deleting Models" on page 22-10

Importing Models

To import models into the Linear System Analyzer, select **File** > **Import**. The **Import System Data** dialog box opens, as shown below.

Use the **Import System Data** dialog box to import LTI models into or from the Linear System Analyzer workspace.

To import a model:

- Click on the desired model in the LTI Browser List. To perform multiple selections:

  - Hold the Control key and click on nonadjacent models.
  - Hold the Shift key while clicking to select multiple adjacent models.

- Click the **OK** or **Apply** Button

Note that the **LTI Browser** lists only the LTI models in the MATLAB workspace.
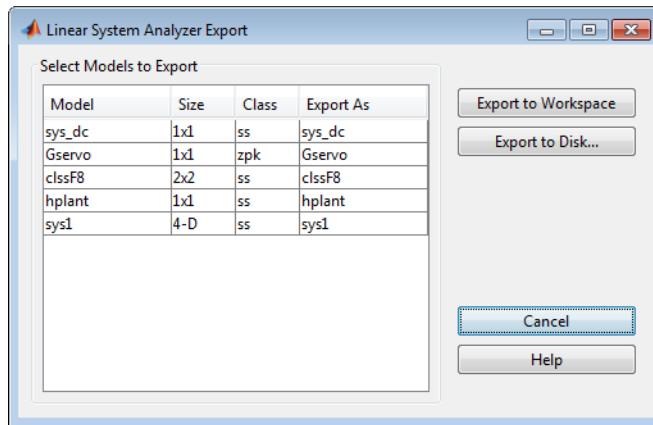
Alternatively, you can directly import a model into the Linear System Analyzer using the linearSystemAnalyzer function, as in

```
linearSystemAnalyzer({'step','bode'},modelname)
```

See the `linearSystemAnalyzer` reference page for more information.

Exporting Models

Use **Export** in the **File** menu to open the **Linear System Analyzer Export** window, shown below.
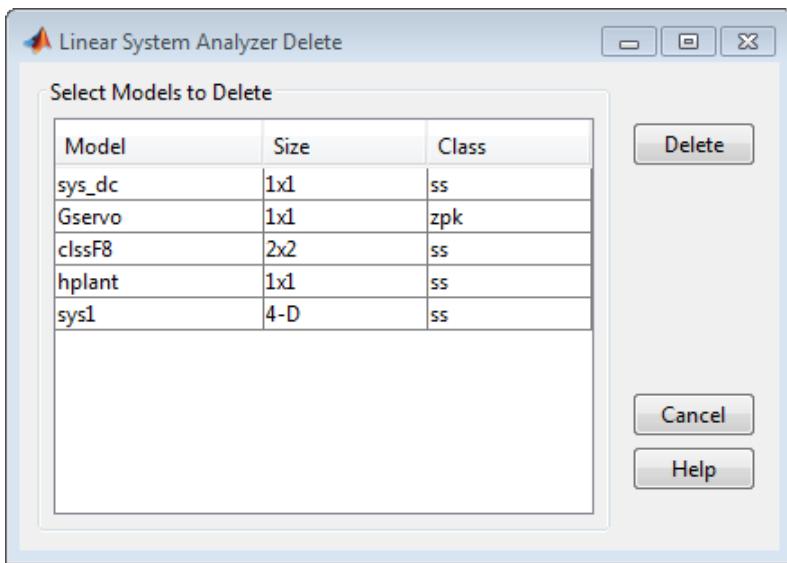

The **Linear System Analyzer Export** window lists all the models with responses currently displayed in your Linear System Analyzer. You can export models back to the MATLAB workspace or to disk.

To export single or multiple models, follow the steps described in the importing models section above. To save your models to disk in a MAT-file, choose **Export to Disk**.

Deleting Models

To remove models from the Linear System Analyzer workspace, select **Edit > Delete Systems**. The **Linear System Analyzer Delete** dialog box opens.

To delete a model:

- Click on the desired model in the Model list. To perform multiple selections:

    a   Click and drag over several variables in the list.

    b   Hold the Control key and click on individual variables.

    c   Hold the Shift key while clicking, to select a range.

Click the **Delete** button.

See Also
linearSystemAnalyzer

More About
- "Linear System Analyzer Overview" on page 22-2

Selecting Response Types

| **In this section...** |
| --- |
| |
| |
| |
| |

Methods for Selecting Response Types

There are two methods for selecting response plots in the Linear System Analyzer:

- Selecting **Plot Type** from the right-click menus
- Opening the **Plot Configurations** window

Right Click Menu: Plot Type
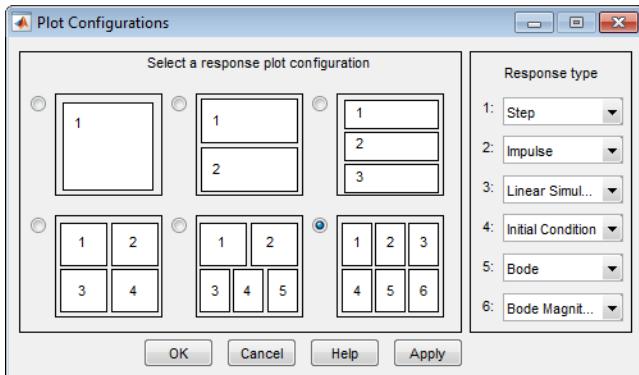
If you have a plot open in the Linear System Analyzer, you can switch to any other response plot available by selecting **Plot Type** from the right click menu.

To change the response plot, select the new plot type from the **Plot Type** submenu. The Linear System Analyzer automatically displays the new response plot.
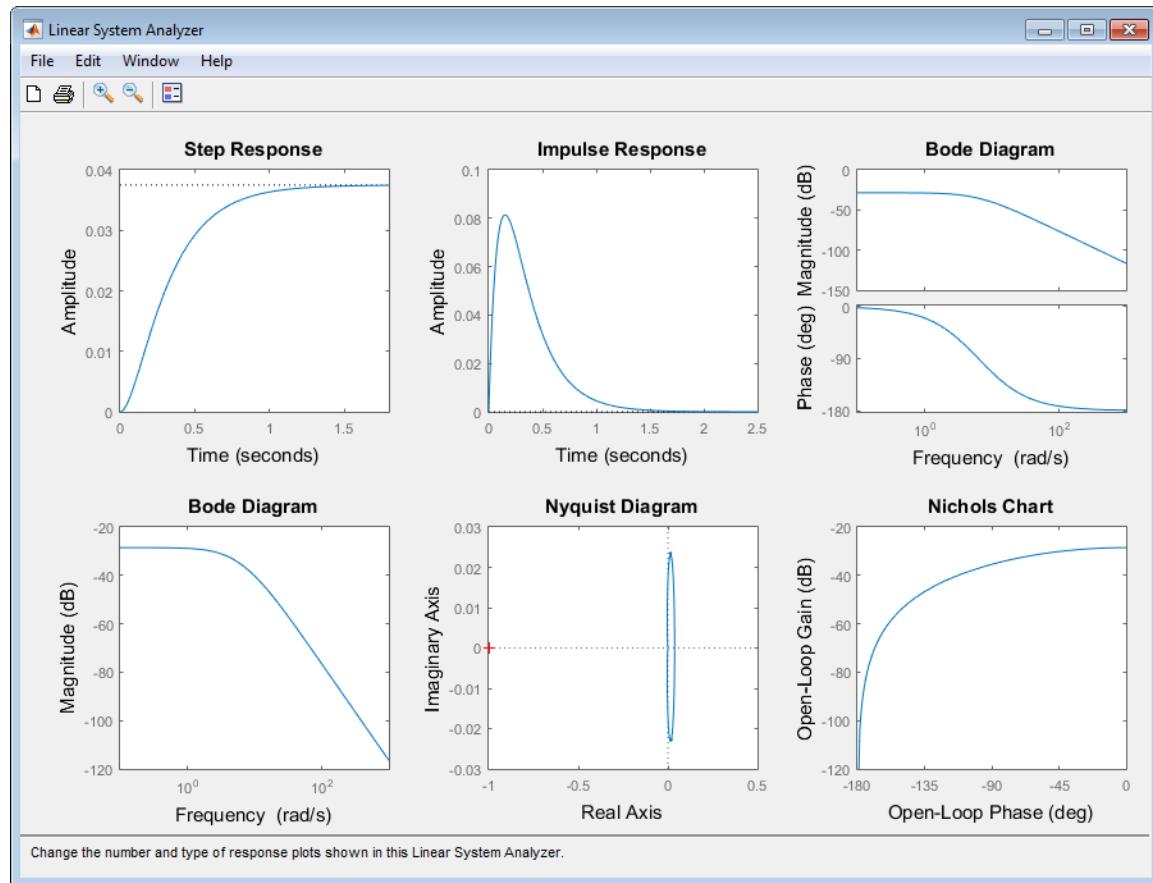
Plot Configurations Window

The Plot Type feature of the right-click menu works on existing plots, but you can also add plots to a Linear System Analyzer by using the **Plot Configurations** window. By default, the Linear System Analyzer opens with a closed-loop step response. To reconfigure an open viewer, select **Plot Configuration** in the **Edit** menu.

Use the radio buttons to select the number of plots you want displayed in your Linear System Analyzer. For each plot, select a response type from the menus located on the right-hand side of the window.
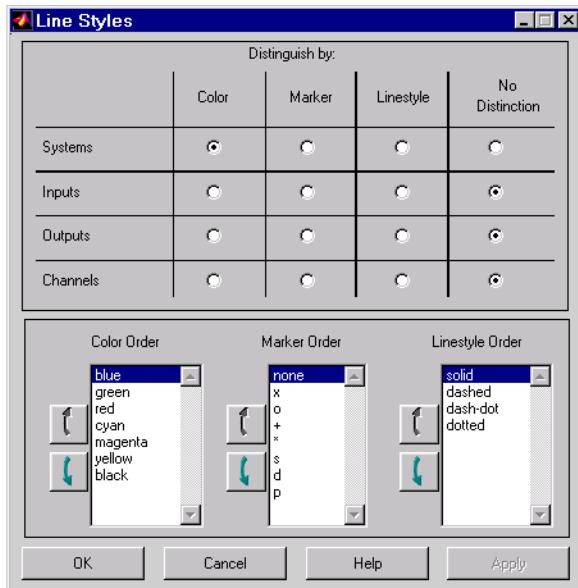
It's possible to configure a single Linear System Analyzer to contain up to six response plots.

Available response plots include: step, impulse, Bode (magnitude and phase, or magnitude only), Nyquist, Nichols, sigma, pole/zero maps, and I/O pole/zero maps.

Line Styles Editor

Select **Edit** > **Line Styles** to open the **Line Styles** editor.

The **Line Styles** editor is particularly useful when you have multiple systems imported. You can use it change line colors, add and rearrange markers, and alter line styes (solid, dashed, and so on).

You can use the **Linestyle Preferences** window to customize the appearance of the response plots by specifying:

- The line property used to distinguish different systems, inputs, or outputs
- The order in which these line properties are applied

Each Linear System Analyzer has its own **Linestyle Preferences** window.

Setting Preferences

You can use the "Distinguish by" matrix (the top half of the window) to specify the line property that will vary throughout the response plots. You can group multiple plot curves by systems, inputs, outputs, or channels (individual input/output relationships). Note that the Line Styles editor uses radio buttons, which means that you can only assign one property setting for each grouping (system, input, etc.).

Ordering Properties

The **Order** field allows you to change the default property order used when applying the different line properties. You can reorder the colors, markers, and linestyles (e.g., solid or dashed).

To change any of the property orders, click the up or down arrow button to the left of the associated property list to move the selected property up or down in the list.

See Also
linearSystemAnalyzer

Related Examples
• "Joint Time- and Frequency-Domain Analysis" on page 7-20

More About
• "Linear System Analyzer Overview" on page 22-2

Analyzing MIMO Models
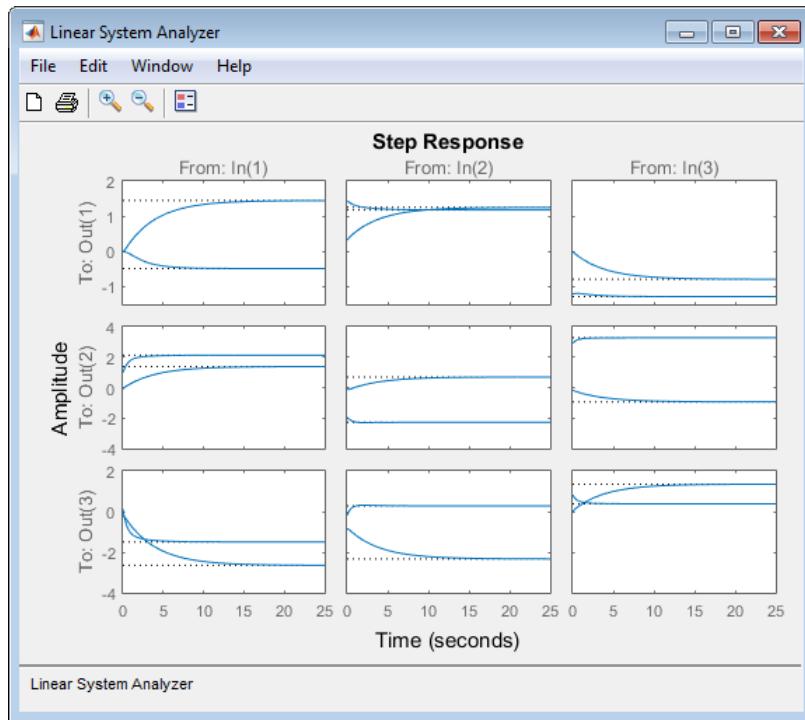
| **In this section...** |
|---|
| "Overview of Analyzing MIMO Models" on page 22-17 |
| "Array Selector" on page 22-18 |
| "I/O Grouping for MIMO Models" on page 22-20 |
| "Selecting I/O Pairs" on page 22-21 |

Overview of Analyzing MIMO Models

If you import a MIMO system, or an LTI array containing multiple linear models, you can use special features of the right-click menu to group the response plots by input/output (I/O) pairs, or select individual plots for display. For example, generate an array of two random 3-input, 3-output MIMO systems and view them in the Linear System Analyzer:
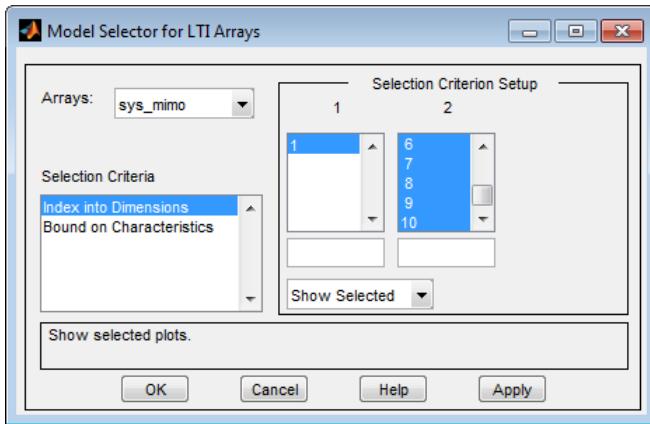
```
sys_mimo=stack(1,rss(3,3,3),rss(3,3,3));
linearSystemAnalyzer(sys_mimo);
```

A set of 9 plots appears, one from each input to each output, each showing the step responses of the corresponding I/Os of both models in the array.

Array Selector

If you import an LTI model array into the Linear System Analyzer, **Array Selector** appears as an option in the right-click menu. Selecting this option opens the **Model Selector for LTI Arrays**, shown below.

You can use this window to include or exclude models within the LTI array using various criteria.
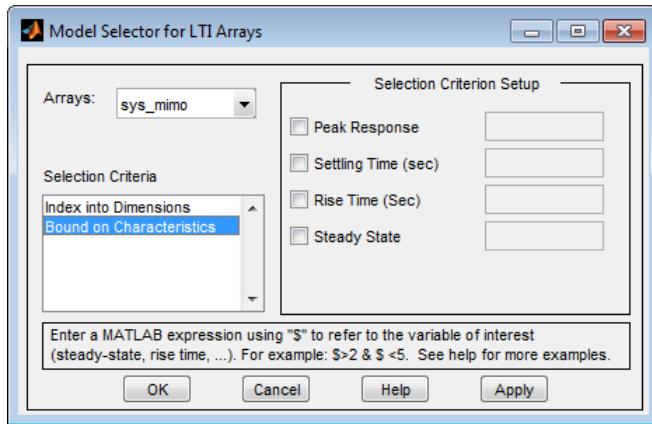
Arrays

Select the LTI array for model selection using the **Arrays** list.

Selection Criteria

There are two selection criteria. The default, **Index into Dimensions**, allows you to include or exclude specified indices of the LTI Array. Select systems from the **Selection Criterion Setup** section of the dialog box. Then, Specify whether to show or hide the systems using the pull-down menu below the Setup lists.

The second criterion is **Bound on Characteristics**. Selecting this options causes the Model Selector to reconfigure. The reconfigured window is shown below
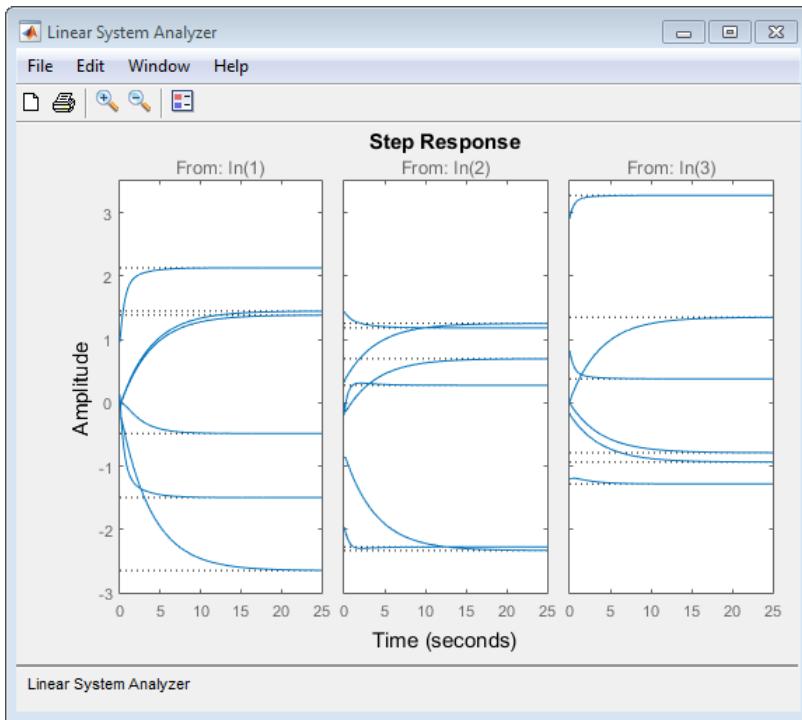
Use this option to select systems for inclusion or exclusion in your Linear System Analyzer based on their time response characteristics. The panel directly above the buttons describes how to set the inclusion or exclusion criteria based on which selection criteria you select from the reconfigured **Selection Criteria Setup** panel.

I/O Grouping for MIMO Models

You can group the plots by inputs, by outputs, or both by selecting **I/O Grouping** from the right-click menu, and then selecting **Inputs**, **Outputs**, or **All**.
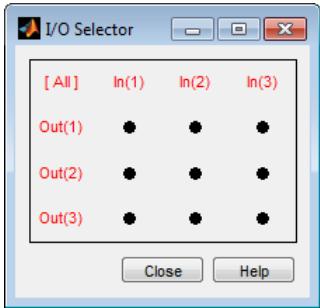
For example, if you select **Outputs**, the step plot reconfigures into 3 plots, grouping all the outputs together on each plot. Each plot now displays the responses from one of the inputs to all of the MIMO system's outputs, for all of the models in the array.

Selecting **None** returns to the default configuration, where all I/O pairs are displayed individually.

Selecting I/O Pairs

Another way to organize MIMO system information is to choose **I/O Selector** from the right-click menu, which opens the **I/O Selector** window.

This window automatically configures to the number of I/O pairs in your MIMO system. You can select:

- Any individual plot (only one at a time) by clicking on a button
- Any row or column by clicking on Y(*) or U(*)
- All of the plots by clicking [all]

Using these options, you can inspect individual I/O pairs, or look at particular I/O channels in detail.

See Also
linearSystemAnalyzer

More About
- "Model Arrays" on page 2-96

Customizing the Linear System Analyzer
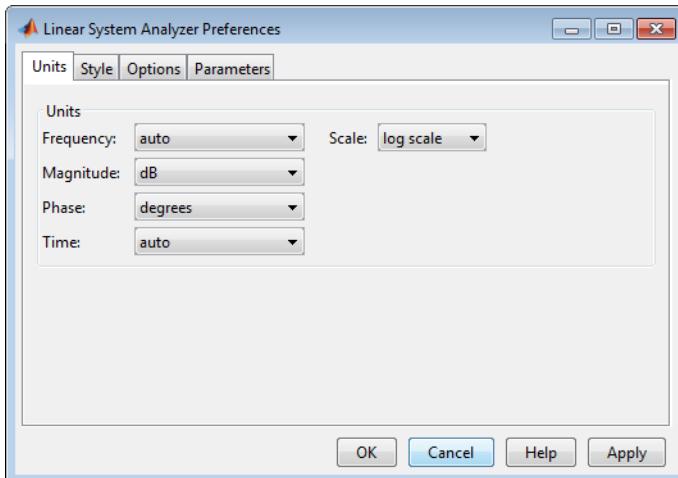
| In this section... |
|---|
| "Overview of Customizing the Linear System Analyzer" on page 22-23 |
| "Linear System Analyzer Preferences Editor" on page 22-23 |

Overview of Customizing the Linear System Analyzer

The Linear System Analyzer has a tool preferences editor, which allows you to set default characteristics for specific instances of Linear System Analyzers. If you open a new instance of either, each defaults to the characteristics specified in the **Toolbox Preferences** editor.

Linear System Analyzer Preferences Editor

Select **Edit** > **Linear System Analyzer Preferences** to open the preferences editor.

The **Linear System Analyzer Preferences** editor contains four panes:

- Units--Convert between various units, including rad/sec and Hertz
- Style--Customize grids, fonts, and colors

- Characteristics--Specify response plot characteristics, such as settling time tolerance
- Parameters--Set time and frequency ranges, stop times, and time step size

For more information about using the options in these panes in an instance of the Linear System Analyzer, see "Linear System Analyzer Preferences Editor" on page 18-2.

If you want to customize the settings for all instances of Linear System Analyzers, see the ??? editor.

See Also
linearSystemAnalyzer

More About
- "Linear System Analyzer Overview" on page 22-2