Control System Toolbox™

User's Guide

MATLAB®

R2015b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

Revision History

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)

Contents

Linear System Modeling

1 Linear System Model Objects

5 Model Transformation

Linear System Modeling

Linear System Model Objects

What Are Model Objects?

In this section...

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 using Robust Control Toolbox tuning commands.

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.

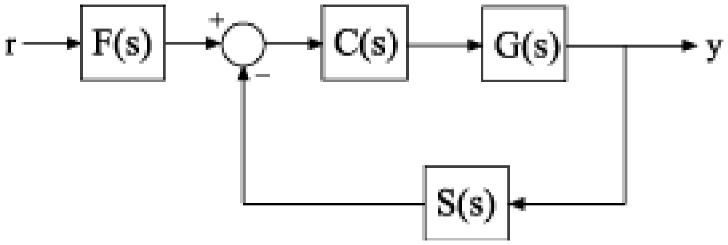
More About

- “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:

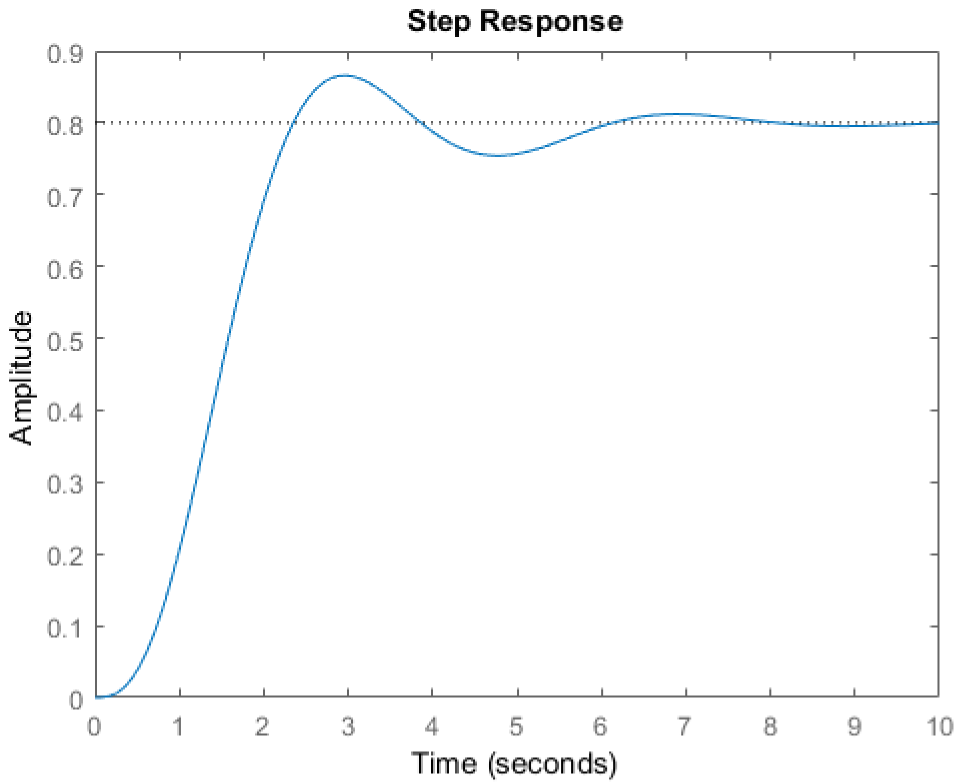
```
T = feedback(G*C,S);
```

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)
```



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

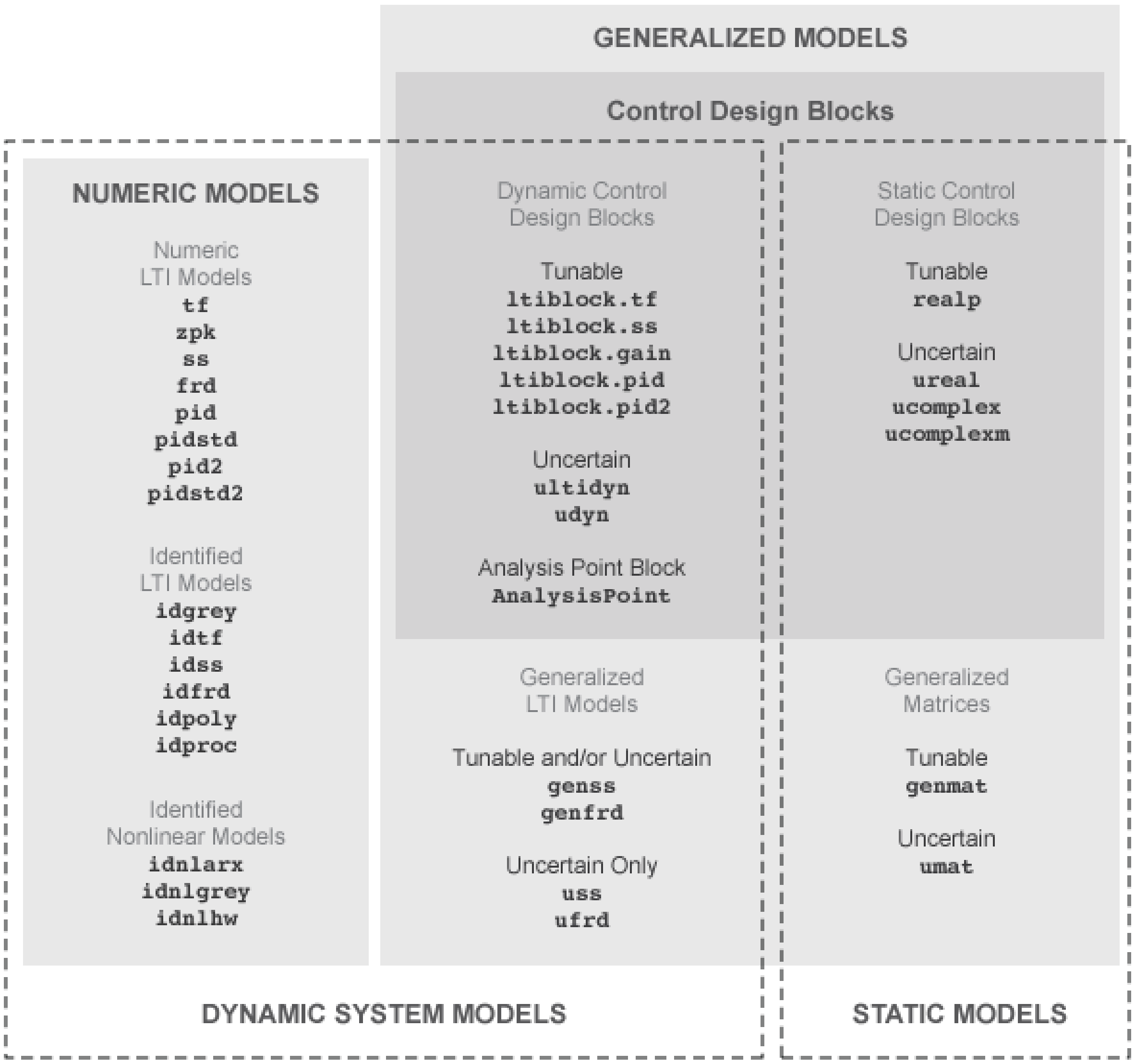
- "Numeric Model of SISO Feedback Loop" on page 4-6
- "Multi-Loop Control System" on page 4-10
- "MIMO Control System" on page 4-19

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.



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

- "What Are Model Objects?" on page 1-2
- "Dynamic System Models" on page 1-10
- "Static Models" on page 1-12
- "Numeric Models" on page 1-13
- "Generalized Models" on page 1-16

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	`ltiblock.gain`
	`ltiblock.tf`
	`ltiblock.ss`
	`ltiblock.pid`
	`ltiblock.pid2`
	`ultidyn`
	`udyn`
	`AnalysisPoint`

More About

- “Numeric Linear Time Invariant (LTI) Models” on page 1-13
- “Identified LTI Models” on page 1-14
- “Identified Nonlinear Models” on page 1-14
- “Generalized and Uncertain LTI Models” on page 1-16
- “Control Design Blocks” on page 1-16

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:

- "Transfer Functions" on page 2-3
- "State-Space Models" on page 2-7
- "Frequency Response Data (FRD) Models" on page 2-11
- "Proportional-Integral-Derivative (PID) Controllers" on page 2-14

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 "SISO Design Tool" 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	`ltiblock.gain`	Tunable gain block
	`ltiblock.tf`	SISO fixed-order transfer function with tunable coefficients
	`ltiblock.ss`	Fixed-order state-space model with tunable coefficients
	`ltiblock.pid`	One-degree-of-freedom PID controller with tunable coefficients
	`ltiblock.pid2`	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

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 2-108.

If you have Robust Control Toolbox software, you can use tunable Generalized LTI models for tuning fixed control structures using tuning commands such as `systune` or the Control System Tuner app. See “Control System Tuning” in the Robust Control Toolbox documentation.

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.

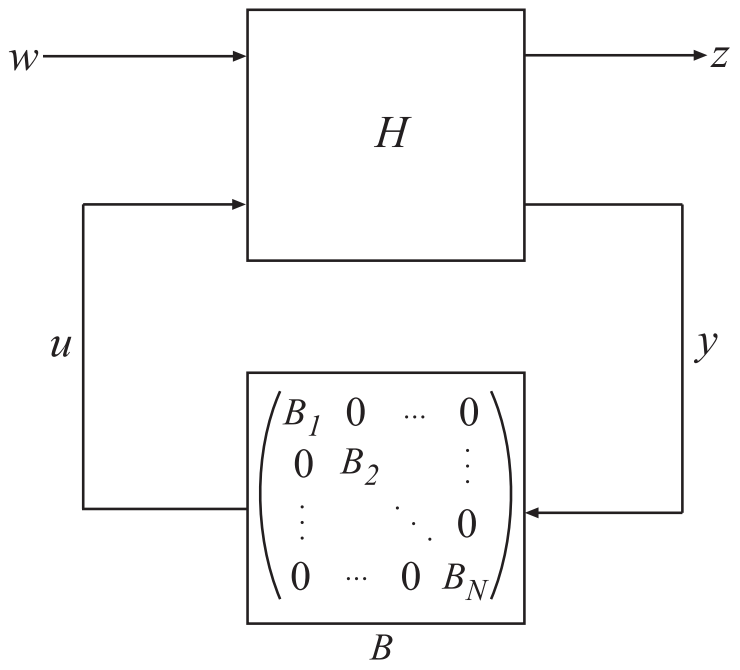
The resulting model is:

- 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.

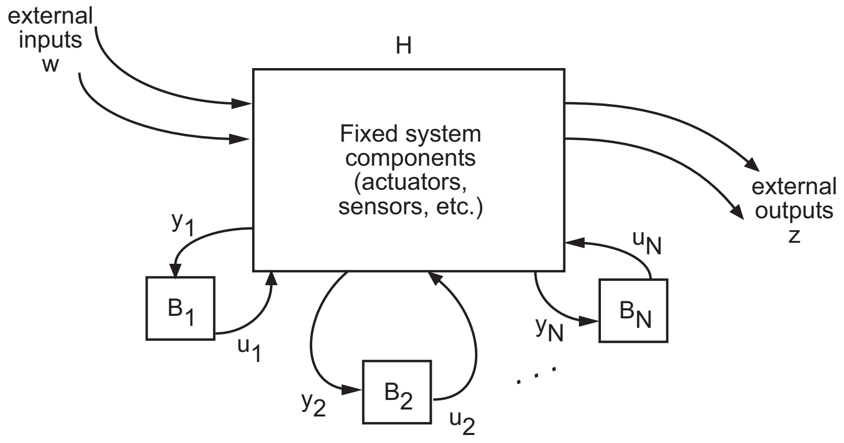
H represents all portions of the Generalized model that have fixed (non-parametric) coefficients. *H* is:

- 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.

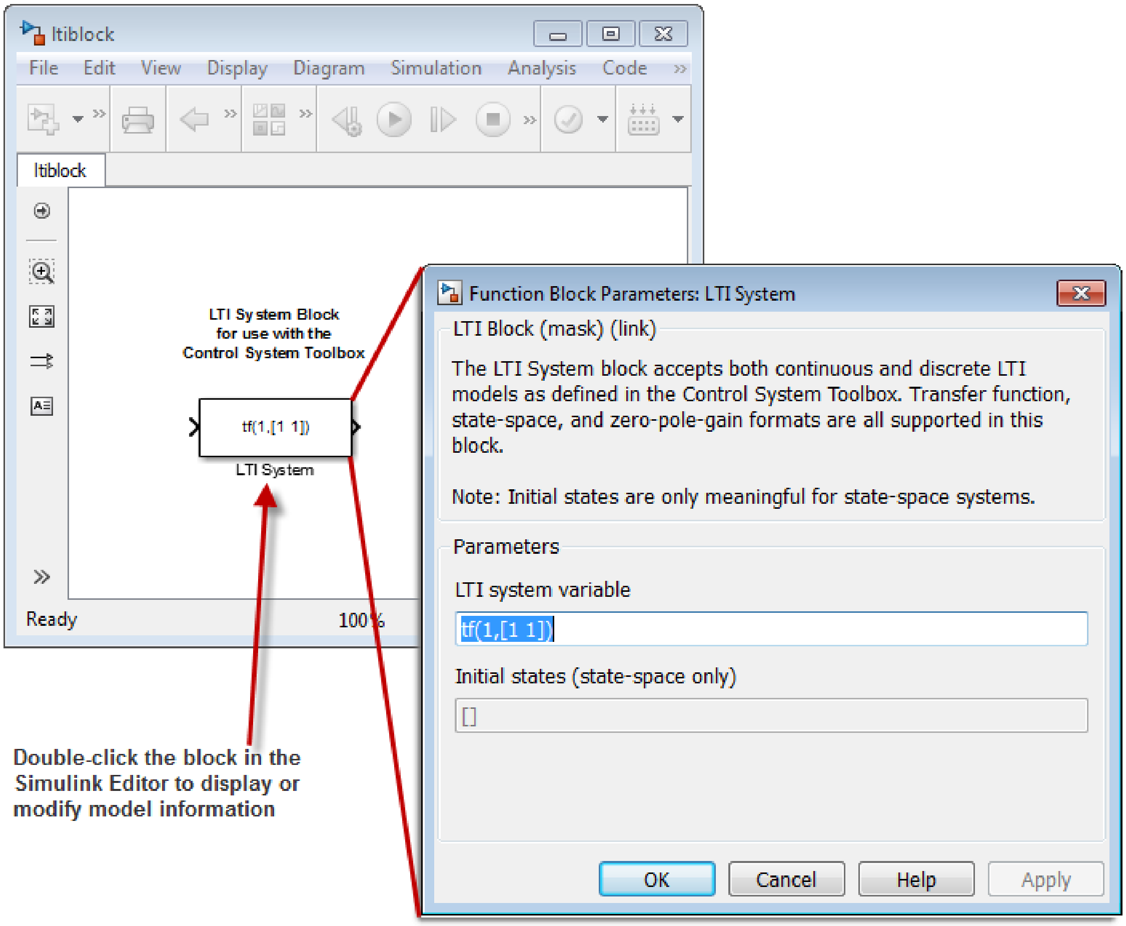
Using Model Objects

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", "SISO Feedback Loops", or "Linear-Quadratic-Gaussian Control".
  - Manually tune many control architectures using the SISO Design Tool. See "SISO Feedback Loops".
  - 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 (requires Robust Control Toolbox software). See "Control System Tuning" in the Robust Control Toolbox documentation.

Simulink Block for LTI Systems

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


ltiblock
File Edit View Display Diagram Simulation Analysis Code
ltiblock
LTI System Block
for use with the
Control System Toolbox
tf(1,[1 1])
LTI System
Ready
100%
Double-click the block in the Simulink Editor to display or modify model information
Function Block Parameters: LTI System
OK
Cancel
Help
Apply

LTI Block (mask) (link)

The LTI System block accepts both continuous and discrete LTI models as defined in the Control System Toolbox. Transfer function, state-space, and zero-pole-gain formats are all supported in this block.

Note: Initial states are only meaningful for state-space systems.

Parameters

LTI system variable: tf(1,[1 1])

Initial states (state-space only): []

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.

Model Creation

Transfer Functions

In this section...

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 $G(s)=5\frac{s}{(s+1+i)(s+1-i)(s+2)}$:

```
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

- "MIMO Transfer Functions" on page 2-29
- "State-Space Models" on page 2-7

- “Discrete-Time Numeric Models” on page 2-24

More About

- “What Are Model Objects?” on page 1-2
- “Model Properties” on page 3-2

State-Space Models

In this section...

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:

$$\begin{aligned}\frac{dx}{dt} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

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:

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] \\ y[n] &= Cx[n] + Du[n] \end{aligned}$$

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:

$$\begin{aligned} E\frac{dx}{dt} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

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:

$$\begin{aligned}\frac{dx}{dt} &= Ax + Bu\\ y &= Cx + Du\end{aligned}$$

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:

$$\begin{aligned}E\frac{dx}{dt} &= Ax + Bu\\ y &= Cx + Du\end{aligned}$$

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

- “MIMO State-Space Models” on page 2-32
- “Transfer Functions” on page 2-3
- “Discrete-Time Numeric Models” on page 2-24

More About

- “What Are Model Objects?” on page 1-2
- “Model Properties” on page 3-2

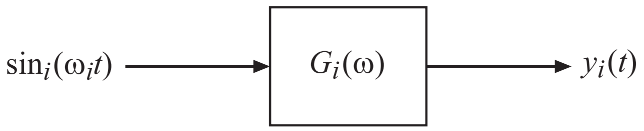
Frequency Response Data (FRD) Models

In this section...

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:



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
- "Model Properties" 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: $C = 29.5 + \frac{26.2}{s} - \frac{4.3s}{0.06s+1}$.

```
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: $C = 29.5\left(1 + \frac{1}{1.13s} + \frac{0.15s}{\frac{0.15}{2.3}s+1}\right)$.

```
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
- "Model Properties" on page 3-2

Two-Degree-of-Freedom PID Controllers

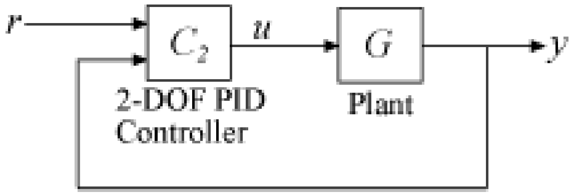
Two-degree-of-freedom (2-DOF) PID controllers include setpoint weighting on the proportional and derivative terms. A 2-DOF PID controller is capable of fast disturbance rejection without significant increase of overshoot in setpoint tracking. 2-DOF PID controllers are also useful to mitigate the influence 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)	$u = K_p(br - y) + \frac{K_i}{s}(r - y) + \frac{K_d s}{T_f s + 1}(cr - y).$

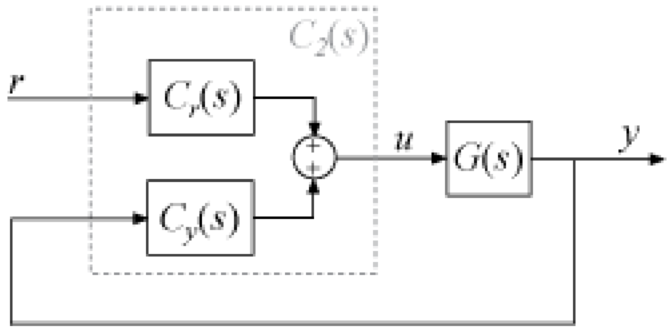
Form	Formula
	In this representation: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time • b = setpoint weight on proportional term • c = setpoint weight on derivative term
Standard (`pidstd2` object)	$u = K_p \left[(br - y) + \frac{1}{T_i s}(r -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

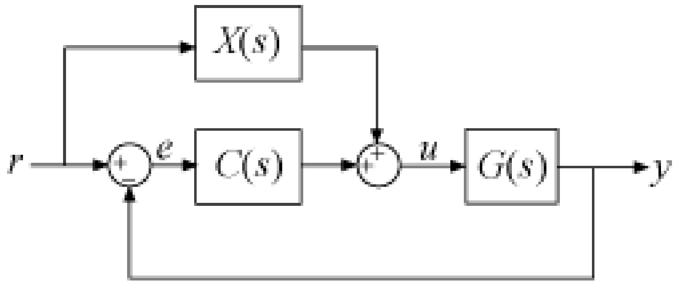
Alternatively, use the connect command to build an equivalent closed-loop system directly with the 2-DOF controller C2. To do so, set the InputName and OutputName properties of G and C2.

```
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`.

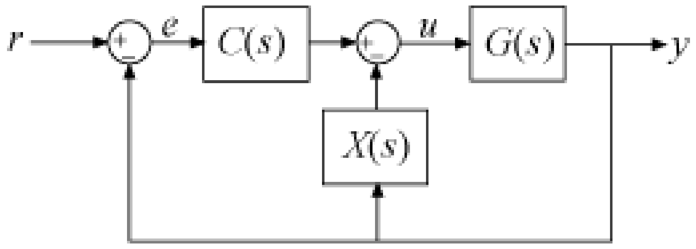
```
[C,X] = getComponents(C2,'feedforward');
```

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`.

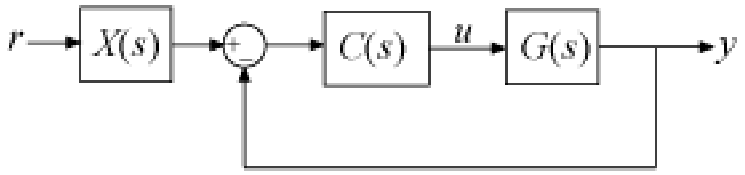
```
[C,X] = getComponents(C2,'feedback');
```

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:

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

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

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 5-6.

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
- “Model Properties” on page 3-2

Discrete-Time Numeric Models

In this section...

Create Discrete-Time Transfer Function Model

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

Create the transfer function $G(z) = \frac{z}{z^2 - 2z - 6}$ 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`)	$$C = K_p \left(1 + \frac{1}{T_i} IF(z) + \frac{T_d}{\frac{T_d}{N} + DF(z)} \right),$$ 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)	$\frac{T_s}{z-1}$
`BackwardEuler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

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 $IF(z) = \frac{T_s}{2}\frac{z+1}{z-1}$ and $DF(z) = \frac{T_s z}{z-1}$.

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
- “Model Properties” 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

$$N(s) = \begin{bmatrix} s-1 \\ s+2 \end{bmatrix}, \quad D(s) = \begin{bmatrix} s+1 \\ s^2+4s+5 \end{bmatrix}.$$

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 *ij*th entry of the transfer matrix $H(s)$.

See Also

`tf` | `zpk`

Related Examples

- “Transfer Functions” on page 2-3

More About

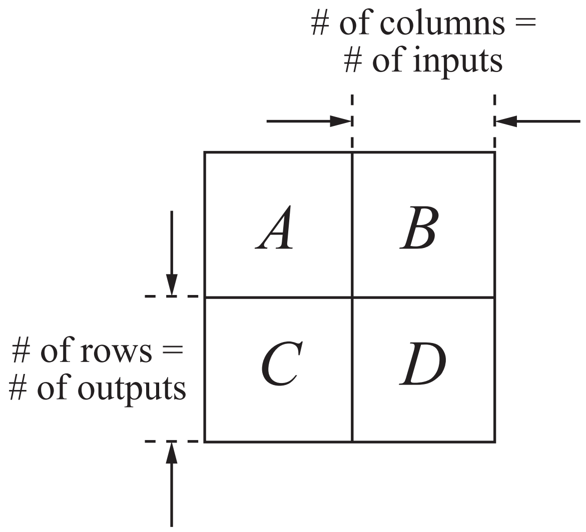
- “What Are Model Objects?” on page 1-2

- “Model Properties” 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:

$$\begin{aligned} E\frac{dx}{dt} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

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

$$\begin{aligned} J\frac{d\omega}{dt} + F\omega &= T \\ y &= \omega. \end{aligned}$$

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
- "Model Properties" 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
- “Model Properties” 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

- "Model Properties" 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

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:

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

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.

Specify time delays for discrete-time models in the same way as for continuous-time models, except for discrete-time models, delay values must be integer multiples of the sample time. For example, create the following first-order transfer function, with a sample time of $T_s = 0.1$ s:

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

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

Setting `InputDelay` to 25 results in a delay of 25 sampling periods.

Tip Use `thiran` to approximate discrete-time models with delays that are a fractional multiple of the sample time.

Related Examples

- “Closing Feedback Loops with Time Delays” on page 2-46
- “Convert Time Delay in Discrete-Time Model to Factors of 1/z” on page 2-65

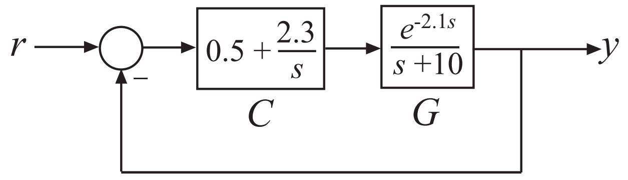
More About

- “Time-Delay Approximation” on page 2-49

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.

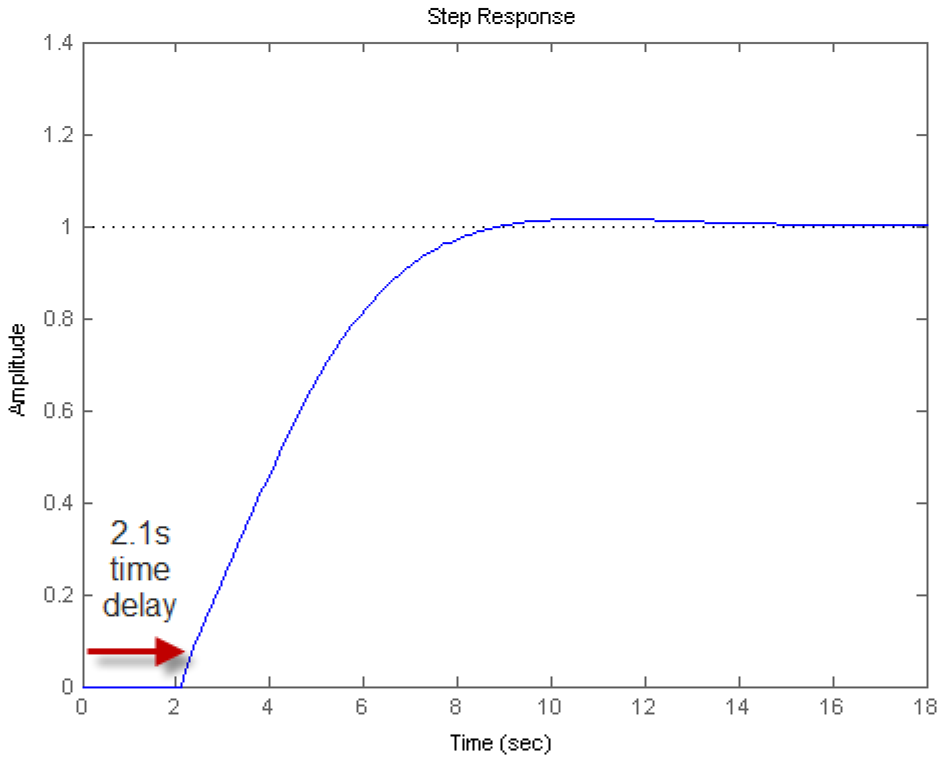
Note: In addition to `feedback`, any system interconnection function (including `parallel` and `series`) can give rise to internal delays.

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.

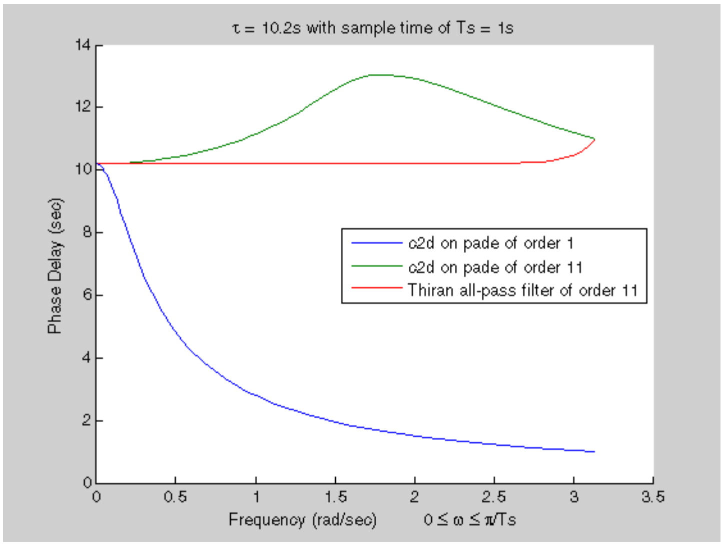
Use the `thiran` command to approximate a time delay that is a fractional multiple of the sample time as a Thiran all-pass filter.

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

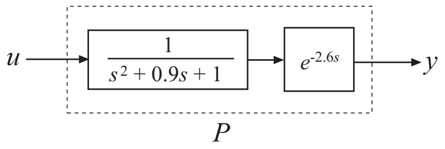
- "Time-Delay Approximation in Continuous-Time Open-Loop Model" on page 2-51
- "Convert Time Delay in Discrete-Time Model to Factors of 1/z" on page 2-65
- "Approximate Different Delays with Different Approximation Orders" on page 2-61

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)
```

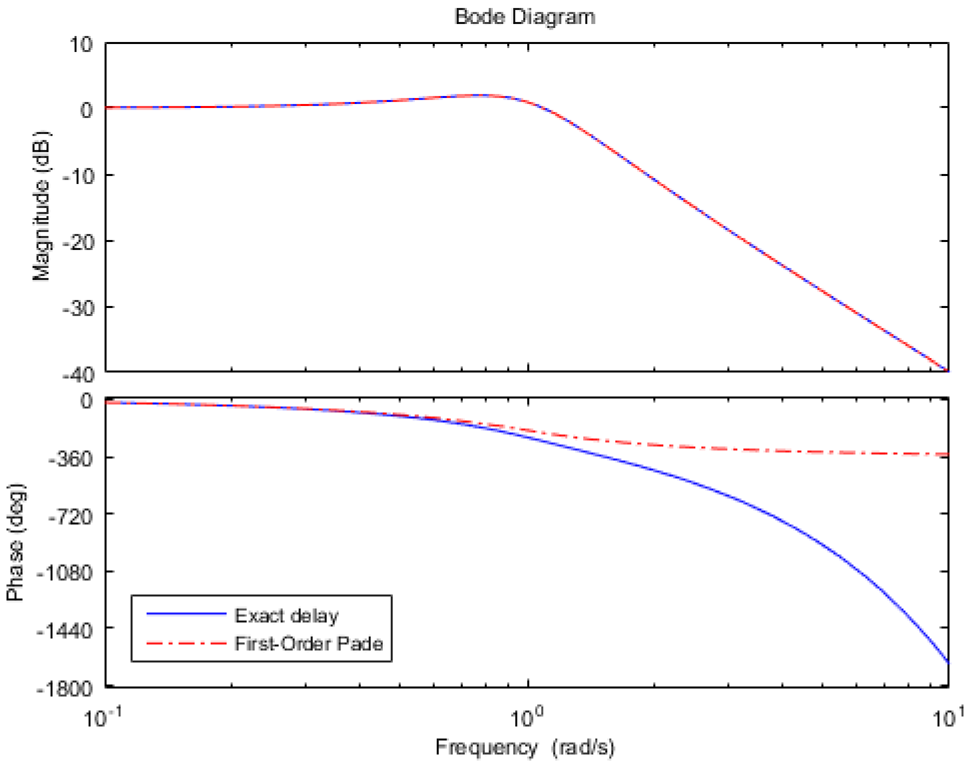
```
Pnd1 =

              -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')
```



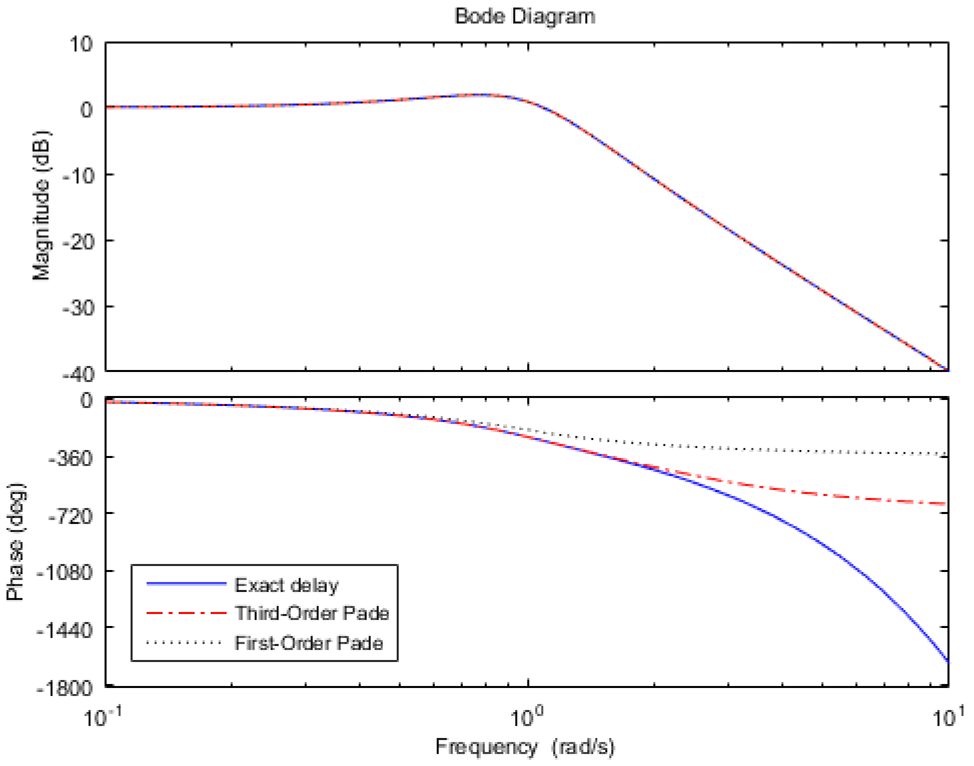
The magnitude of `P` and `Pnd1` match exactly. However, the phase of `Pnd1` deviates from the phase of `P` beyond approximately 1 rad/s.

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`.

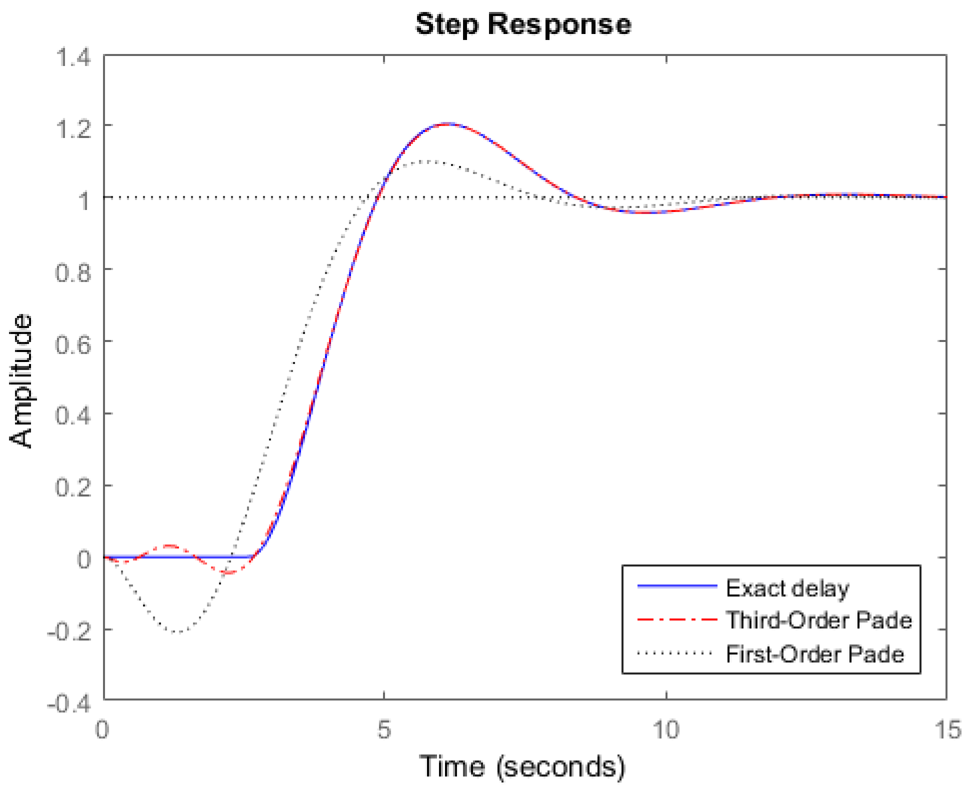
```
bodeplot(P,'-b',Pnd3,'-.r',Pnd1,':k',{0.1 10},h)
legend('Exact delay','Third-Order Pade','First-Order Pade',...
       'Location','SouthWest')
```



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

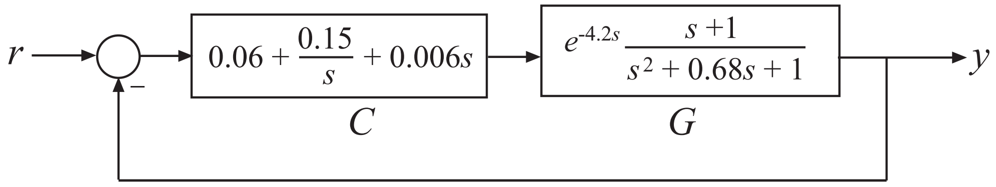
- “Time-Delay Approximation” on page 2-49
- “Internal Delays” on page 2-72

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.


r
−
$0.06 + \frac{0.15}{s} + 0.006s$
C
$e^{-4.2s} \frac{s+1}{s^2 + 0.68s + 1}$
G
y

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`.

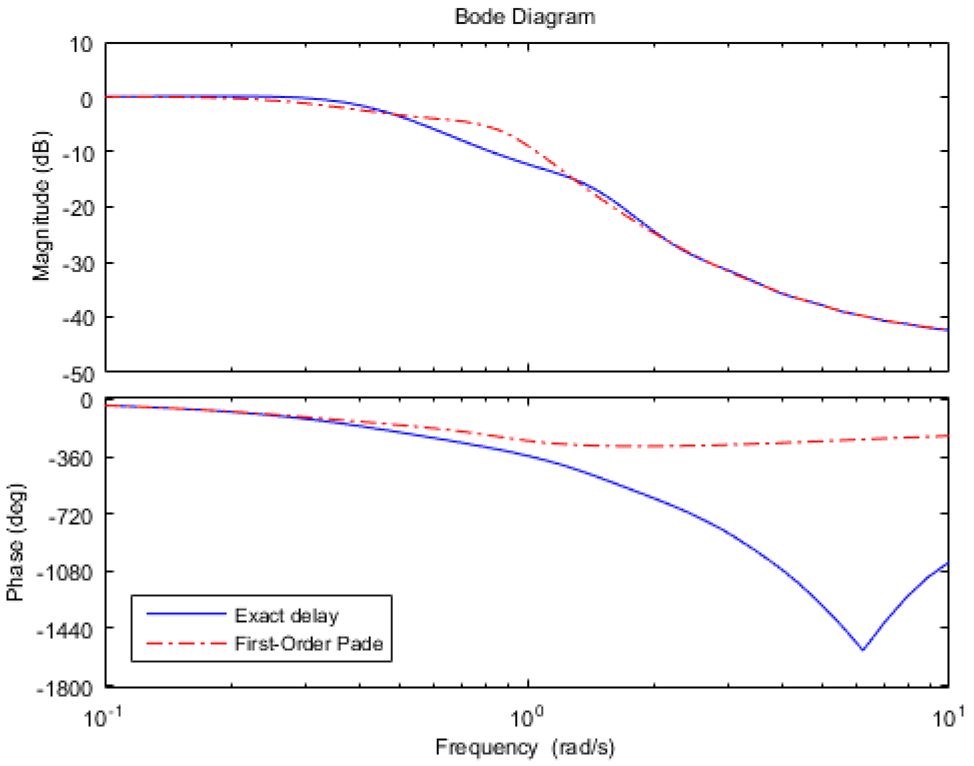
```
Tnd1 = pade(Tcl,1);
```

`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);
```

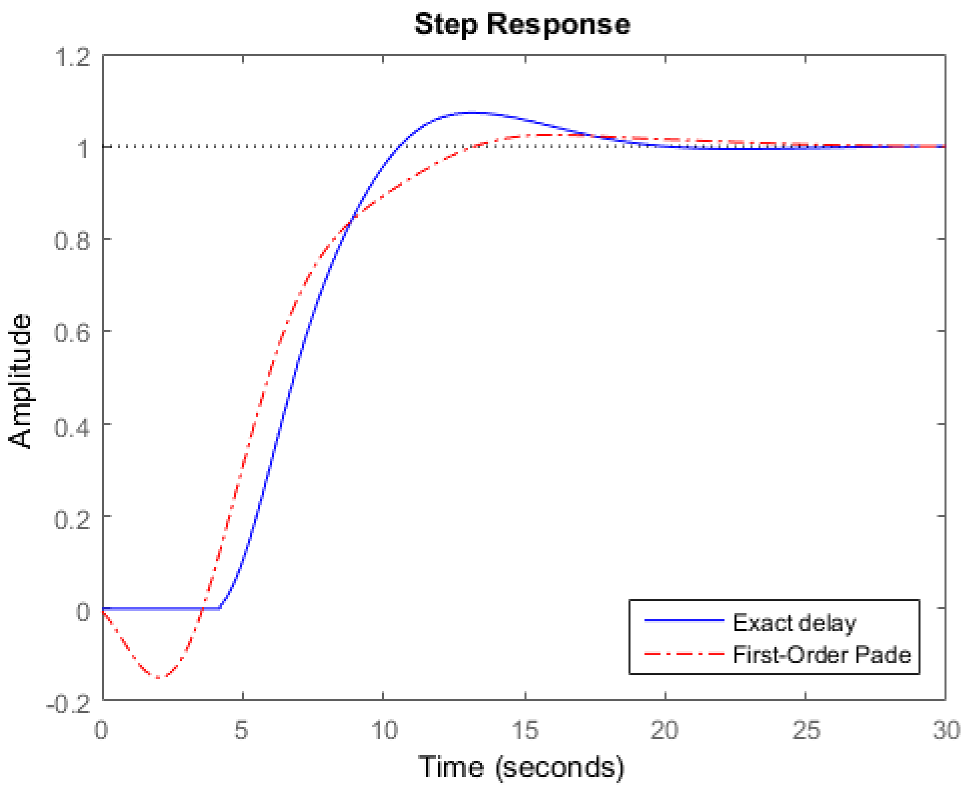
```
legend('Exact delay','First-Order Pade','Location','SouthWest');
```



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

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

```
stepplot(Tcl,'-b',Tnd1,'-.r');
legend('Exact delay','First-Order Pade','Location','SouthEast');
```



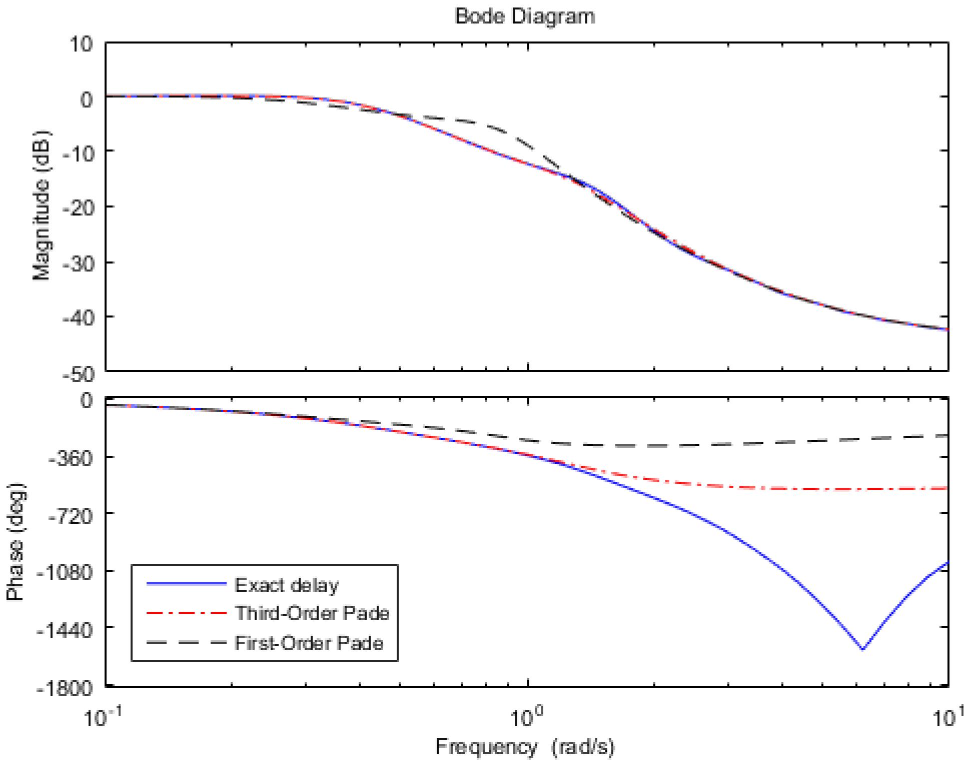
Using the Padé approximation introduces a nonminimum phase artifact (“wrong way” effect) in the initial transient response.

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

- "Approximate Different Delays with Different Approximation Orders" on page 2-61

More About

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

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.

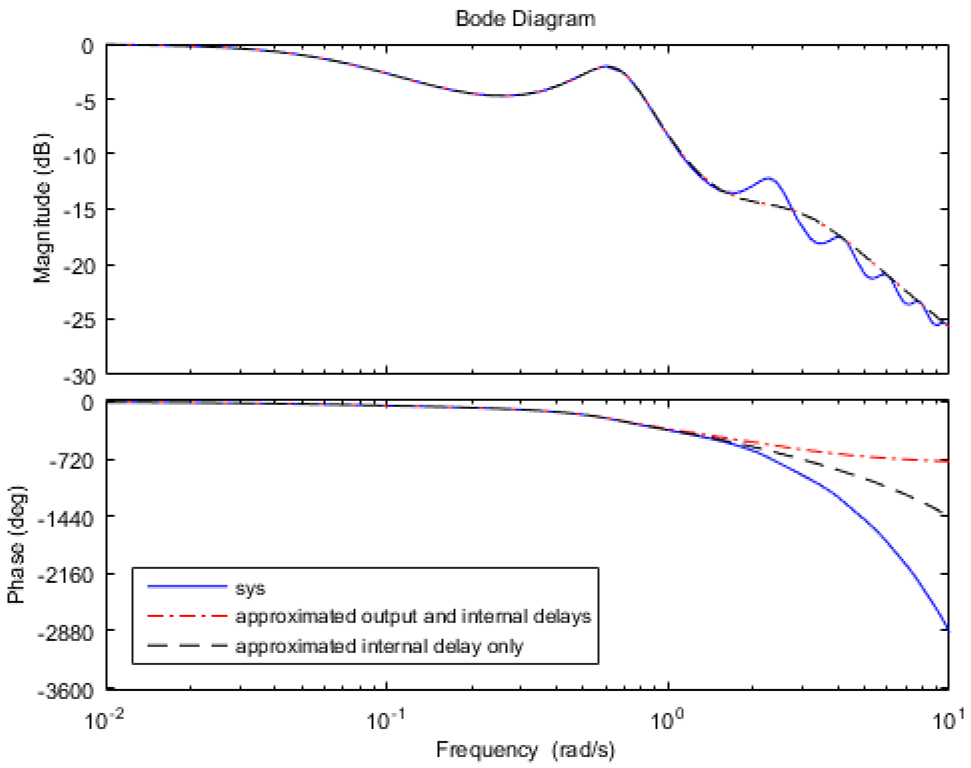
For comparison, approximate only the internal delay of sys, leaving the output delay intact.

```
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 onl
    'location','SouthWest')
```



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

See Also

`pade`

Related Examples

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

More About

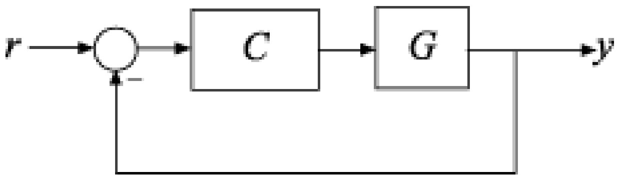
- “Time-Delay Approximation” on page 2-49
- “Internal Delays” on page 2-72

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
```

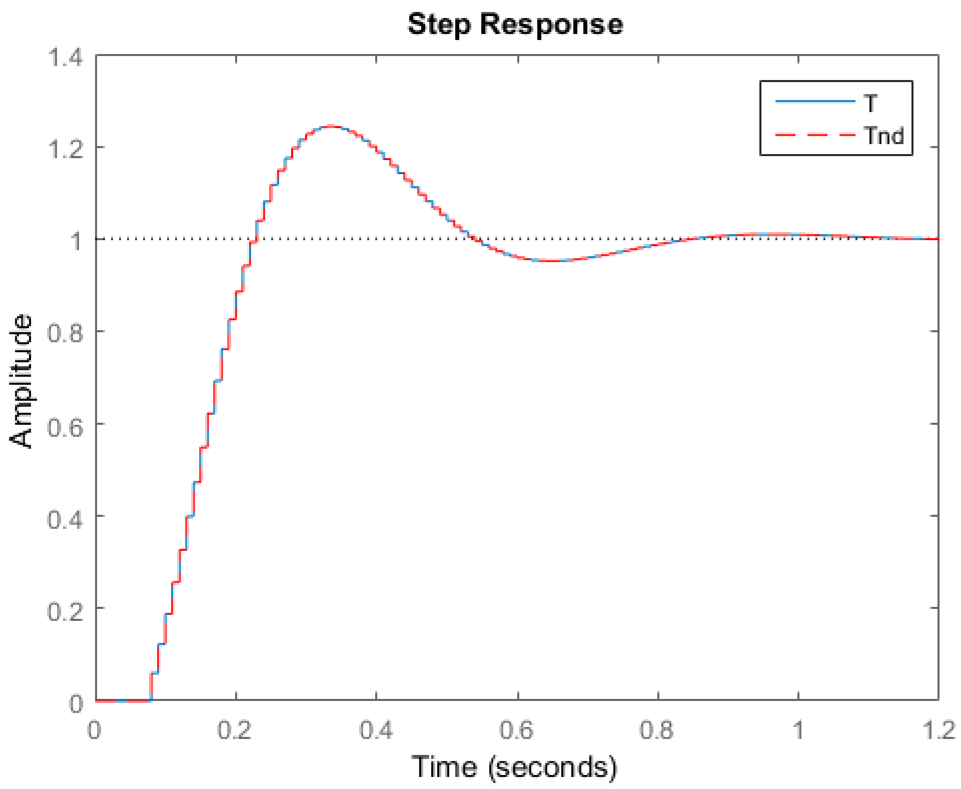
```
Tnd.InternalDelay
```

```
ans =

   Empty matrix: 0-by-1
```

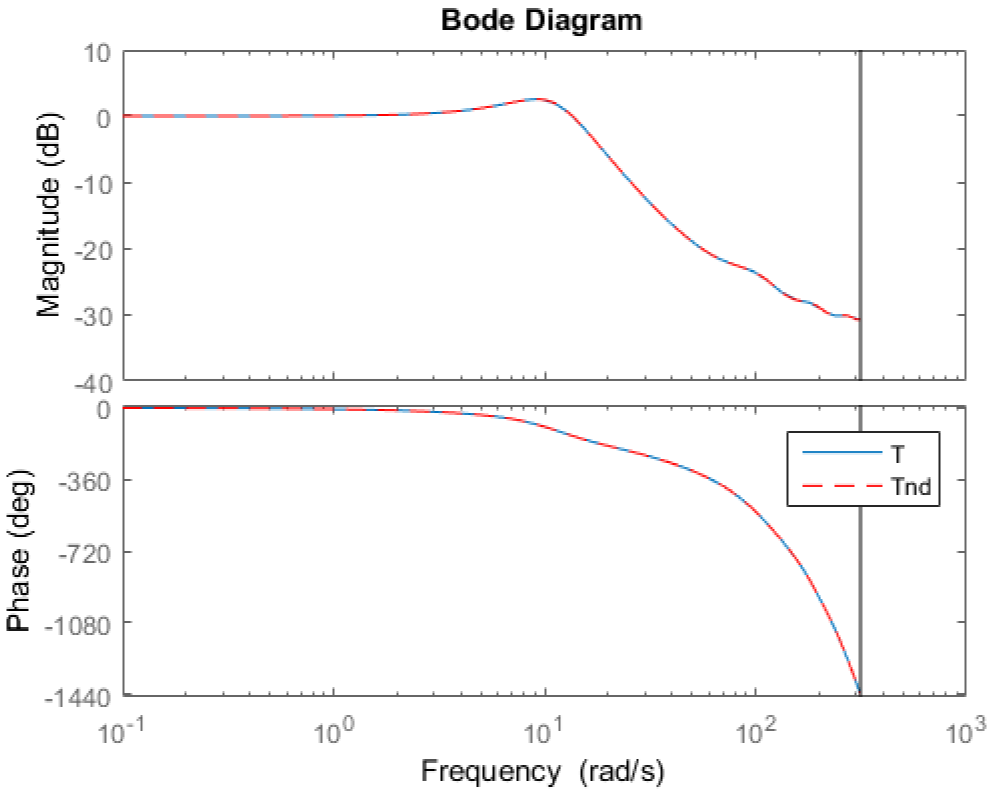
Tnd has no internal delay, but it is a ninth-order model, due to the seven extra poles introduced by absorbing the seven-unit delay into the model dynamics.

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

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



```
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.

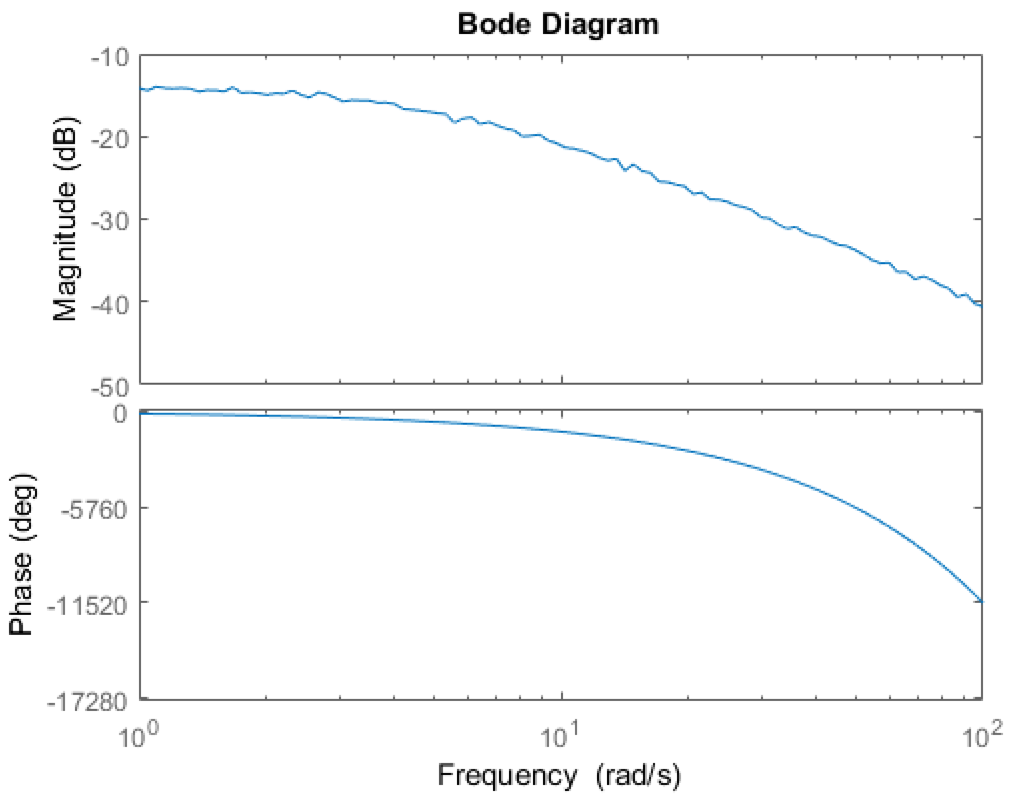
The `frd` model `fsys` includes a transport delay of 2 s. Load the model into the MATLAB® workspace and inspect the time delay.

```
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)
```



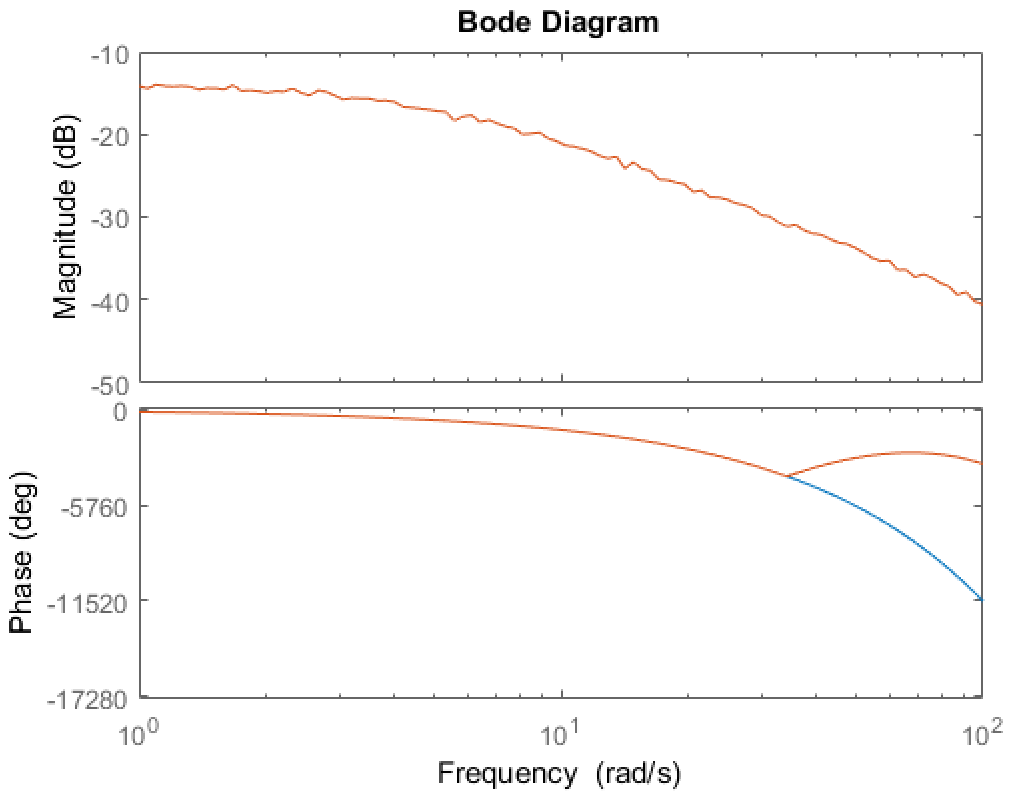
The `absorbDelay` command absorbs all time delays directly into the frequency response, resulting in an `frd` model with `ioDelay = 0`.

```
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:

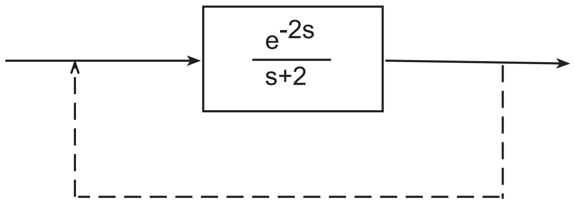
- Concatenating state-space models with input and output delays
- Feeding back a delayed signal
- Converting MIMO `tf` or `zpk` models with transport delays to state-space form

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:

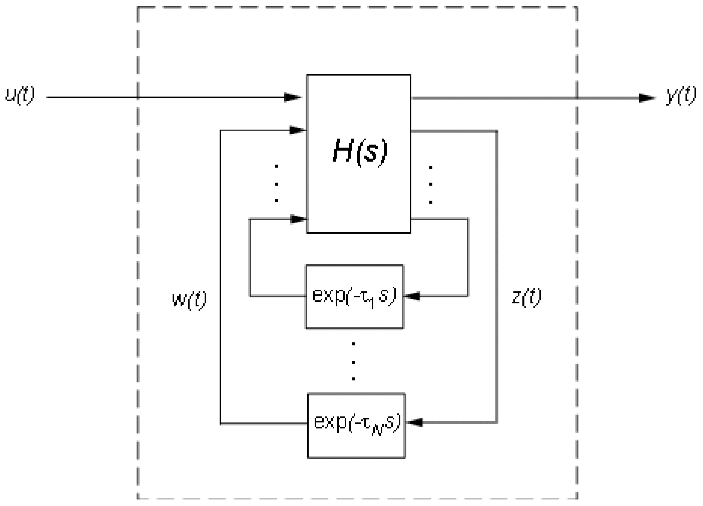
- Entering `sys` returns only sizes for the matrices of a system named `sys`.
- Entering `sys.a` produces an error.

  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:

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

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 `ltiblock.tf` to represent F, because the numerator and denominator coefficients of an `ltiblock.tf` 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

- “Tunable Second-Order Filter” on page 2-78
- “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

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:

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

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

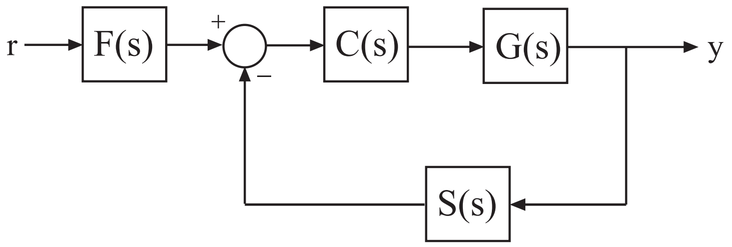
- “Tunable Low-Pass Filter” on page 2-77
- “Tunable Second-Order Filter” on page 2-78
- “Control System with Tunable Components” on page 2-82

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 = ltiblock.pid('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 `ltiblock.pid` 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 ltiblock.pid]
    a: [1x1 realp]
```

If you have Robust Control Toolbox software, 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

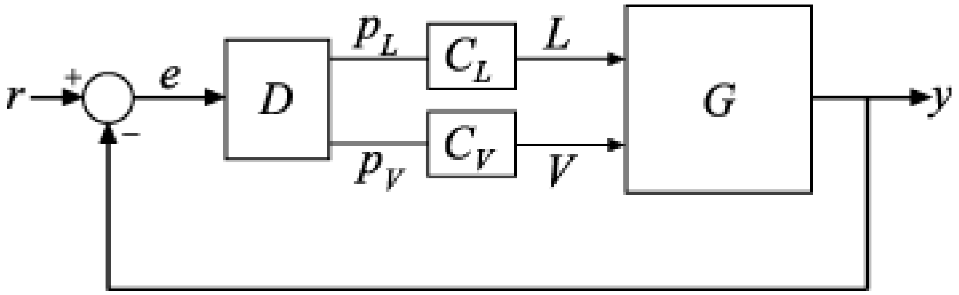
More About

- “Models with Tunable Coefficients” on page 1-19

Control System with Multichannel Analysis Points

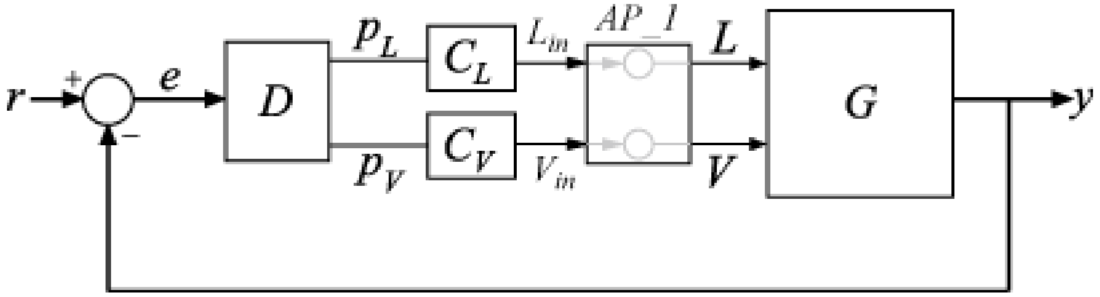
This example shows how to insert multichannel analysis points in a generalized state-space model of a MIMO control system.

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 = ltiblock.gain('Decoupler',eye(2));
C_L = ltiblock.pid('C_L','pi');
C_V = ltiblock.pid('C_V','pi');
```

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

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

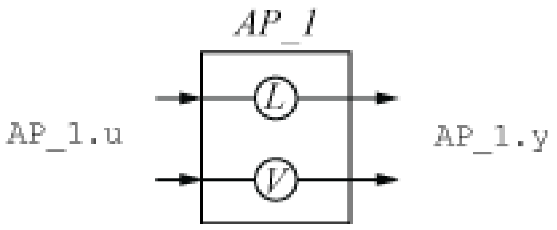
```
AP_1 =

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.

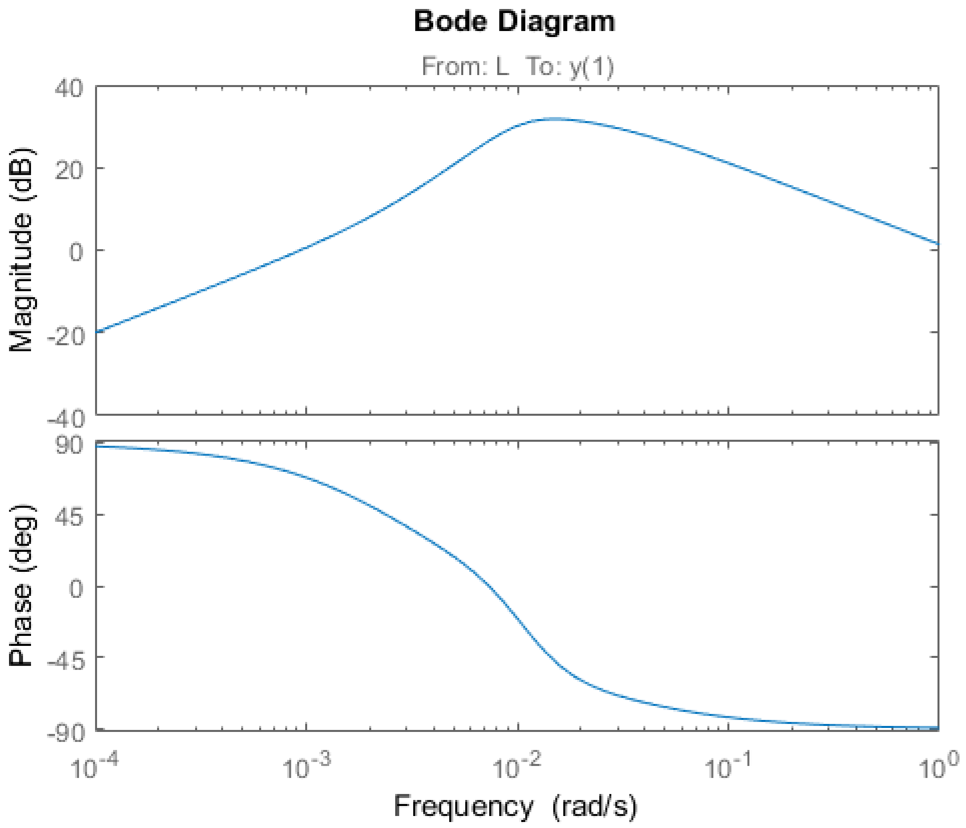
```
GC = G*AP_1*append(C_L,C_V)*D;
```

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

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`:

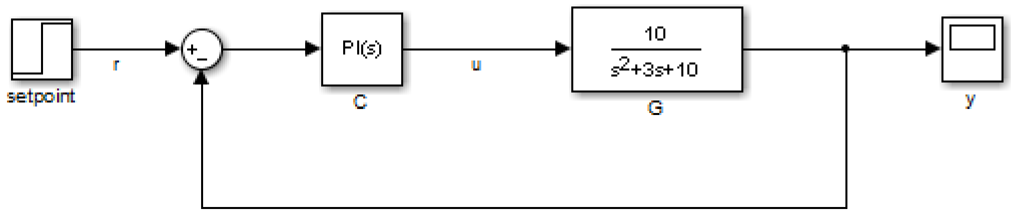


Figure 1: Simple Feedback Loop

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.

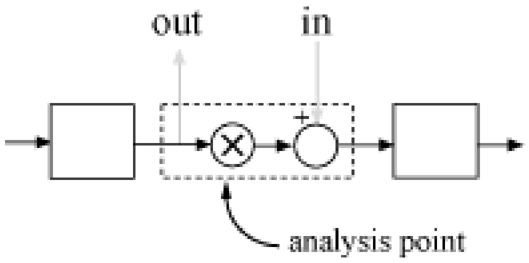


Figure 2: Analysis Point as Input, Output, and Loop Opening

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 (requires Robust Control Toolbox software). 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);
```

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`.

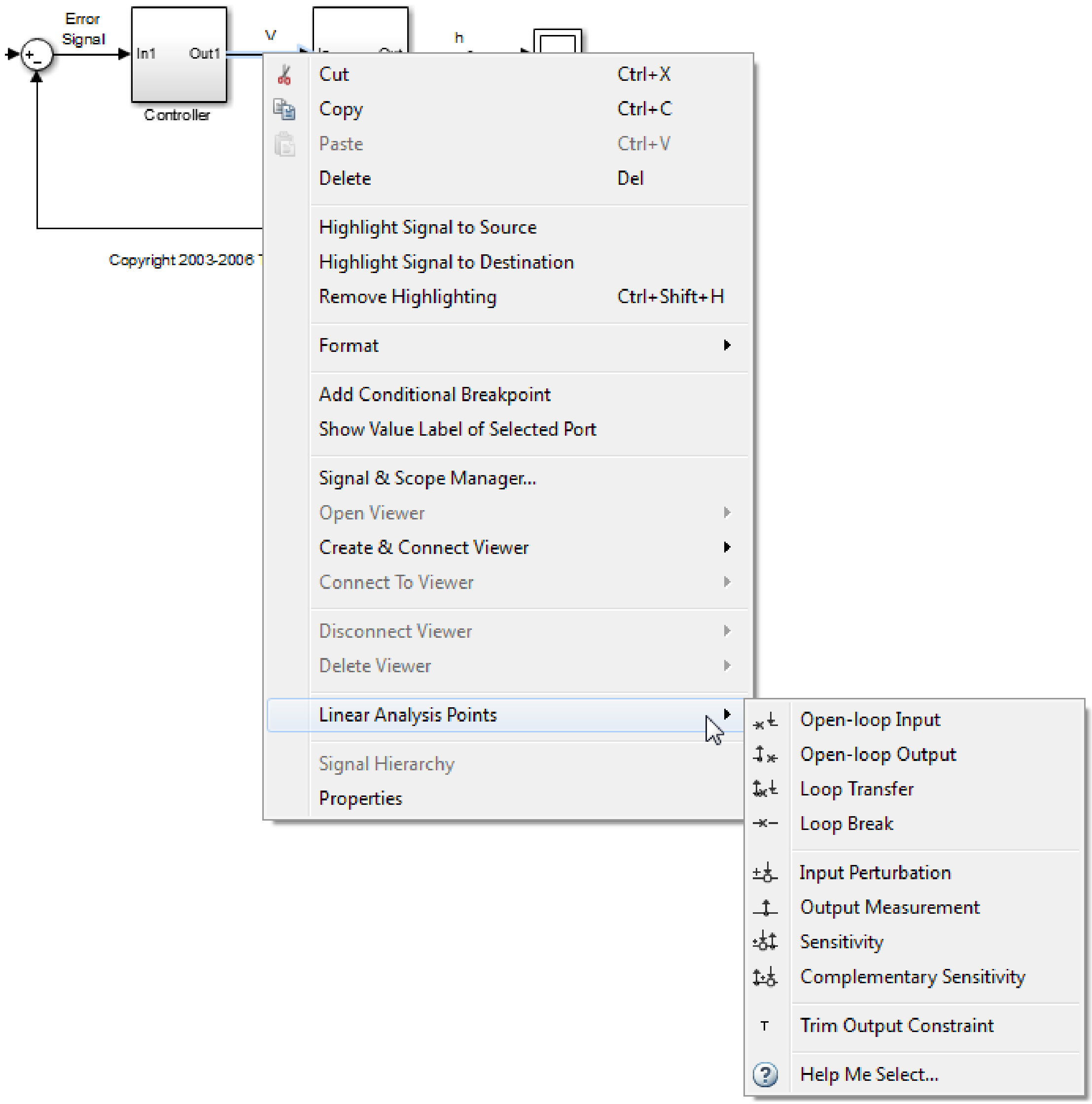


Figure 3: Marking Analysis Points in a Simulink Model

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.

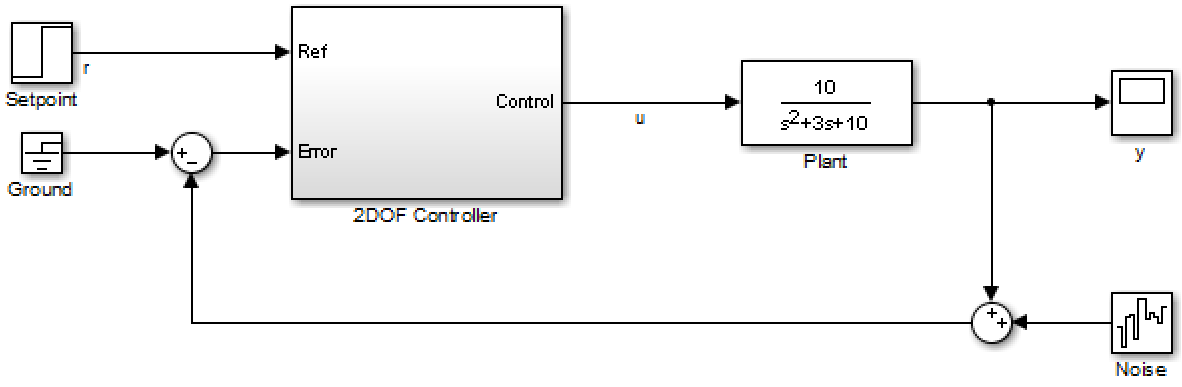


Figure 4: Simple Feedback Loop in Simulink

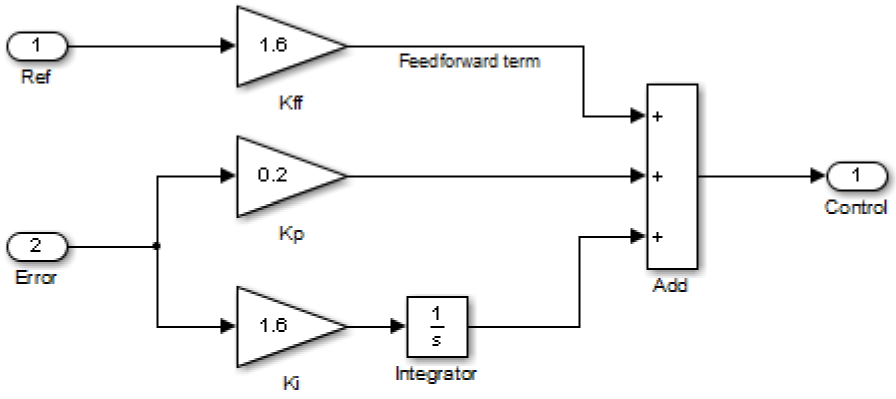


Figure 5: 2DOF Controller Subsystem

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

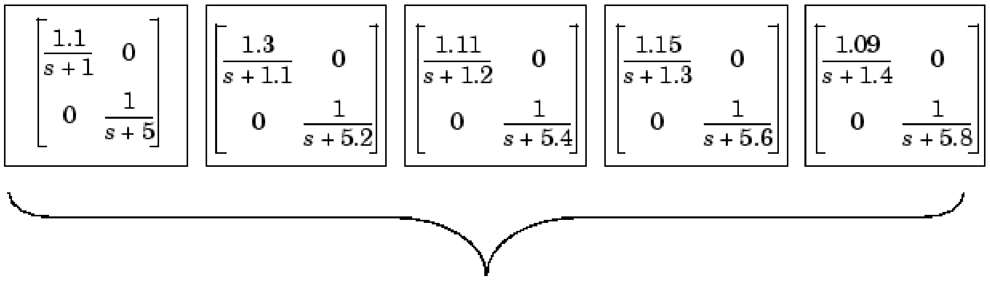
Uses of model arrays include:

- 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

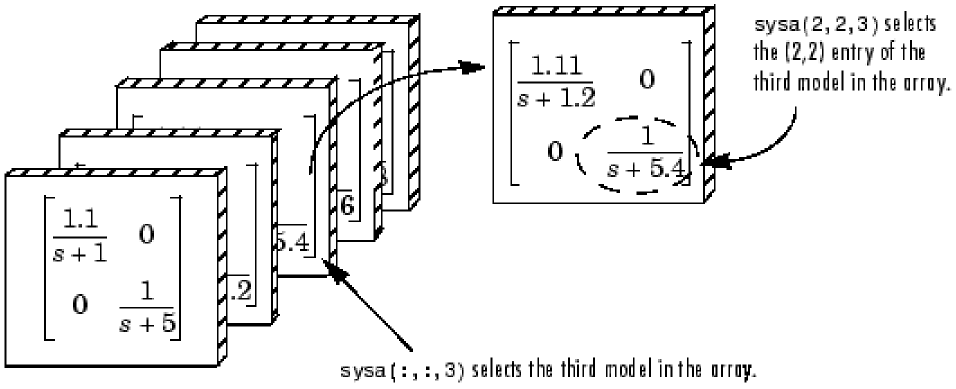
To visualize the concept of a model array, consider the set of five transfer function models shown below. In this example, each model has two inputs and two outputs. They differ by parameter variations in the individual model components.


$\begin{bmatrix} \frac{1.1}{s+1} & 0 \\ 0 & \frac{1}{s+5} \end{bmatrix}$
$\begin{bmatrix} \frac{1.3}{s+1.1} & 0 \\ 0 & \frac{1}{s+5.2} \end{bmatrix}$
$\begin{bmatrix} \frac{1.11}{s+1.2} & 0 \\ 0 & \frac{1}{s+5.4} \end{bmatrix}$
$\begin{bmatrix} \frac{1.15}{s+1.3} & 0 \\ 0 & \frac{1}{s+5.6} \end{bmatrix}$
$\begin{bmatrix} \frac{1.09}{s+1.4} & 0 \\ 0 & \frac{1}{s+5.8} \end{bmatrix}$

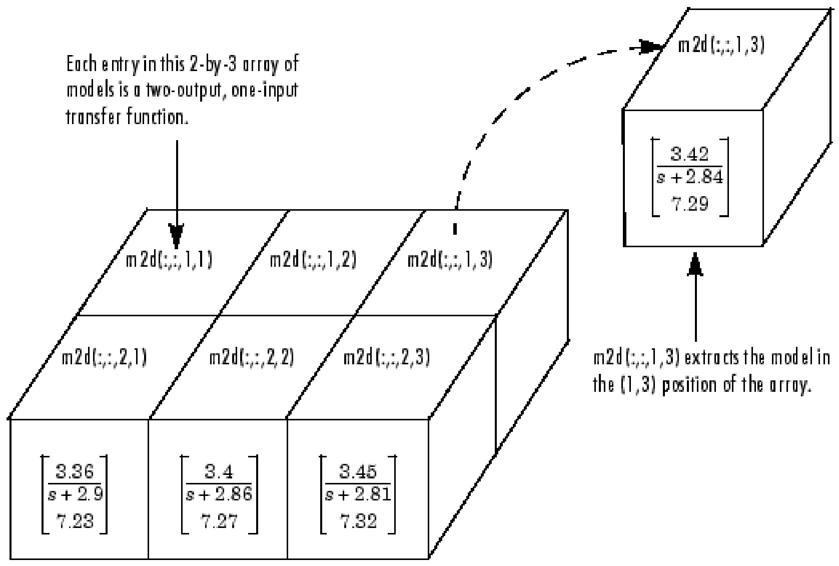
Just as you might collect a set of two-by-two matrices in a multidimensional array, you can collect this set of five transfer function models as a list in a model array under one variable name, say, `sys`. Each element of the model array is a single model object.

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.


$\begin{bmatrix} \frac{1.1}{s+1} & 0 \\ 0 & \frac{1}{s+5} \end{bmatrix}$
$\begin{bmatrix} \frac{1.11}{s+1.2} & 0 \\ 0 & \frac{1}{s+5.4} \end{bmatrix}$
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`.


Each entry in this 2-by-3 array of models is a two-output, one-input transfer function.
m2d(:,:,1,1)
m2d(:,:,1,2)
m2d(:,:,1,3)
m2d(:,:,2,1)
m2d(:,:,2,2)
m2d(:,:,2,3)
$\begin{bmatrix} \frac{3.36}{s+2.9} \\ 7.23 \end{bmatrix}$
$\begin{bmatrix} \frac{3.4}{s+2.86} \\ 7.27 \end{bmatrix}$
$\begin{bmatrix} \frac{3.45}{s+2.81} \\ 7.32 \end{bmatrix}$
m2d(:,:,1,3)
$\begin{bmatrix} \frac{3.42}{s+2.84} \\ 7.29 \end{bmatrix}$
m2d(:,:,1,3) extracts the model in the (1,3) position of the array.

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.

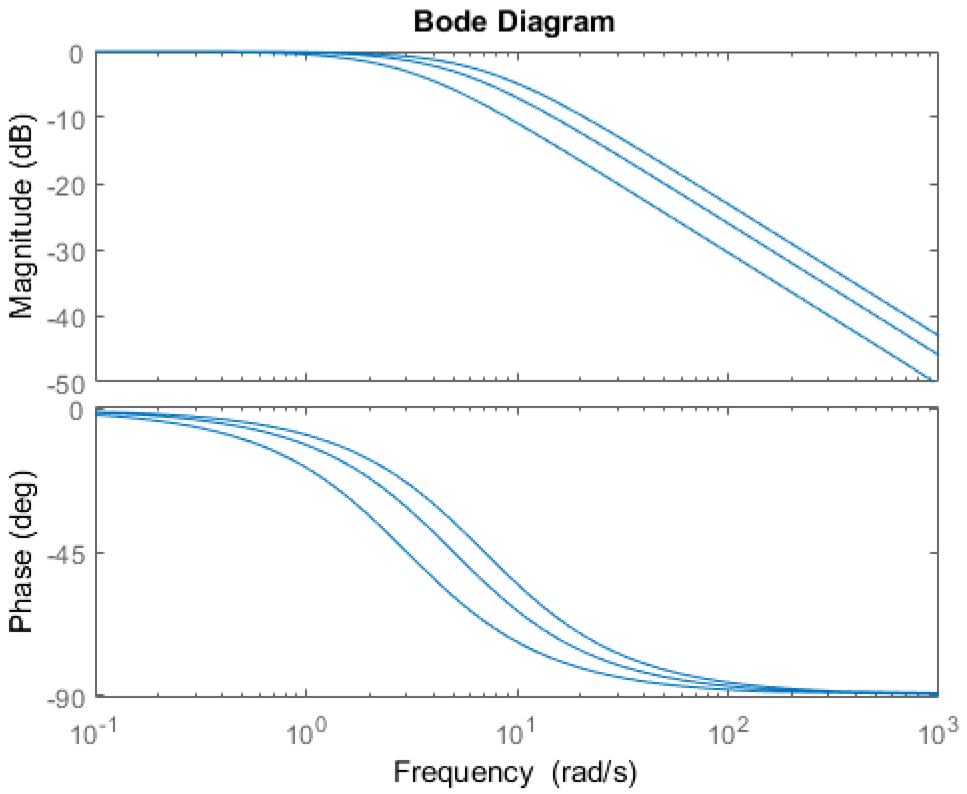
```
Farray.SamplingGrid = struct('alpha',[3 5 7]);
Farray
```

```
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.

More About

- "Model Arrays" on page 2-96

Select Models from Array

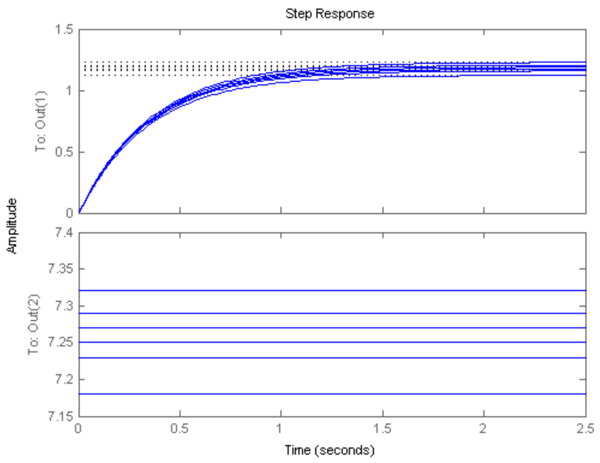
This example shows how to select individual models or sets of models from a model array using array indexing.

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 *i*th 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)`.

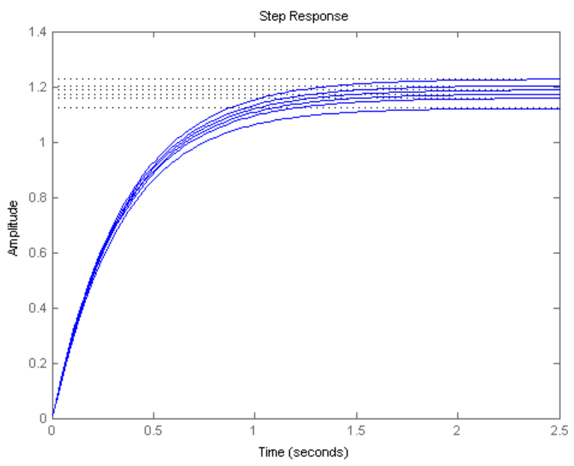
---

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)
```

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

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

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:

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

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];
```

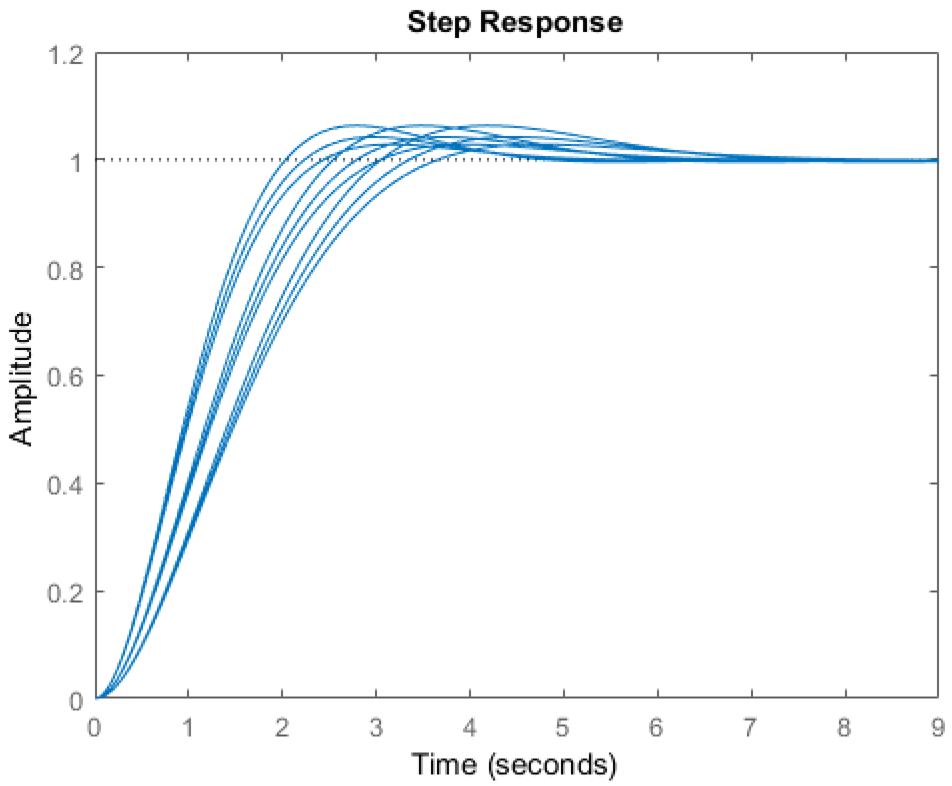
Build the array by looping through all combinations of parameter values.

```
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.

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 `replaceBlock`.

Consider the second-order filter represented by:

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

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, anc
    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];
[wngrid,zetagrid] = ndgrid(wnvals,zetavals);
Fsample = replaceBlock(F,'wn',wngrid,'zeta',zetagrid);
```

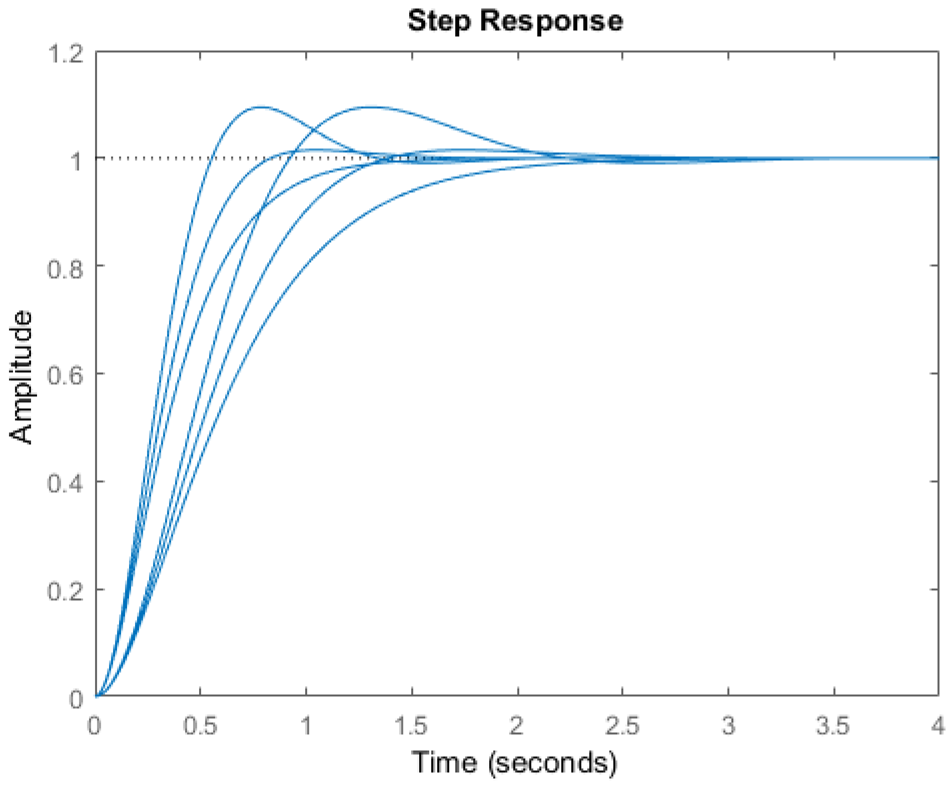
The `ndgrid` command produces a full 2-by-3 grid of parameter combinations. `Fsample` is a 2-by-3 array of state-space models. Each entry in the array is a state-space model that represents `F` evaluated at the corresponding (`wn`, `zeta`) pair. For example, `Fsample(:,:,2,3)` has `wn` = 5 and `zeta` = 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.

```
Fsample.SamplingGrid = struct('wn',wngrid,'zeta',zetagrid);
```

When you display `Fsample`, 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 `wn` and `zeta` values as specified in `Fsample.SamplingGrid`.

More About

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

Linear Parameter-Varying Models

In this section...

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:

$$\begin{aligned}
&dx(t) = A(p)x(t) + B(p)u(t) \\
&y(t) = C(p)x(t) + D(p)u(t) \\
&x(0) = x_0
\end{aligned}$$

where

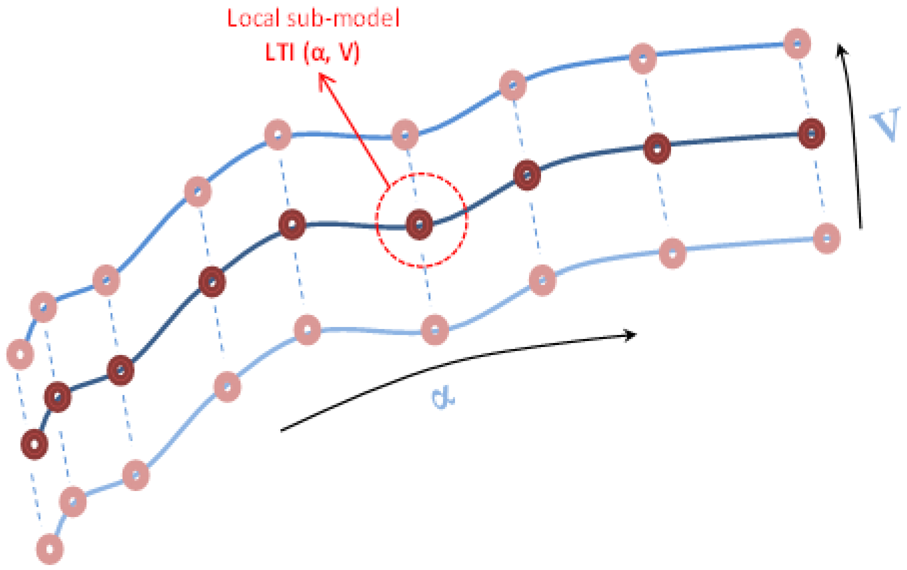
- `u(t)` are the inputs
- `y(t)` the outputs
- `x(t)` are the model states with initial value `x0`
- $dx(t)$ is the state derivative vector $\dot{x}$ for continuous-time systems and the state update vector $x(t+\Delta T)$ for discrete-time systems. ΔT is the sample time.
- `A(p)`, `B(p)`, `C(p)` and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

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

$\overline{dx}(p),\ \bar{x}(p),\ \bar{u}(p),\ \bar{y}(p)$ 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.

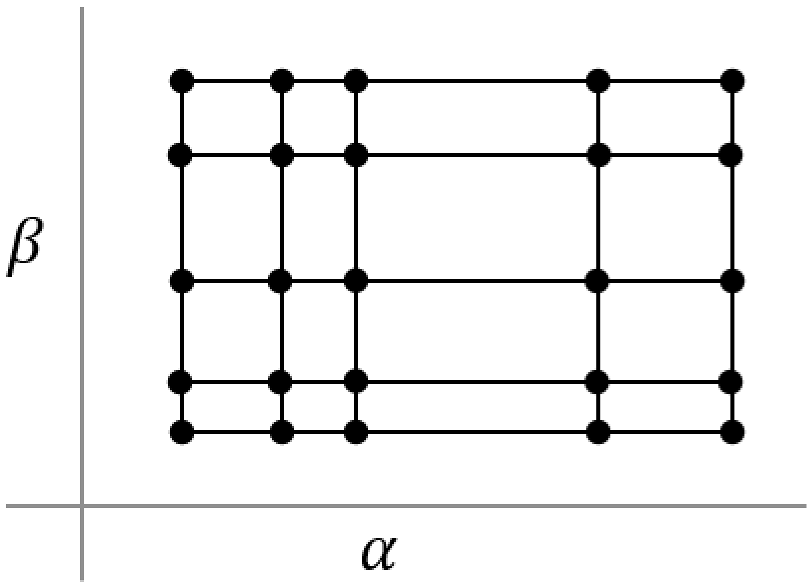
In the affine representation, the linear model at a given point `p = p*` in the scheduling space is given by:

$$\begin{aligned} d\Delta x(t,p^*) &= A\left(p^*\right)\Delta x\left(t,p^*\right) + B\left(p^*\right)\Delta u\left(t,p^*\right) \\ \Delta y\left(t,p^*\right) &= C\left(p^*\right)\Delta x\left(t,p^*\right) + D\left(p^*\right)\Delta u\left(t,p^*\right) \end{aligned}$$

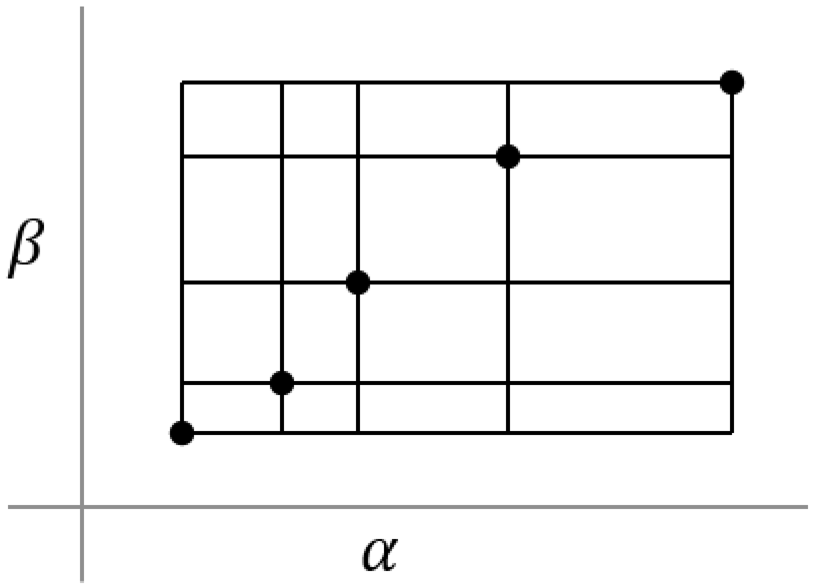
The states of this linear model are related to the states of the overall LPV model (Equation 2-2) by $\Delta x\left(t,p^*\right) = x(t) - \bar{x}\left(p^*\right)$. Similarly, $\Delta y\left(t,p^*\right) = y(t) - \bar{y}\left(p^*\right)$ and $\Delta u\left(t,p^*\right) = u(t) - \bar{u}\left(p^*\right)$.

Regular vs. Irregular Grids

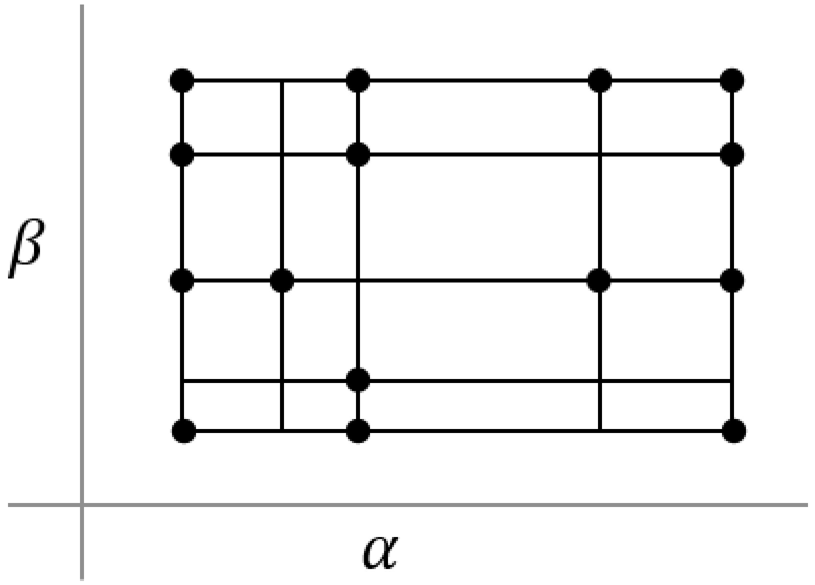
Consider a system that uses two scheduling parameters – α and β. When α and β vary monotonically, a regular grid is formed, as shown in the next figure. The state space array contains a value at every combination of α and β values. Regular grid does not imply uniform spacing between values.



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:

$$\begin{aligned}\dot{x}_1 &= x_1^2 + x_2^2 \\ \dot{x}_2 &= -2x_1 - 3x_2 + 2u \\ y &= x_1 + 2\end{aligned}$$

Suppose you use $p(t) = \dot{x}_1$ as a scheduling variable. At a given time instant t = t_0, you have:

$$\begin{aligned}\dot{x}_1 &\approx 2x_1(t_0)x_1 + 2x_2(t_0)x_2 - \dot{x}_1(t_0) \\ \dot{x}_2 &= -2x_1 - 3x_2 + 2u \\ y &= x_1 + 2\end{aligned}$$

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” on page 2-117
- “Proxy Modeling for Faster Simulations” on page 2-118

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 2-119.

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 SimMechanics™ and SimPowerSystems™ 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 2-119
- “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^o 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.

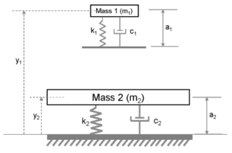


Figure 1: Elastic body bouncing on a fixed elastic surface.

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.

```
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.

```
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]);
```

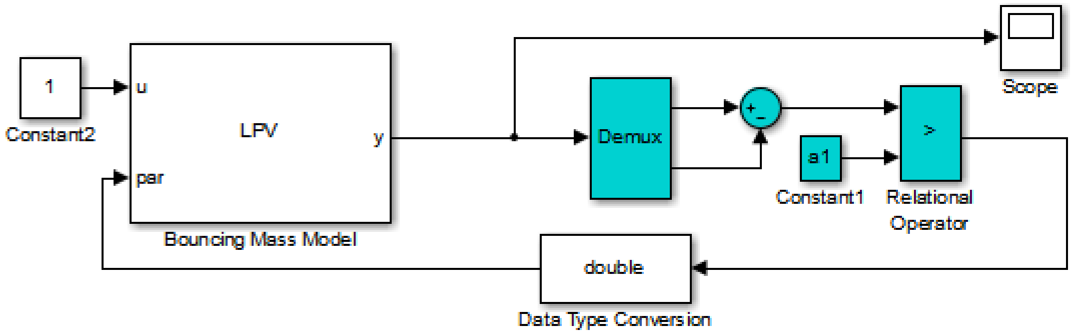
Whether the masses are in contact or not is decided by the relative positions of the two masses; when $y_1 - y_2 > a_1$, the masses are not in contact.

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')
```



The block called "Bouncing Mass Model" is an LPV System block. Its parameters are specified as follows:

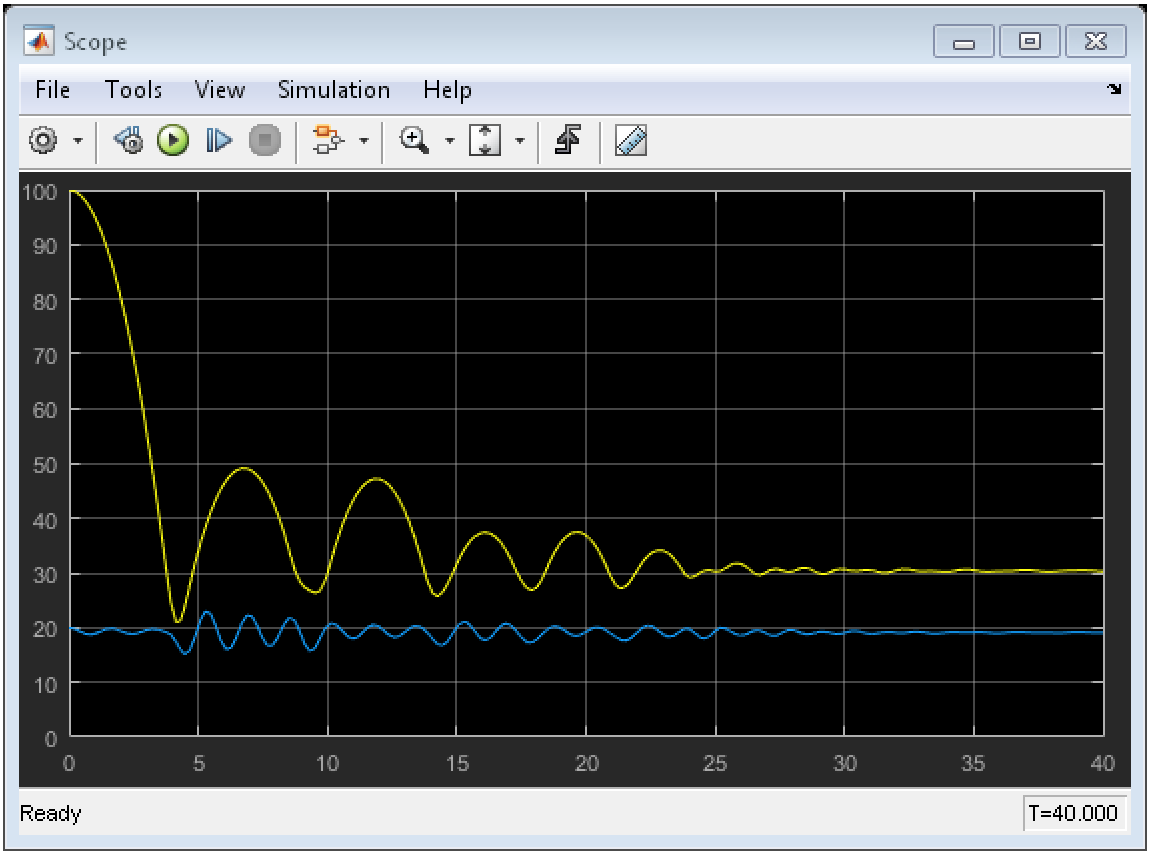
- For "State-space array" field, specify the state-space model array `sys` that was created above.
- For "Initial state" field, specify the initial positions and velocities of the two masses. Note that the state vector is: $[y_1, \dot{y}_1, y_2, \dot{y}_2]$. Specify its value as [h1 0 h2 0]'.
- Under 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

Model Properties

In this section...

Model Properties

Model properties are data fields that let you specify model attributes, such as coefficients, sample time, and metadata such as channel names.

For help on the properties associated with each model type, see the corresponding reference page, such as `tf`, `zpk`, or `ss`.

Specify Model Properties at Model Creation

This example shows how to specify transfer function model properties when you create the model. You can use the technique of this example to access and set properties of any type of model.

To specify model properties at model creation, use `Name,Value` pair syntax. For example, assign a transport delay and an output name to a new transfer function model:

```
H = tf(1,[1 10],'ioDelay',6.5,'OutputName','velocity')
```

Examine and Specify Properties of an Existing Model

This example shows how to examine and specify properties of an existing `ss` model using `get`, `set`, and dot notation. You can use the techniques of this example to examine and specify properties of any type of model.

1 Load an existing model object.

   ```
   load PadeApproximation1 sys;
   ```

   `sys` is a state-space (`ss`) model.
2 Examine the current property values.

   ```
   get(sys)
   ```

This command displays all the properties of `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: []
```

3 Specify input delay and channel names.

You can use dot notation to directly reference property values.

```
sys.InputDelay = 4.2;
sys.InputName = 'thrust';
sys.OutputName = 'velocity';
```

Alternatively, you can use the `set` command to specify multiple property values at one time.

```
set(sys,'InputDelay',4.2,'InputName','thrust',...
                         'OutputName','velocity');
```

Caution Changing certain properties, such as `Ts` and `TimeUnits`, can cause undesirable changes in system behavior. See the property descriptions in the model reference pages for more information.

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: {[0 1]}
            den: {[1 12]}
       Variable: 's'
        ioDelay: 0
     InputDelay: 0
    OutputDelay: 2.5000
             Ts: 0
       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 3-7

More About

- "Model Properties" on page 3-2

Attach Metadata to Models

In this section...

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 12-2 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 12-2 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.

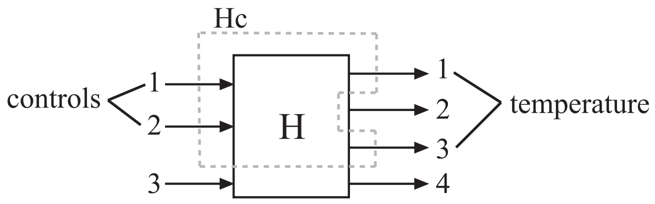
3 Extract the subsystem corresponding to the `controls` inputs and the `temperature` outputs.

   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

- "Query Model Characteristics" on page 3-12

More About

- "Model Properties" on page 3-2

Query Model Characteristics

In this section...
"Query Model Dynamics" on page 3-12
"Query Array Size" on page 3-13

How to query model characteristics such as stability, time domain, and number of inputs and outputs.

Query Model Dynamics

This example shows how to query model dynamics such as stability, time domain, and model order. You can use the techniques of this example on type of model.

1. Load a saved state-space model.

   ```
   load queryexample.mat T;
   ```

2. Query whether `T` has stable dynamics.

   ```
   isstable(T)
   ```

   The `isstable` command returns a Boolean value of `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` returns `0` (false).

3. Query whether `T` has any time delays.

   ```
   hasdelay(T)
   ```

   This command returns `1`, which indicates that `T` has a time delay. For a state-space model, time delay can be stored as input delay, output delay, or internal delay. Use `get(T)` to determine which properties of `T` hold the time delay.

4. Query whether `T` is proper.

   ```
   isproper(T)
   ```

   This command returns `1`, if the system has relative degree ≤ 0.

5. Query the order of `T`.

   ```
   order(T)
   ```

For a state-space model, `order` returns the number of states. For a `tf` or `zpk` model, the order is the number of states required for a state-space realization of the system.

6 Query whether `T` is a discrete-time model.

```
isdt(T)
```

This command returns `1`, indicating that `T` is a discrete-time model. Similarly, you can use `isct` to query whether `T` is a continuous-time model.

Query Array Size

This example shows how to query the size of model arrays, including the number of inputs and outputs and the number of models in an array.

You can also use these commands on individual models.

1 Load a saved model array.

```
load queryexample sysarr
```

2 Query the array dimensions.

```
size(sysarr)
```

This command displays the array size, and the number of inputs, outputs, and states of the models in the array.

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

```
dims = size(sysarr)
```

This command produces the result:

```
dims =

     3     1     2     4
```

The first two entries `3,1` are the number of outputs and inputs, respectively. The remaining entries `2,4` are the array dimensions.

Tip To query the number of array dimensions, use `ndims`.

3. Query the number of models in the array.

   ```
   N = nmodels(sysarr)
   ```

   Because `sysarr` is a 2-by-4 array, this command returns a result of 2 × 4 = 8.
4. Query whether the models in `sysarr` are single-input, single-output (SISO).

   ```
   issiso(sysarr)
   ```

   The command `issiso` returns a Boolean value of `1` (true) if the models in the array are SISO, and `0` (false) if the models are not SISO.
5. Query the number of inputs and outputs in the models in the array using `iosize`.

   ```
   ios = iosize(sysarr)
   ```

   This command returns a numeric array containing the number of outputs and inputs of the models in the array.

See Also

`isproper` | `isstable` | `size`

Related Examples

- “Select Models from Array” on page 2-102

More About

- “Model Properties” 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 $H(z) = \frac{z-1}{z^2 - 3z + 2}$

   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

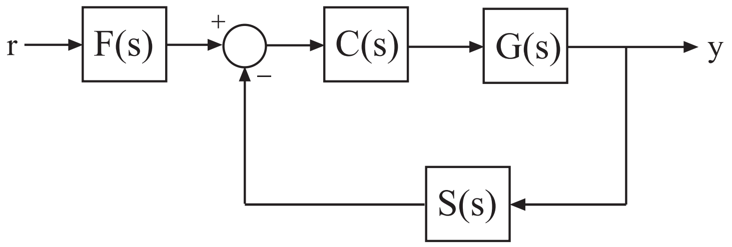
- "Transfer Functions" on page 2-3

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

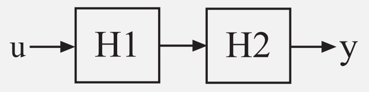
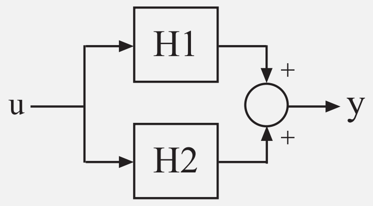
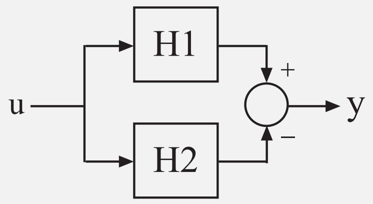
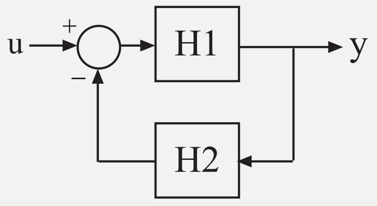
- "Catalog of Model Interconnections" on page 4-3

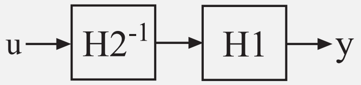
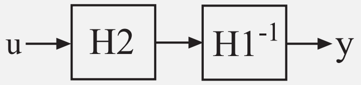
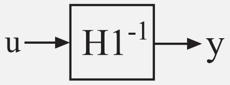
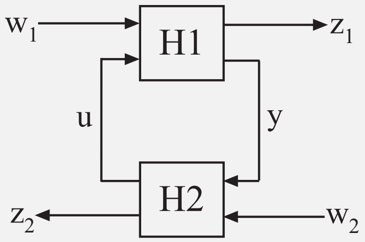
Catalog of Model Interconnections

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

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
`'`	Pertransposition
`.'`	Transposition

Operation	Description
`^`	Powers of a dynamic system model (as in s^2)

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

- “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

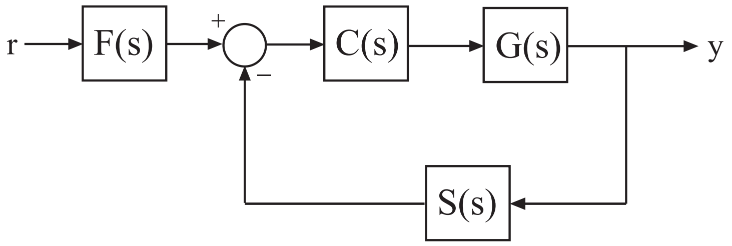
More About

- “How the Software Determines Properties of Connected Models” on page 4-26
- “Recommended Model Type for Building Block Diagrams” on page 4-29

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 $T(s) = \frac{H}{1 + HS}$.

```
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

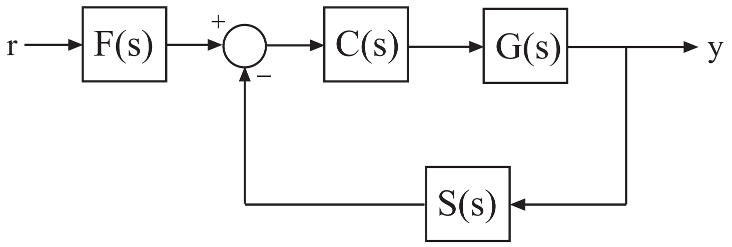
- “Tunable Model of SISO Feedback Loop” on page 4-8
- “Multi-Loop Control System” on page 4-10
- “MIMO Control System” on page 4-19

More About

- “Catalog of Model Interconnections” on page 4-3

Tunable Model of SISO Feedback Loop

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 = ltiblock.pid('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 `ltiblock.pid` 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 ltiblock.pid]
    a: [1x1 realp]
```

If you have Robust Control Toolbox software, you can use tuning commands such as `systune` to tune the free parameters of `T` to meet design requirements you specify.

See Also

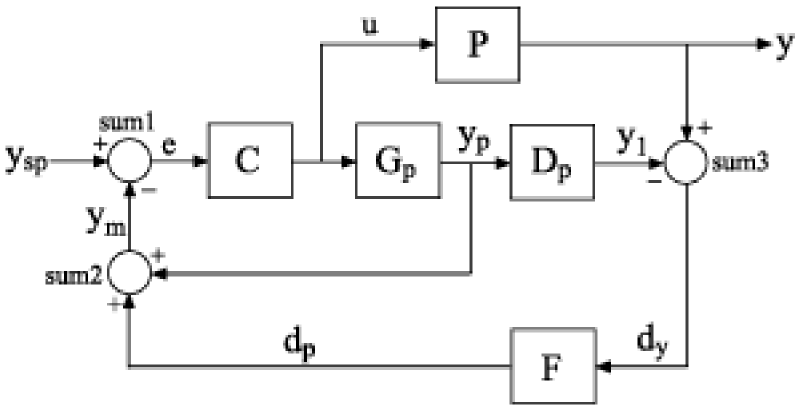
`feedback` | `ltiblock.pid`

More About

- “Control Design Blocks” on page 1-16
- “Dynamic System Models” on page 1-10

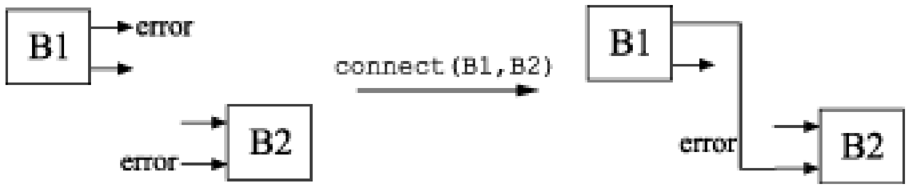
Multi-Loop Control System

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

- "Tunable Model of SISO Feedback Loop" on page 4-8
- "MIMO Control System" on page 4-19
- "Mark Analysis Points in Closed-Loop Models" on page 4-13

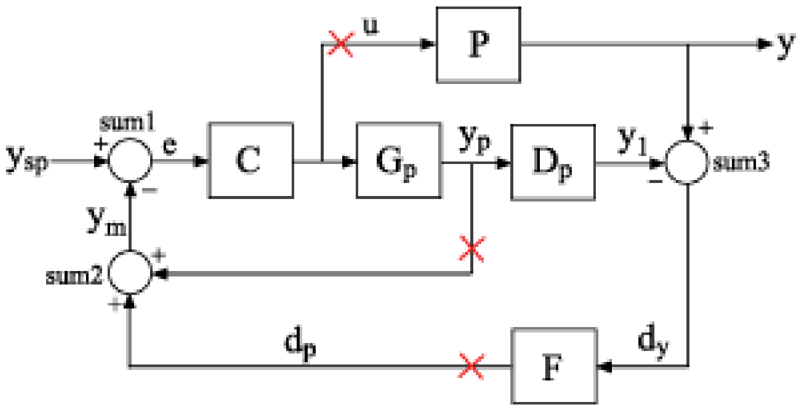
More About

- "How the Software Determines Properties of Connected Models" on page 4-26

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
```

```
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).

```
sum1 = sumblk('e = ysp - ym');
sum2 = sumblk('ym = yp + dp');
sum3 = sumblk('dy = y - y1');
```

Assemble the complete model.

```
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
```

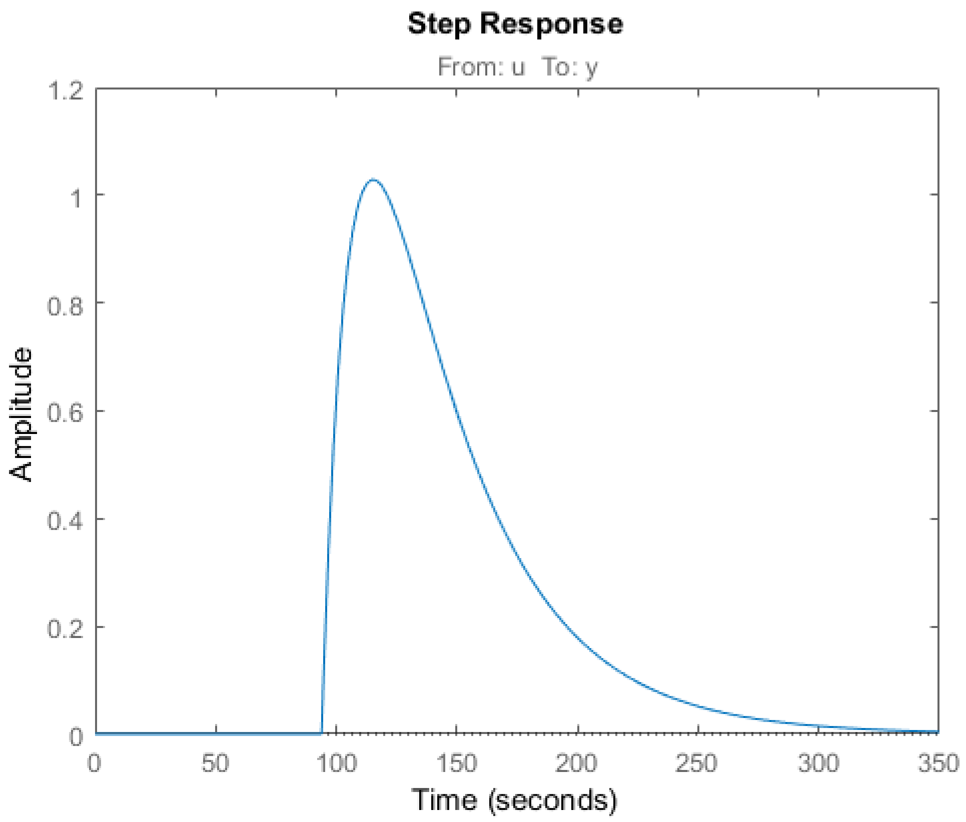
When you use the `APs` input argument, the `connect` command automatically inserts an `AnalysisPoint` block into the generalized state-space (`genss`) model, `T`. The automatically generated block is named `AnalysisPoints_`. The three channels of `AnalysisPoints_` correspond to the three locations specified 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'
```

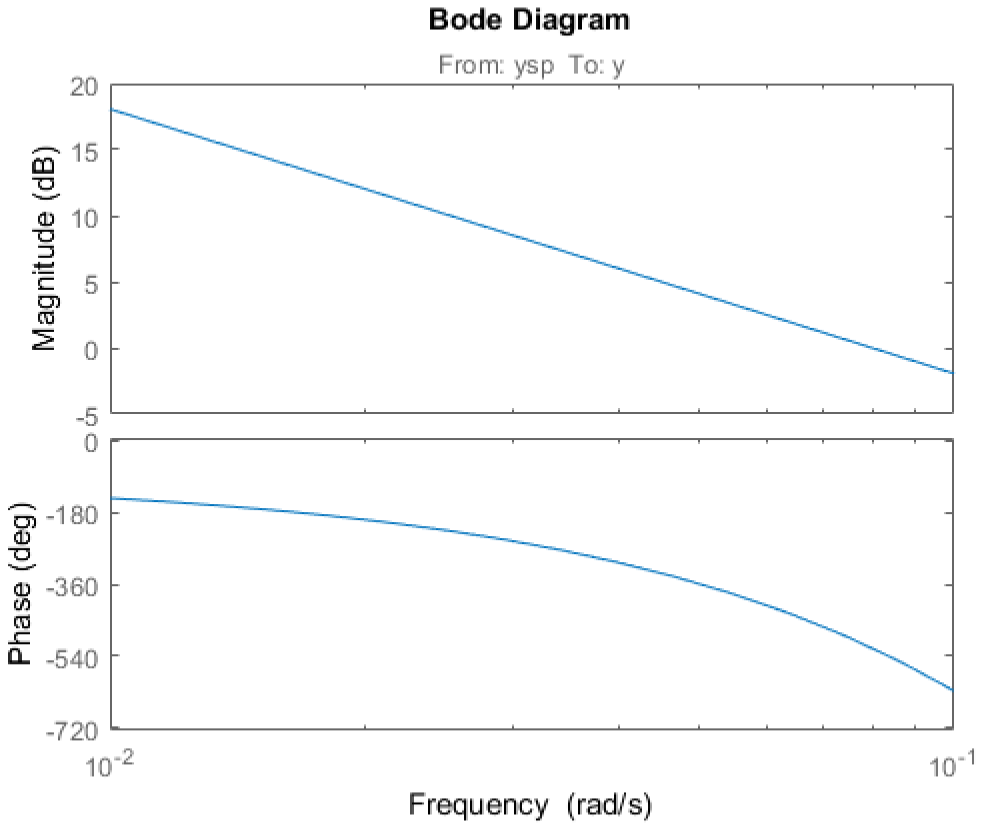
Use these locations as inputs, outputs, or loop openings when you extract responses from the model. For example, extract and plot the response at the system output to a step disturbance at the plant input, *u*.

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



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)
```



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

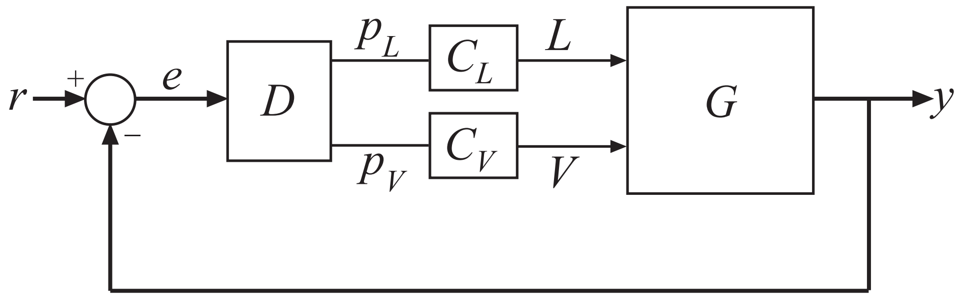
More About

- “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 = ltiblock.gain('Decoupler',eye(2));
D.u = 'e';
D.y = {'pL','pV'};

C_L = ltiblock.pid('C_L','pi');  C_L.TimeUnit = 'minutes';
C_L.u = 'pL'; C_L.y = 'L';

C_V = ltiblock.pid('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

- "Tunable Model of SISO Feedback Loop" on page 4-8
- "Multi-Loop Control System" on page 4-10
- "MIMO Control System" on page 4-19

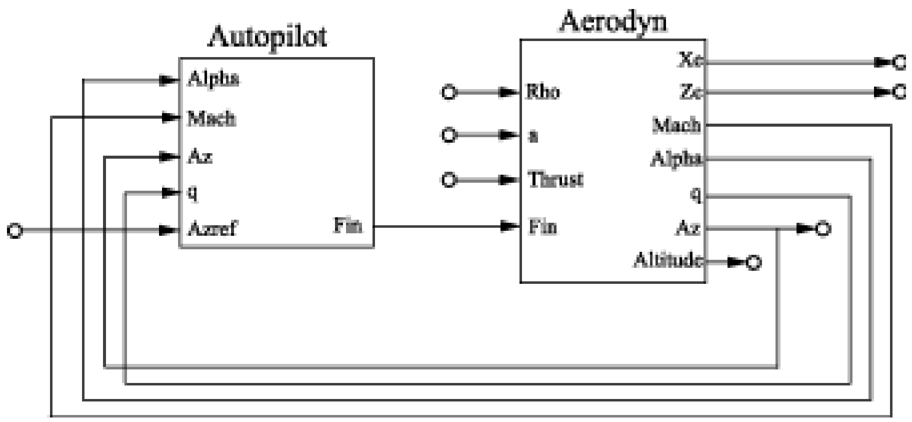
More About

- "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`.

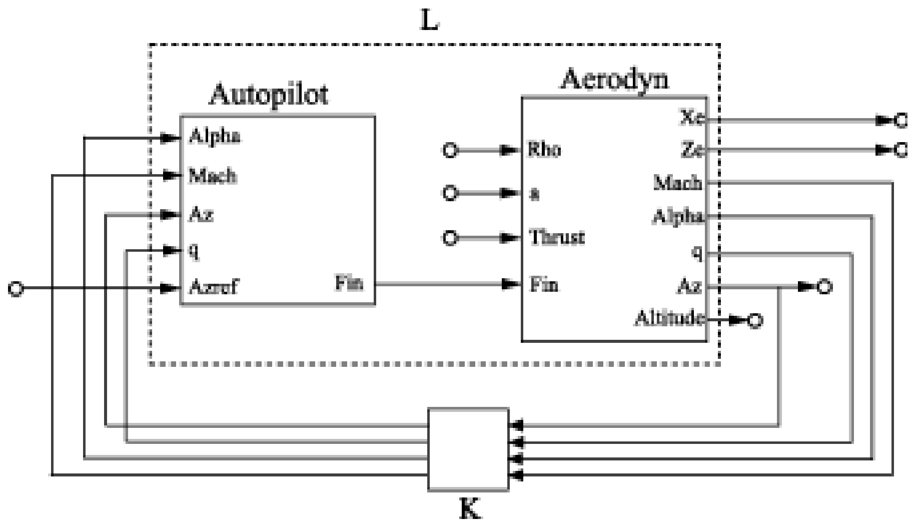
```
T1 = connect(Autopilot,Aerodyn,'Azref','Az');
```

```
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.

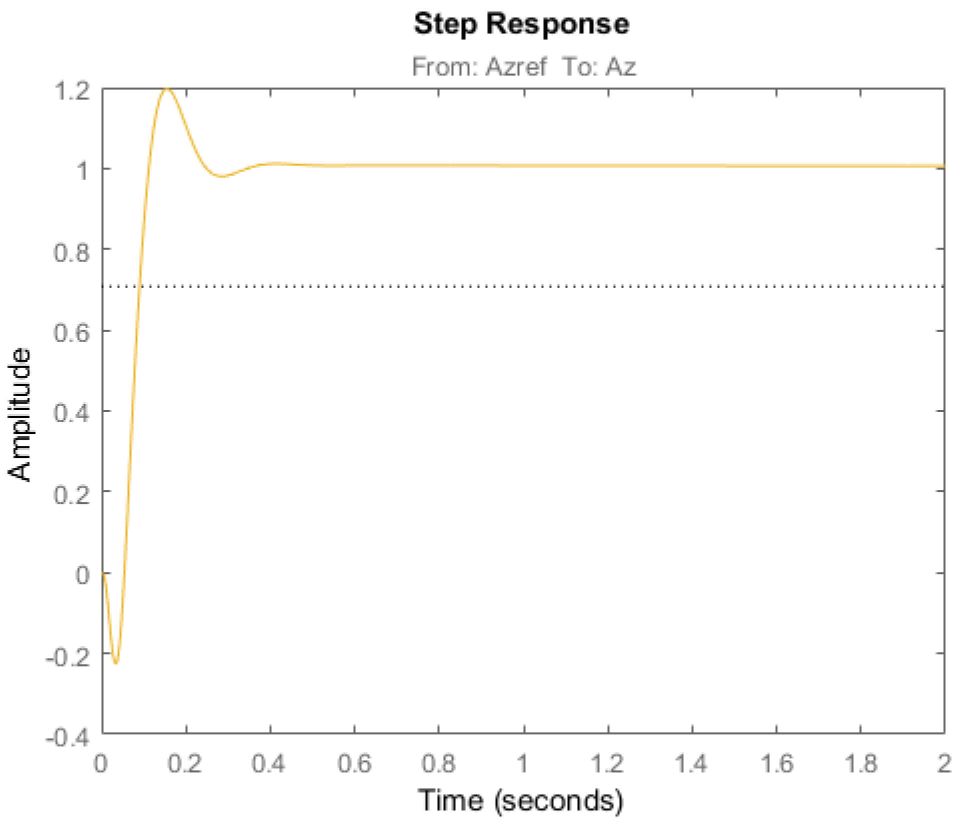
```
L = series(Autopilot,Aerodyn,1,4);
K = ss(eye(4));
T3 = feedback(L,K,[1 2 3 4],[4 3 6 5],+1);
```

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)
```



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 = ltiblock.tf('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

- “Multi-Loop Control System” on page 4-10

More About

- “How the Software Determines Properties of Connected Models” on page 4-26
- “Recommended Model Type for Building Block Diagrams” on page 4-29

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.

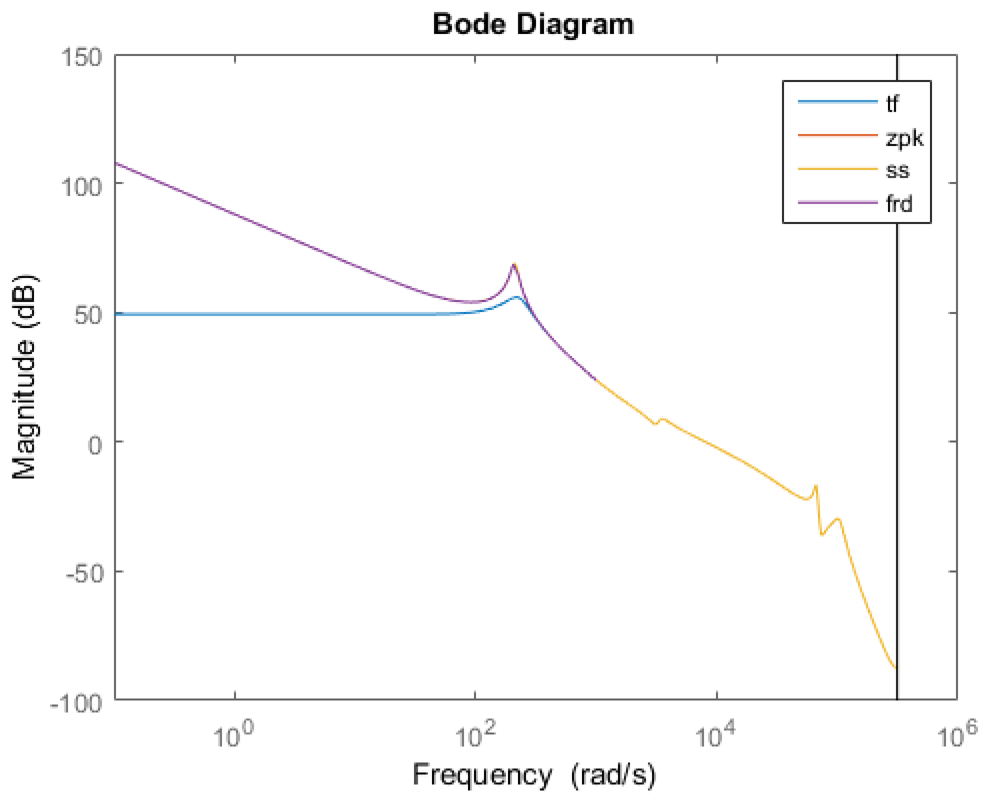
```
load numdemo Pd Cd
```

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

Model Transformation

Conversion Between Model Types

In this section...

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 16-2.

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.

Convert PID Controller to Transfer Function

This example shows how to convert a PID controller model to a transfer function model.

You can use the technique of this example to convert any type of model to another type of model.

Create a PID controller.

```
pid_sys = pid(1,1.5,3);
```

Convert `pid_sys` to a transfer function model.

```
C = tf(pid_sys);
```

`C` is a `tf` model representation of `pid_sys`.

You can similarly convert transfer function models to `pid` models, provided the `tf` model object represents a parallel-form PID controller with `Tf` ≥ 0.

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 Robust Control Toolbox command such as `systune` or `looptune`.

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` or `looptune`. 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`.

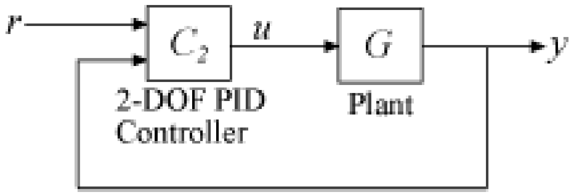
Decompose a 2-DOF PID Controller into SISO Components

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.

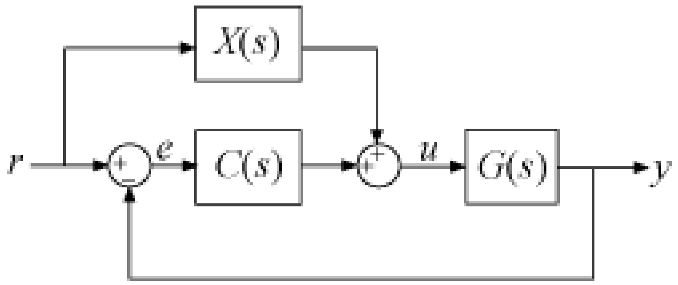
Obtain a 2-DOF PID controller. For this example, create a plant model, and tune a 2-DOF PID controller for it.

```
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')
```

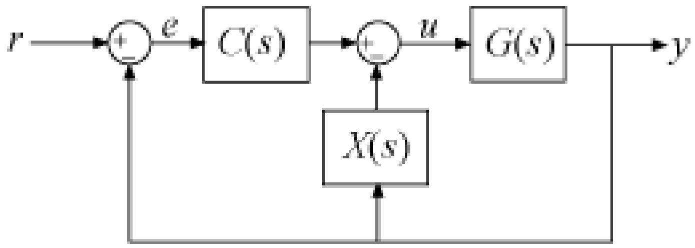
```
Cff =

             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);
```

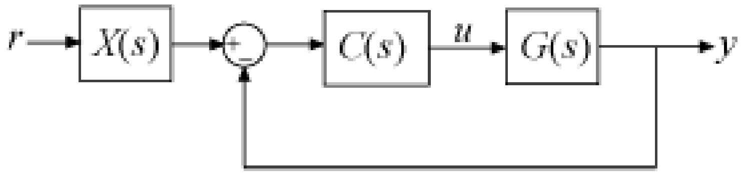
In the feedback configuration, the 2-DOF controller is represented as a SISO PID controller and an additional feedback compensator.



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.

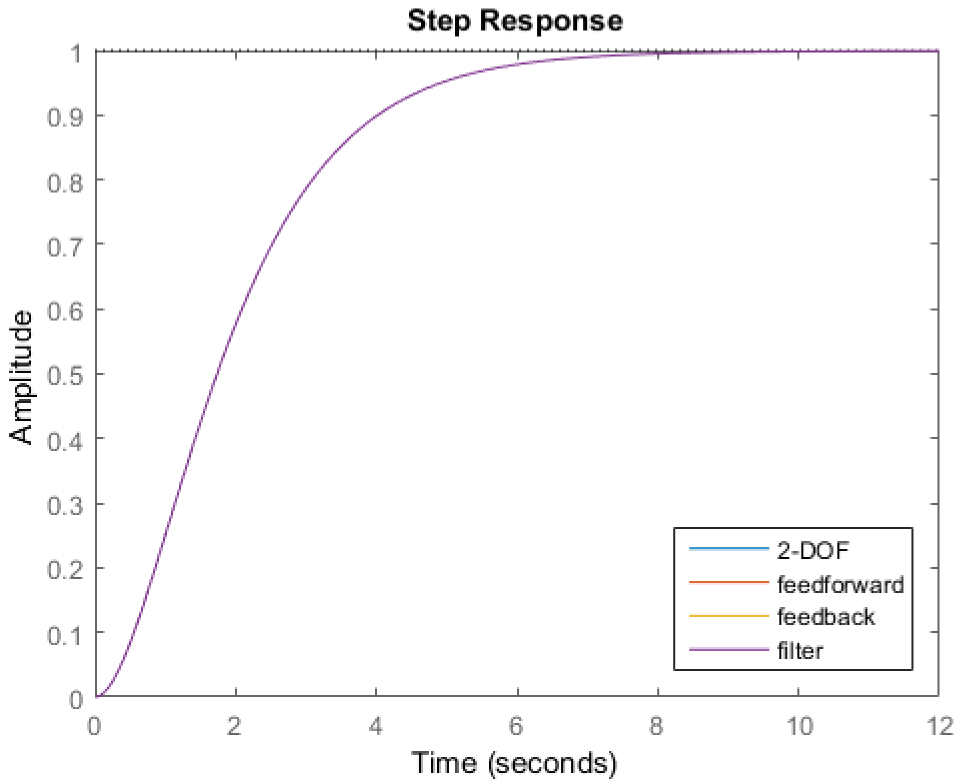
```
[Cfr, Xfr] = getComponents(C2,'filter');
Tfr = Xfr*feedback(G*Cfr,1);
```

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

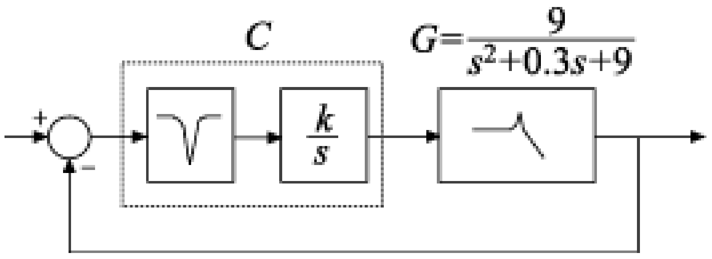
- "Two-Degree-of-Freedom PID Controllers" on page 2-17

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.

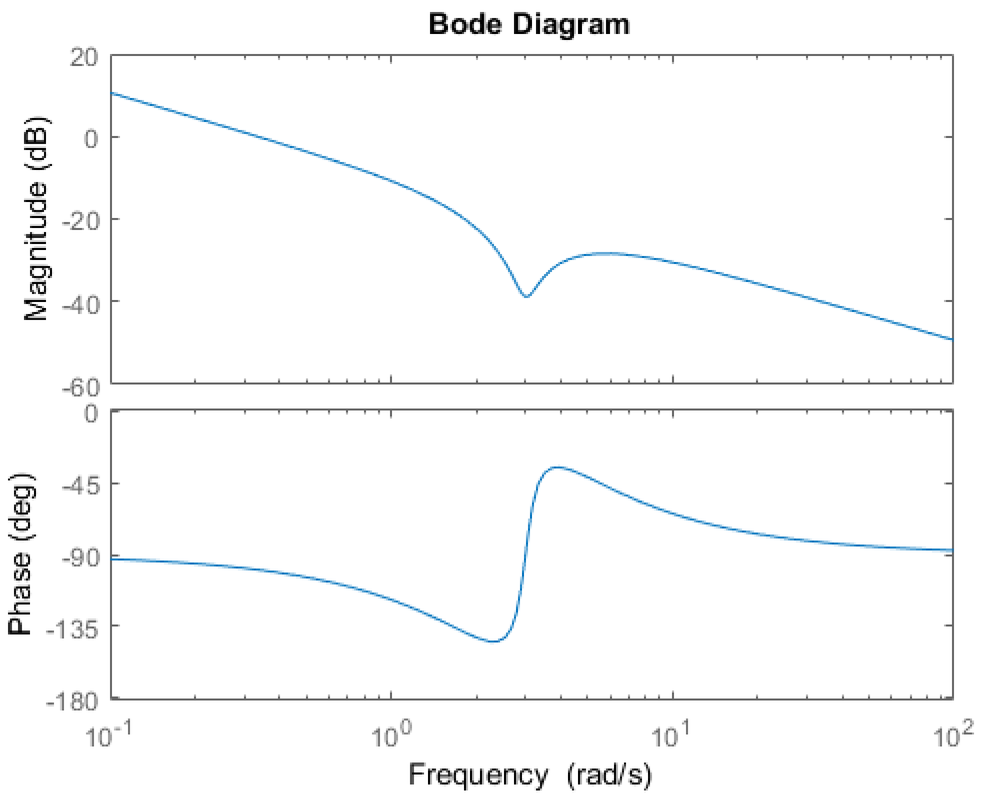
You might design a compensator in continuous time, and then need to convert it to discrete time for a digital implementation. When you do so, you want the discretization to preserve frequency-domain characteristics that are essential to your performance and stability requirements.

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)
```



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.

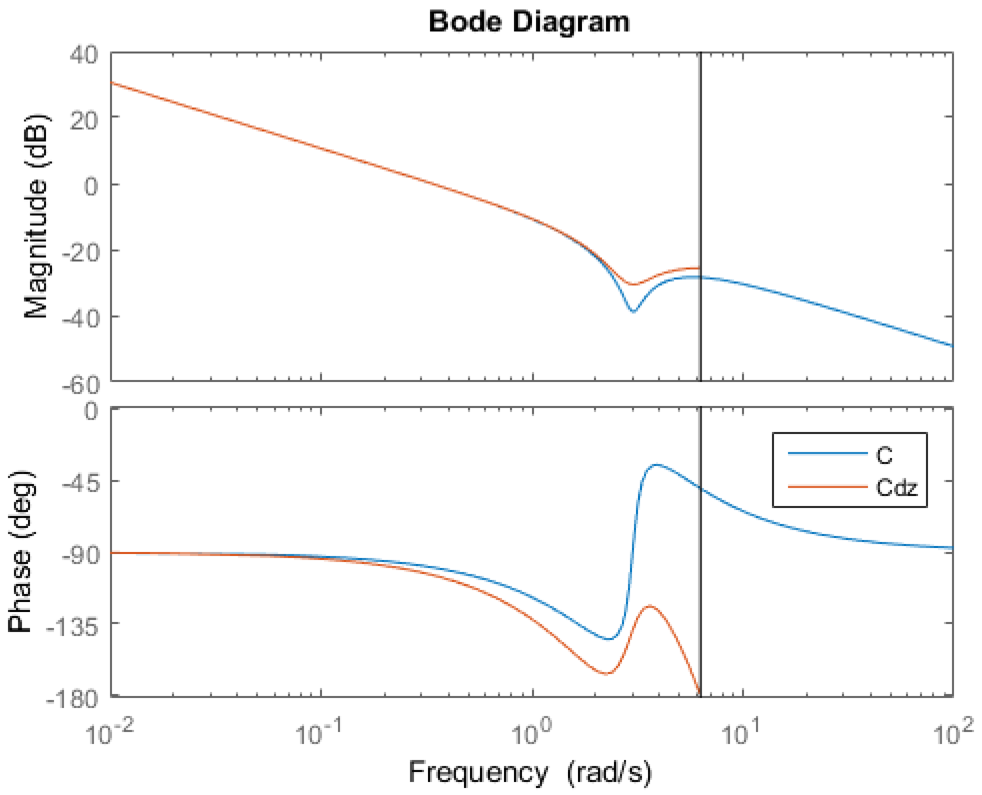
```
Cdz = c2d(C,0.5);
```

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`.

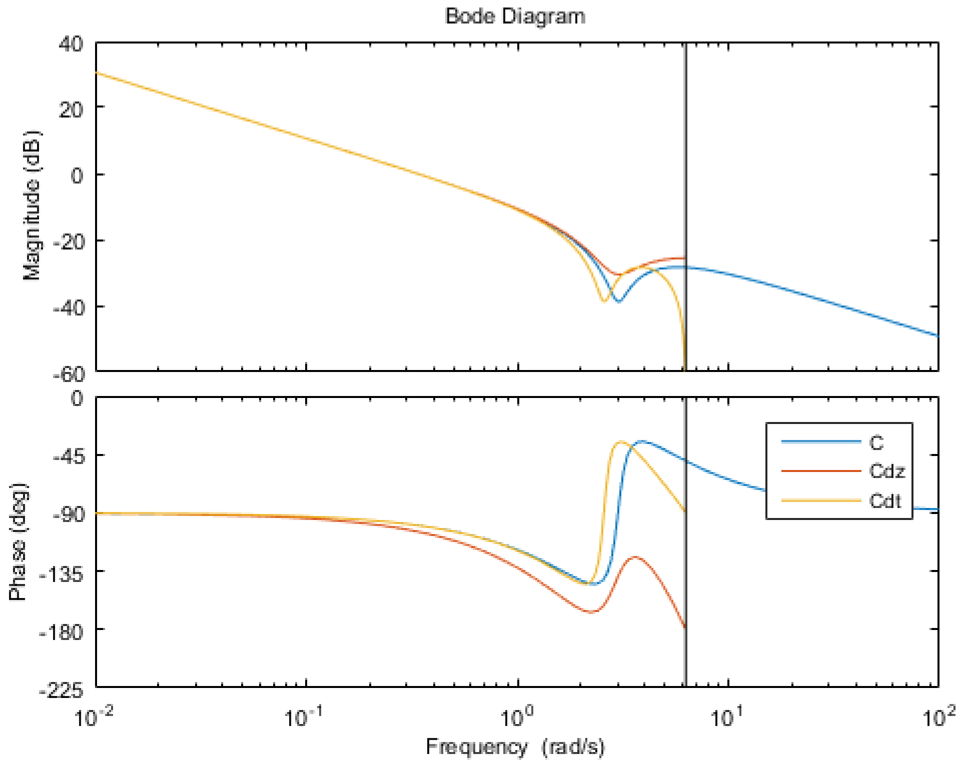
```
bodeplot(C,Cdz)
legend('C','Cdz');
```



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.

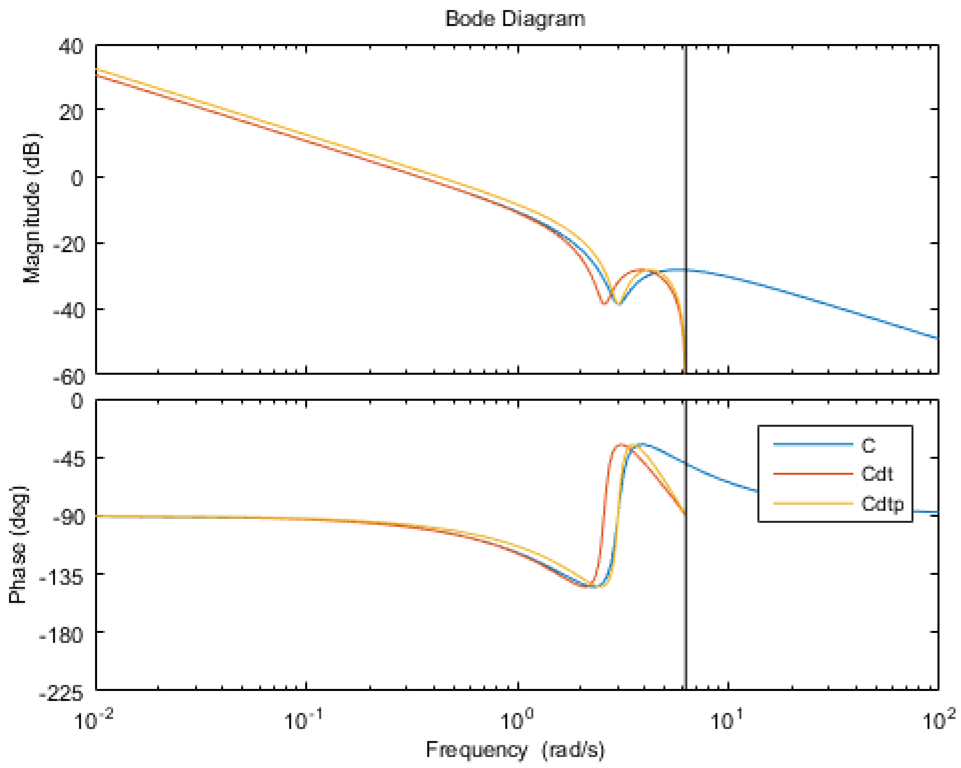
```
Cdt = c2d(C,0.5,'tustin');
plotopts = bodeoptions;
plotopts.Ylim = {[-60,40],[-225,0]};
bodeplot(C,Cdz,Cdt,plotopts)
legend('C','Cdz','Cdt')
```



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 5-16

More About

- "Continuous-Discrete Conversion Methods" on page 5-23

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.

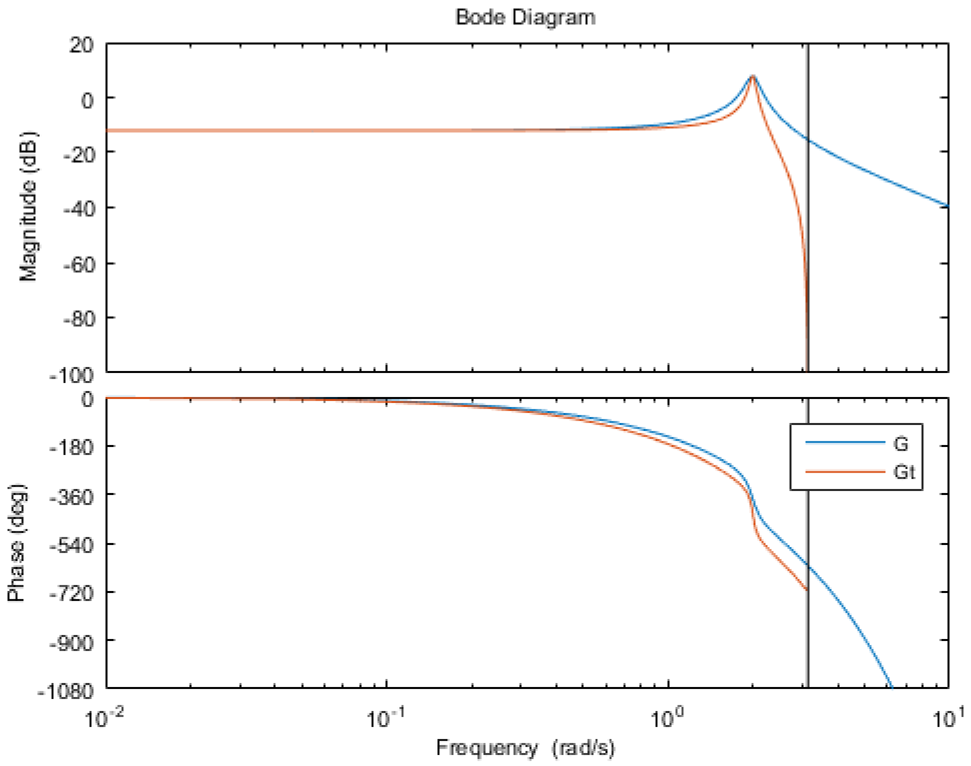
```
discopts = c2dOptions('Method','tustin','PrewarpFrequency',2);
Gt = c2d(G,1,discopts)
```

```
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')
```



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.

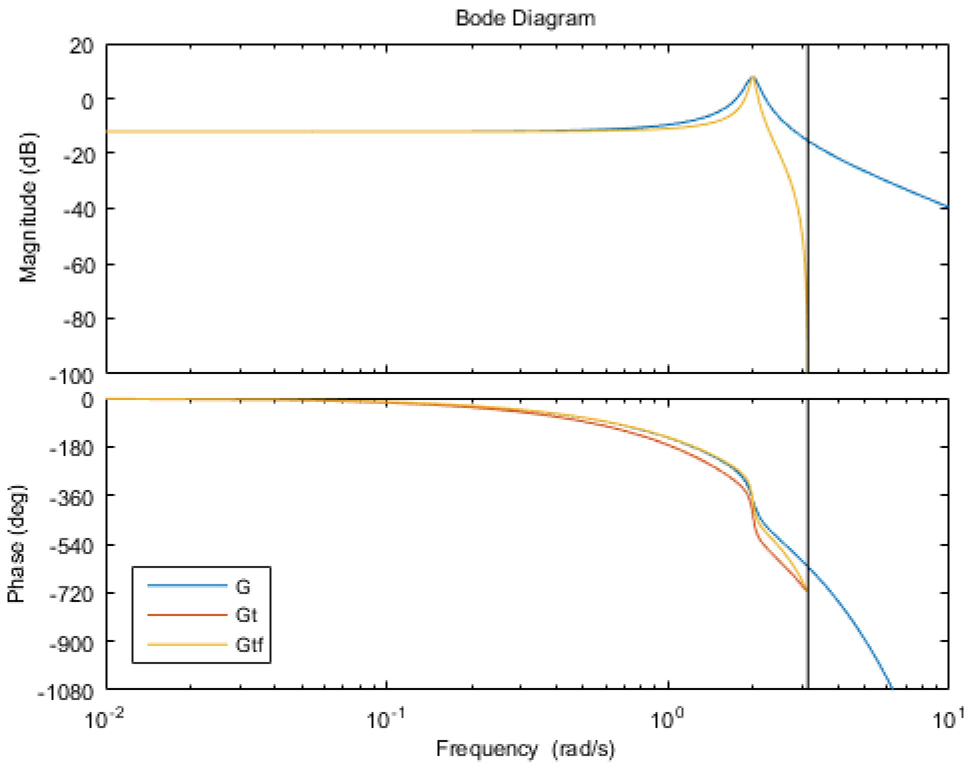
```
discopts.FractDelayApproxOrder = 3;
```

```
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 5-10

More About

- "Continuous-Discrete Conversion Methods" on page 5-23

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.

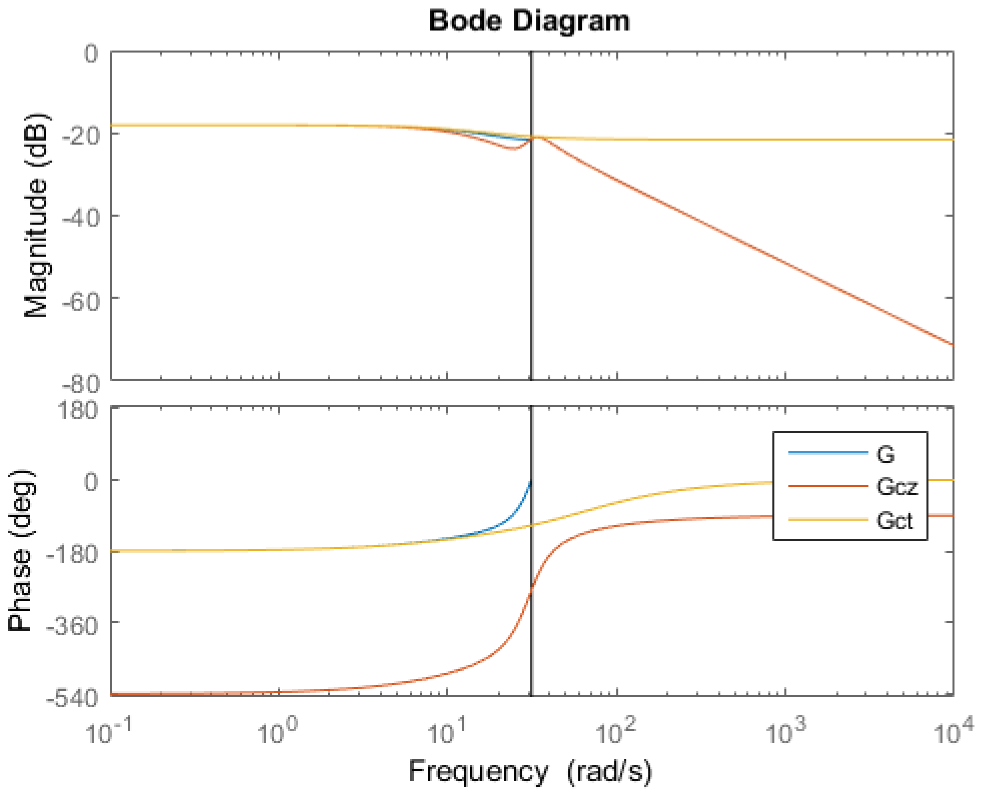
```
Gct = d2c(G,'tustin')
```

```
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')
```



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 5-10

More About

- "Continuous-Discrete Conversion Methods" on page 5-23

Continuous-Discrete Conversion Methods

In this section...

Choosing a Conversion Method

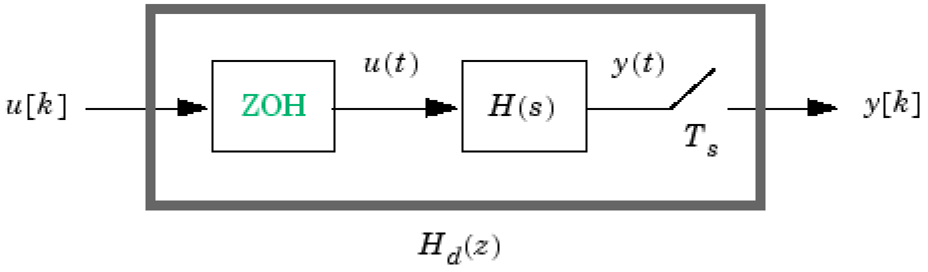
The `c2d` command discretizes continuous-time models. Conversely, `d2c` converts discrete-time models to continuous time. Both commands support several discretization and interpolation methods, as shown in the following table.

Discretization Method	Use when:
"Zero-Order Hold" on page 5-24	You want an exact discretization in the time domain for staircase inputs.
"First-Order Hold" on page 5-25	You want an exact discretization in the time domain for piecewise linear inputs.
"Impulse-Invariant Mapping" on page 5-26 (`c2d` only)	You want an exact discretization in the time domain for impulse train inputs.
"Tustin Approximation" on page 5-27	• 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 5-31	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 $H_d(z)$.

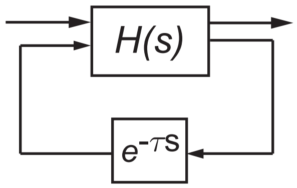
The ZOH discrete-to-continuous conversion has the following limitations:

- `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 5-20 for an example.

ZOH Method for Systems with Time Delays

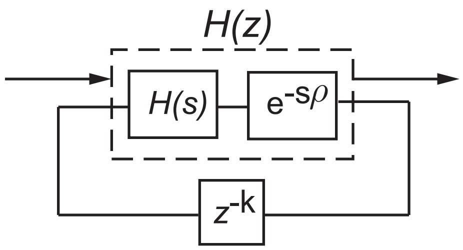
You can use the ZOH method to discretize SISO or MIMO continuous-time models with time delays. The ZOH method yields an exact discretization for systems with input delays, output delays, or transfer delays.

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:

$$u(t) = u[k] + \frac{t - kT_s}{T_s}\left(u[k+1] - u[k]\right), \quad kT_s \le t \le (k+1)T_s$$

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.

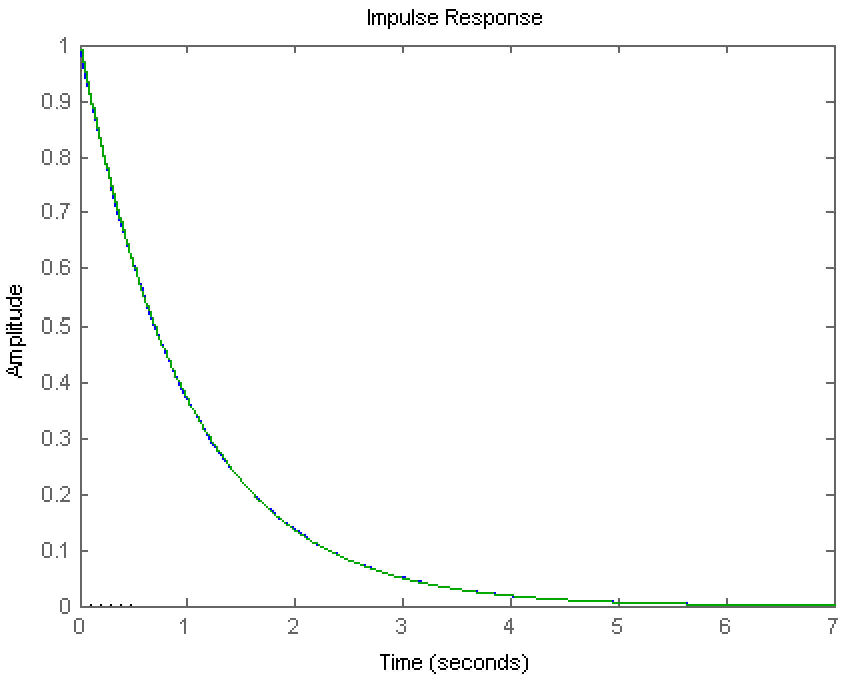
FOH Method for Systems with Time Delays

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 5-25.

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:

$$H(j\omega) = H_d\left(e^{j\omega T_s}\right)$$

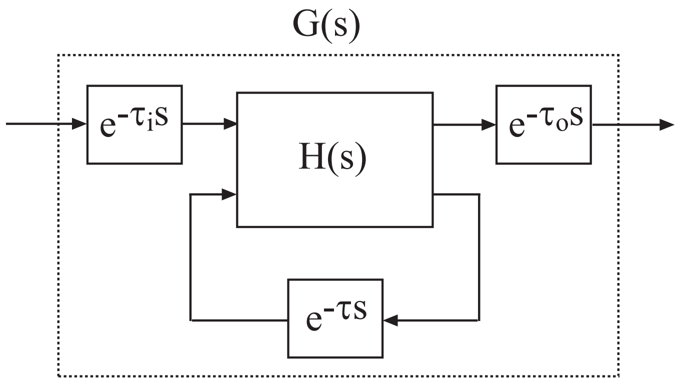
Tustin Approximation for Systems with Time Delays

You can use the Tustin approximation to discretize SISO or MIMO continuous-time models with time delays.

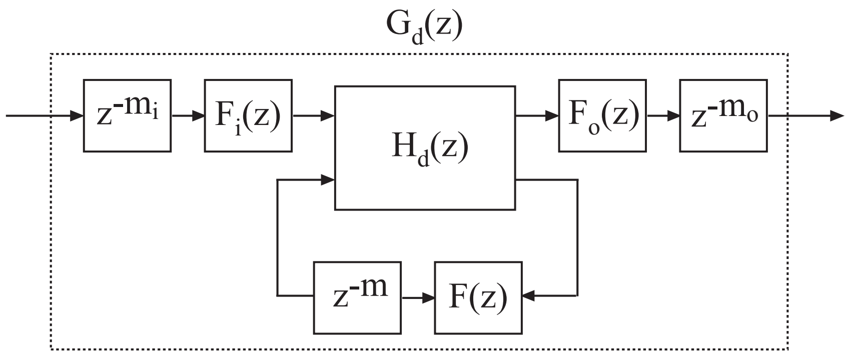
By default, the Tustin method rounds any time delay to the nearest multiple of the sample time. Therefore, for any time delay `tau`, the integer portion of the delay, `k*Ts`, maps to a delay of `k` sampling periods in the discretized model. This approach ignores the residual fractional delay, `tau - 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 5-16 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.



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 = round(τ_i /T_s), m_o = round(τ_o/T_s), and m = round(τ/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:
order($F_i(z)$) = max(ceil(τ_i /T_s), `FractDelayApproxOrder`).

If ceil(τ_i /T_s) < `FractDelayApproxOrder`, the Thiran filter $F_i(z)$ approximates the entire input delay τ_i. If ceil(τ_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 = ceil(τ_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 5-29.

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 5-10
- “Improve Accuracy of Discretized System with Time Delay” on page 5-16
- “Convert Discrete-Time System to Continuous Time” on page 5-20

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 $G(z) = \frac{z + 0.4}{z - 0.7}$ 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)
```

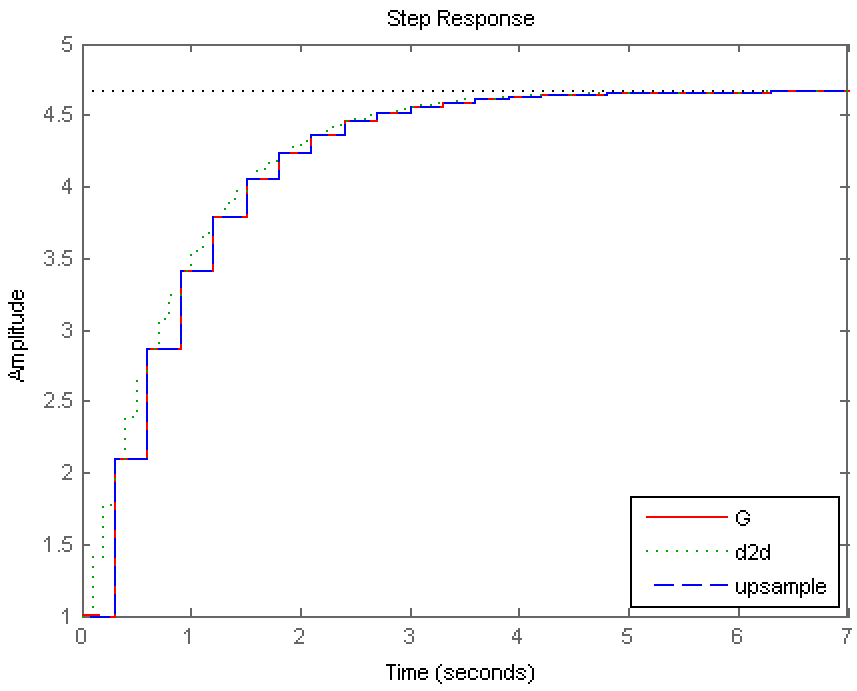
```
G_up =

  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`.

Compare the step responses of the original model G with the resampled models G_d2d and G_up.

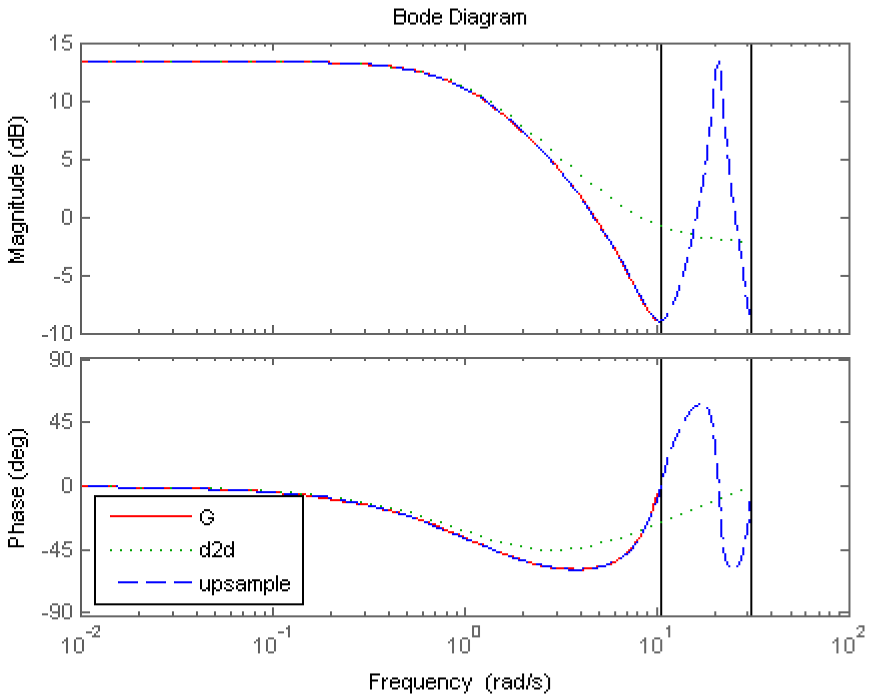
```
step(G,'-r',G_d2d,':g',G_up,'--b')
legend('G','d2d','upsample','Location','SouthEast')
```



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 5-36

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 5-33

Why Simplify Models?

Working with simpler models result in faster and more reliable computations than higher-order models. Simpler models are also easier to understand and manipulate. Therefore, it can be useful to reduce model order while preserving model characteristics that are important to your application.

Some cases where you might want to reduce model order include:

- You are working with a relatively high-order model obtained from linearizing a Simulink model, performing a finite-element calculation, identification with System Identification Toolbox software, or other source.
- You design a high-order controller that you wish 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 might reduce the plant order before synthesis, reduce the controller order after synthesis, or both.
- 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.

In general, when you reduce model order, you want to preserve model characteristics that are important for your application. For example, for control design, you should verify that the behavior of the reduced order model adequately matches the behavior of the original model where the open-loop gain is close to 1 (in the gain crossover region).

Related Examples

- "Approximate Model with Lower-Order Model" on page 5-40
- "Approximate Model with Unstable or Near-Unstable Pole" on page 5-50
- "Eliminate States by Pole-Zero Cancellation" on page 5-54

More About

- "Cancellation Versus Approximation" on page 5-38

Cancellation Versus Approximation

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	Commands
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. Eliminating structurally disconnected states is a good first step in model reduction because the process does not involve any numerical computation. • `minreal` — Eliminate canceling or near-canceling pole-zero pairs from transfer functions. Eliminate unobservable or uncontrollable states from state-space models.
Approximation — compute a lower-order approximation	`balred` — Compute a lower-order approximation of your model by neglecting states that have relatively low effect on the overall model response

In some cases, approximation can yield better results, even if the model looks like a good candidate for simplification. For example, models with near pole-zero cancellations may be 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 `bode` or `sigma`. Or, compare the open-loop responses for the original and reduced plant and controller models.

See Also

`balred` | `minreal` | `sminreal`

Related Examples

- “Approximate Model with Lower-Order Model” on page 5-40
- “Approximate Model with Unstable or Near-Unstable Pole” on page 5-50
- “Eliminate States by Pole-Zero Cancellation” on page 5-54

Approximate Model with Lower-Order Model

This example shows how to compute a reduced-order approximation of a model using `balred`.

To compute a reduced-order approximation, you first choose an order for the reduced model by examining the contribution of the various states to the overall model behavior.

Load a high-order model.

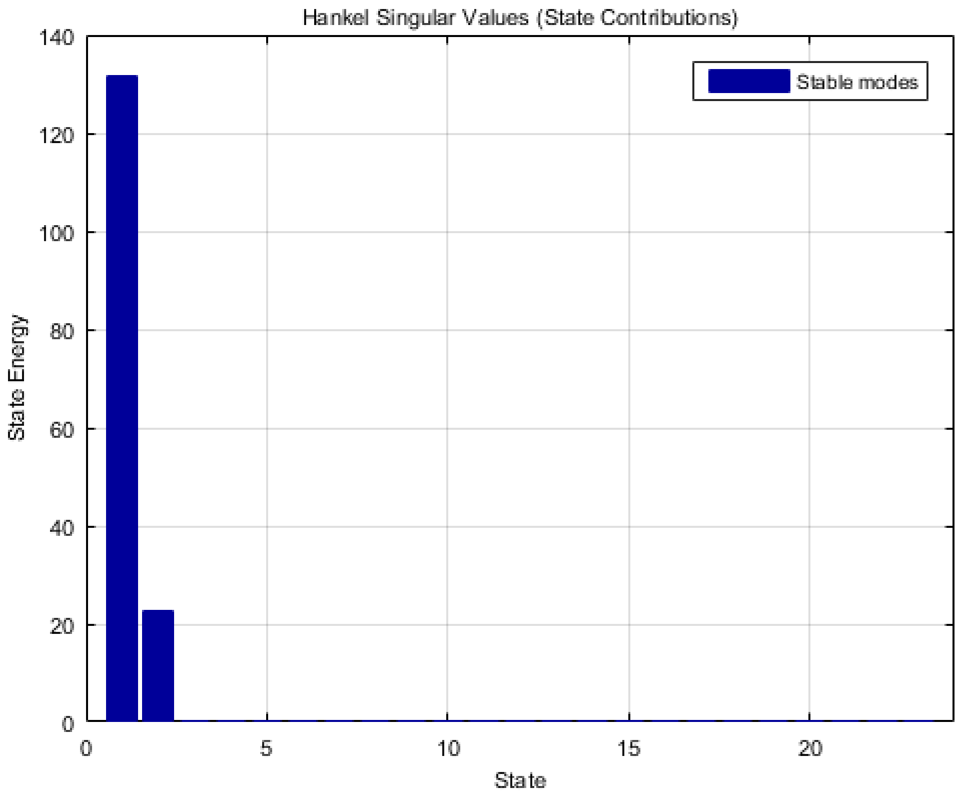
```
load ltiexamples hplant
order(hplant)
```

```
ans =

    23
```

`hplant` is a 23rd-order SISO model. When choosing an order for the reduced system, it is helpful to know how many states make a significant contribution to the overall model behavior.

Examine the relative amount of energy per state 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.

Compute first-order and second-order reduced approximations of `hplant`.

```
opts = balredOptions('StateElimMethod','Truncate');
hplant1 = balred(hplant,1,opts);
hplant2 = balred(hplant,2,opts);
```

The second argument to `balred` specifies the target approximation order.

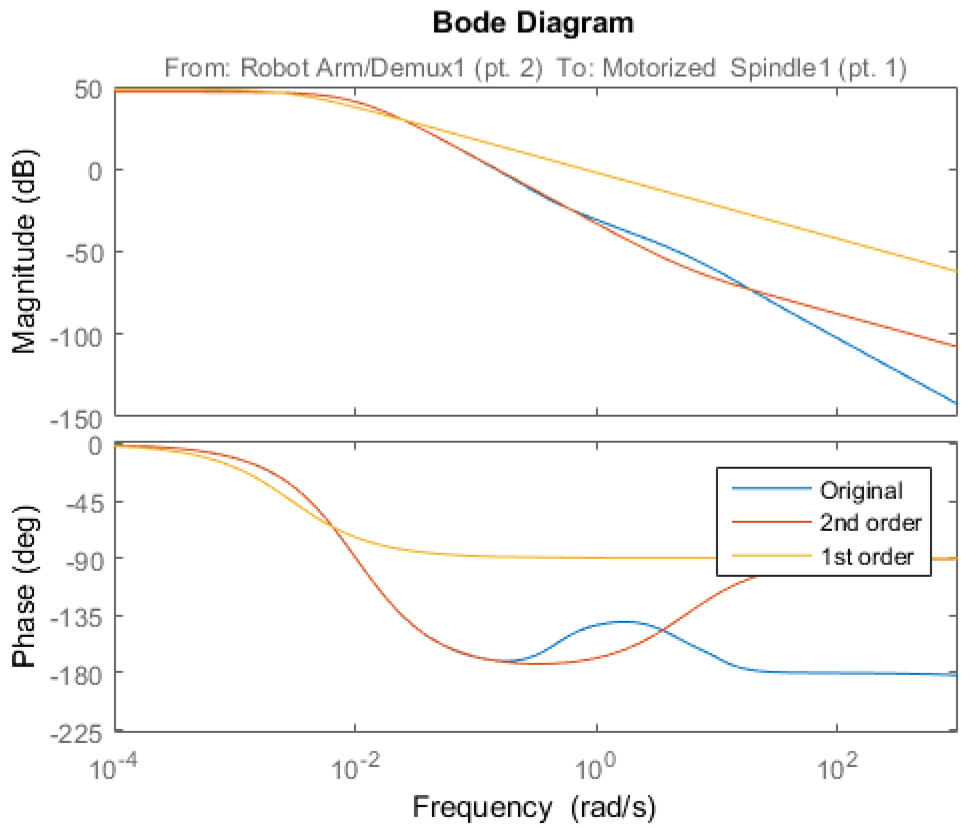
By default, `balred` discards the states with the smallest Hankel singular values, and alters the remaining states to preserve the DC gain of the system. Setting the

`StateElimMethod` option to `Truncate` causes `balred` to discard low-energy states without altering the remaining states.

Compare the frequency responses of the original and approximated systems.

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.

```
bodeplot(hplant,hplant2,hplant1)
legend('Original','2nd order','1st order')
```



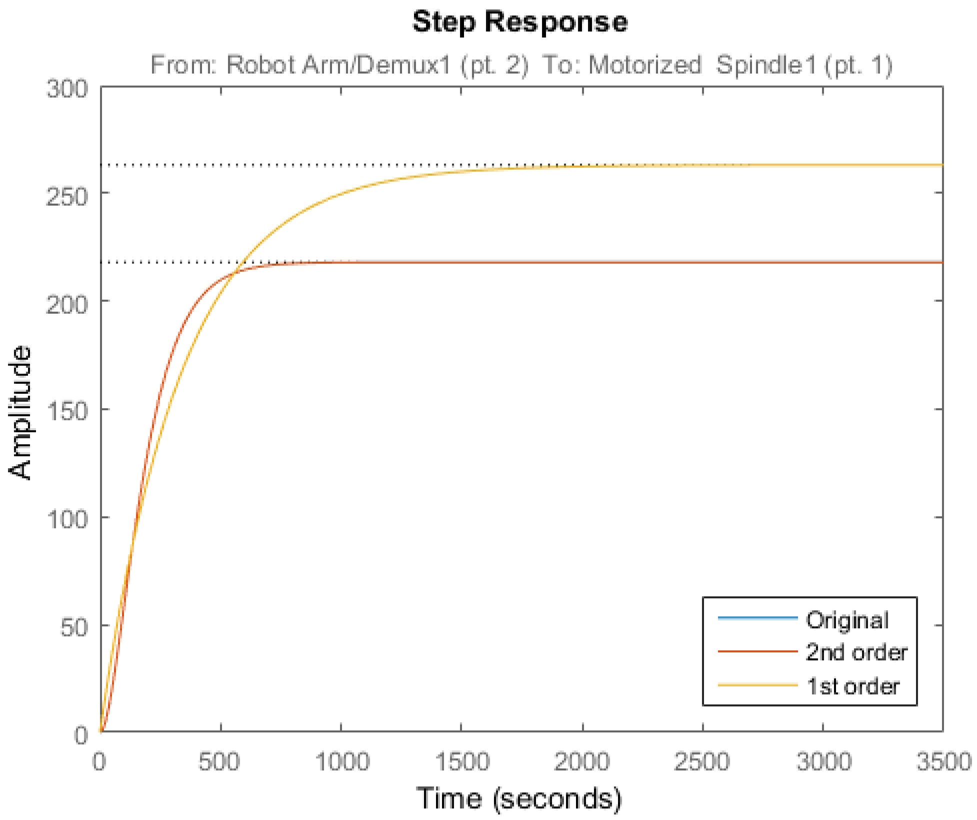
When working with MIMO systems, you can use `sigma` to examine a singular value plot rather than a Bode plot.

In this case, the second-order system 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.

Examine the time-domain responses of the original and reduced-order systems.

```
stepplot(hplant,hplant2,hplant1)
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.

See Also

`balred` | `hsvplot`

Related Examples

- "Approximate Model with Unstable or Near-Unstable Pole" on page 5-50
- "Choose a Low-Order Approximation Method" on page 5-45
- "Eliminate States by Pole-Zero Cancellation" on page 5-54

More About

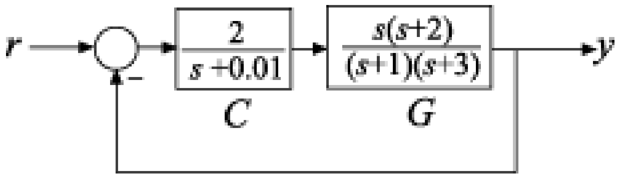
- "Cancellation Versus Approximation" on page 5-38

Choose a Low-Order Approximation Method

This example shows how to compute a low-order approximation in two ways and compares the results.

By default, balred discards the states with that make the smallest contribution to system behavior, altering 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. Which method you choose depends upon what dynamics are 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.

In this example, compare the two methods of computing a reduced-order approximation of the following closed-loop system from *r* to *y*.



Create the closed-loop model 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 near pole-zero cancellation close to $s = 0$. Therefore, it is a good candidate for order reduction by approximation.

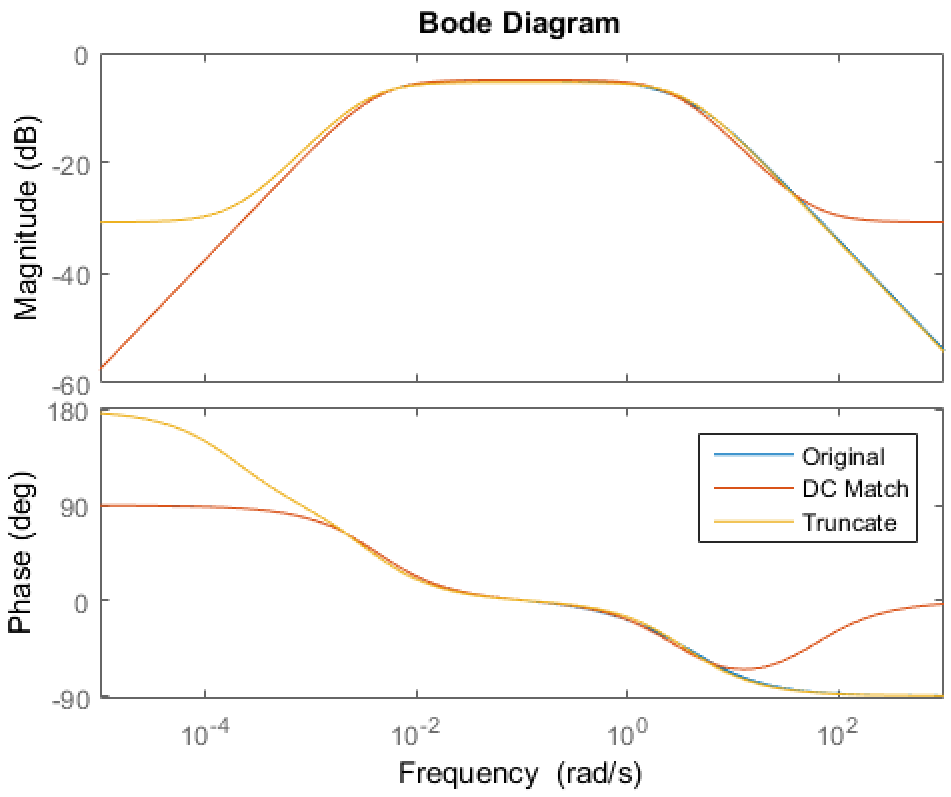
Compute two second-order approximations to T, one that preserves the DC gain and one that truncates the lowest-energy state without changing the other states.

```
matchopt = balredOptions('StateElimMethod','MatchDC');
truncopt = balredOptions('StateElimMethod','Truncate');
Tmatch = balred(T,2,matchopt);
Ttrunc = balred(T,2,truncopt);
```

Use `balredOptions` to specify the approximation method.

Compare the frequency responses of the approximated models.

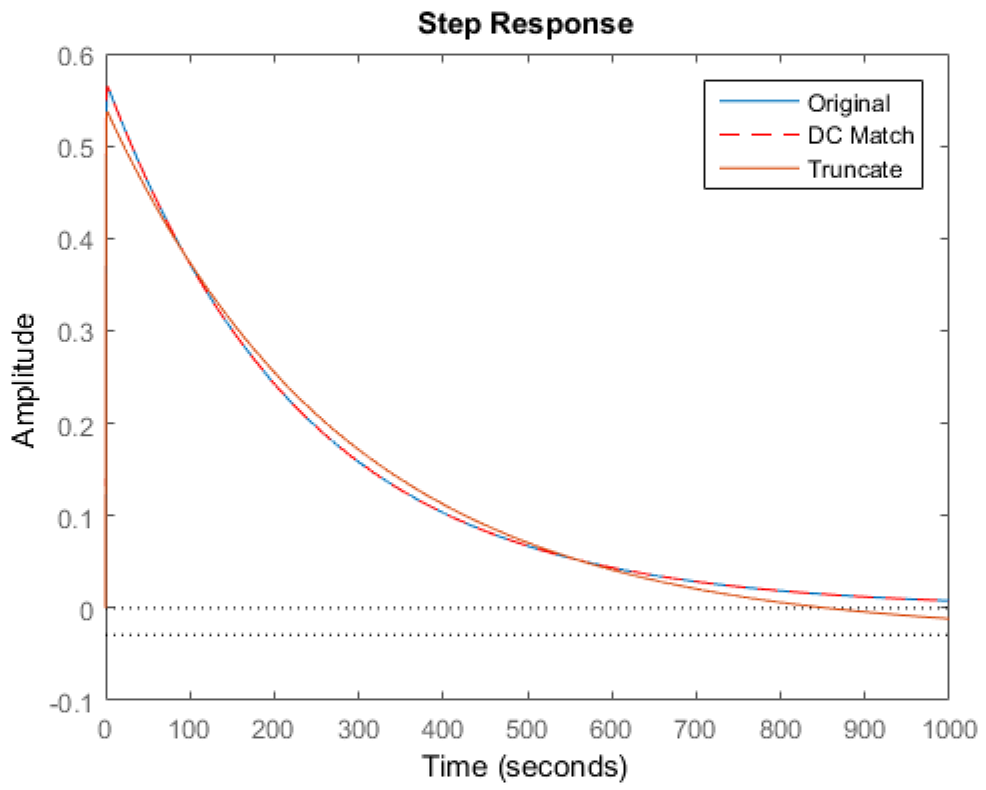
```
bodeplot(T,Tmatch,Ttrunc)
legend('Original','DC Match','Truncate')
```



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.

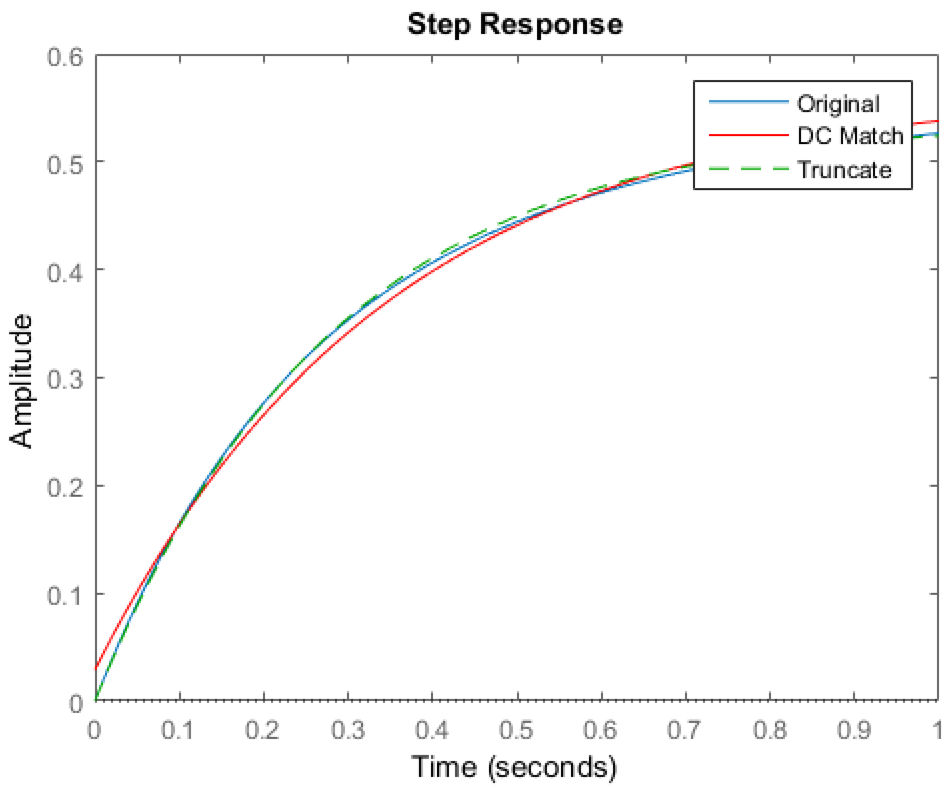
```
stepplot(T,Tmatch,'r--',Ttrunc,1000)
legend('Original','DC Match','Truncate')
```



As expected, on long time scales the DC matched approximation `Tmatch` is a good match to the original model.

Compare the fast transients in the step response.

```
stepplot(T,Tmatch,'r',Ttrunc,'g--',1)
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.

See Also

`balred`

Related Examples

- "Approximate Model with Lower-Order Model" on page 5-40
- "Approximate Model with Unstable or Near-Unstable Pole" on page 5-50
- "Eliminate States by Pole-Zero Cancellation" on page 5-54

More About

- “Cancellation Versus Approximation” on page 5-38

Approximate Model with Unstable or Near-Unstable Pole

This example shows how to compute a reduced-order approximation of a system using `balred` when the system has unstable or near-unstable poles.

`balred` does not eliminate unstable poles when computing a reduced-order approximation, because doing so would fundamentally change the system dynamics. Instead, `balred` first decomposes the model into stable and unstable parts. Then, `balred` reduces the stable part of the model.

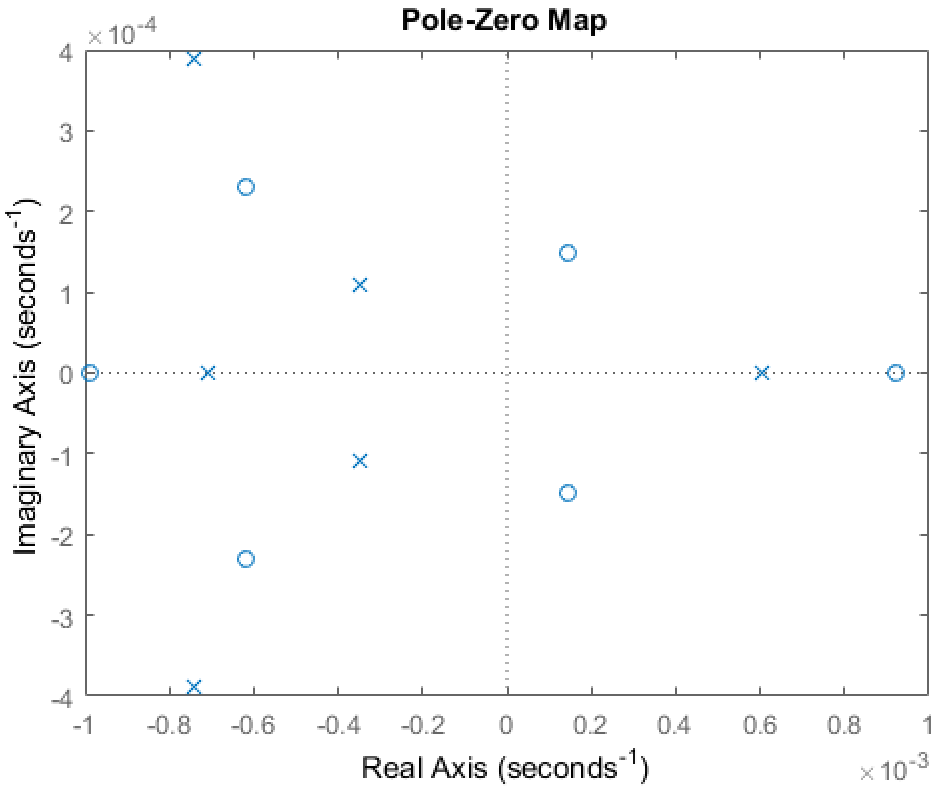
Load a model with unstable poles.

```
load(fullfile(matlabroot,'examples','control','reduce.mat'),'gasf35unst')
```

`gasf35unst` is a 25-state SISO model with two unstable poles ($s > 0$).

Examine the system poles to find near-unstable poles.

```
pzplot(gasf35unst)
axis([-0.001 0.001 -0.0004 0.0004])
```



The pole-zero plot shows several poles (marked by `x`) that are stable, but relatively close to the imaginary axis. In your application, you might want to ensure that `balred` does not discard these near-unstable poles.

To do so, calculate a reduced-order system that preserves the two stable poles within 0.0005 s^{-1} of the imaginary axis.

```
opts = balredOptions('Offset',0.0005);
gasf_arr = balred(gasf35unst,[10 15],opts);
```

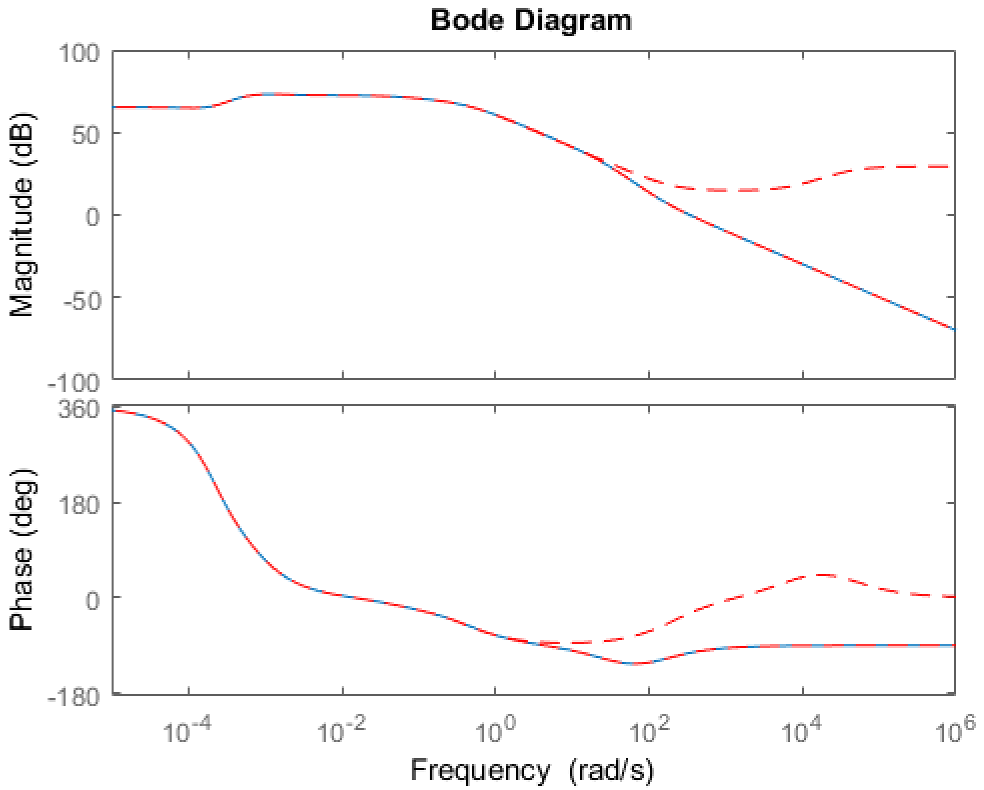
The `Offset` option of `balredOptions` sets the boundary between poles that `balred` can discard, and poles that `balred` must preserve (treat as unstable). Setting `Offset` to

0.0005 preserves the two stable poles of `gasf35unst` that are closest to the imaginary axis.

Providing `balred` an array of target approximation orders `[10 15]` causes `balred` to return an array of approximated models. The array `gasf_arr` is 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 too large to be acceptable. The 15th-order approximation is probably a better choice.

See Also

`balred`

Related Examples

- “Approximate Model with Lower-Order Model” on page 5-40
- “Eliminate States by Pole-Zero Cancellation” on page 5-54

More About

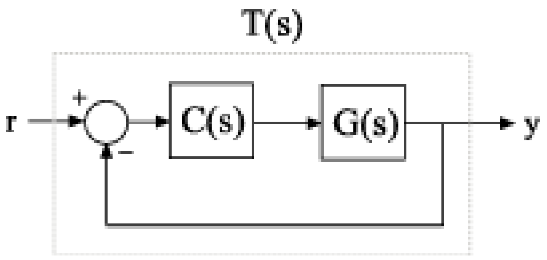
- “Cancellation Versus Approximation” on page 5-38

Eliminate States by Pole-Zero Cancellation

This example shows how to use to reduce the order of a transfer function with cancelling or near-cancelling factors in the numerator and denominator using `minreal`.

Canceling or near-canceling pole-zero pairs can arise from system dynamics, or from building models using model interconnection functions.

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 cancellation of the integrator with the zero near the origin.

```
Tred = minreal(T,1e-7)
```

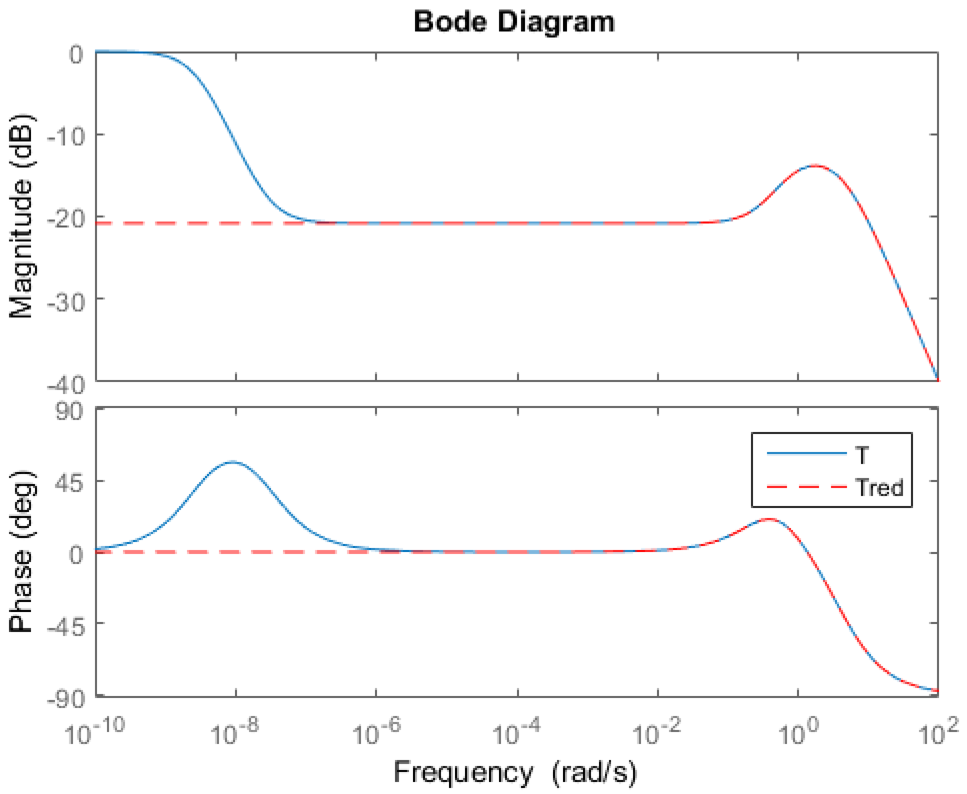
```
Tred =

        (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 exactly match, 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`

Related Examples

- “Approximate Model with Lower-Order Model” on page 5-40

- “Approximate Model with Unstable or Near-Unstable Pole” on page 5-50
- “Choose a Low-Order Approximation Method” on page 5-45

More About

- “Cancellation Versus Approximation” on page 5-38

Linear Analysis

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

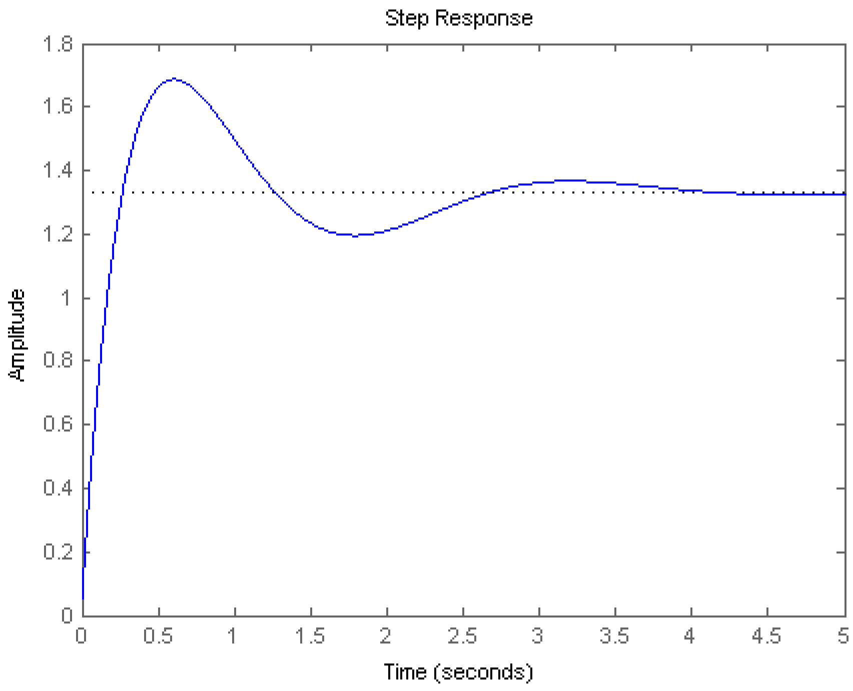
- "Time-Domain Response Data and Plots" on page 6-3
- "Joint Time- and Frequency-Domain Analysis" on page 6-20

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.

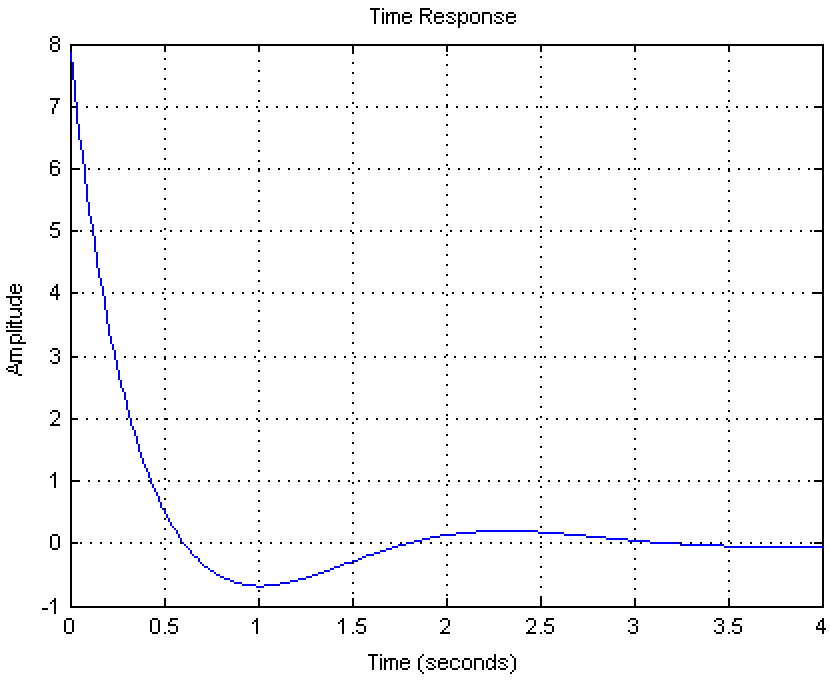
Calculate the step response data from $t = 0$ (application of the step input) to $t = 8$ s.

```
[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 6-6
- “Time-Domain Responses of Multiple Models” on page 6-16
- “Joint Time- and Frequency-Domain Analysis” on page 6-20

More About

- “Choosing a Time-Domain Analysis Command” on page 6-2

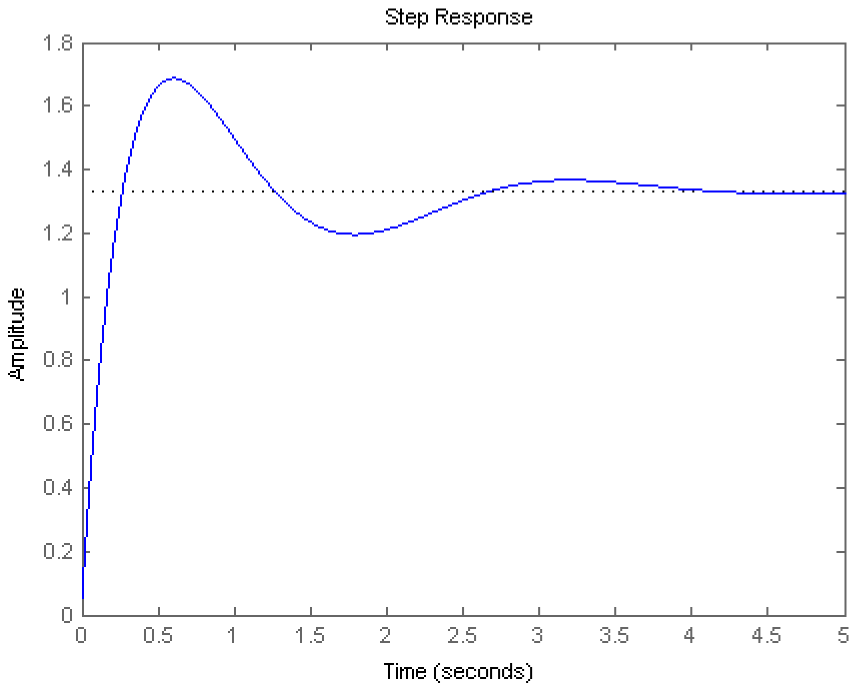
System Characteristics on Response Plots

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.

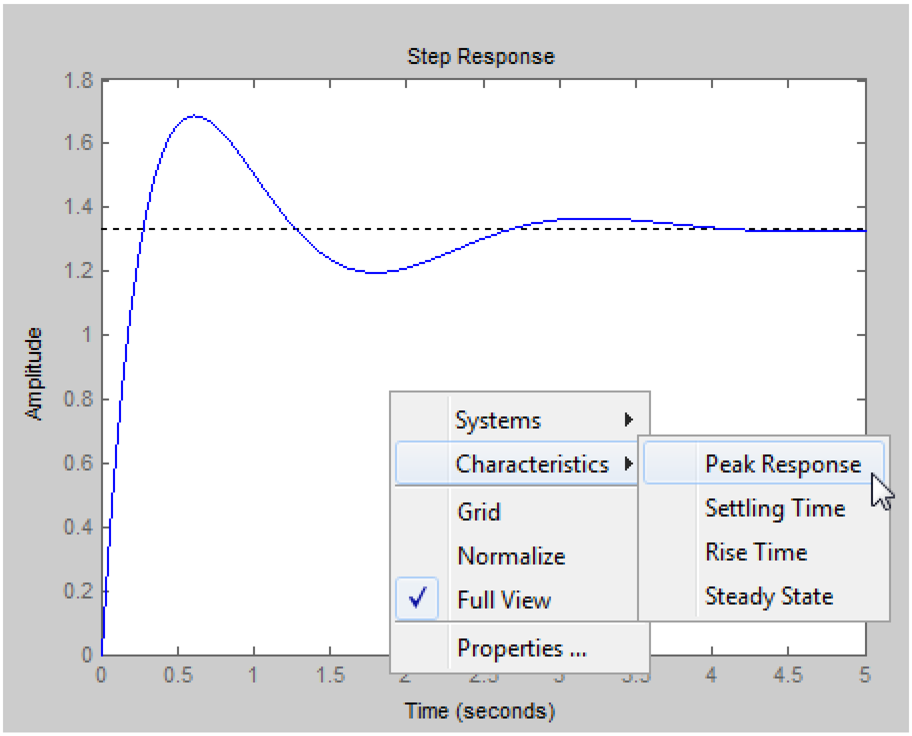
Create a transfer function model and plot its response to a step input at $t = 0$.

```
H = tf([8 18 32],[1 6 14 24]);
stepplot(H)
```

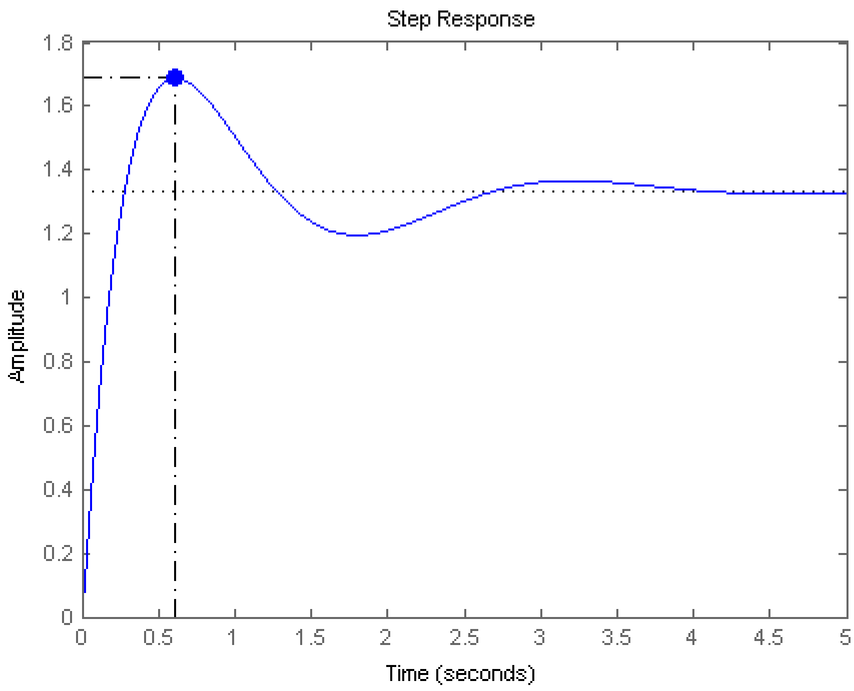


Display the peak response on the plot.

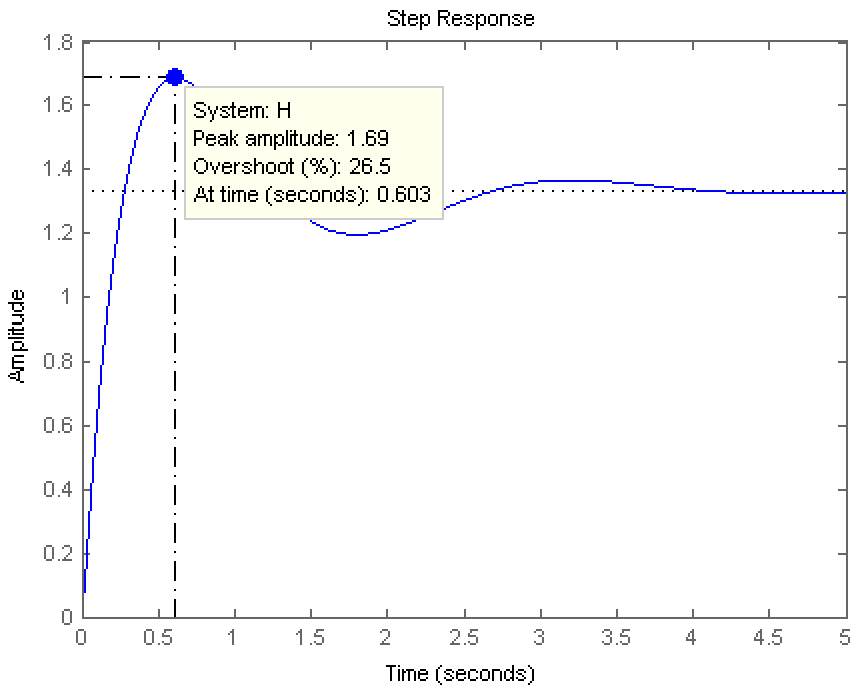
Right-click anywhere in the figure and select **Characteristics** > **Peak Response** from the menu.



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 6-10
- "Joint Time- and Frequency-Domain Analysis" on page 6-20

More About

- "Choosing a Time-Domain Analysis Command" on page 6-2

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 6-6
- “Joint Time- and Frequency-Domain Analysis” on page 6-20

More About

- “Choosing a Time-Domain Analysis Command” on page 6-2

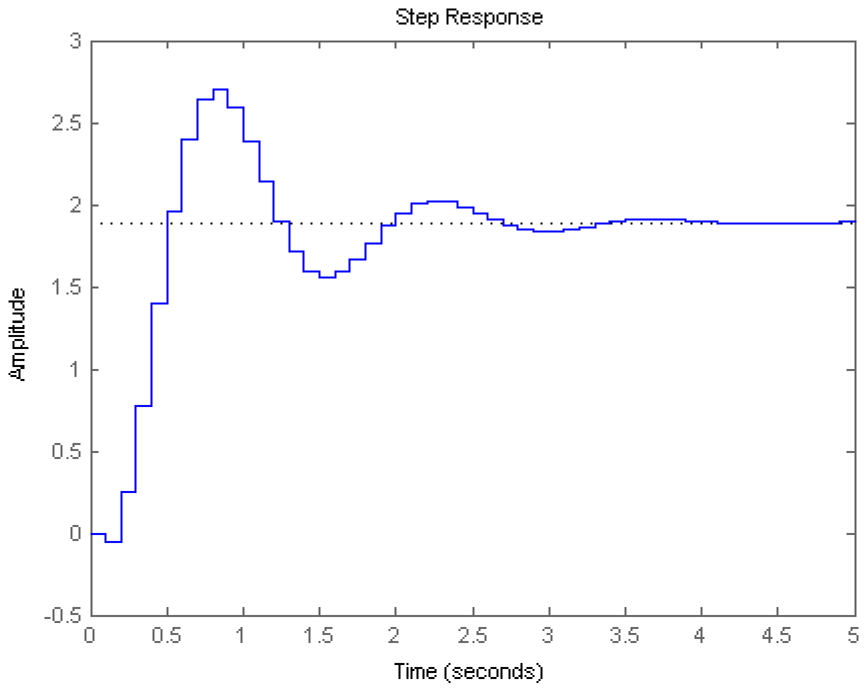
Time-Domain Responses of Discrete-Time Model

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 6-14
- "Time-Domain Responses of Multiple Models" on page 6-16
- "Joint Time- and Frequency-Domain Analysis" on page 6-20

More About

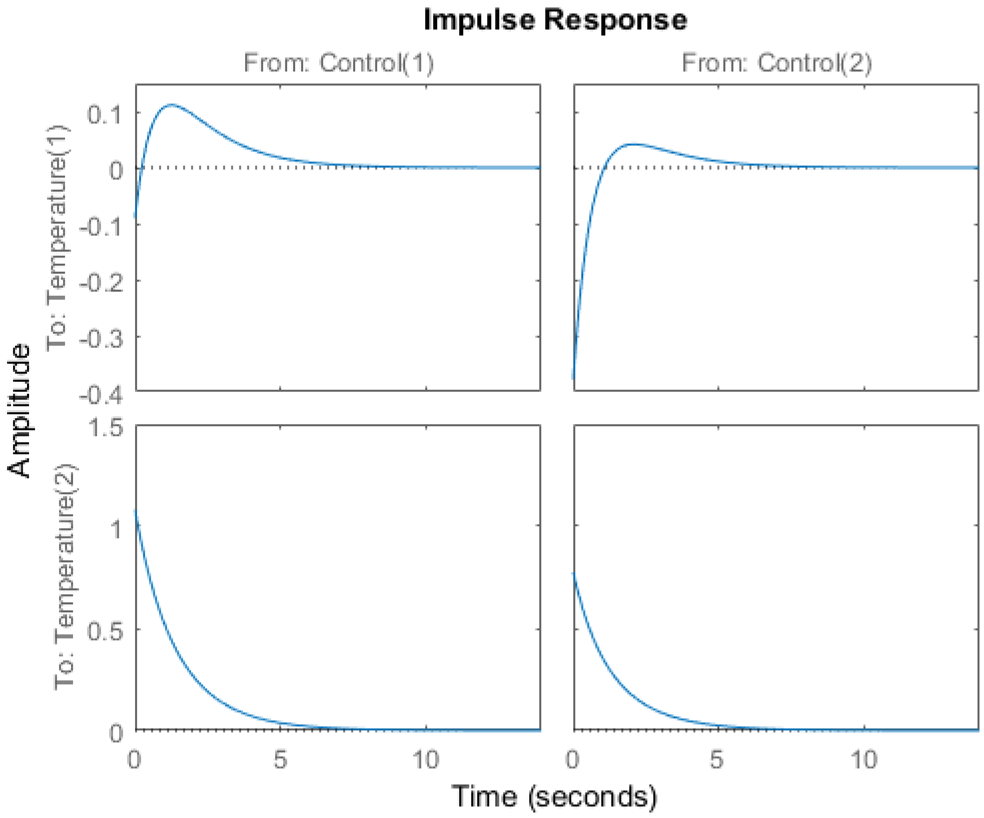
- "Choosing a Time-Domain Analysis Command" on page 6-2

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 6-16
- "Joint Time- and Frequency-Domain Analysis" on page 6-20

More About

- "Choosing a Time-Domain Analysis Command" on page 6-2

Time-Domain Responses of Multiple Models

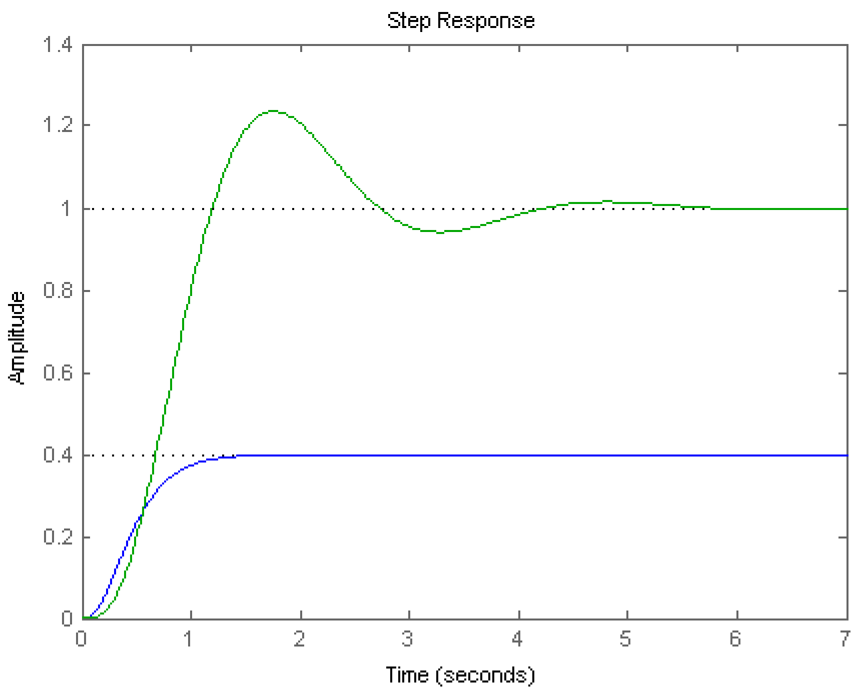
This example shows how to compare the step responses of multiple models on a single plot using `step`. This example compares the step response of an uncontrolled plant to the closed-loop step response of the plant with two different PI controllers.

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.

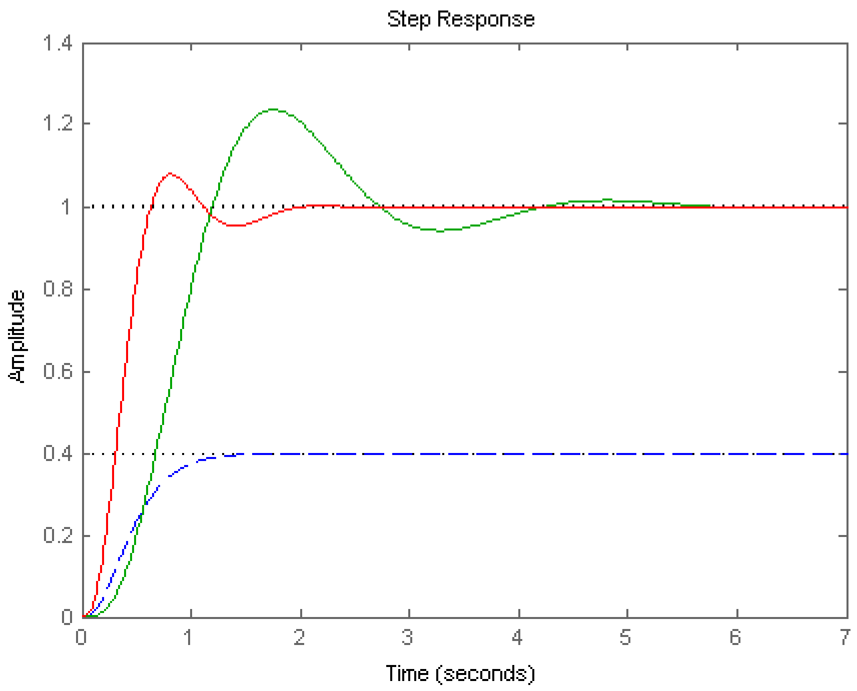
```
G = zpk([],[-5 -5 -10],100);

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 6-14
- "Joint Time- and Frequency-Domain Analysis" on page 6-20

More About

- "Choosing a Time-Domain Analysis Command" on page 6-2

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.

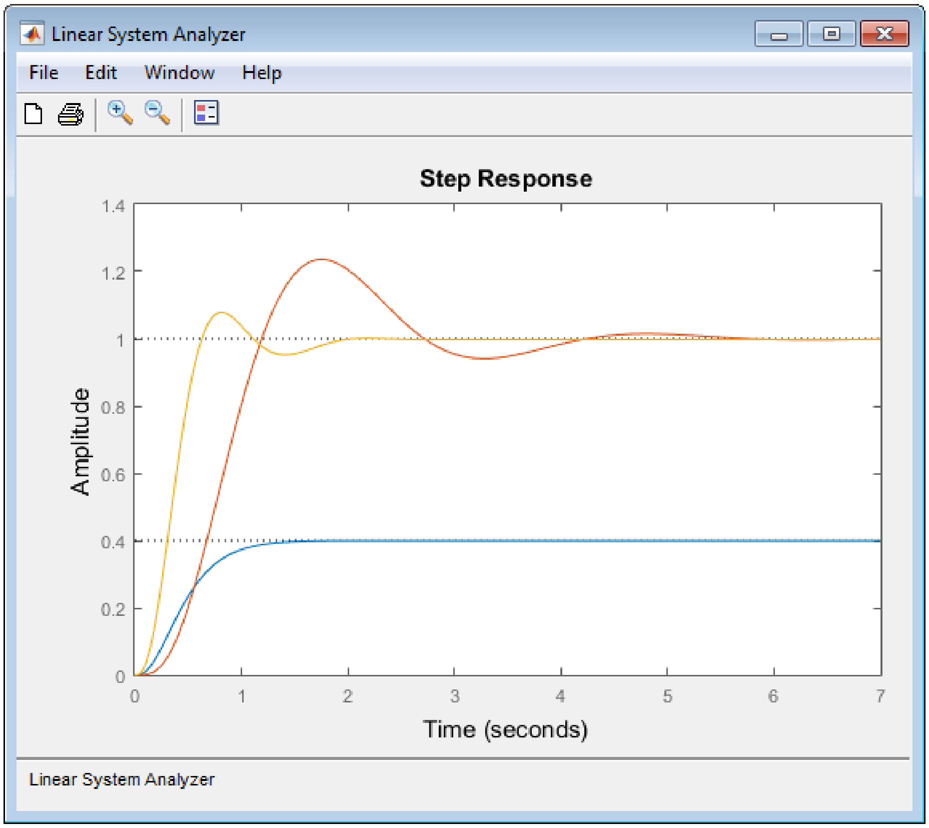
Obtain models whose responses you want to compare.

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.

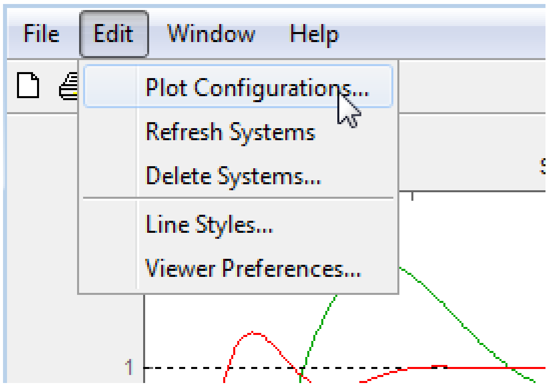
```
linearSystemAnalyzer(G,T1,T2)
```



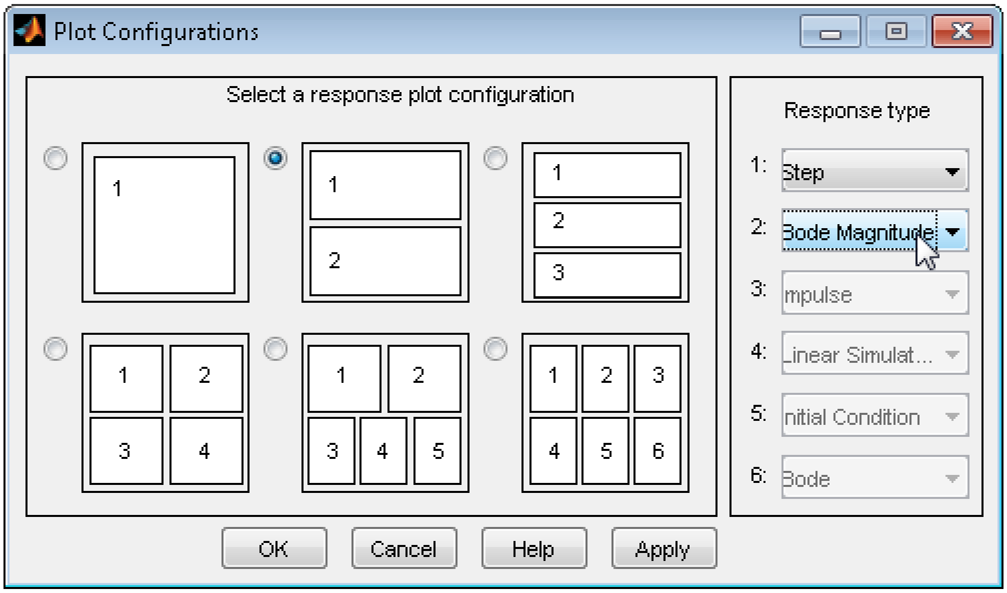
By default, the Linear System Analyzer launches with a plot of the step response of the three systems. Click to add a legend to the plot.

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.



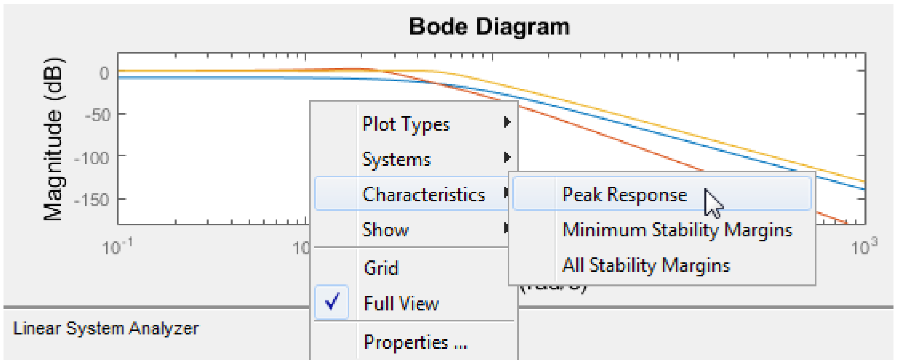
Select the two-plot configuration. In the Response Type area, select `Bode Magnitude` for the second plot type.



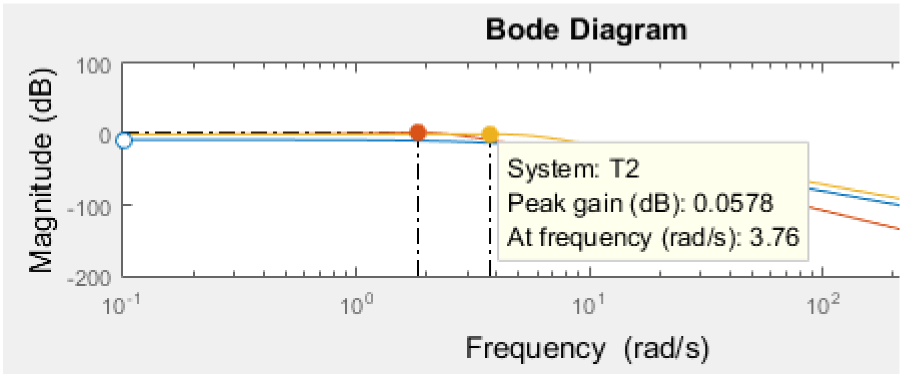
Click **OK** to add the Bode plots to the Linear System Analyzer display.

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.



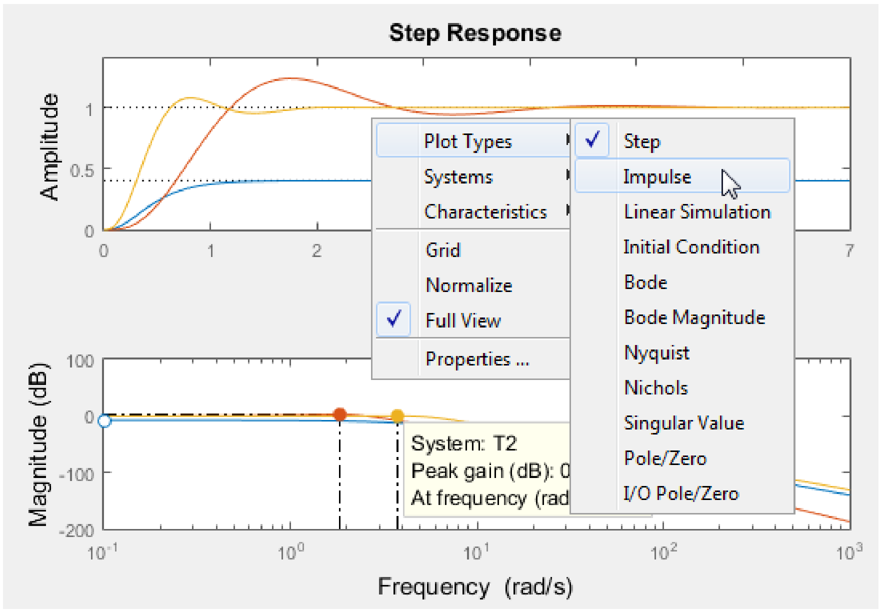
Markers appear on the plot indicating the peak response values. Horizontal and vertical dotted lines indicate the frequency and amplitude of those responses. Click on a marker to view the value of the peak response in a datatip.



System: T2
Peak gain (dB): 0.0578
At frequency (rad/s): 3.76

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 6-16

More About

- "Choosing a Time-Domain Analysis Command" on page 6-2

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.

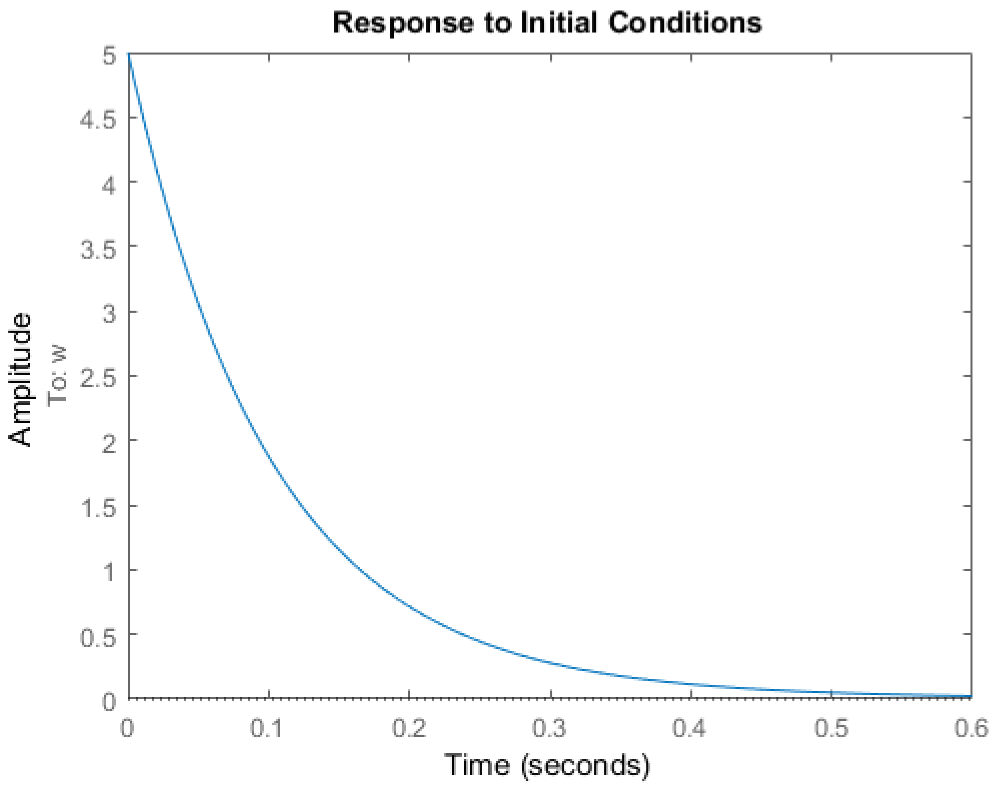
```
load ltiexamples sys_dc
sys_dc.InputName = 'Volts';
sys_dc.OutputName = 'w';
sys_dc.StateName = {'Current','w'};
sys_dc
```

```
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)
```



`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 6-3
- "Numeric Values of Time-Domain System Characteristics" on page 6-10

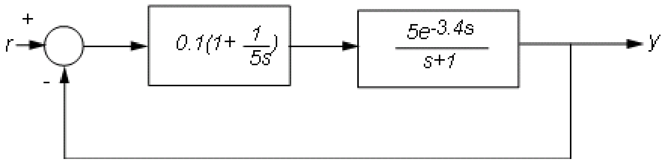
More About

- "Choosing a Time-Domain Analysis Command" on page 6-2

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:



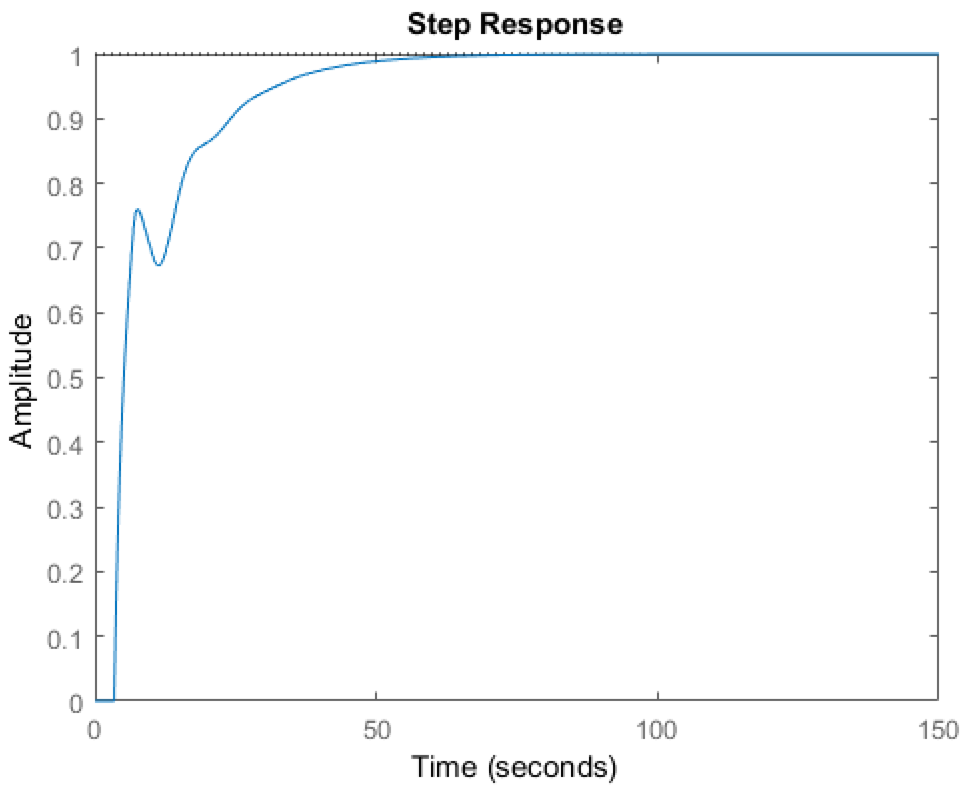
You can model the closed-loop system from `r` to `y` with the following commands:

```
s = tf('s');
P = 5*exp(-3.4*s)/(s+1);
C = 0.1 * (1 + 1/(5*s));
T = feedback(P*C,1);
```

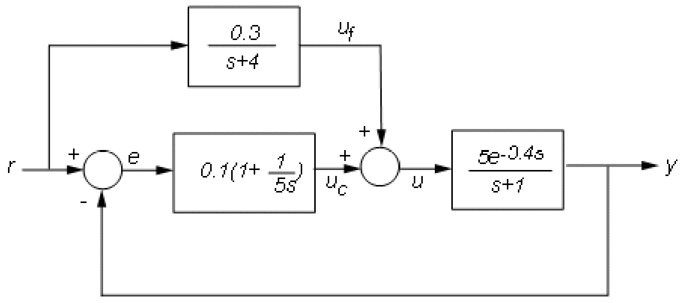
`T` is a state-space model with an internal delay. For more information about models with internal delays, see "Closing Feedback Loops with Time Delays" on page 2-46.

Plot the step response of `T`:

```
stepplot(T)
```



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.

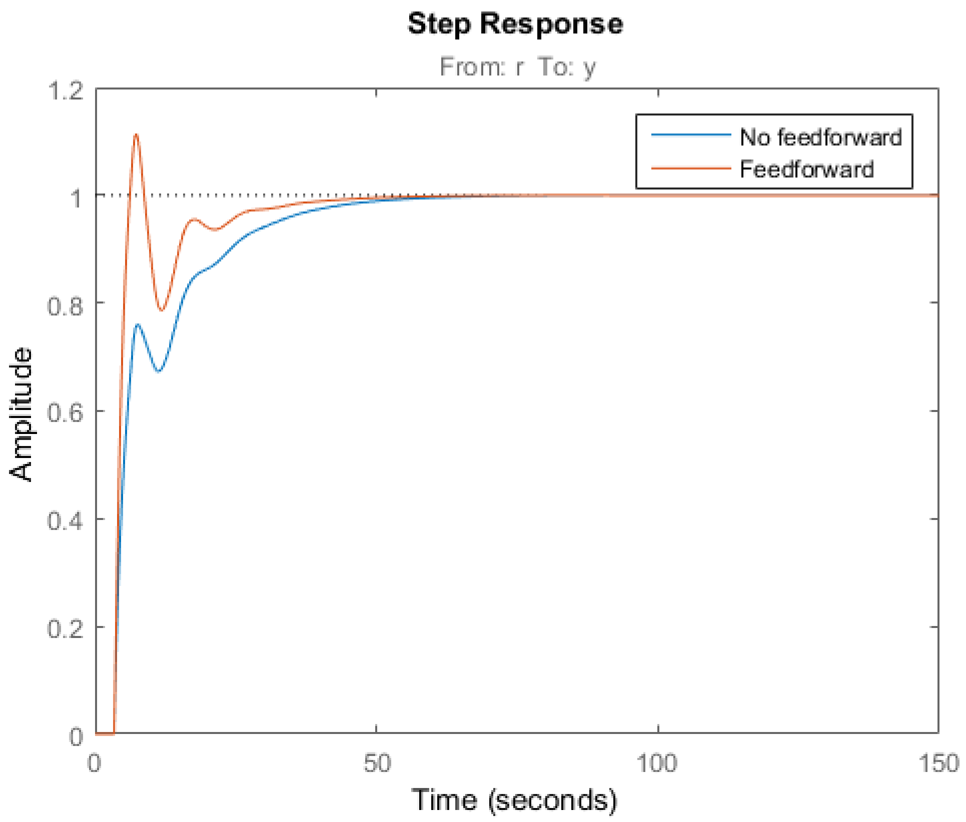


You can derive the corresponding closed-loop model `Tff` by

```
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.

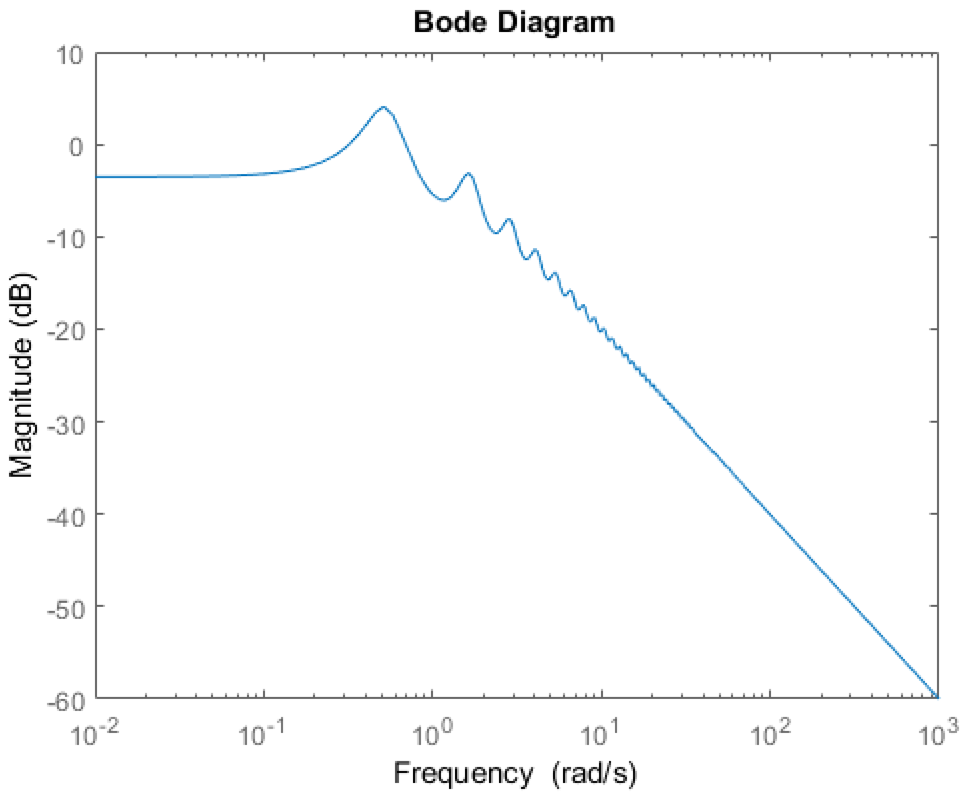
Considerations to Keep in Mind when Analyzing Systems with Internal Time Delays

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:

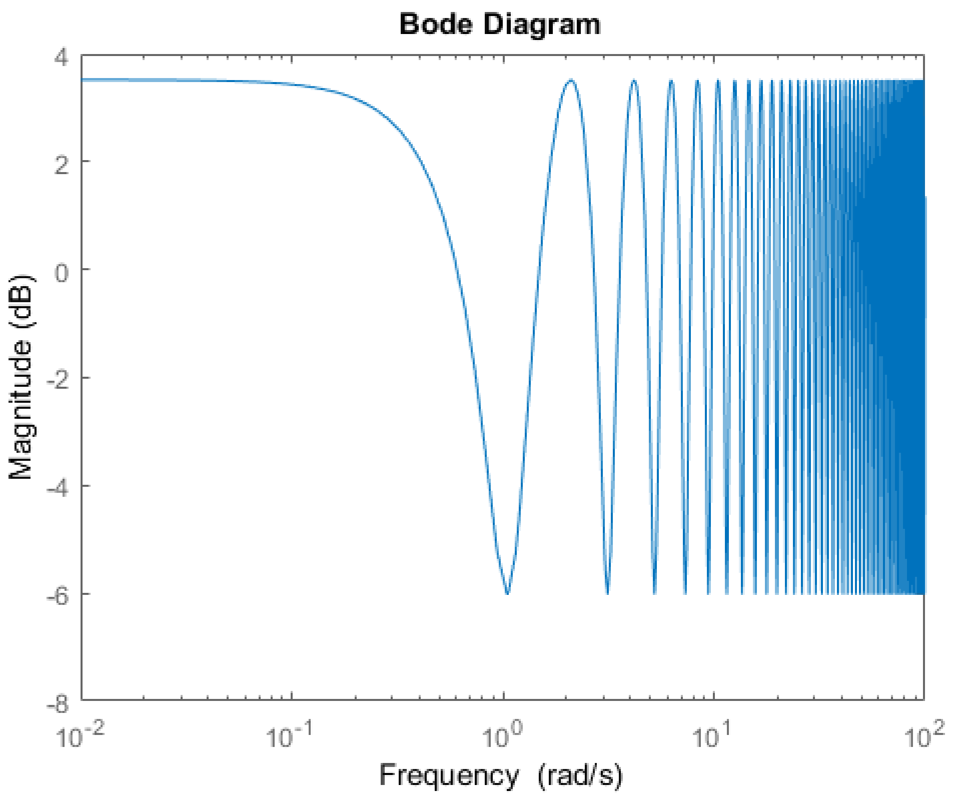
```
s = tf('s');
```

```
G = exp(-5*s)/(s+1);
T = feedback(G,.5);
bodemag(T)
```



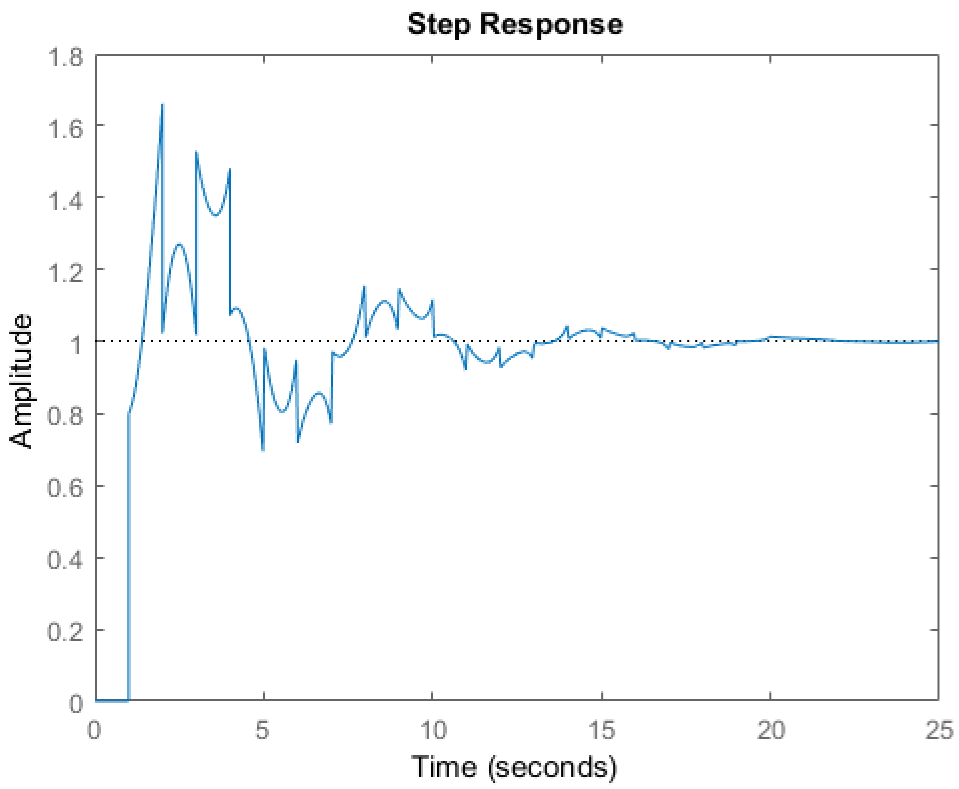
Gain oscillations:

```
G = 1 + 0.5 * exp(-3*s);
bodemag(G)
```



Jagged step response:

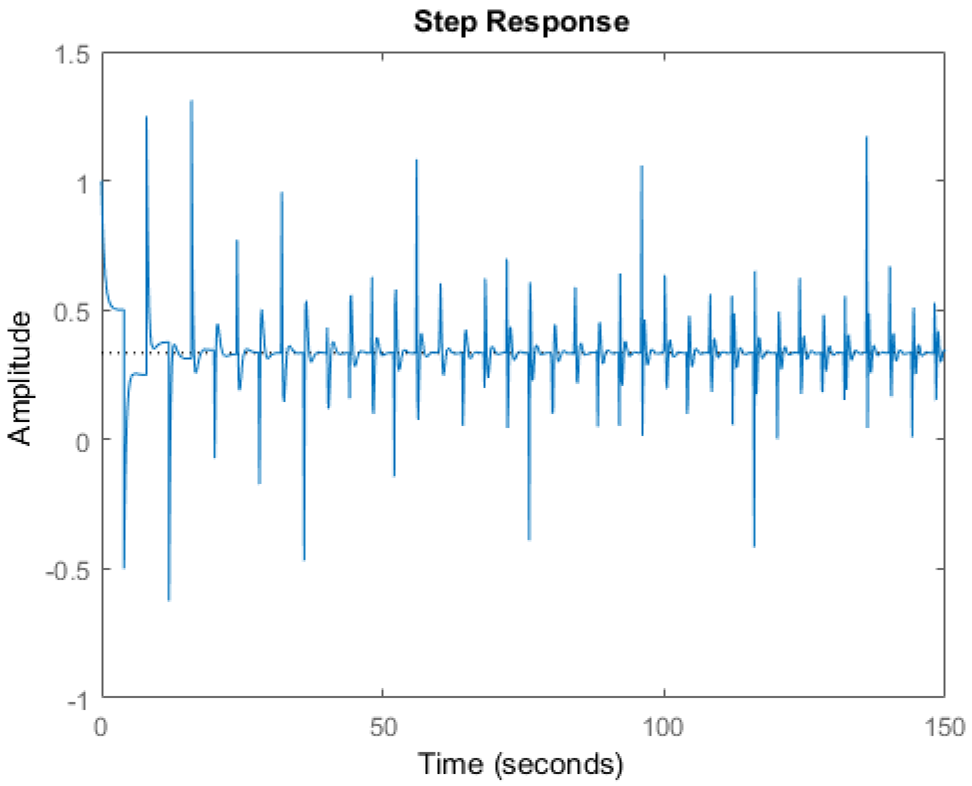
```
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(G,1);
stepplot(T)
```



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

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

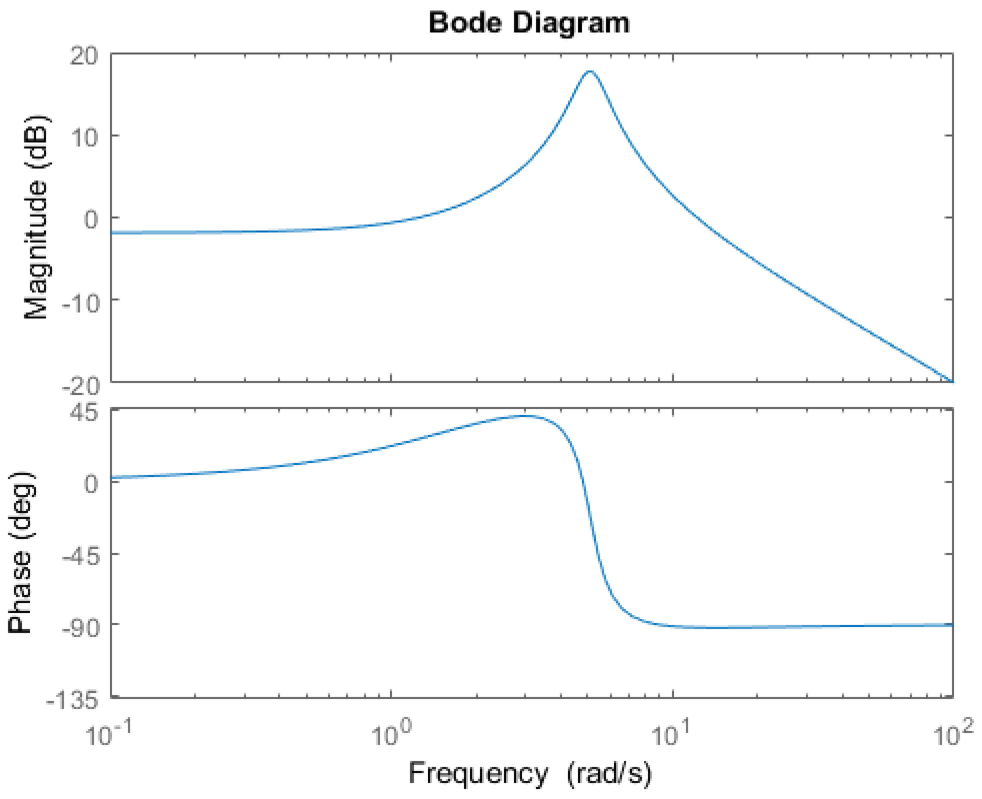
- “Frequency Response of a SISO System” on page 7-4
- “Frequency Response of a MIMO System” on page 7-6
- “Joint Time- and Frequency-Domain Analysis” on page 6-20

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 7-6
- “Numeric Values of Frequency-Domain Characteristics of SISO Model” on page 7-13

More About

- “Choosing a Frequency-Domain Analysis Command” on page 7-2

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.

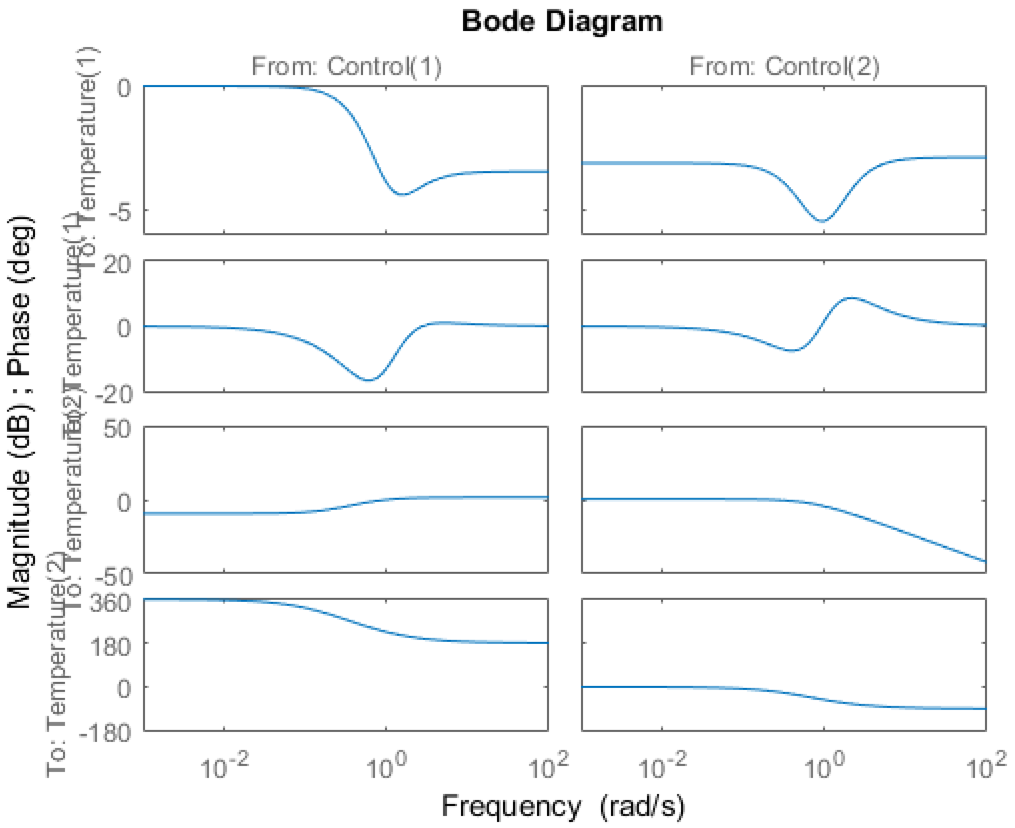
Calculate the frequency response of a MIMO model and examine the size of the output.

```
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`.

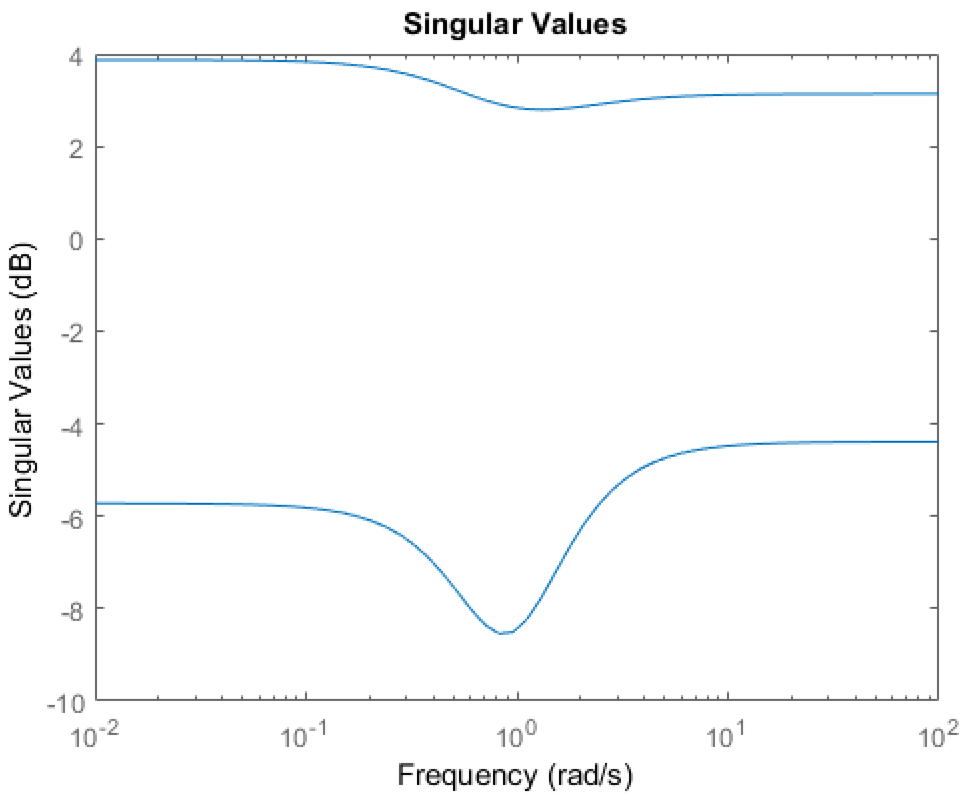
```
bodeplot(H)
```



`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 7-13
- "Joint Time- and Frequency-Domain Analysis" on page 6-20

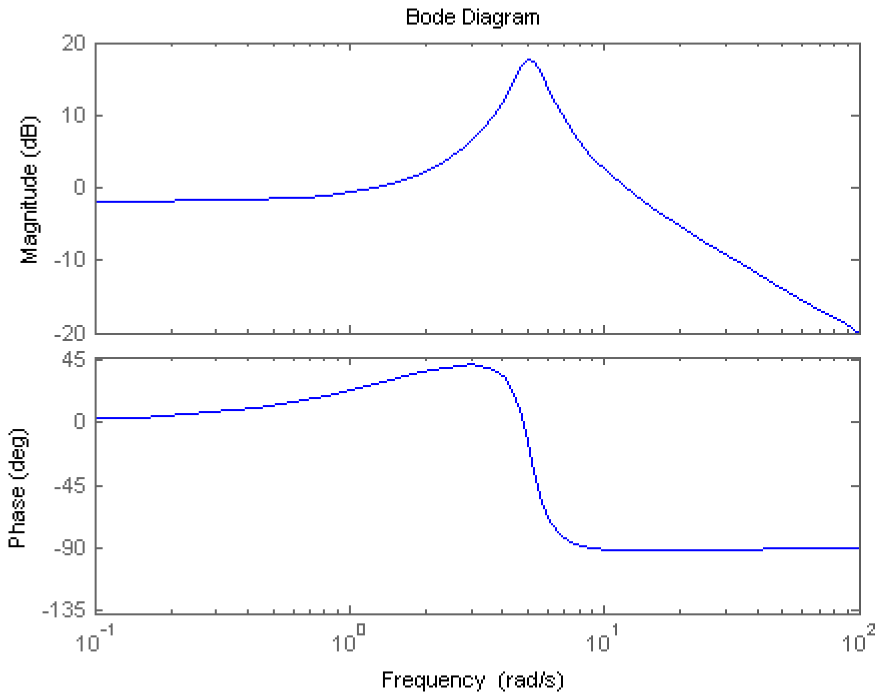
System Characteristics on Response Plots

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.

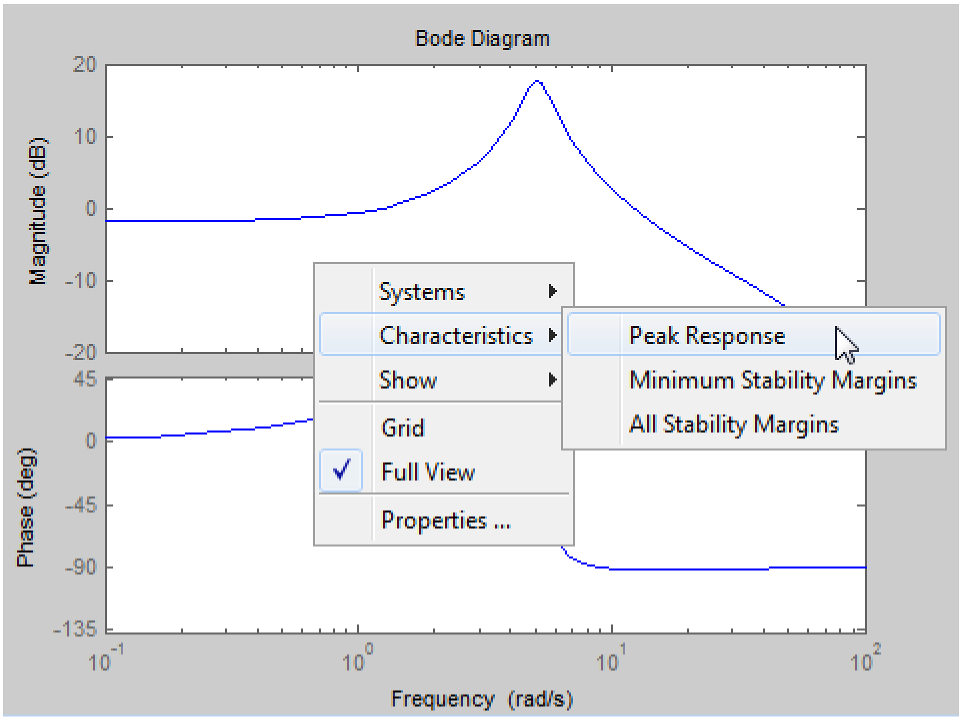
Create a transfer function model and plot its frequency response.

```
H = tf([10,21],[1,1.4,26]);
bodeplot(H)
```

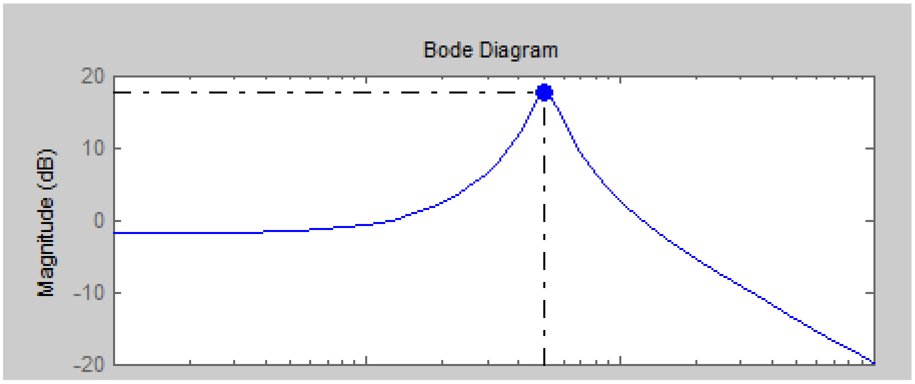


Display the peak response on the plot.

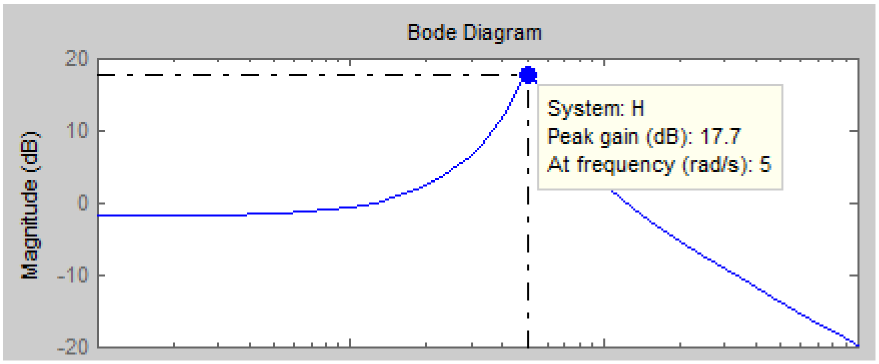
Right-click anywhere in the figure and select **Characteristics** > **Peak Response** from the menu.



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

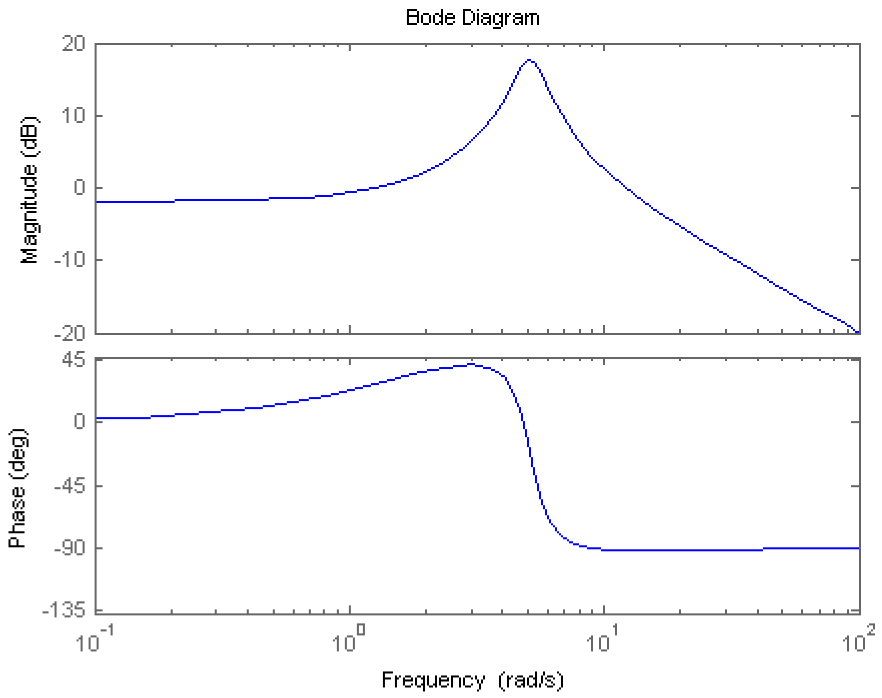
- "Numeric Values of Frequency-Domain Characteristics of SISO Model" on page 7-13
- "Joint Time- and Frequency-Domain Analysis" on page 6-20
- "Pole and Zero Locations" on page 7-16

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 7-16

More About

- “Choosing a Frequency-Domain Analysis Command” on page 7-2

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.

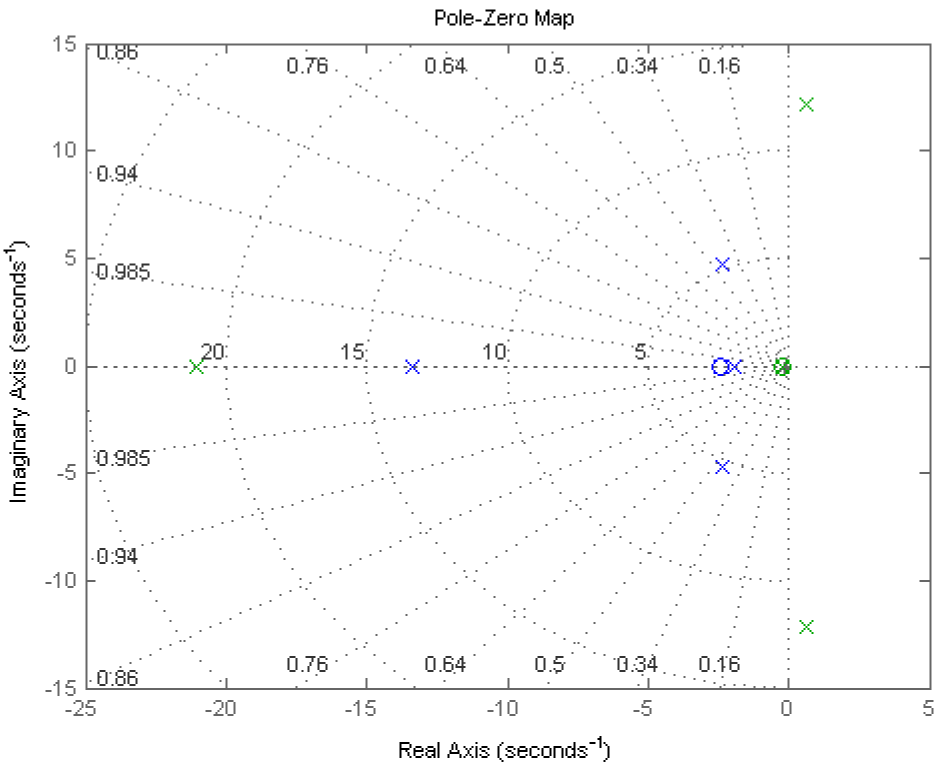
Create dynamic system models representing the two closed-loop systems.

```
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
```



`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);
```

```
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 7-13

More About

- “Choosing a Frequency-Domain Analysis Command” on page 7-2

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:

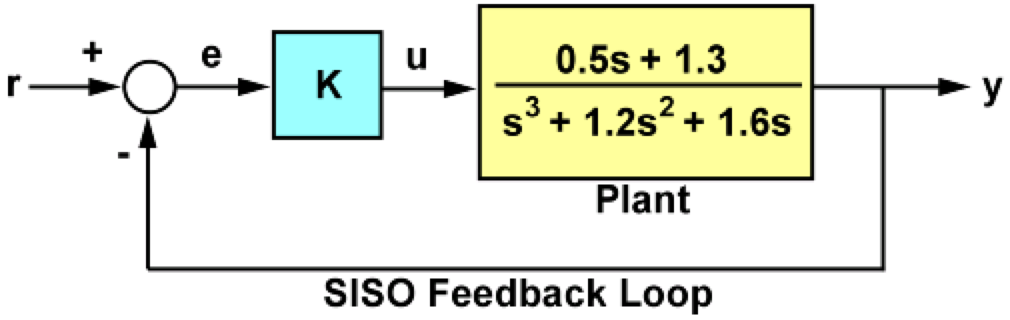


Figure 1: SISO Feedback Loop.

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)
```

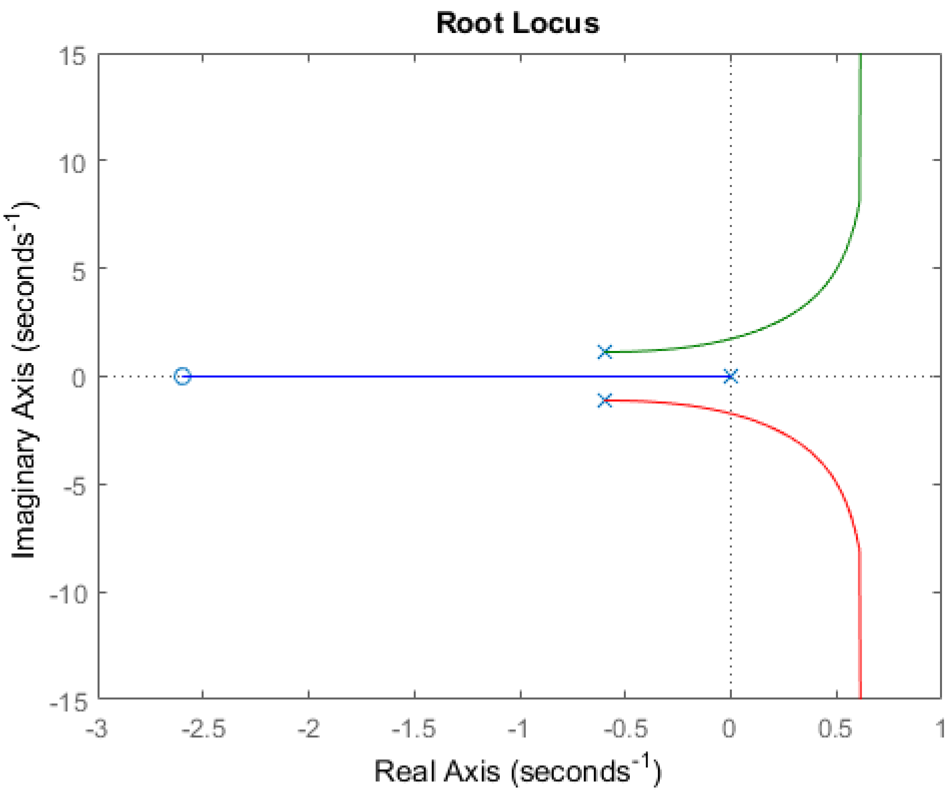
```
ans =

  -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.

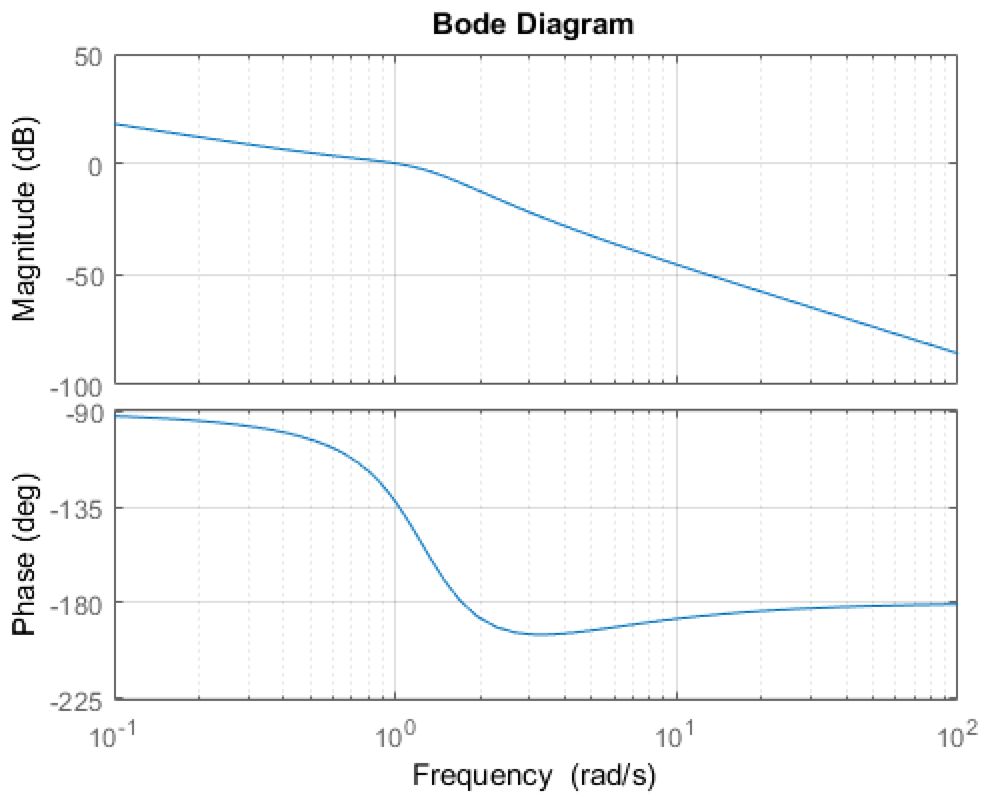
Gain and Phase Margins

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.

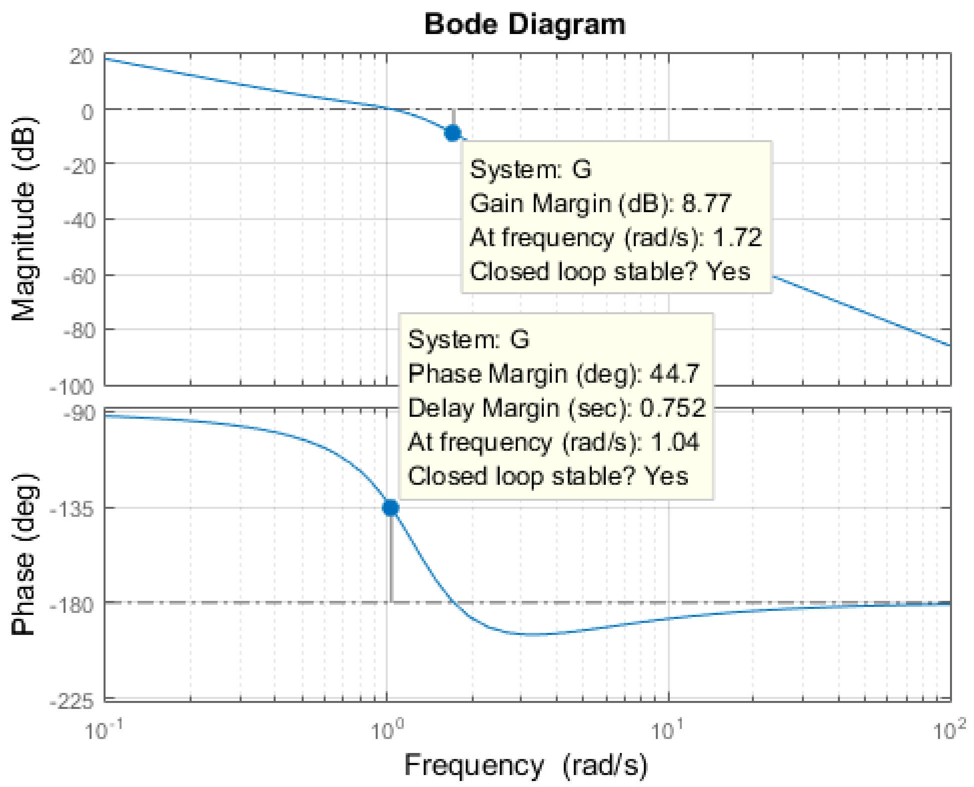
The phase margin measures how much phase variation is needed at the gain crossover frequency to lose stability. Similarly, the gain margin measures what relative gain variation is needed at the gain crossover frequency to lose stability. Together, these two numbers give an estimate of the "safety margin" 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
```

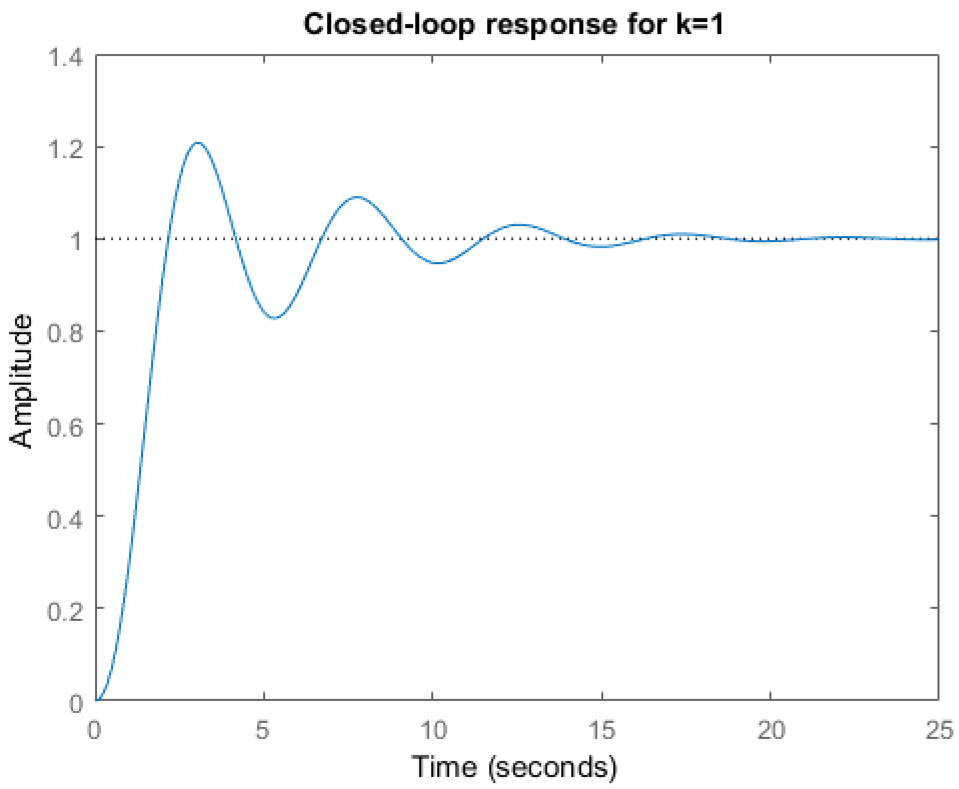


Then, right-click on the plot and select the **Characteristics -> Minimum Stability Margins** submenu. Finally, click on the blue dot markers. The resulting plot is shown below:



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')
```



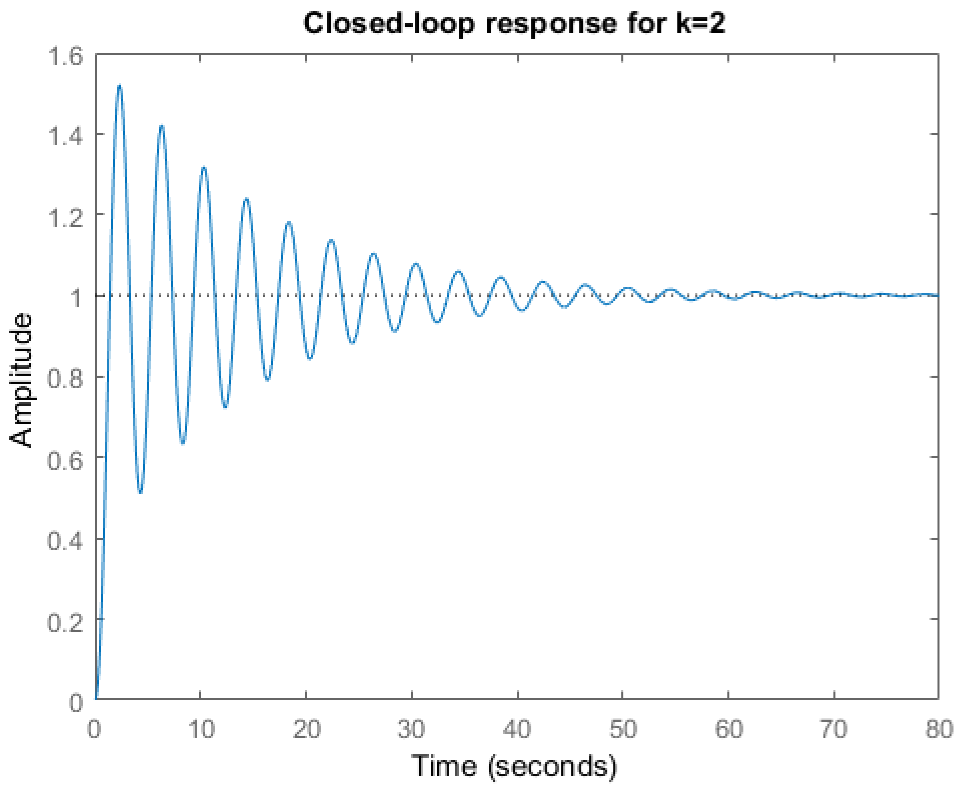
If we increase the gain to `k=2`, the stability margins are reduced to

```
[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

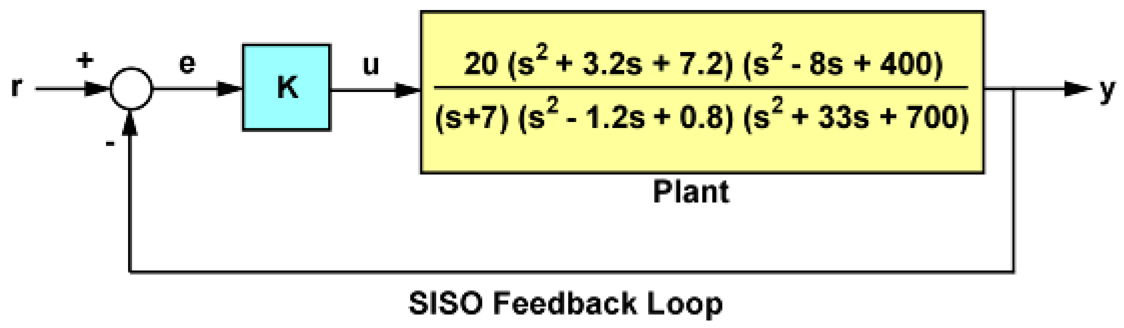
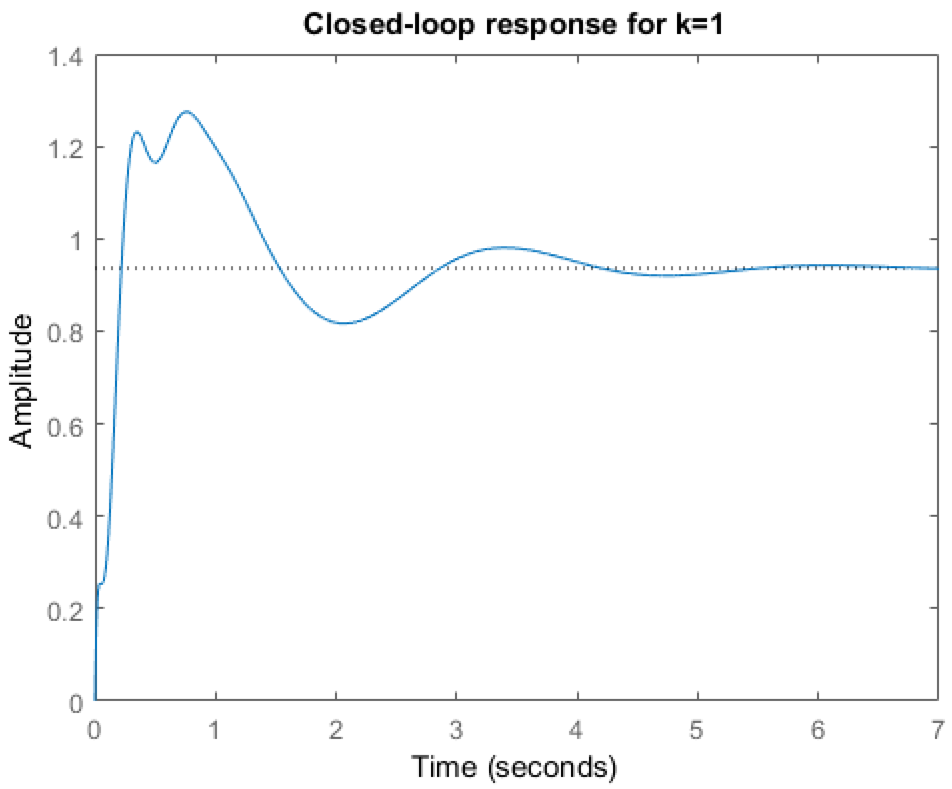

r
+
-
e
K
u
$\frac{20\,(s^2+3.2s+7.2)\,(s^2-8s+400)}{(s+7)\,(s^2-1.2s+0.8)\,(s^2+33s+700)}$
Plant
y
SISO Feedback Loop

Figure 2: Feedback Loop with Multiple Phase Crossovers

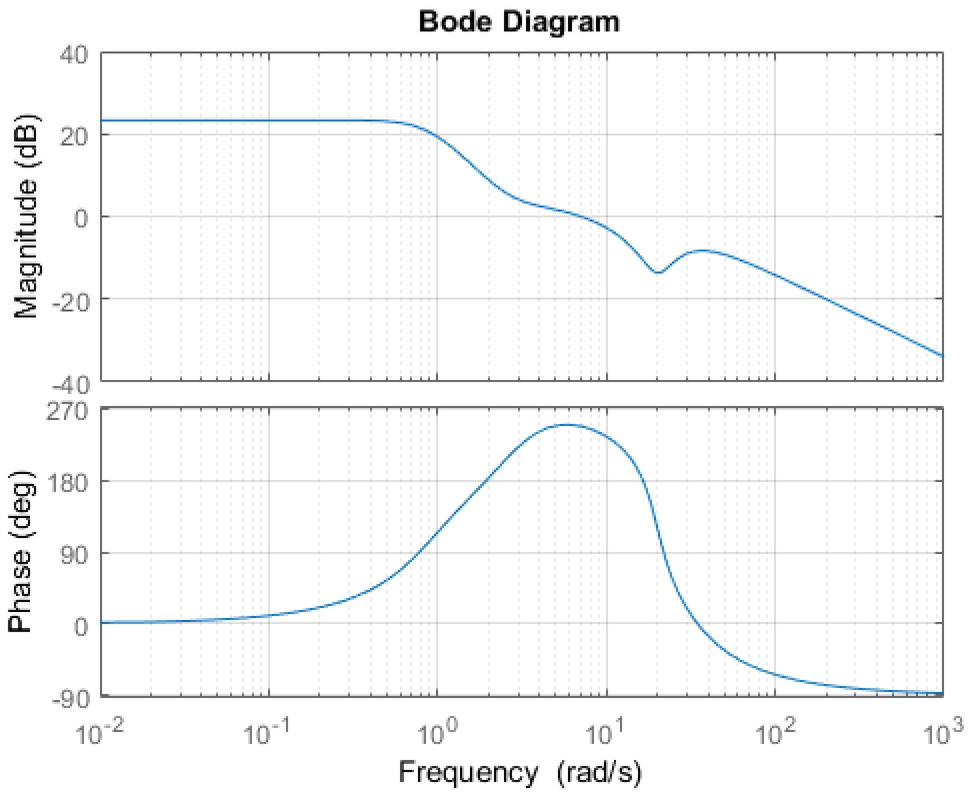
The closed-loop response for `k=1` is stable:

```
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')
```

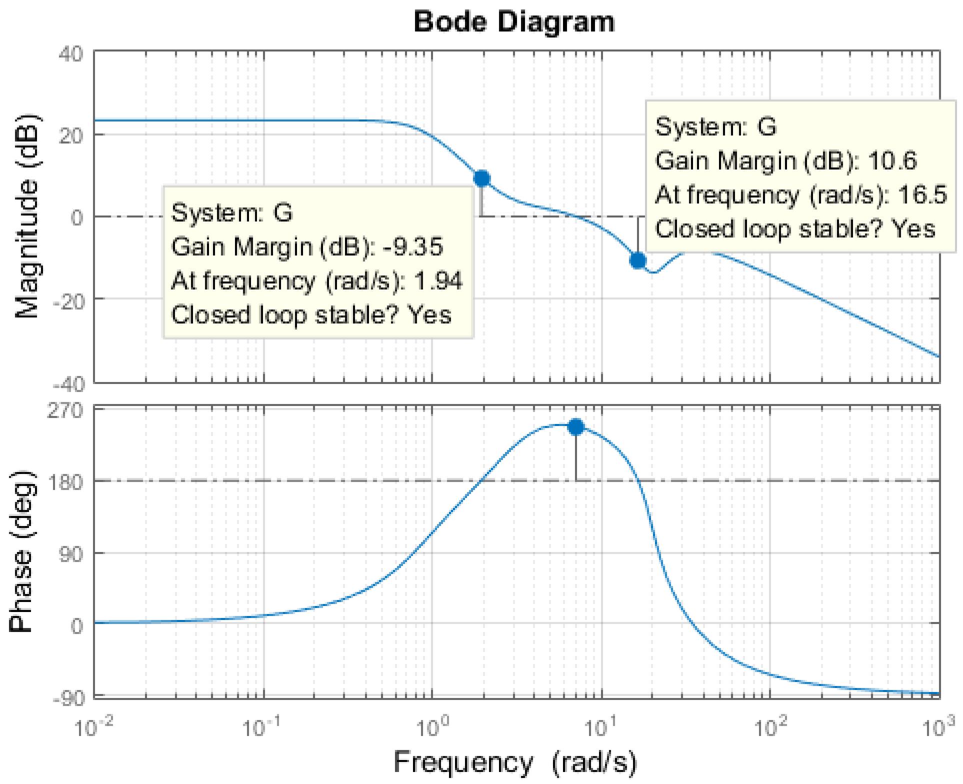


To assess how robustly stable this loop is, plot its Bode response:

```
bode(G), grid
```

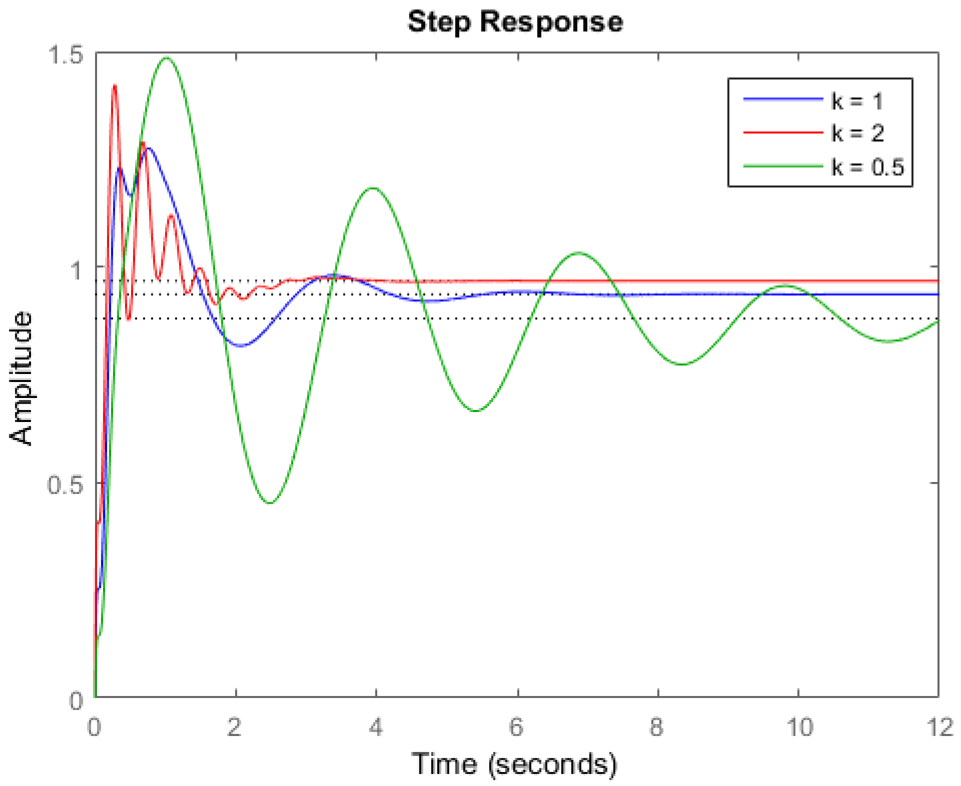


Then, right-click on the plot and select the **Characteristics -> All Stability Margins** submenu to show all the crossover frequencies and associated stability margins. The resulting plot is shown below.



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)
```

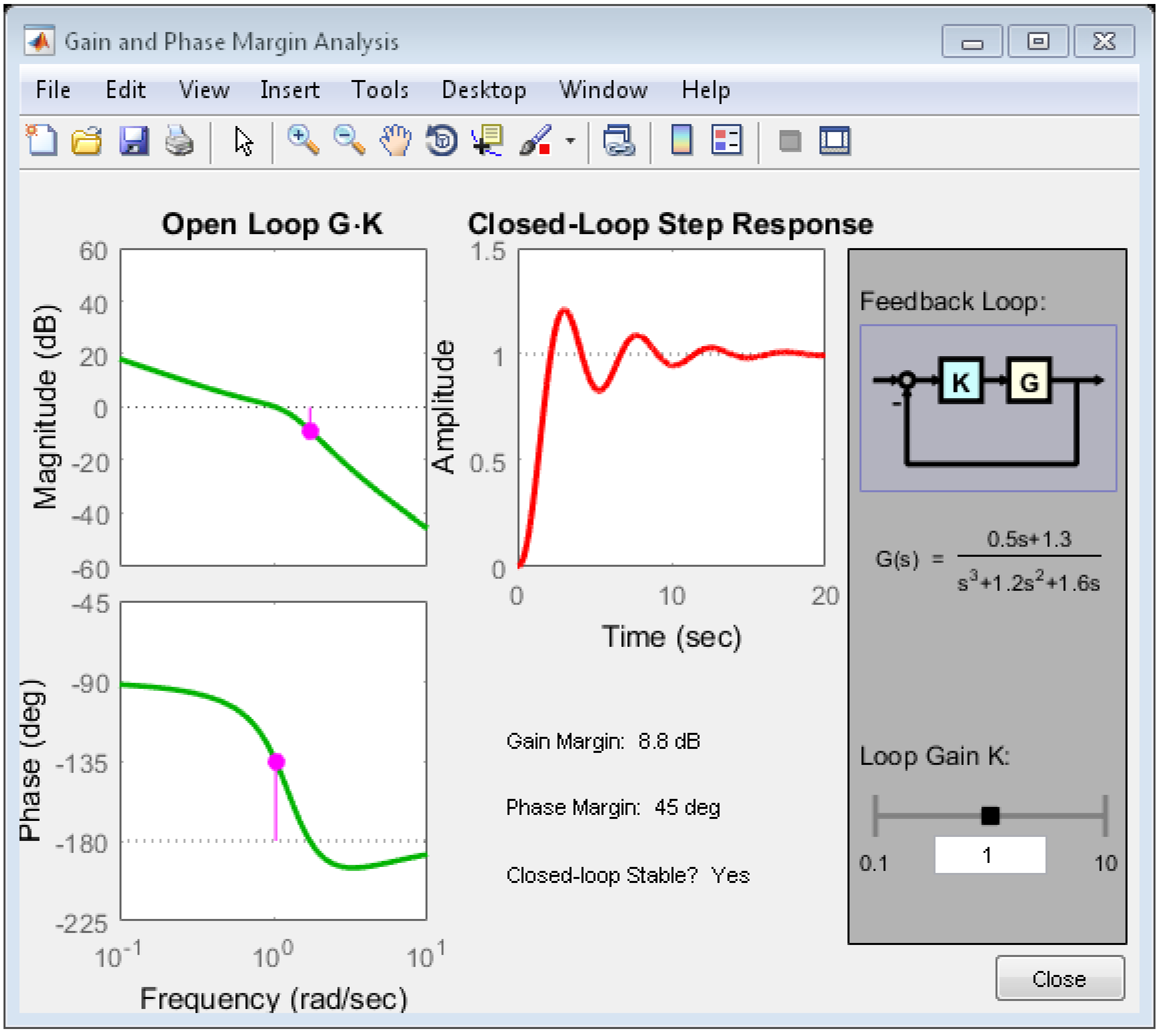
```
m = 

     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
```



$G(s) = \dfrac{0.5s+1.3}{s^3+1.2s^2+1.6s}$

Gain Margin: 8.8 dB

Phase Margin: 45 deg

Closed-loop Stable? Yes

See Also

`margin` | `pole`

Related Examples

- "Pole and Zero Locations" on page 7-16

Analyzing Control Systems with Delays

This example shows how to use Control System Toolbox™ to analyze and design control systems with delays.

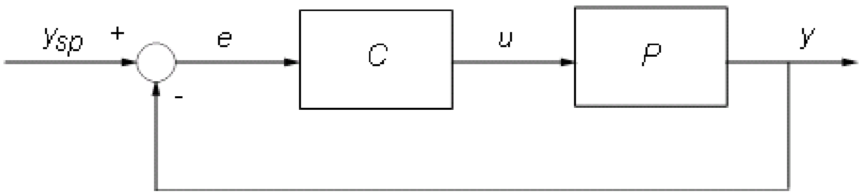
Control of Processes with Delays

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);
```

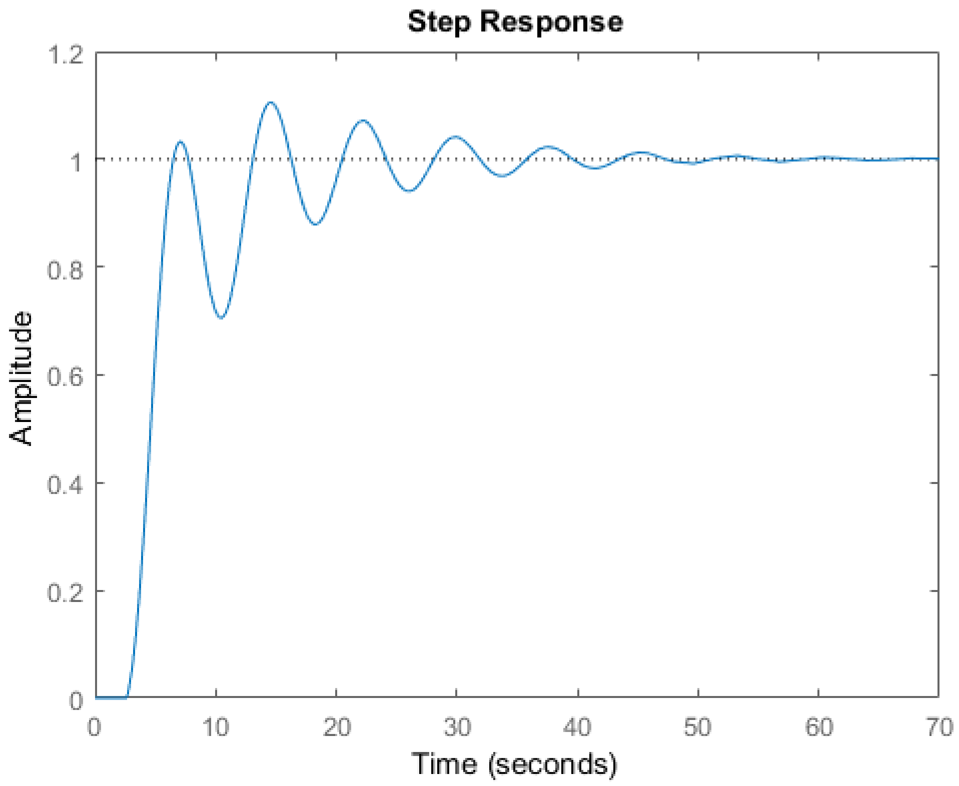
To analyze the closed-loop response, construct a model T of the closed-loop transfer from ysp to y. Because there is a delay in this feedback loop, you must convert P and C to state space and use the state-space representation for analysis:

```
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.

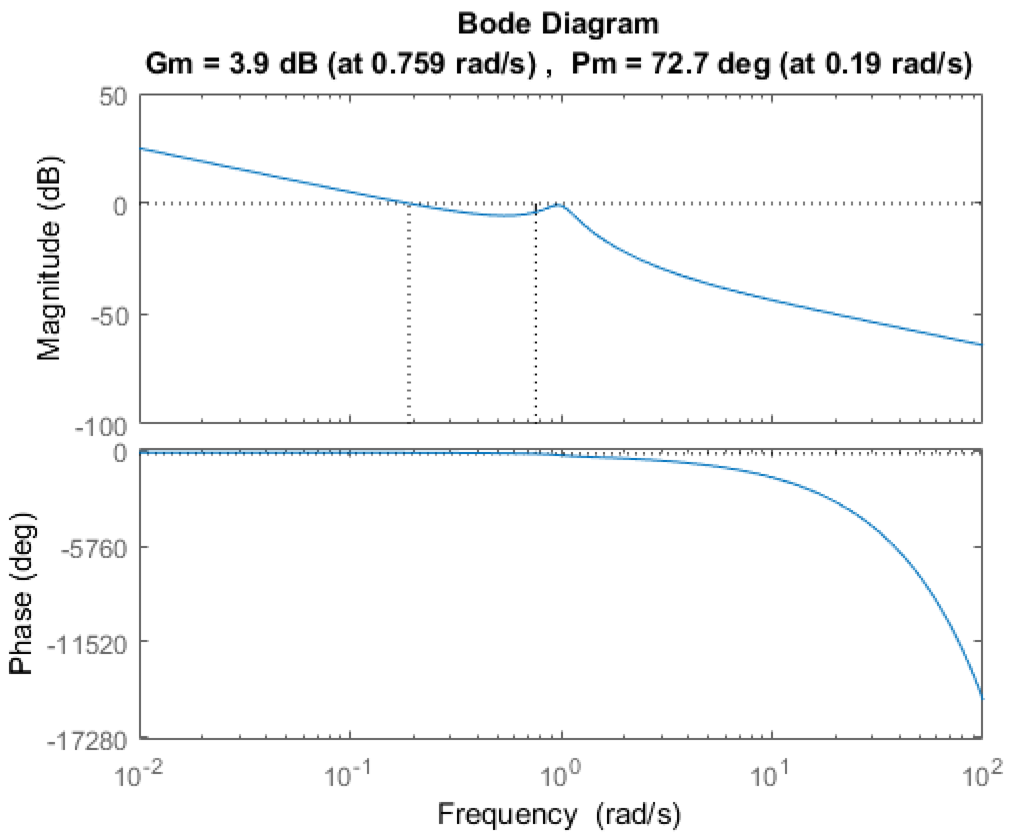
Use the `STEP` command to plot the closed-loop step response from `ysp` to `y`:

```
step(T)
```



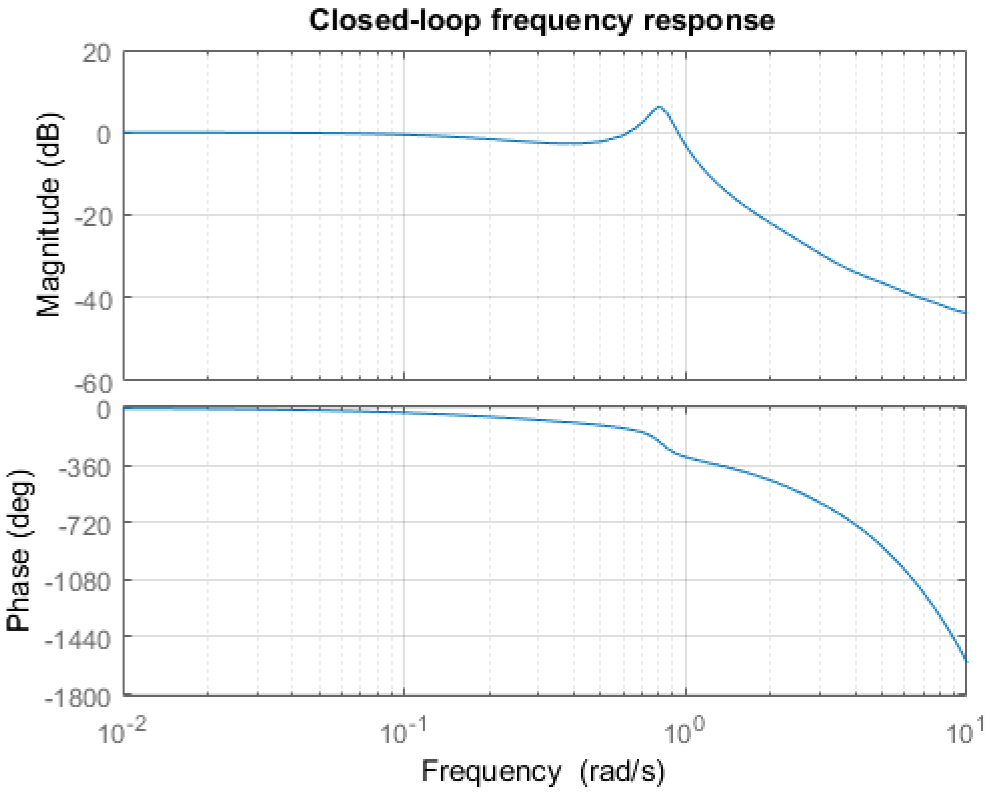
The closed-loop oscillations are due to a weak gain margin as seen from the open-loop response `P*C`:

```
margin(P*C)
```



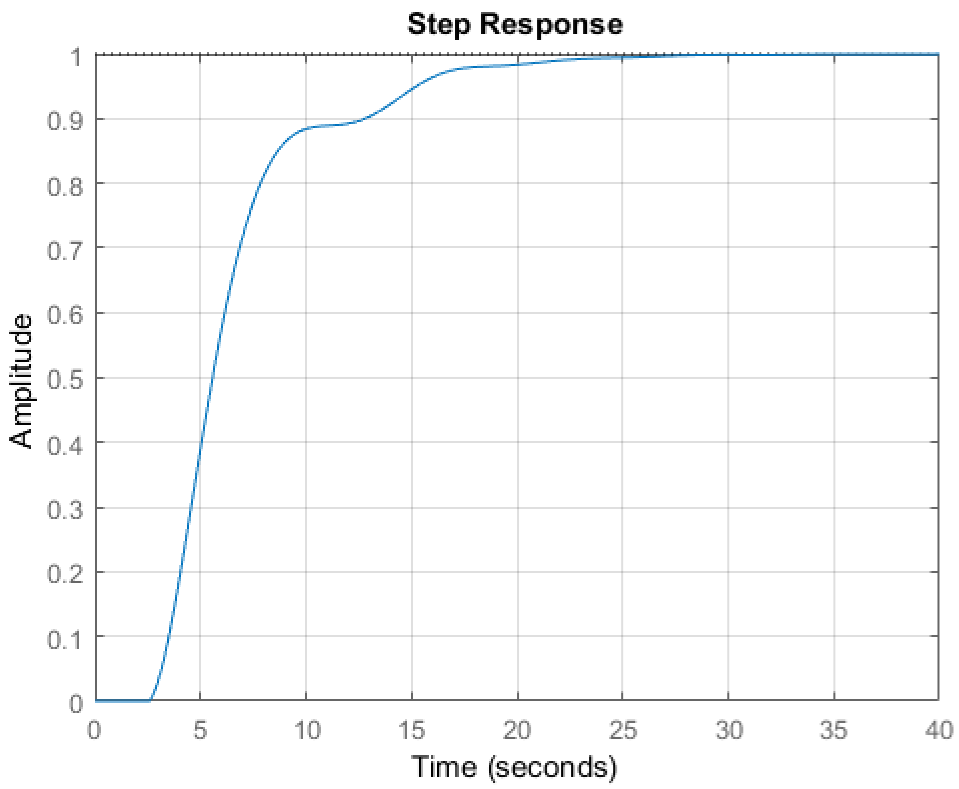
There is also a resonance in the closed-loop frequency response:

```
bode(T)
grid, title('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
```



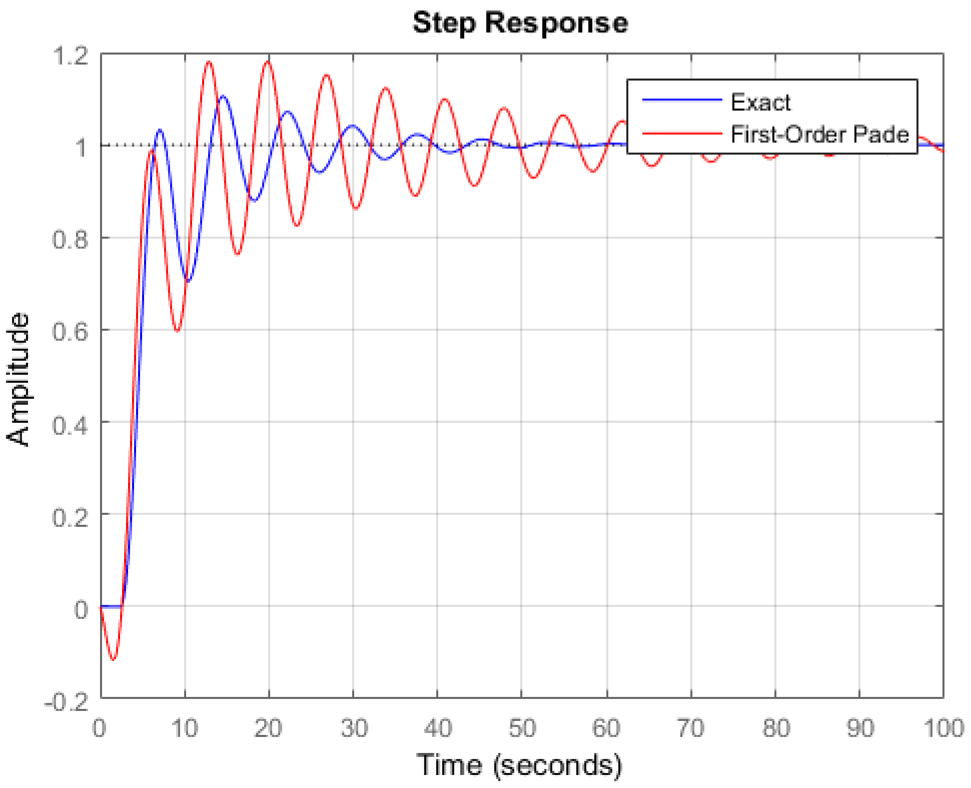
Pade Approximation of Time Delays

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:

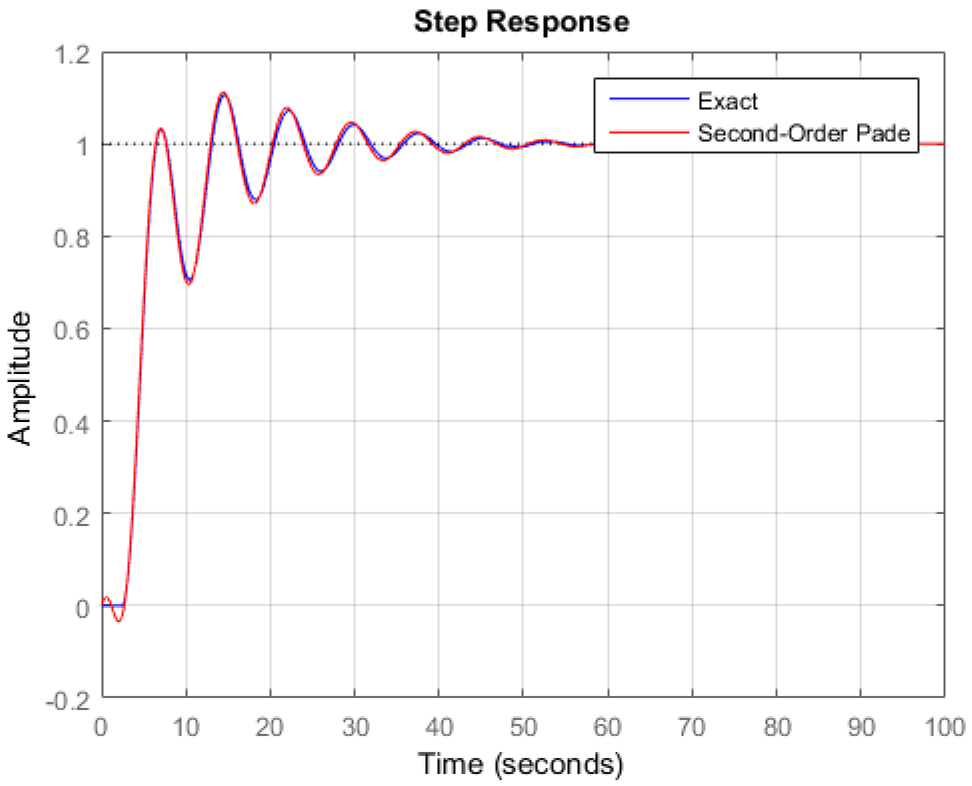
```
T1 = pade(T,1);
```

```
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.

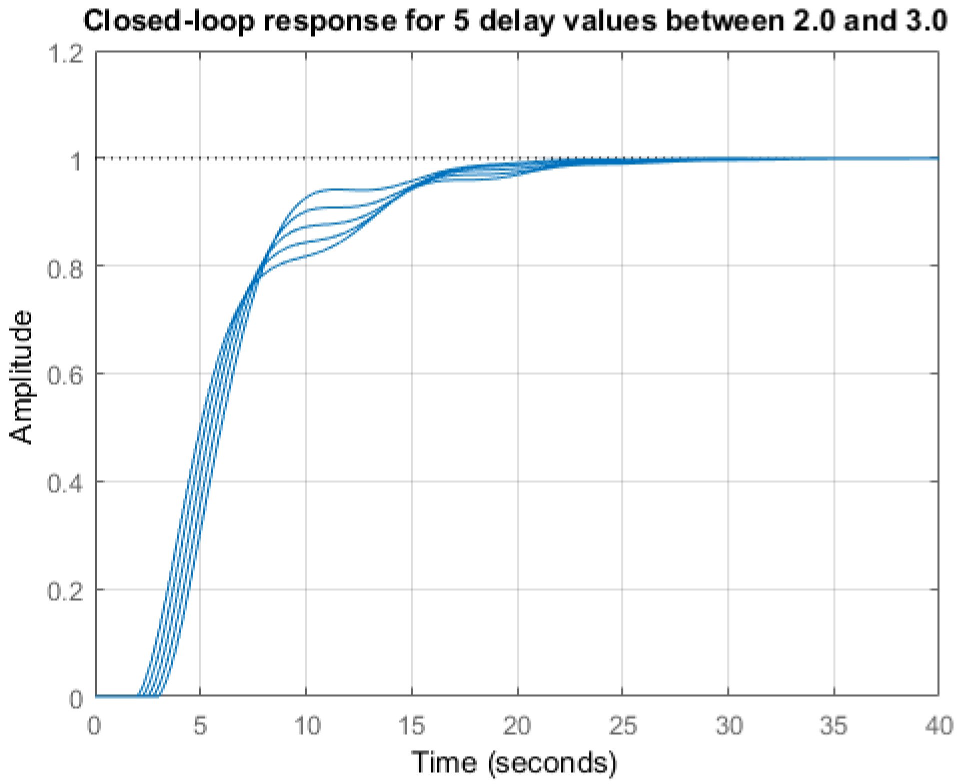
For example, to analyze the sensitivity of the notched PI control above, create 5 models with delay values ranging from 2.0 to 3.0:

```
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
```

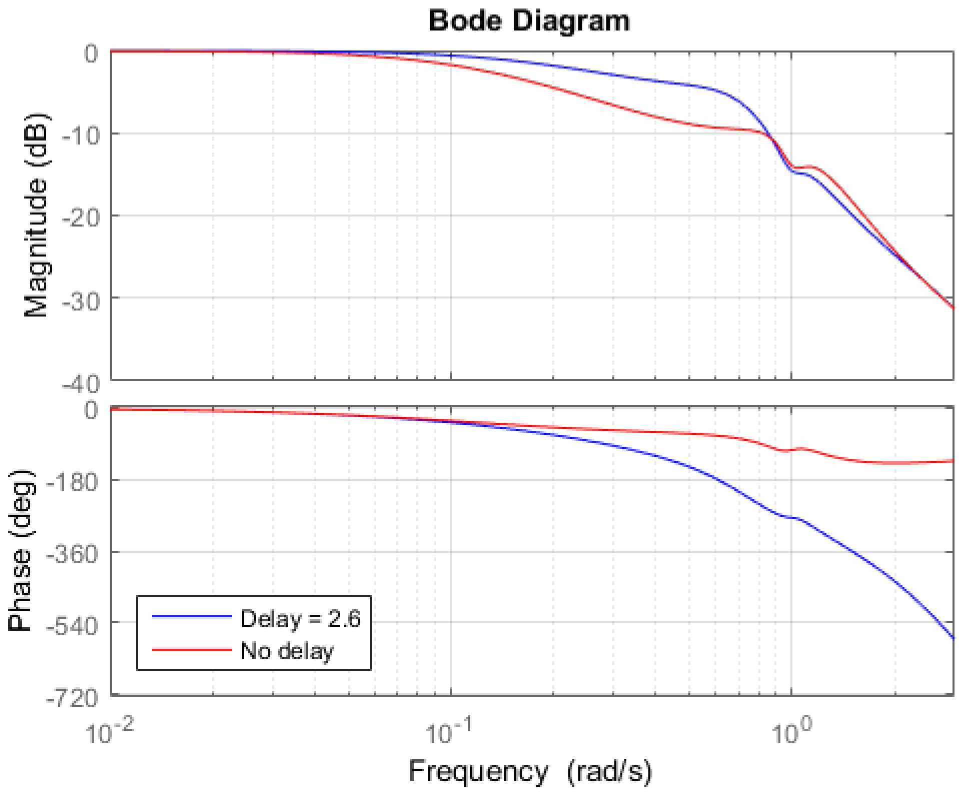
Then use STEP to create an envelope plot:

```
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')
```



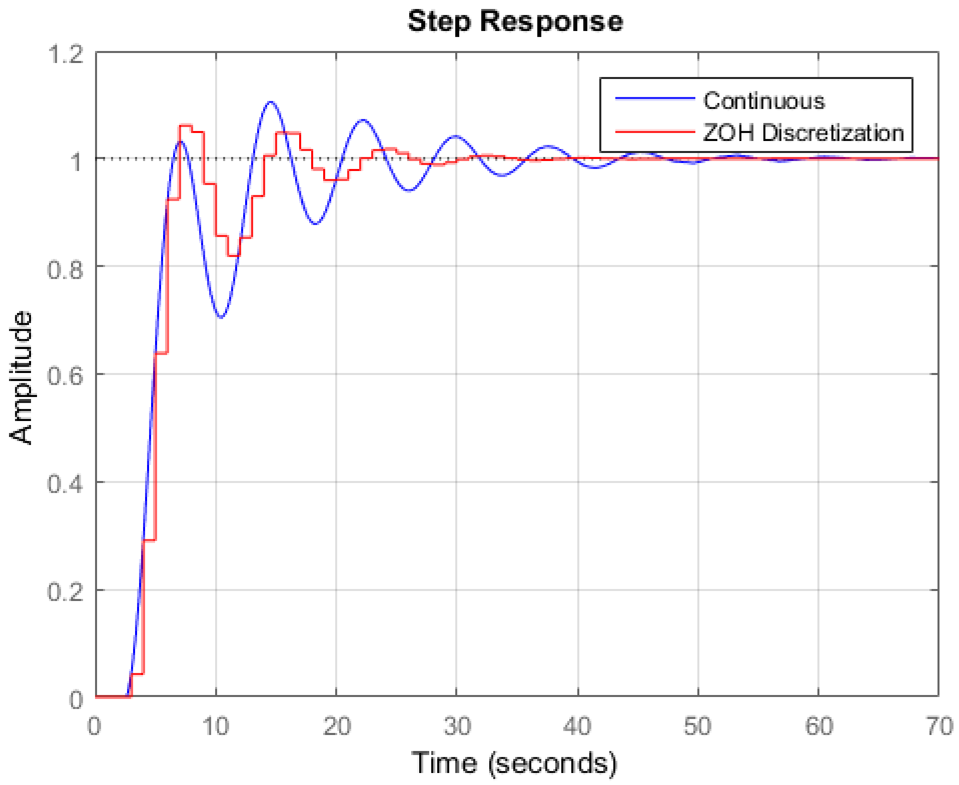
Discretization

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')
```

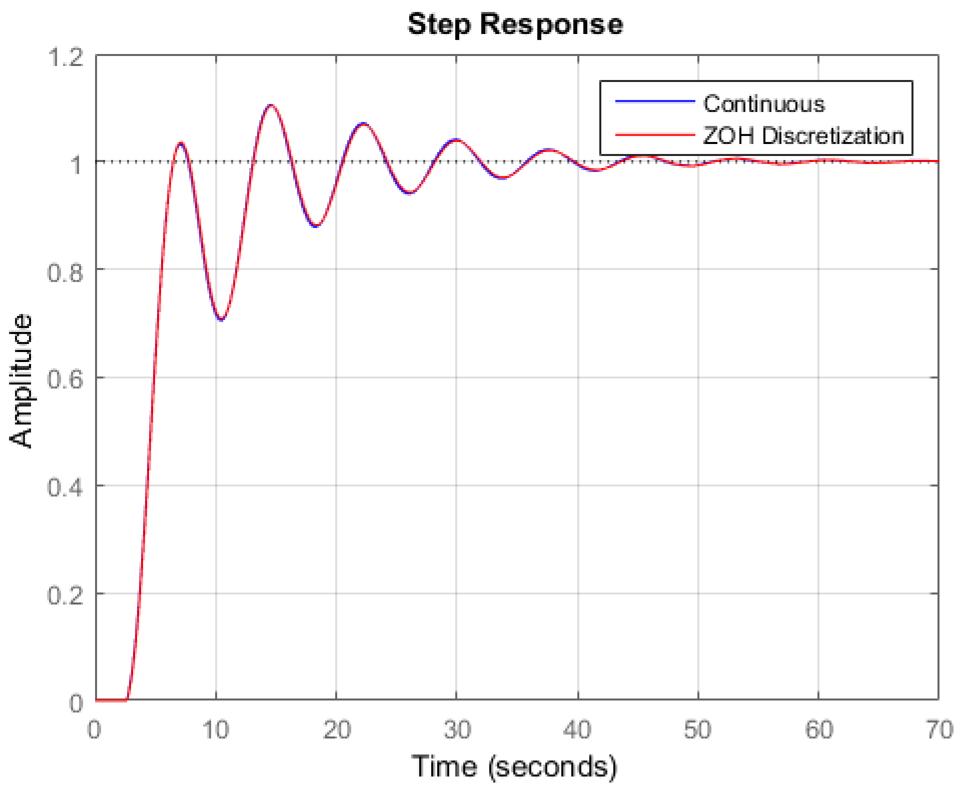
```
Warning: Discretization is only approximate due to internal delays. Use faster
sampling rate if discretization error is large.
```



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.
```



Note that internal delays remain internal in the discretized model and do not inflate the model order:

```
order(Td)
Td.InternalDelay
```

```
ans =

     3


ans =
```

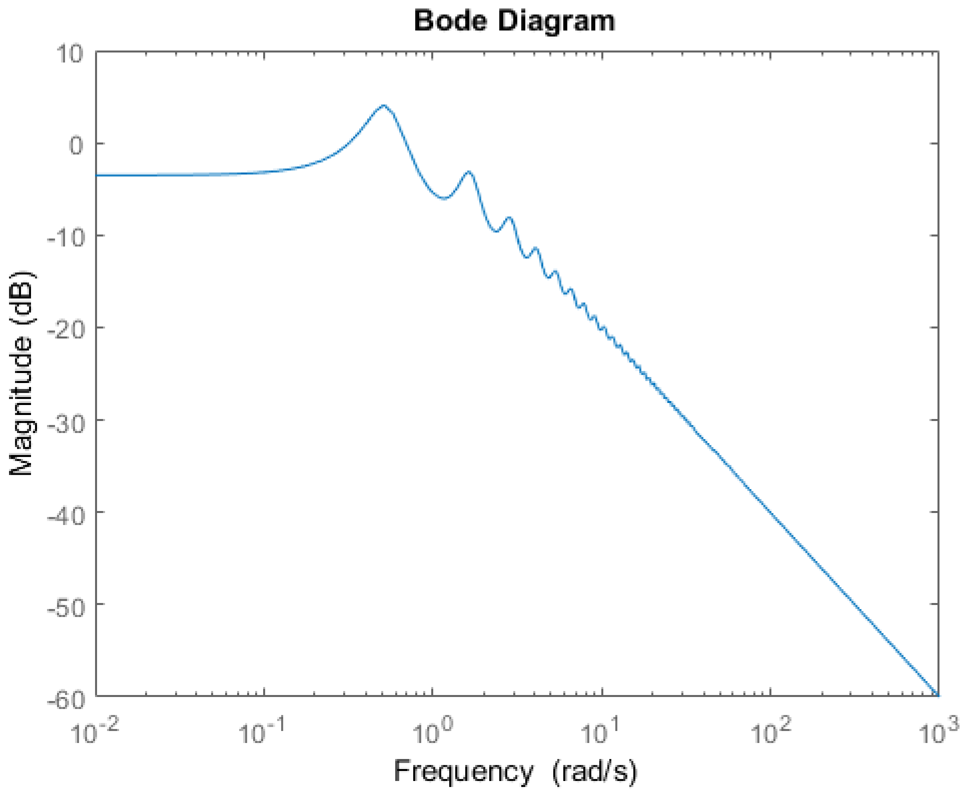
52

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

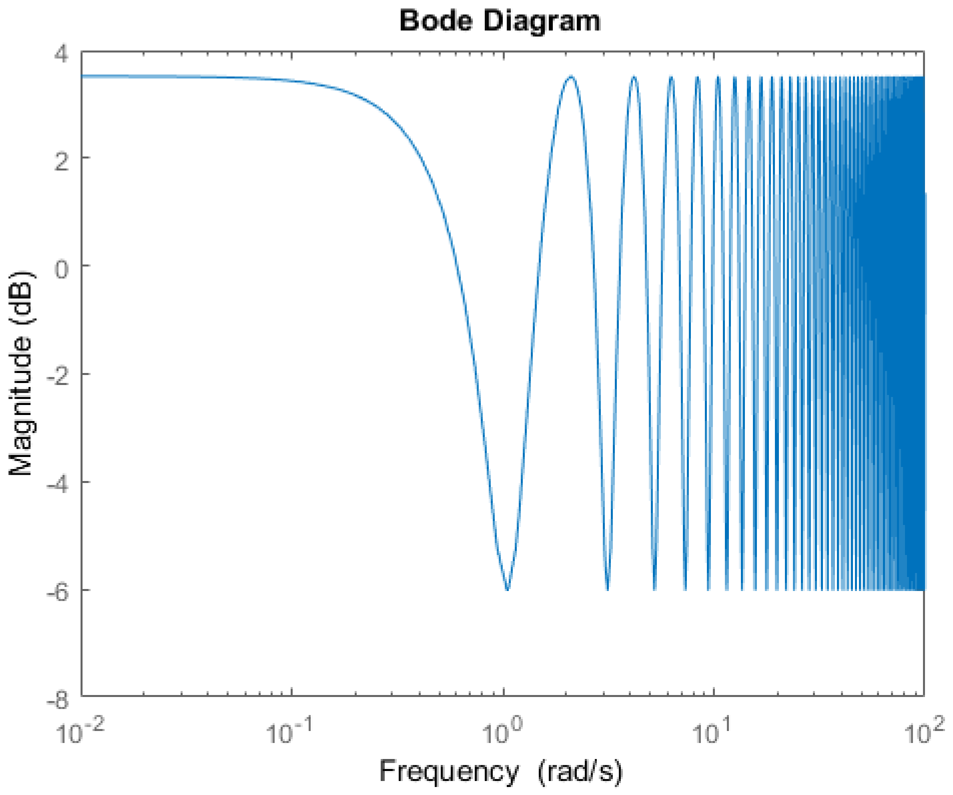
Gain ripples:

```
G = exp(-5*s)/(s+1);
T = feedback(G,.5);
bodemag(T)
```



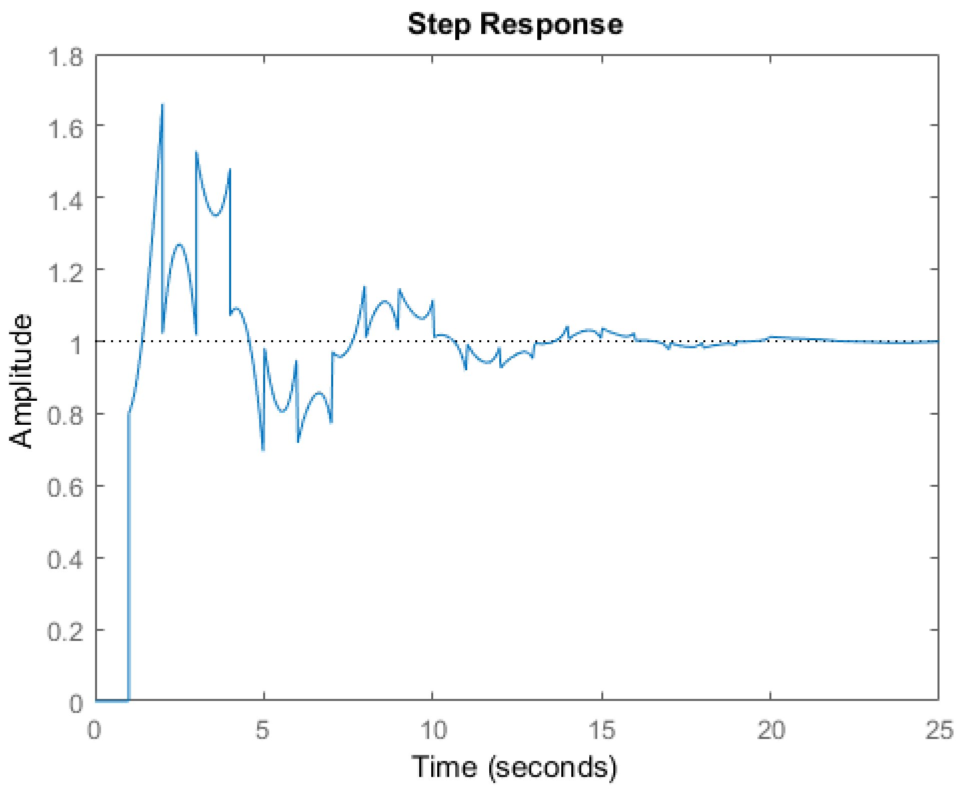
Gain oscillations:

```
G = 1 + 0.5 * exp(-3*s);
bodemag(G)
```



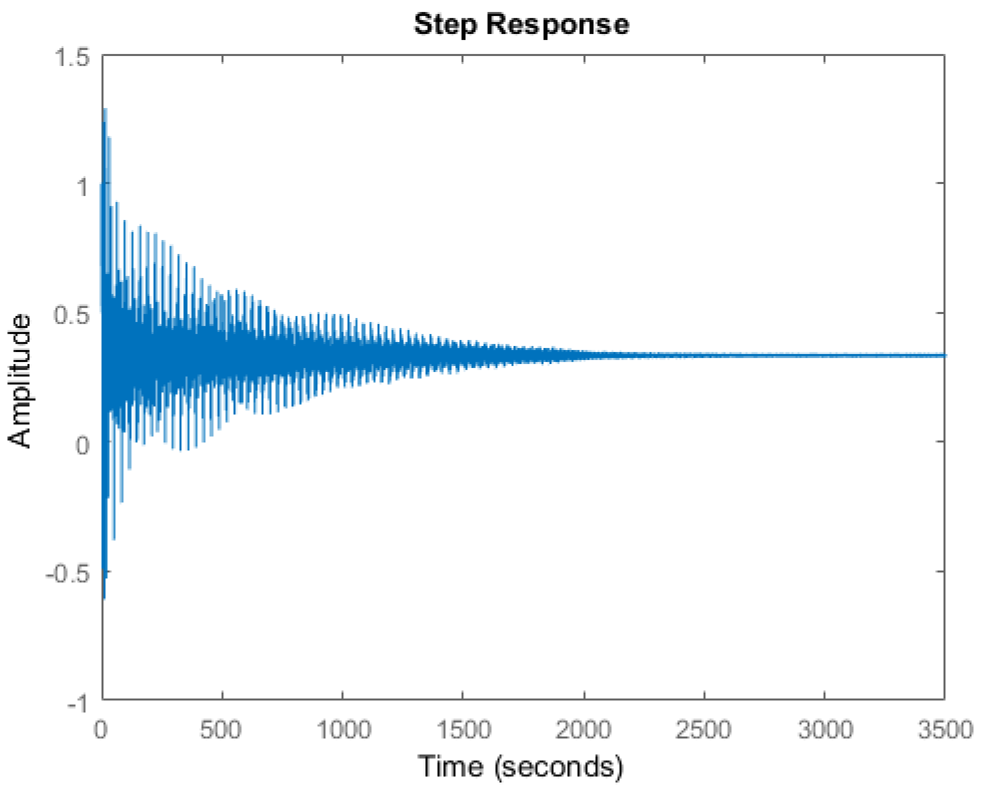
Jagged step response (note the "echoes" of the initial step):

```
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(G,1);
step(T)
```



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 7-49

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:

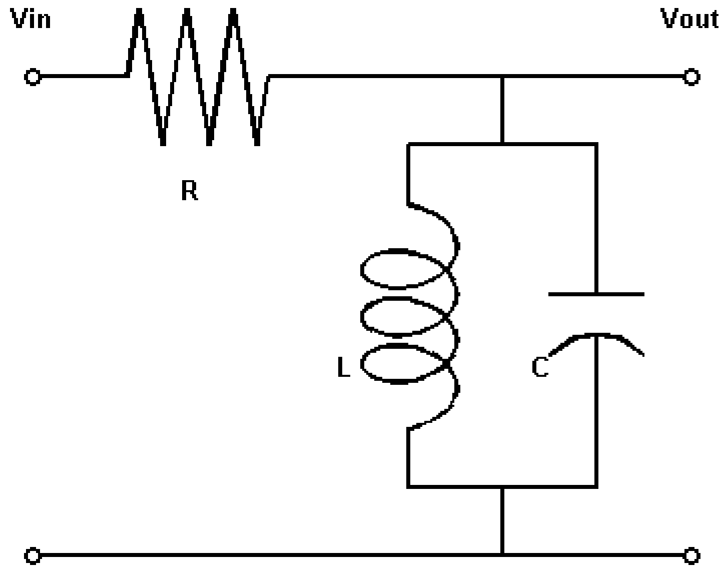


Figure 1: Bandpass RLC Network.

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

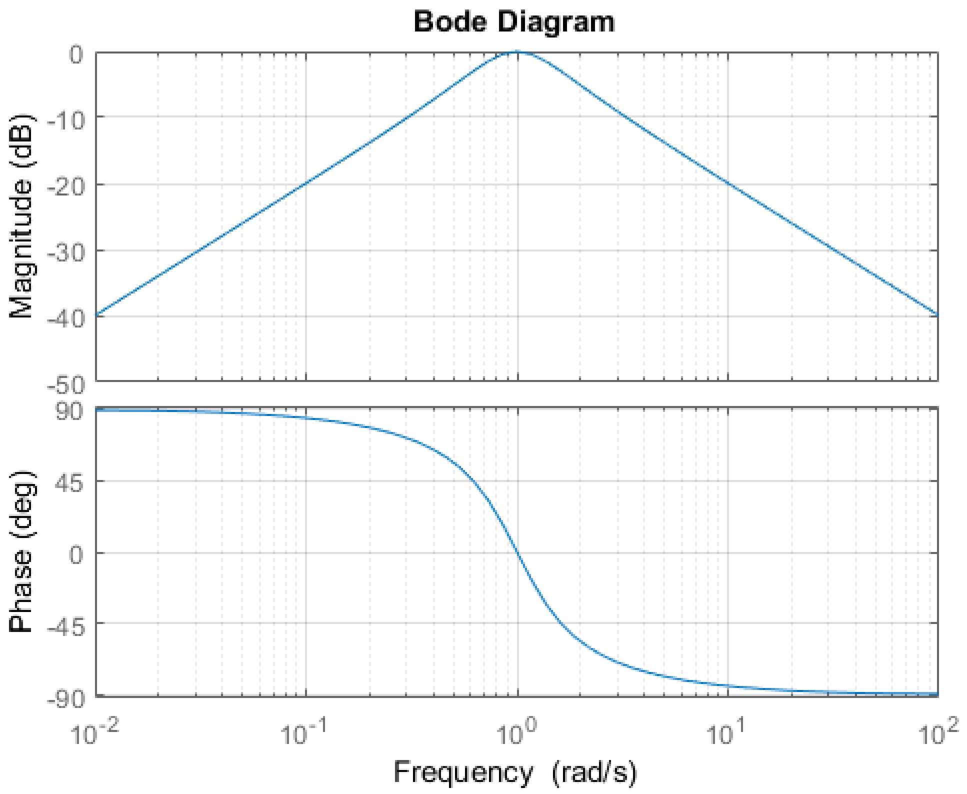
The Bode plot is a convenient tool for investigating the bandpass characteristics of the RLC network. Use `tf` to specify the circuit's transfer function for the values

```
%|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.
```

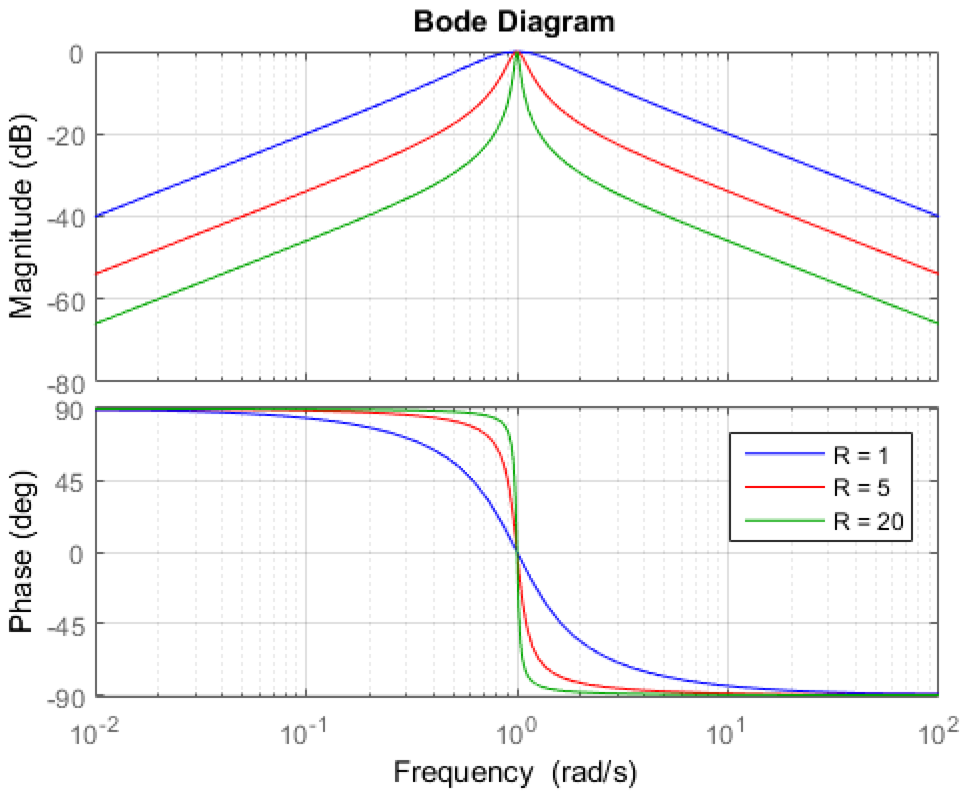
Next, use `bode` to plot the frequency response of the circuit:

```
bode(G), grid
```



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')
```

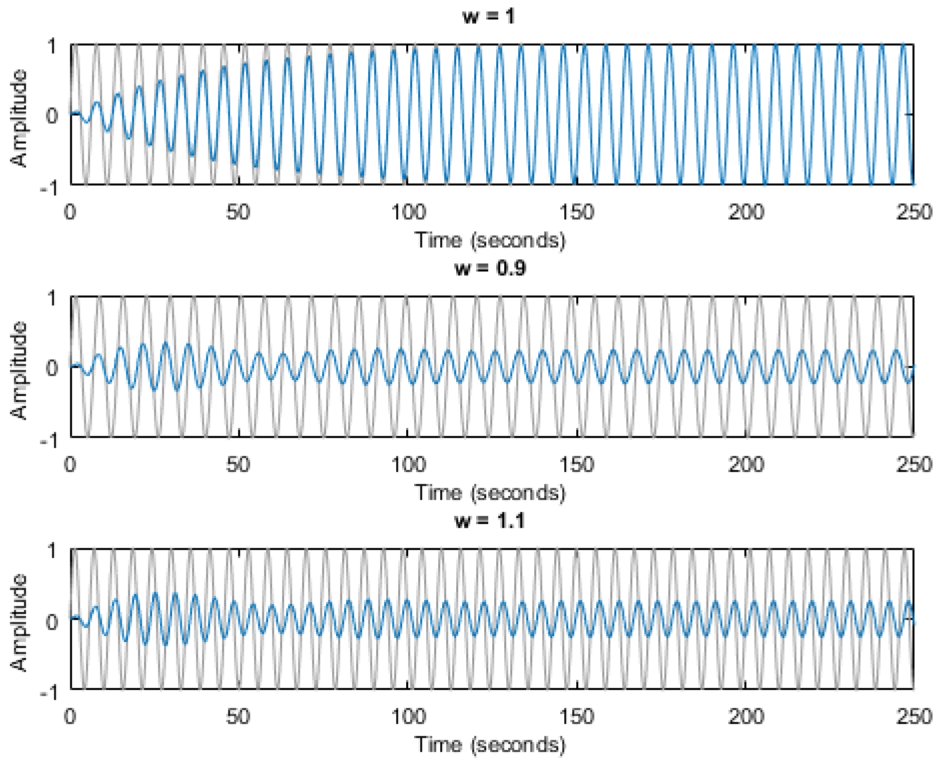


The resistor value R=20 gives a filter narrowly tuned around the target frequency of 1 rad/s.

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))
```

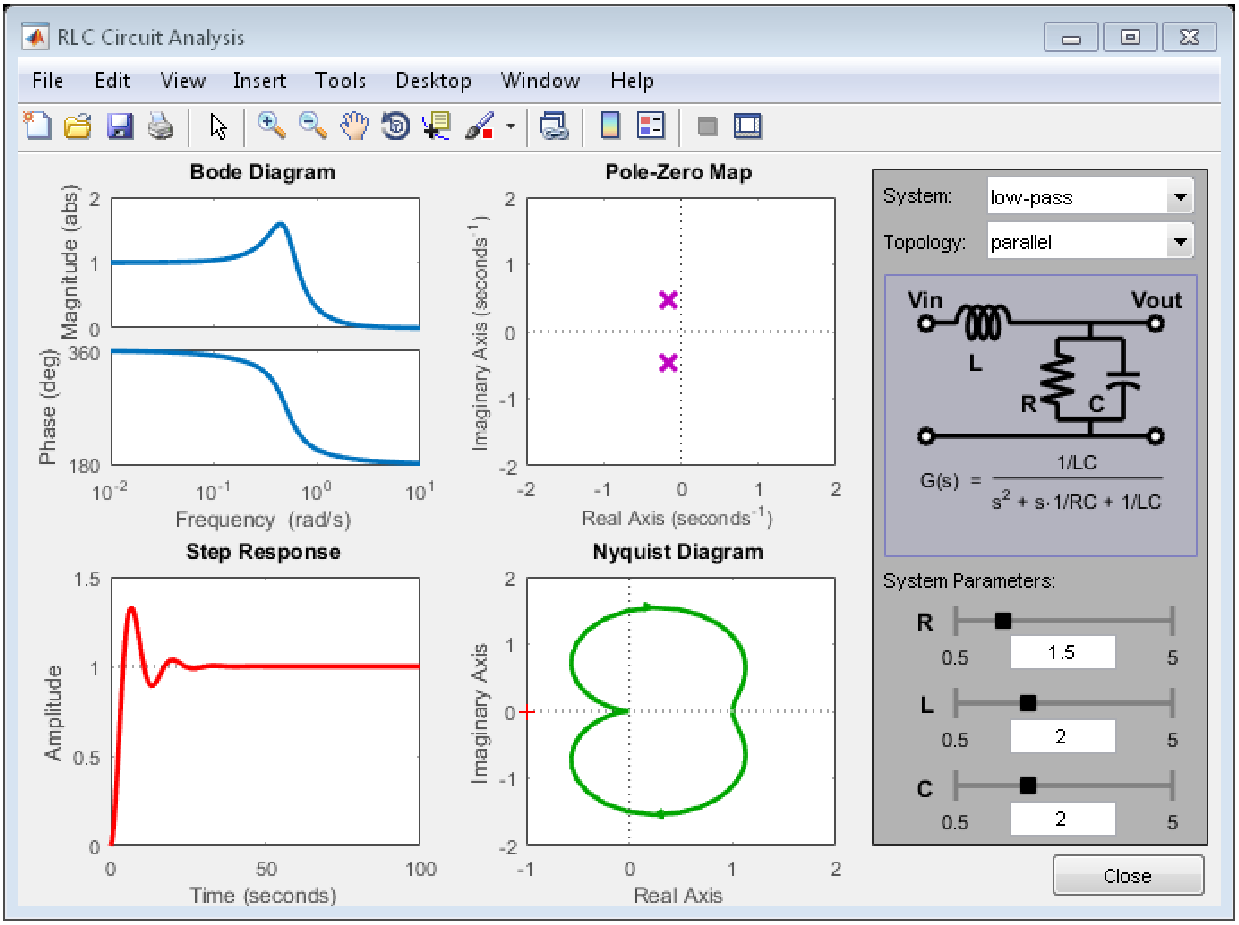
```
          Pole                  Damping       Frequency      Time Constant
                                            (rad/TimeUnit)    (TimeUnit)

 -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
```



System: low-pass

Topology: parallel

See Also

`bodeplot` | `lsim` | `stepplot`

Related Examples

- "Joint Time- and Frequency-Domain Analysis" on page 6-20

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.

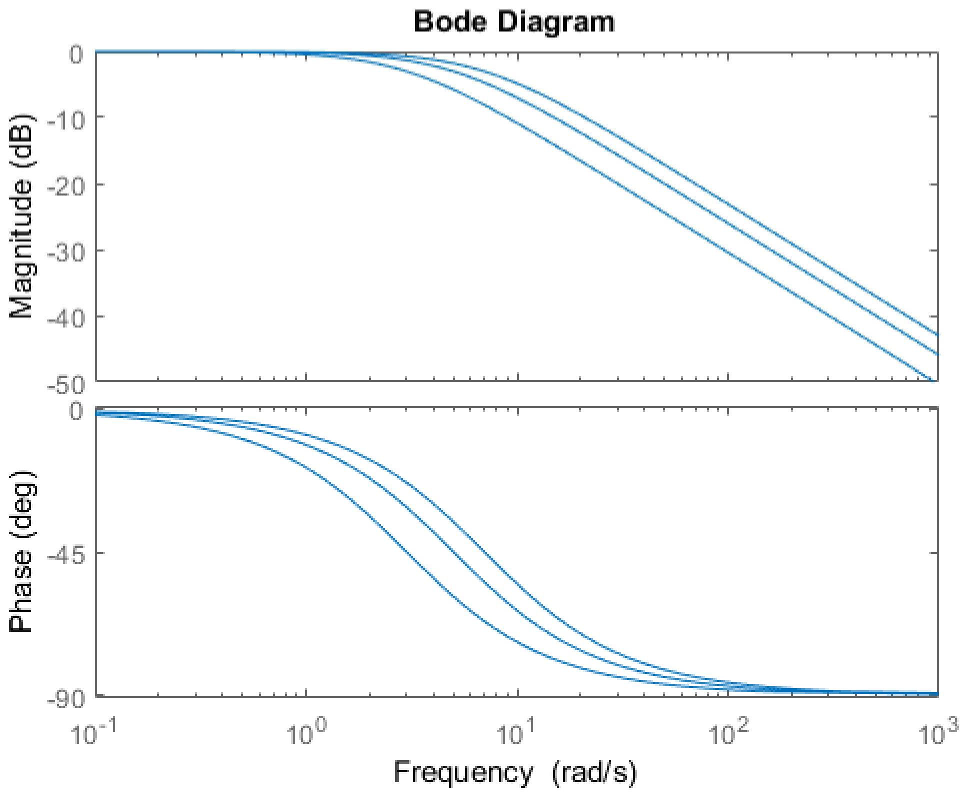
```
Farray.SamplingGrid = struct('alpha',[3 5 7]);
Farray
```

```
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

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

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:

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

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];
```

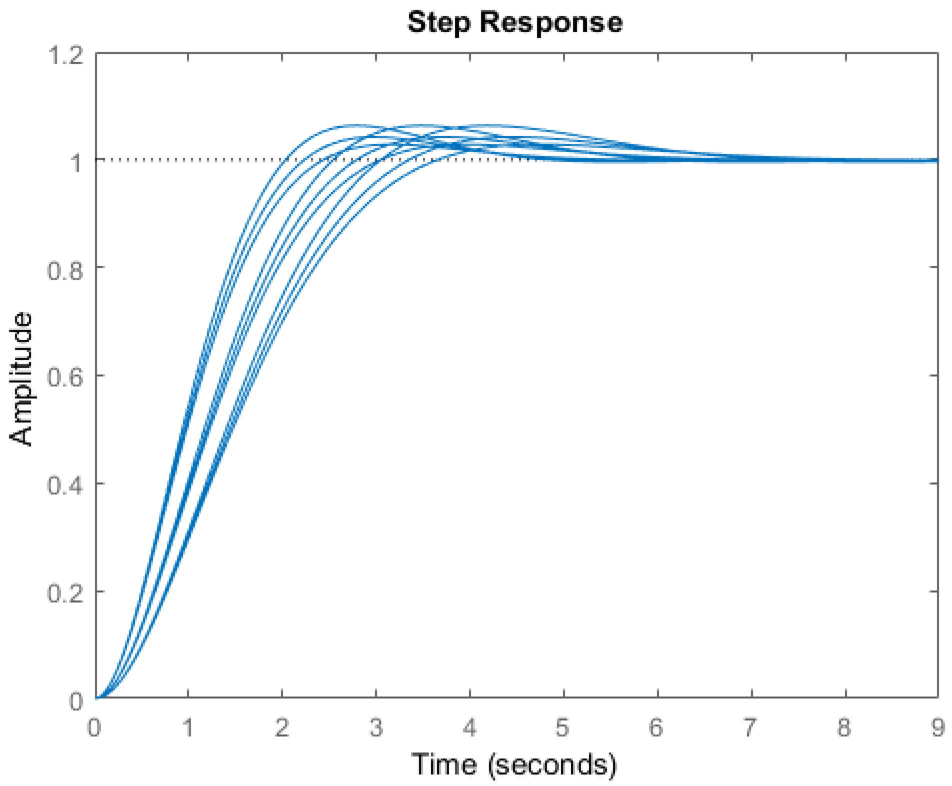
Build the array by looping through all combinations of parameter values.

```
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 `replaceBlock`.

Consider the second-order filter represented by:

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

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" f
```

`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];
[wngrid,zetagrid] = ndgrid(wnvals,zetavals);
Fsample = replaceBlock(F,'wn',wngrid,'zeta',zetagrid);
```

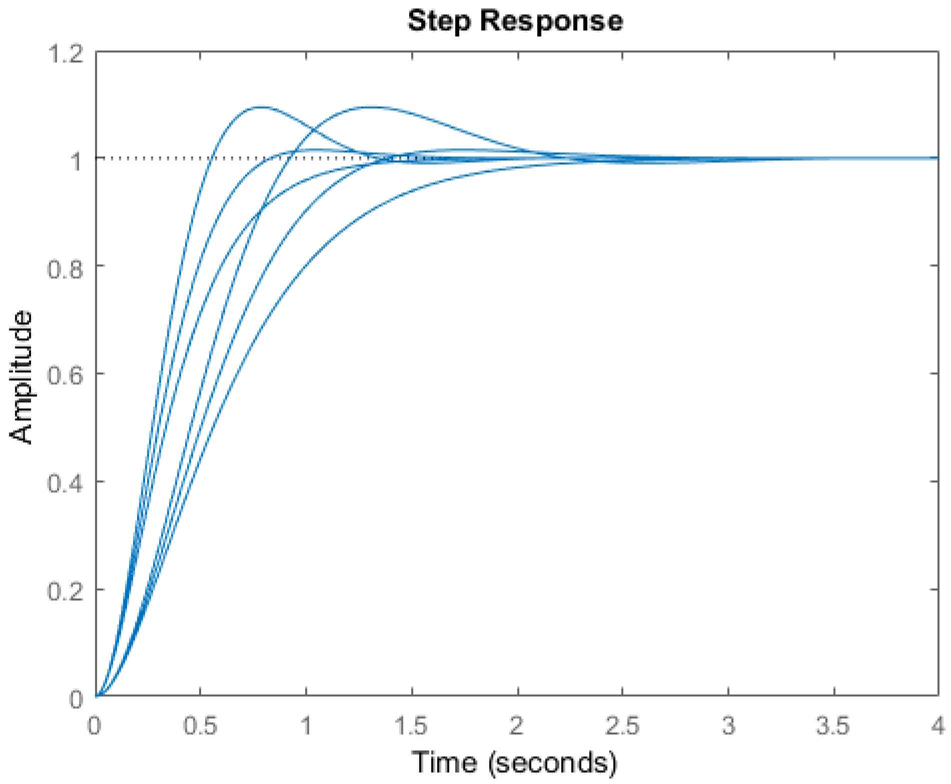
The `ndgrid` command produces a full 2-by-3 grid of parameter combinations. `Fsample` is a 2-by-3 array of state-space models. Each entry in the array is a state-space model that represents `F` evaluated at the corresponding (`wn`, `zeta`) pair. For example, `Fsample(:,:,2,3)` has `wn` = 5 and `zeta` = 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.

```
Fsample.SamplingGrid = struct('wn',wngrid,'zeta',zetagrid);
```

When you display `Fsample`, 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 `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
```

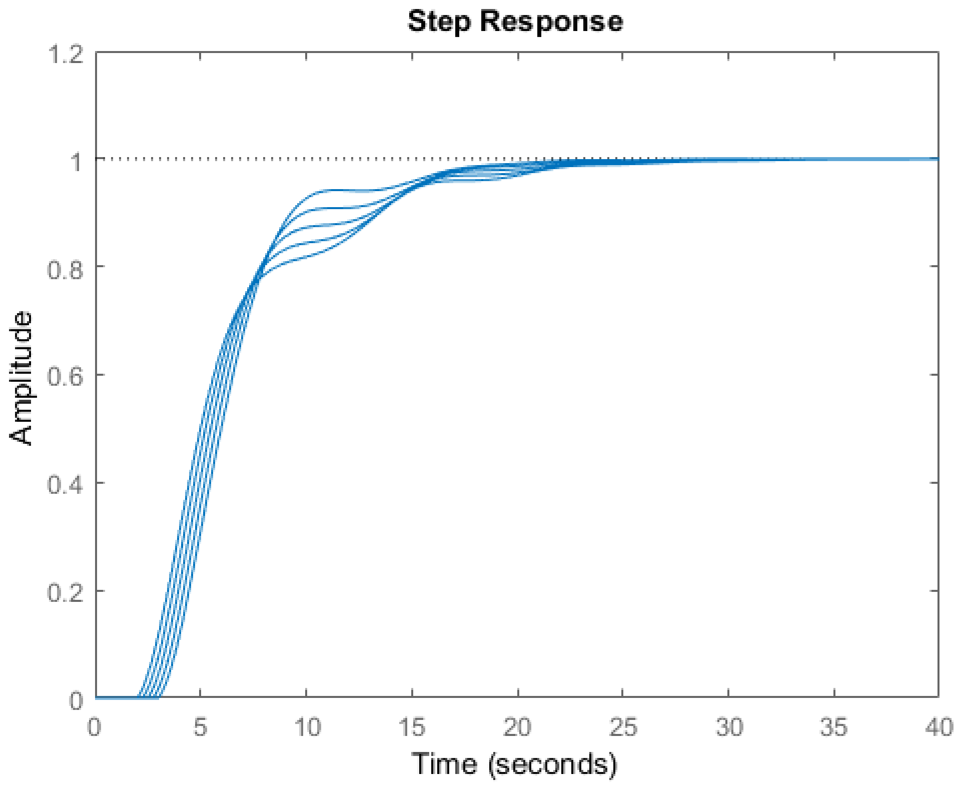
The 2.6-second input delay of the plant `G` becomes an internal delay of 2.6 s in the closed-loop system. To examine the sensitivity of the responses of Tnotch to variations in this delay, create an array of copies of Tnotch. Then, vary the internal delay across the array.

```
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.

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.

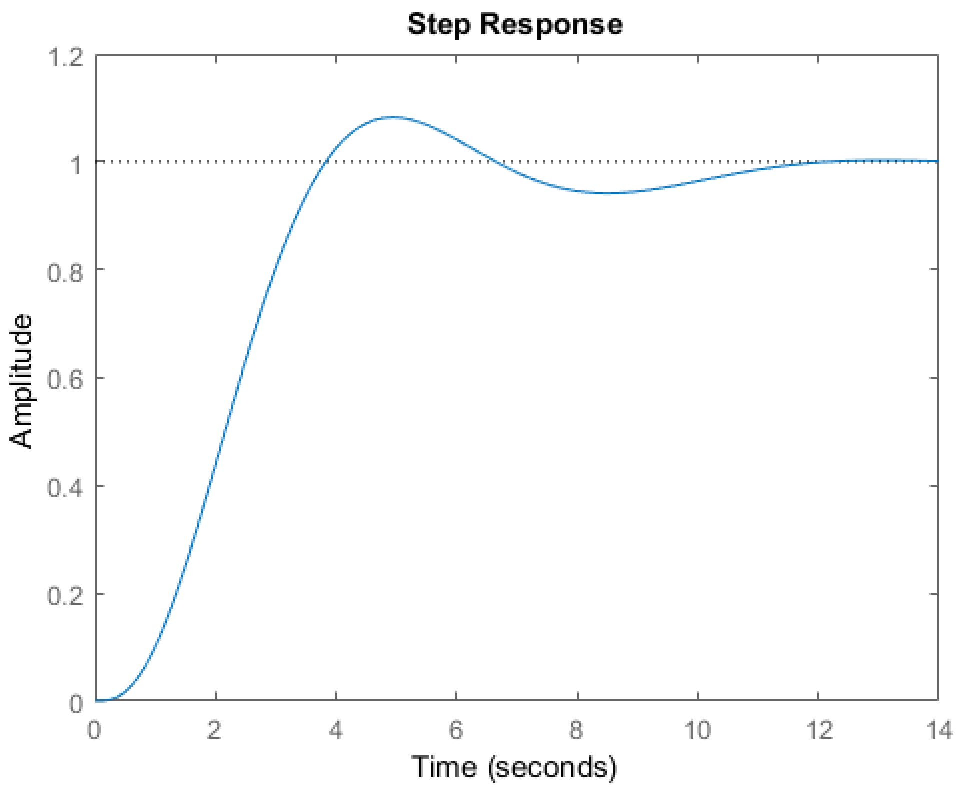
```
sys = zpk([],[-1 -1 -1],1);
[C_pi,info] = pidtune(sys,'PI')
```

```
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)
```



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)
```

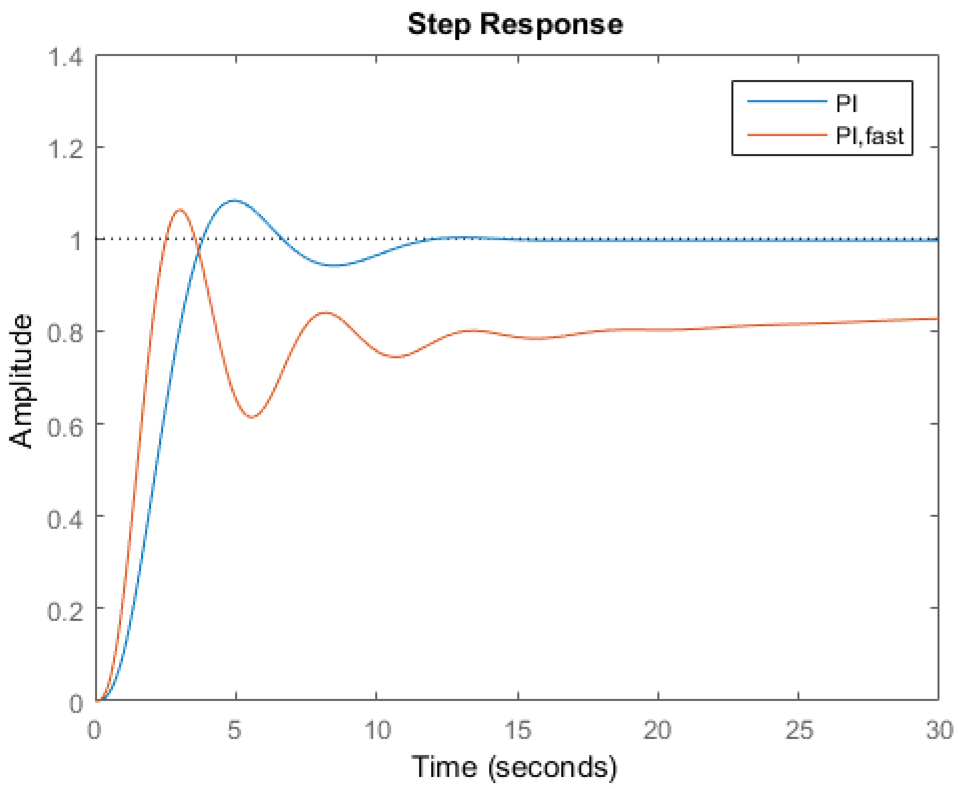
```
C_pi_fast =
 
             1 
  Kp + Ki * ---
             s 

  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.

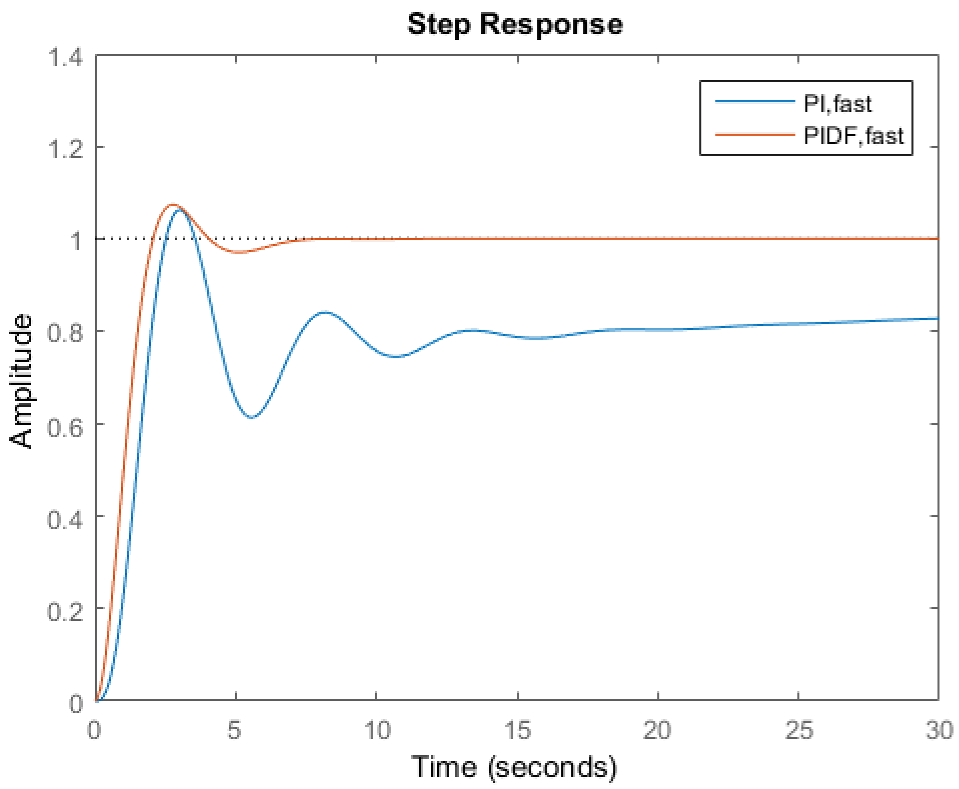
```
[C_pidf_fast,info] = pidtune(sys,'PIDF',1.0)
```

```
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.

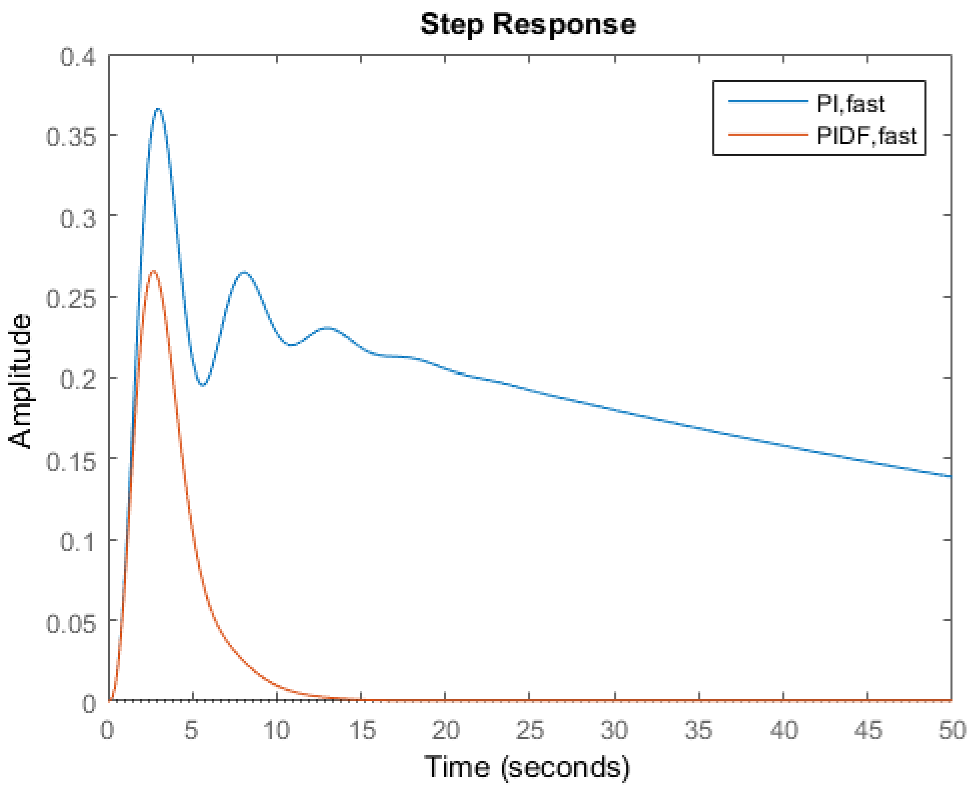
Compare the closed-loop step response and disturbance rejection for the fast PI and PIDF controllers.

```
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 9-10
- "PID Controller Design for Fast Reference Tracking"

More About

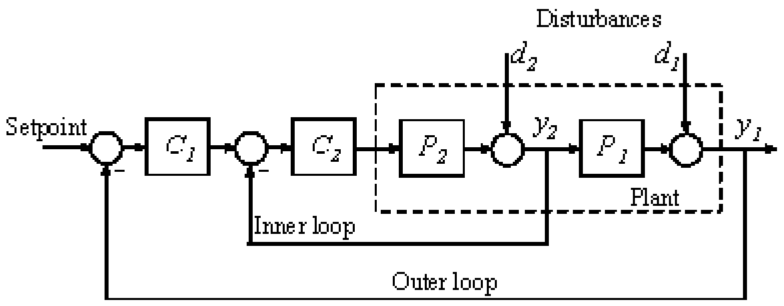
- “Choosing a PID Controller Design Tool”

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

$$P2(s) = \frac{3}{s+2}$$

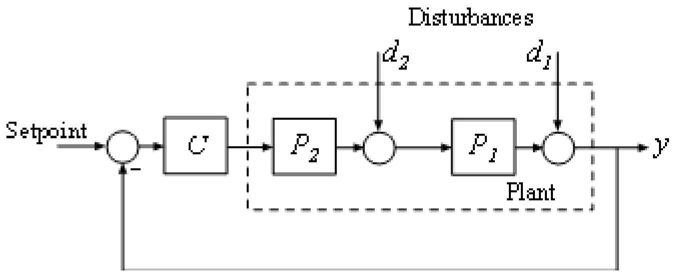
The outer loop plant P1 is

$$P1(s) = \frac{10}{(s+1)^3}$$

```
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.

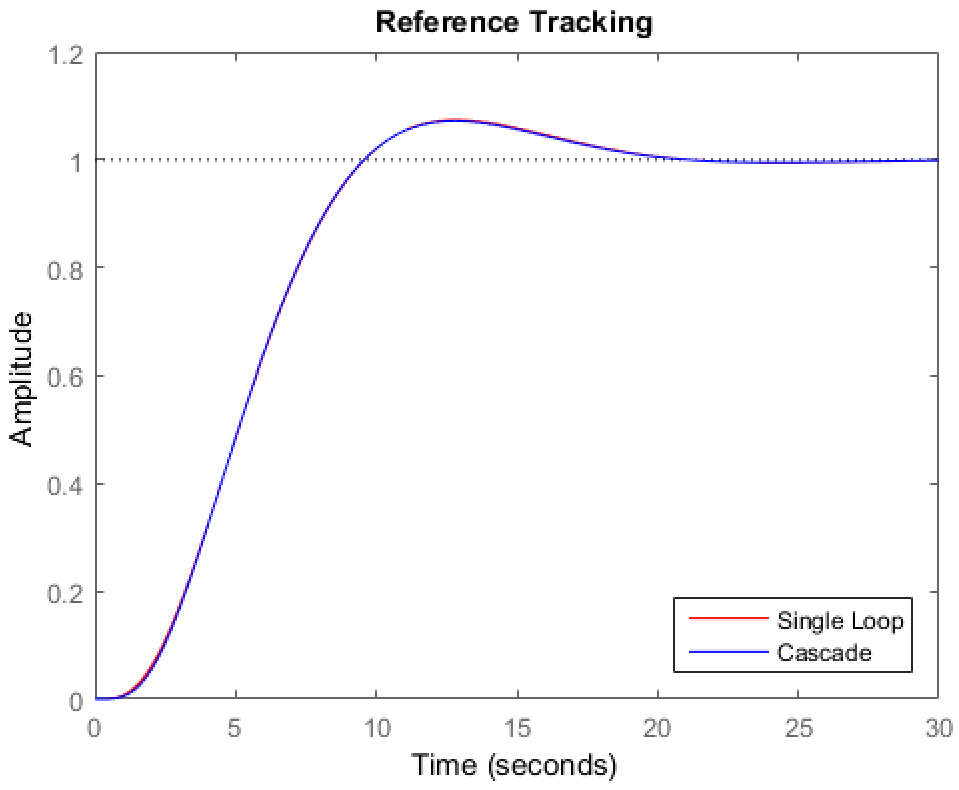
```
% Inner loop system when the control loop is closed first
clsys = feedback(P2*C2,1);
% Plant seen by the outer loop controller C1 is clsys*P1
C1 = pidtune(clsys*P1,pidstd(1,1),0.2);
C1


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
```

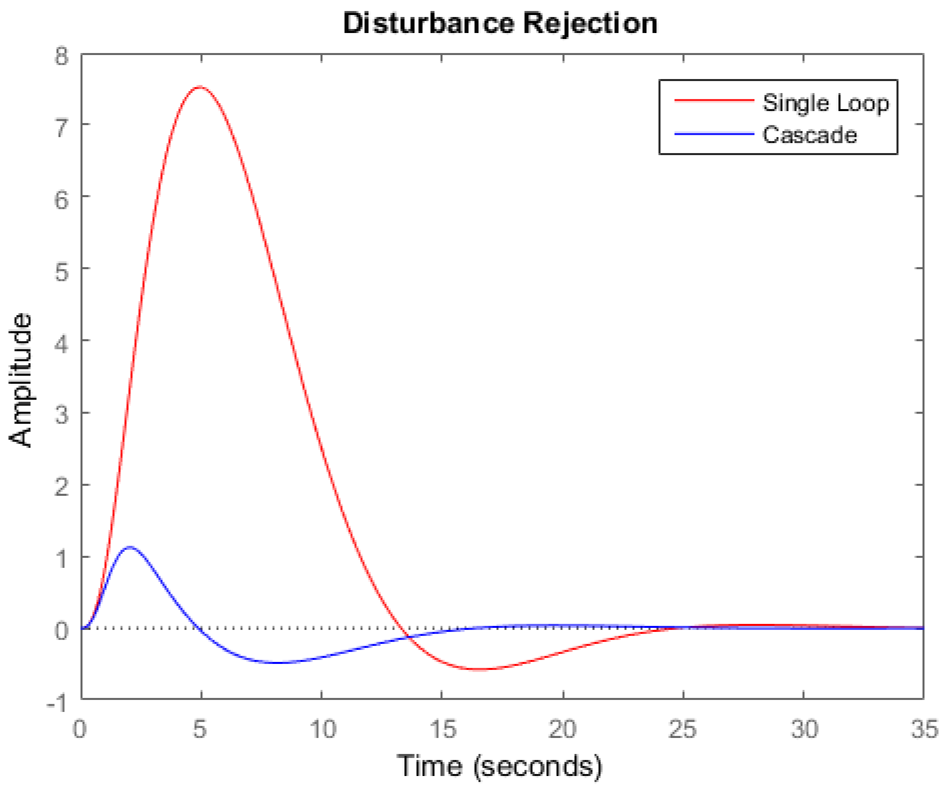
```
sys1 = feedback(P*C,1);
sys1.Name = 'Single Loop';
% cascade system for reference tracking
sys2 = feedback(clsys*P1*C1,1);
sys2.Name = 'Cascade';
% plot step response
figure;
step(sys1,'r',sys2,'b')
legend('show','location','southeast')
title('Reference Tracking')
```



Secondly, plot the step disturbance rejection responses of d2 for both control systems.

```
% 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')
```



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 9-2
- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (Command Line)”

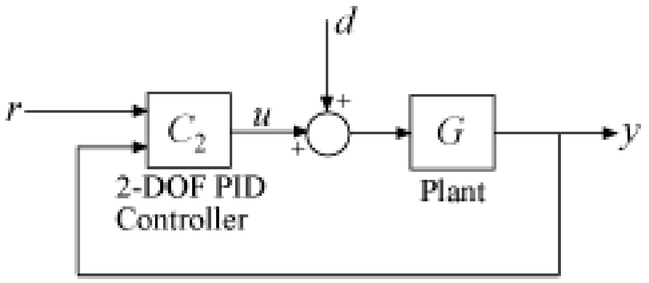
More About

- “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:

$$G(s) = \frac{1}{s^2 + 0.5s + 0.1}.$$

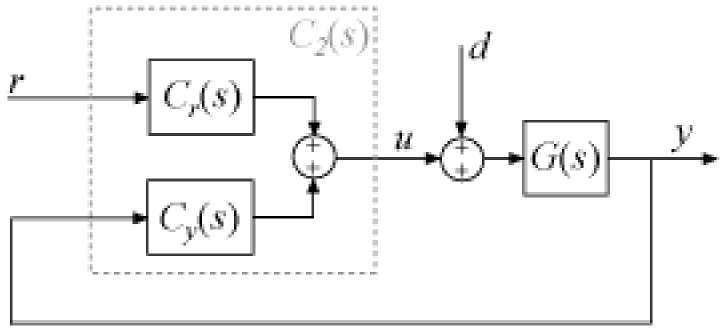
Suppose that your target bandwidth for the system is 1.5 rad/s.

```
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*.

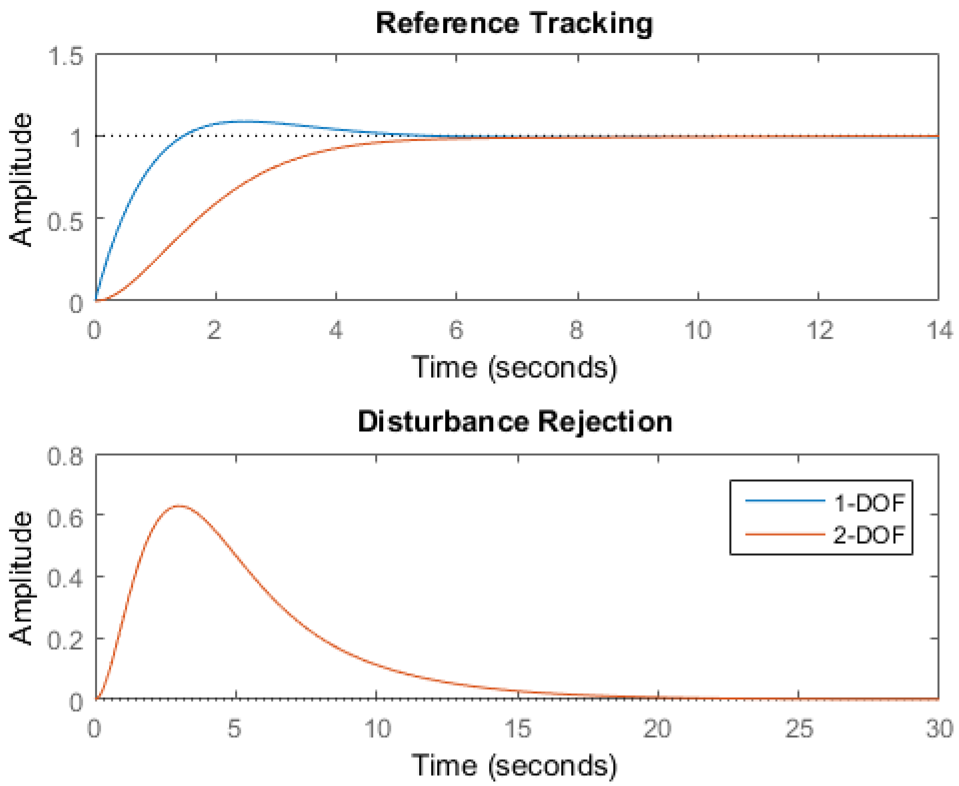
```
C2tf = tf(C2);
Cr = C2tf(1);
Cy = C2tf(2);
T2 = Cr*feedback(G,Cy,+1);
```

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.

```
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.

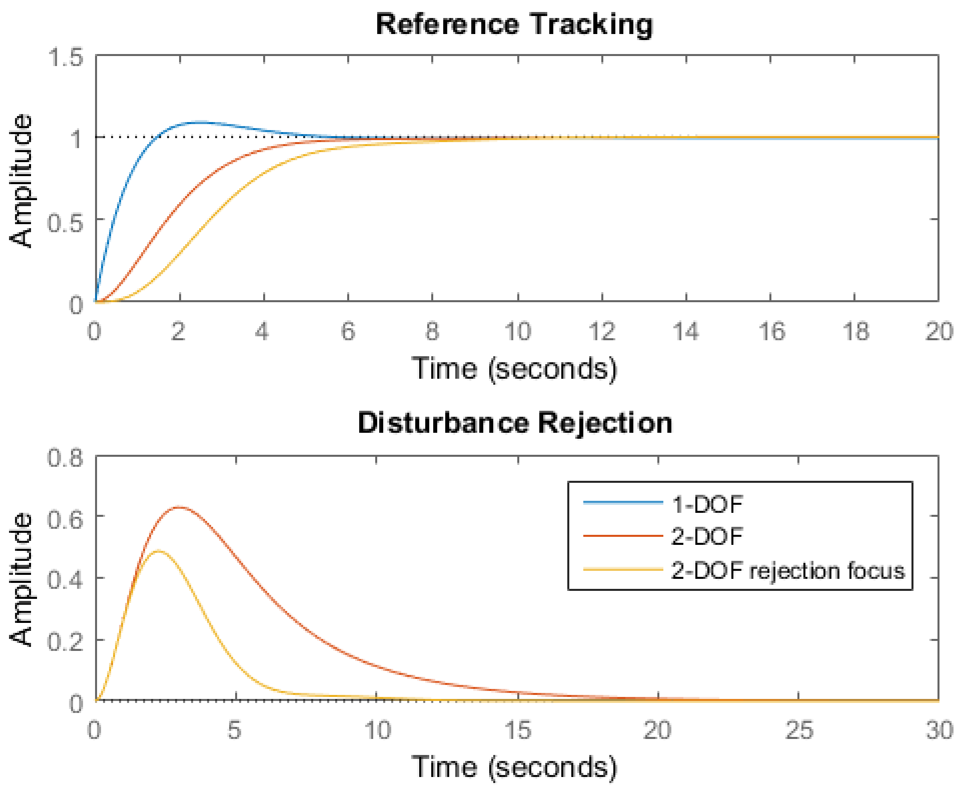
```
opt = pidtuneOptions('DesignFocus','disturbance-rejection');
C2dr = pidtune(G,'PID2',wc,opt)
```

```
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 9-22
- "Analyze Design in PID Tuner"

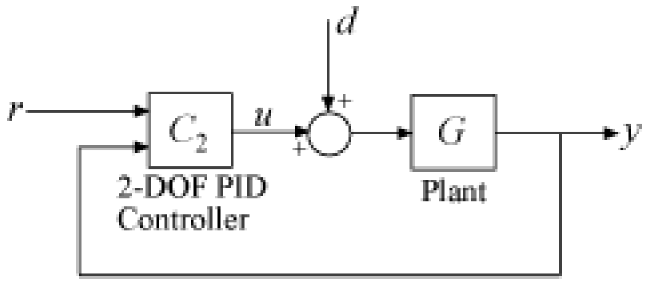
More About

- "Designing PID Controllers with the 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.

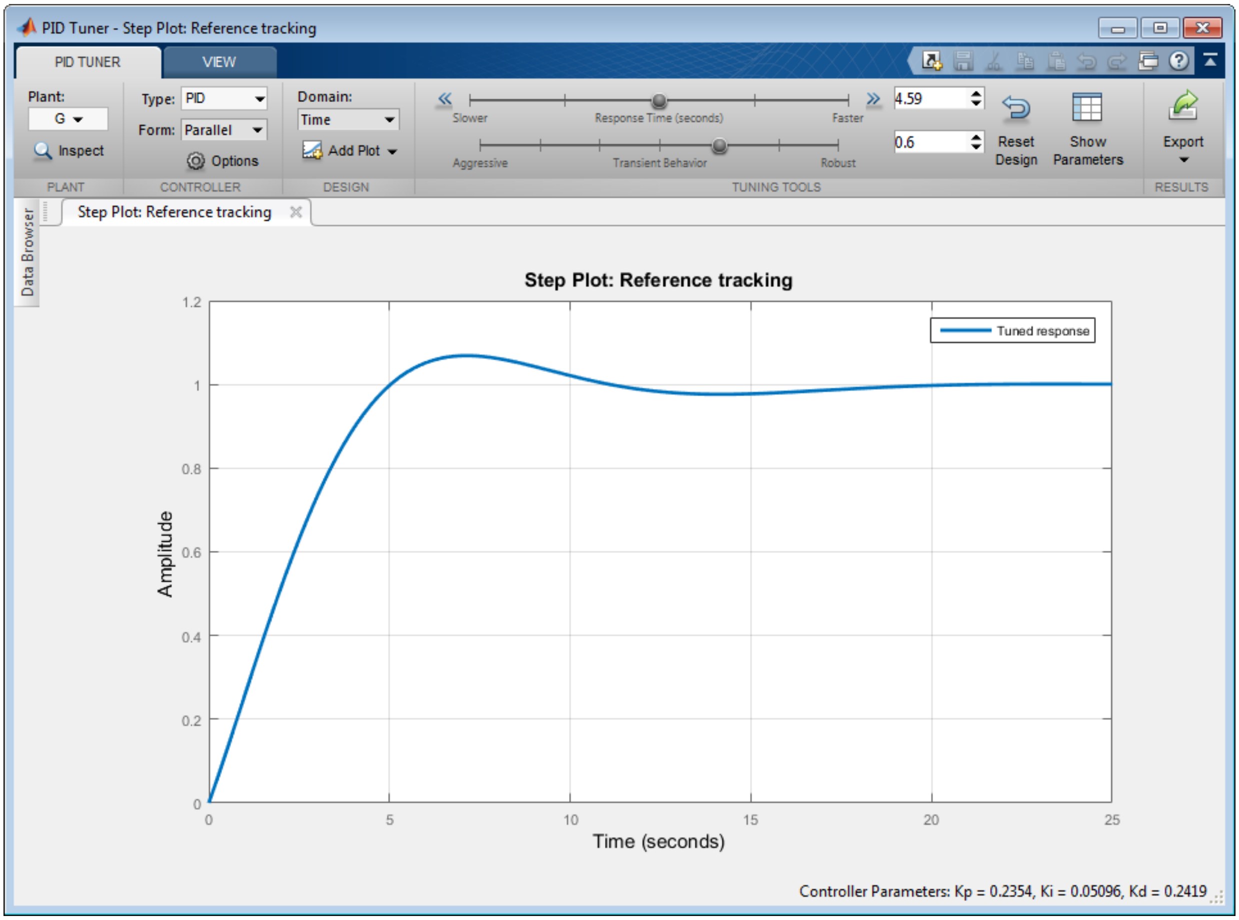
2-DOF PID controllers include setpoint weighting on the proportional and derivative terms. Compared to a 1-DOF PID controller, a 2-DOF PID controller can achieve better disturbance rejection without significant increase of overshoot in setpoint tracking. A typical control architecture using a 2-DOF PID controller is shown in the following diagram.



For this example, first design a 1-DOF controller for the plant given by:

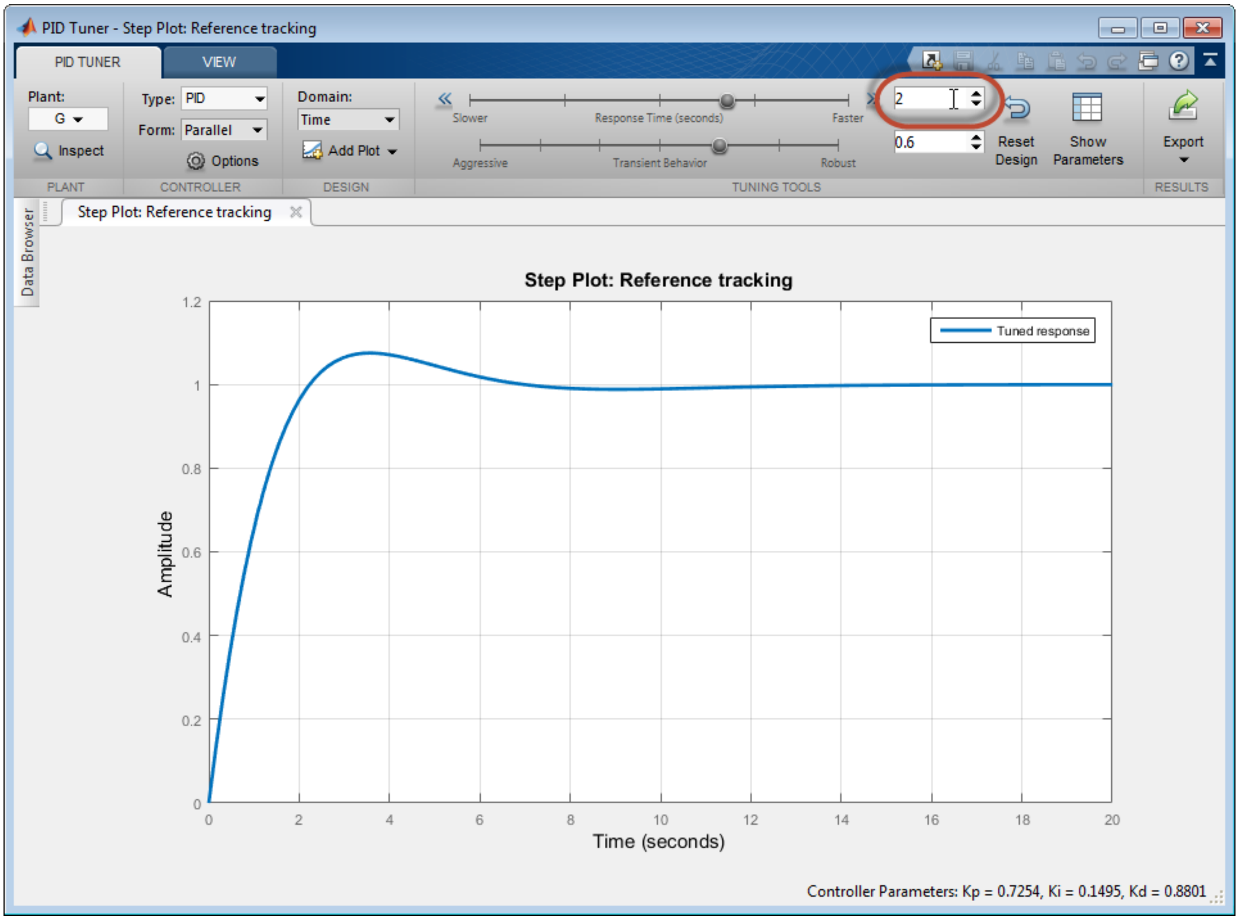
$$G(s) = \frac{1}{s^2 + 0.5s + 0.1}.$$

```
G = tf(1,[1 0.5 0.1]);
pidTuner(G,'PID')
```



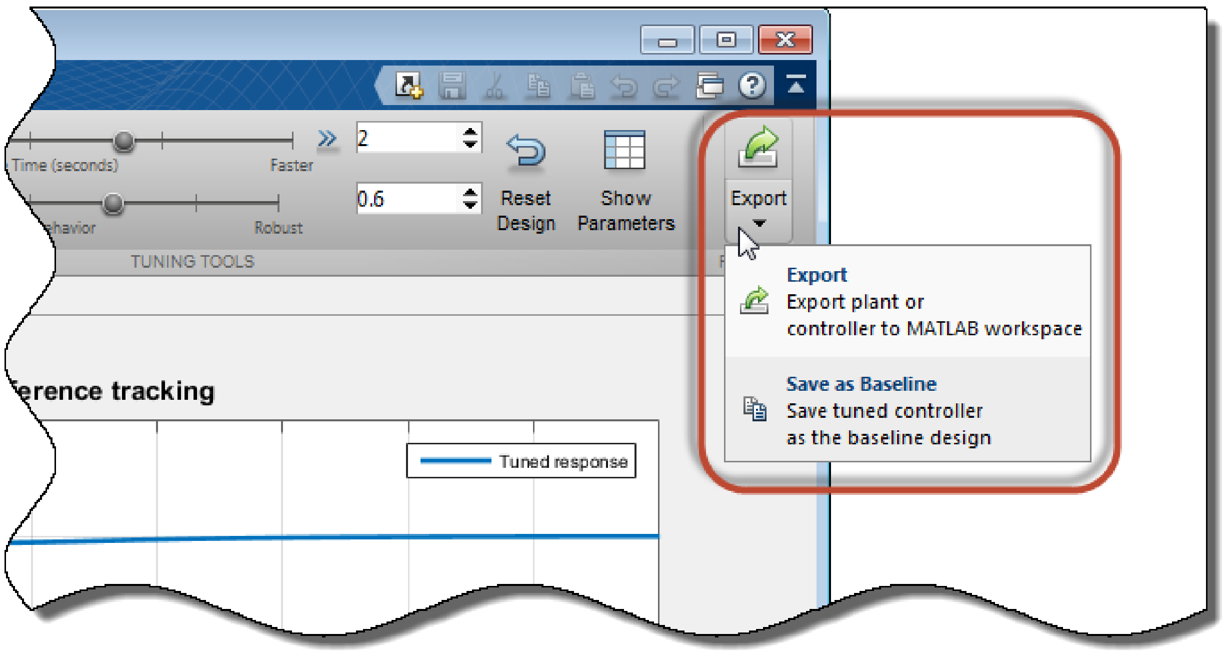
Controller Parameters: Kp = 0.2354, Ki = 0.05096, Kd = 0.2419

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.

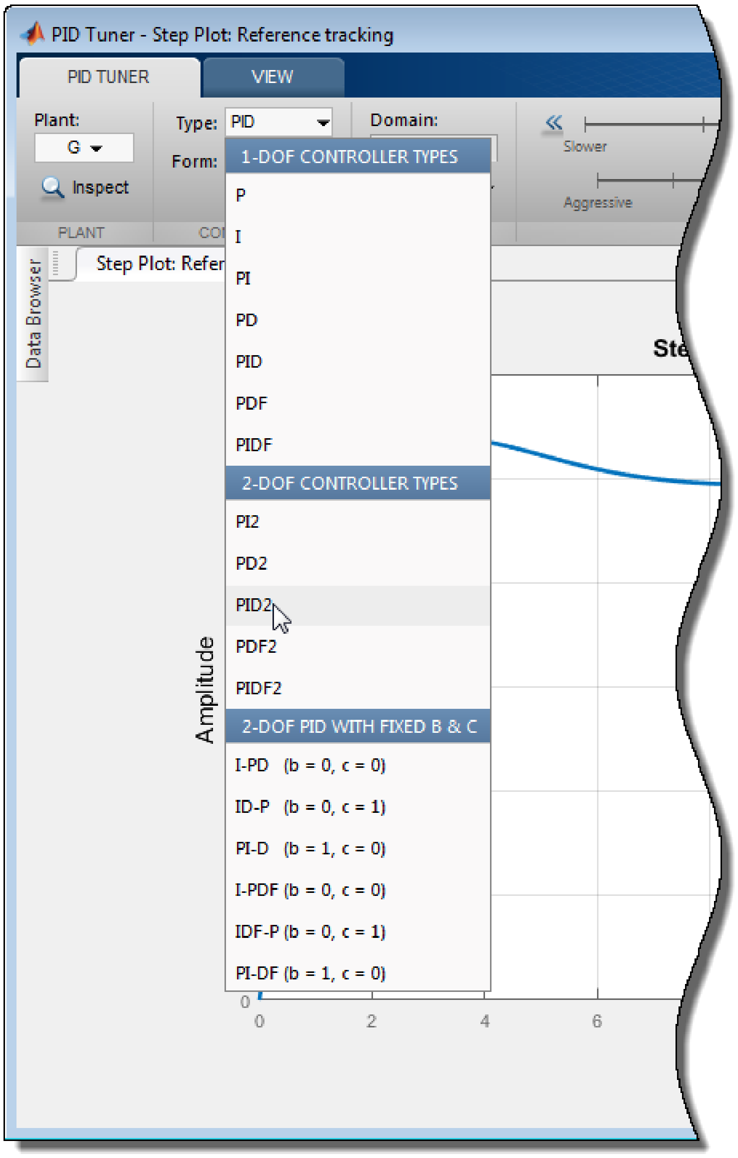


Controller Parameters: Kp = 0.7254, Ki = 0.1495, Kd = 0.8801

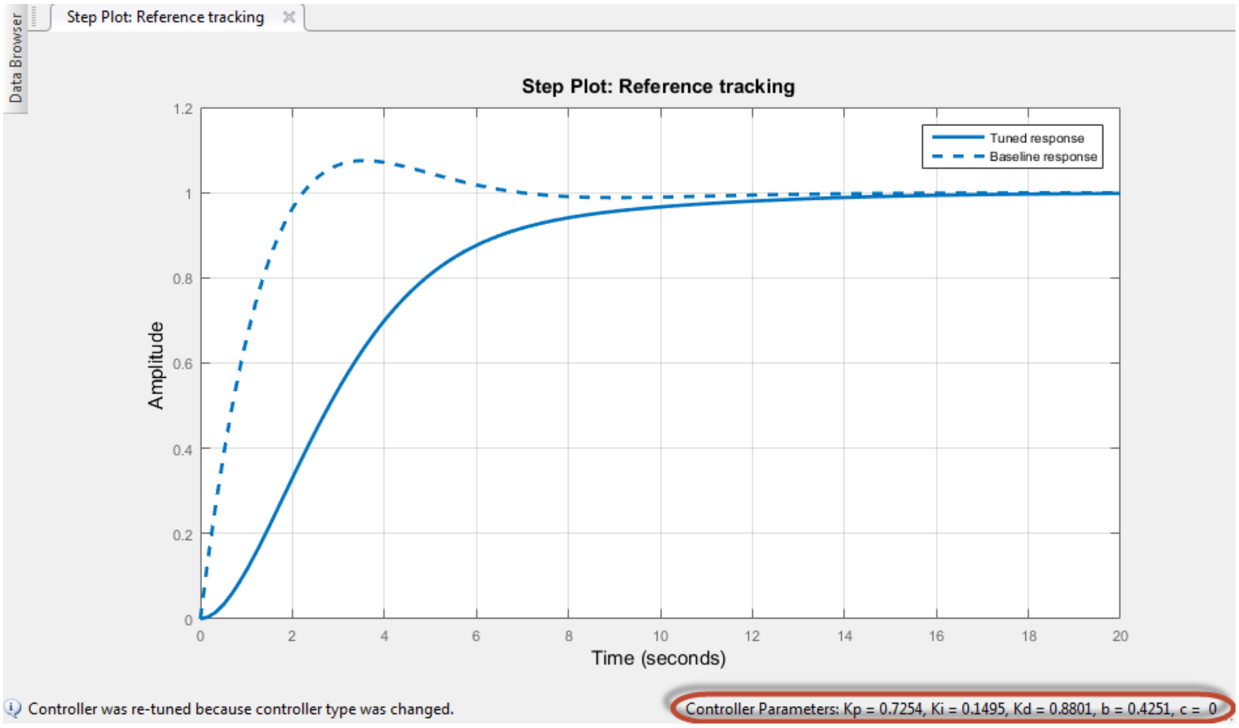
The resulting response is fast, but has a considerable amount of overshoot. Design a 2-DOF controller to improve the overshoot. First, set the 1-DOF controller as the baseline controller for comparison. Click the **Export** arrow ▾ and select `Save as Baseline`.



Design the 2-DOF controller. In the **Type** menu, select `PID2`.



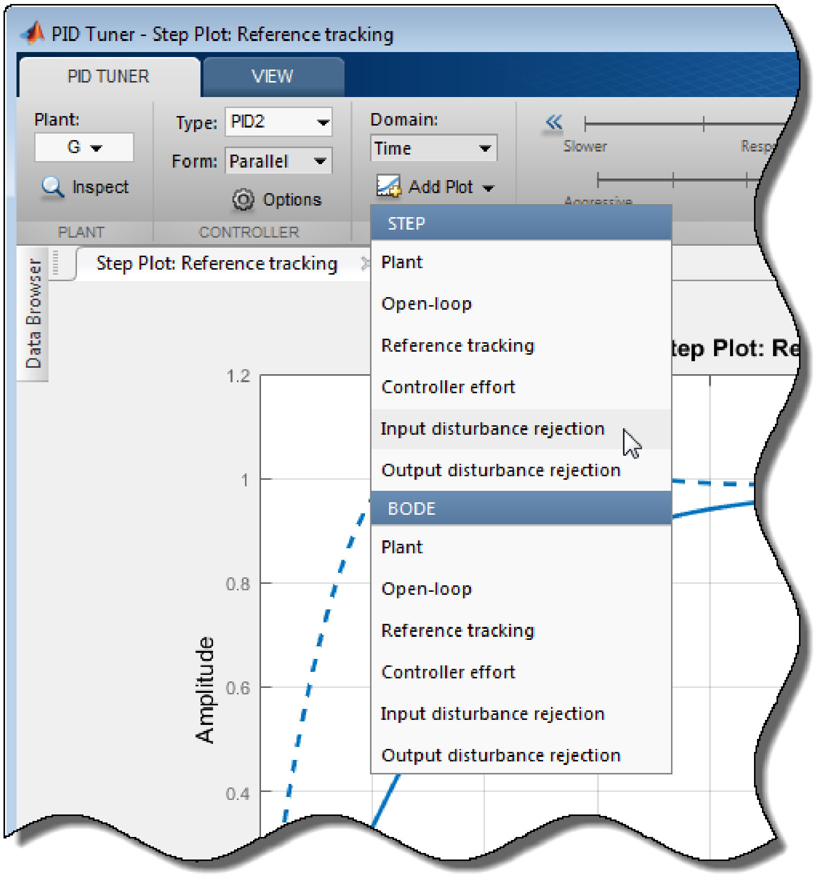
PID Tuner generates a 2-DOF controller with the same target response time. The controller parameters displayed at the bottom right show that PID Tuner tunes all controller coefficients, including the setpoint weights b and c, to balance performance and robustness. Compare the 2-DOF controller performance (solid line) with the performance of the 1-DOF controller that you stored as the baseline (dotted line).



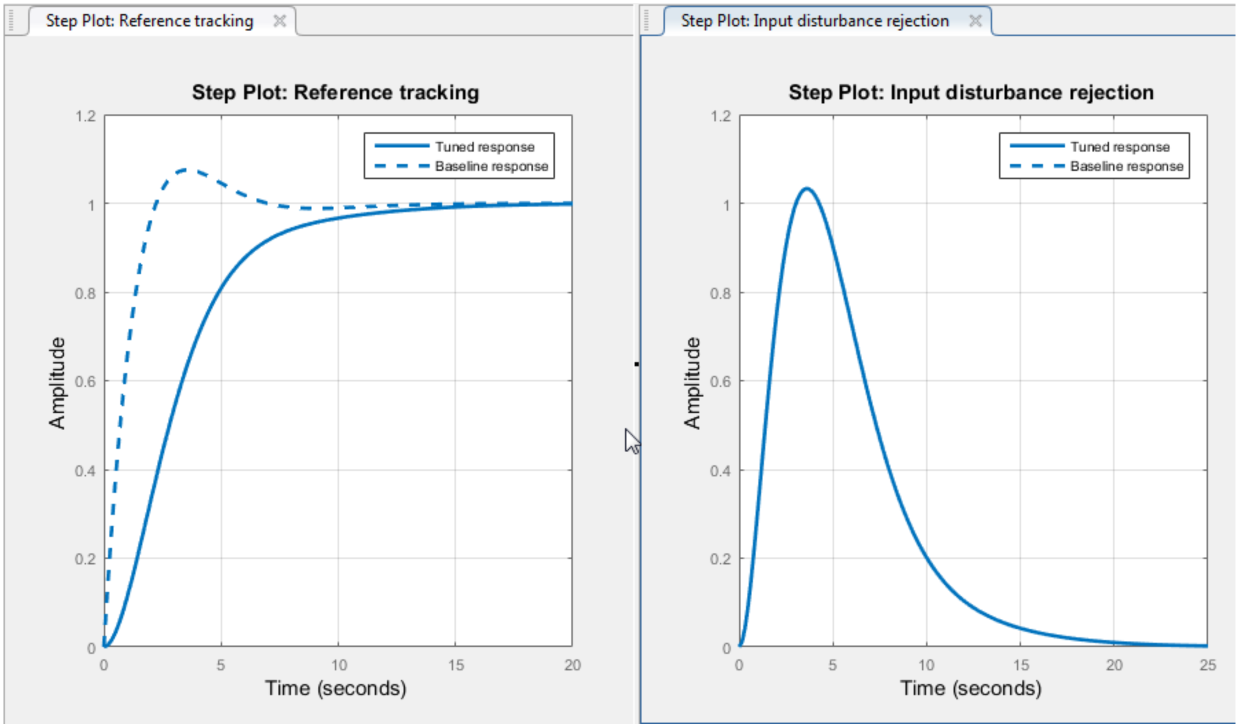
Controller was re-tuned because controller type was changed.

Controller Parameters: Kp = 0.7254, Ki = 0.1495, Kd = 0.8801, b = 0.4251, c = 0

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**.



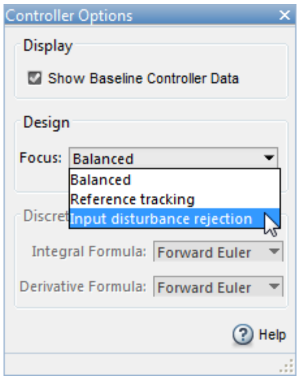
PID Tuner tiles the disturbance-rejection plot side by side with the reference-tracking plot.



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`.



Controller Options

Display

Show Baseline Controller Data

Design

Focus: Balanced

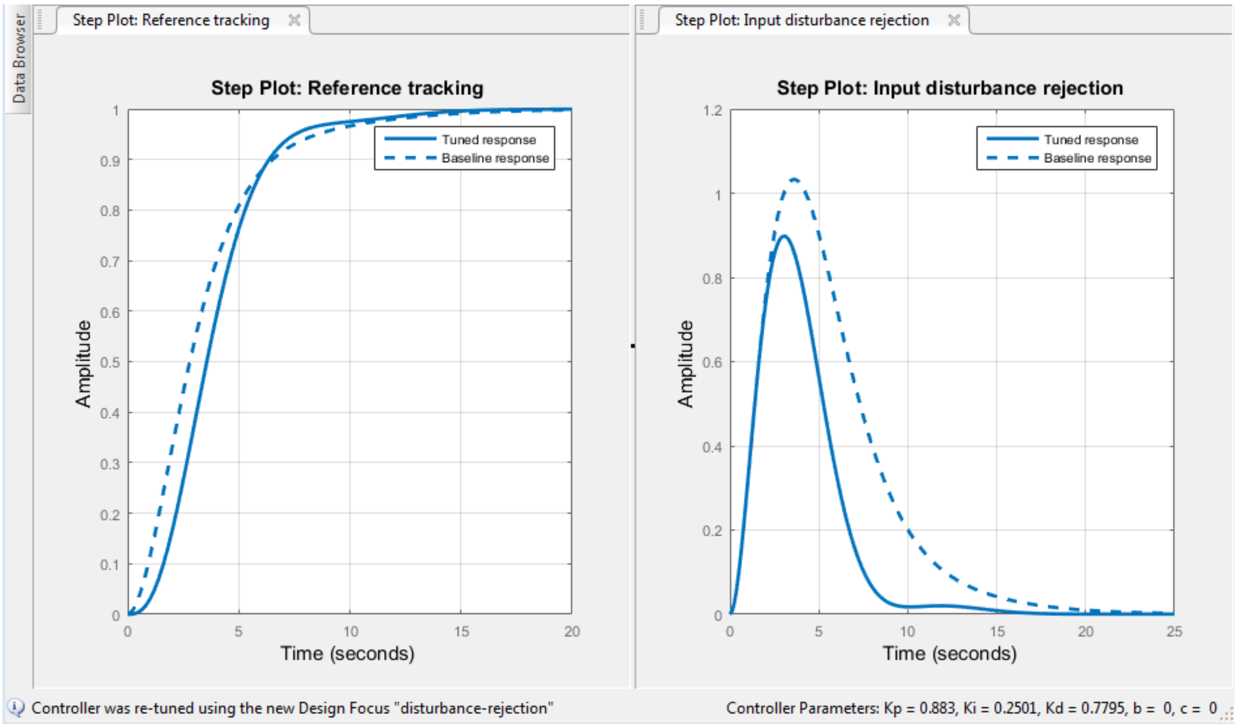
Balanced
Reference tracking
Input disturbance rejection

Discret

Integral Formula: Forward Euler

Derivative Formula: Forward Euler

PID Tuner automatically retunes the controller coefficients with a focus on disturbance-rejection performance.


Data Browser
Step Plot: Reference tracking
Step Plot: Reference tracking
Amplitude
Time (seconds)
Tuned response
Baseline response
Step Plot: Input disturbance rejection
Step Plot: Input disturbance rejection
Amplitude
Time (seconds)
Tuned response
Baseline response

Controller was re-tuned using the new Design Focus "disturbance-rejection"

Controller Parameters: Kp = 0.883, Ki = 0.2501, Kd = 0.7795, b = 0, c = 0

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 9-16
- “Analyze Design in PID Tuner”

More About

- “Designing PID Controllers with the PID Tuner”
- “Two-Degree-of-Freedom PID Controllers” on page 2-17

PID Controller Types for Tuning

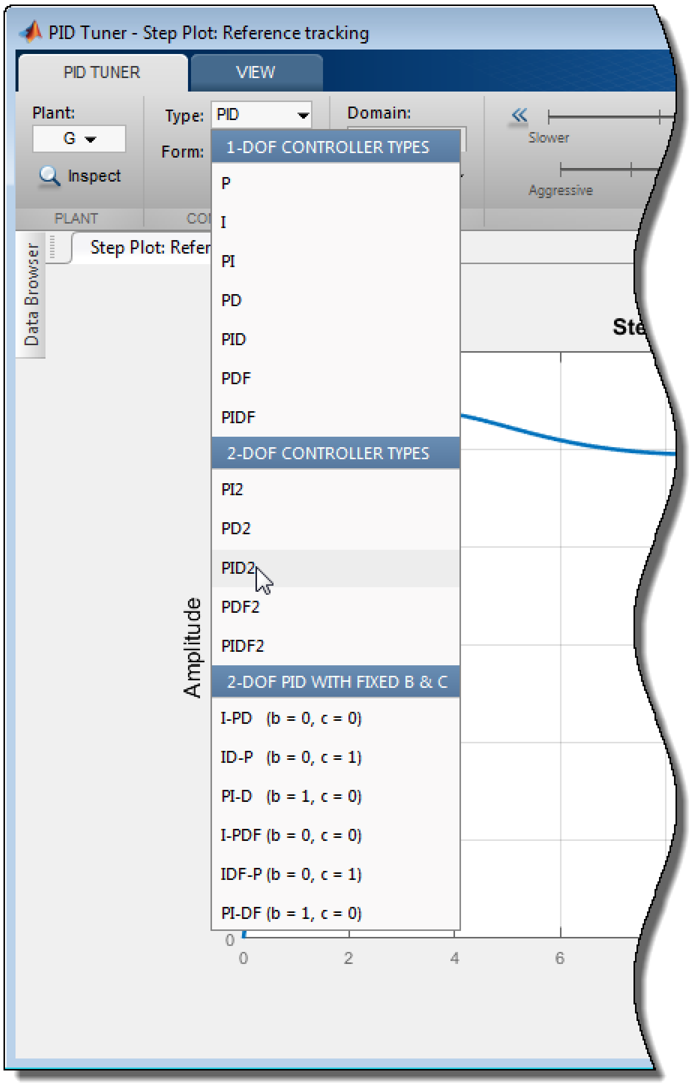
In this section...

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.


PID Tuner - Step Plot: Reference tracking
PID TUNER
VIEW
Plant:
G
Inspect
PLANT
Type: PID
Form:
Domain:
Slower
Aggressive
Data Browser
Step Plot: Refer
1-DOF CONTROLLER TYPES
P
I
PI
PD
PID
PDF
PIDF
2-DOF CONTROLLER TYPES
PI2
PD2
PID2
PDF2
PIDF2
2-DOF PID WITH FIXED B & C
I-PD (b = 0, c = 0)
ID-P (b = 0, c = 1)
PI-D (b = 1, c = 0)
I-PDF (b = 0, c = 0)
IDF-P (b = 0, c = 1)
PI-DF (b = 1, c = 0)
Amplitude
0
0
2
4
6

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	$\frac{K_i}{s}$	$K_i \frac{T_s}{z-1}$
`PI`	Proportional and integral	$K_p + \frac{K_i}{s}$	$K_p + K_i \frac{T_s}{z-1}$
`PD`	Proportional and derivative	$K_p + K_d s$	$K_p + K_d \frac{z-1}{T_s}$
`PDF`	Proportional and derivative with first-order filter on derivative term	$K_p + \frac{K_d s}{T_f s + 1}$	$K_p + K_d \frac{1}{T_f + \frac{T_s}{z-1}}$
`PID`	Proportional, integral, and derivative	$K_p + \frac{K_i}{s} + K_d s$	$K_p + K_i \frac{T_s}{z-1} + K_d \frac{z-1}{T_s}$
`PIDF`	Proportional, integral, and derivative with first-order filter on derivative term	$K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}$	$K_p + K_i \frac{T_s}{z-1} + K_d \frac{1}{T_f + \frac{T_s}{z-1}}$

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	$u = K_p(br-y) + \frac{K_i}{s}($	$u = K_p(br-y) + K_i\frac{T_s}{z-1}(r-y)$
`PD2`	2-DOF proportional and derivative	$u = K_p(br-y) + K_d s$	$u = K_p(br-y) + K_d\frac{z-1}{T_s}(cr-y)$
`PDF2`	2-DOF proportional and derivative with first-order filter on derivative term	$u = K_p(br-y) + K_d$	$u = K_p(br-y) + K_d\frac{1}{T_f+\frac{T_s}{z-1}}(cr$
`PID2`	2-DOF proportional, integral, and derivative	$u = K_p(br-y) + \frac{K_i}{s}($	$u = K_p(br-y) + K_i\frac{T_s}{z-1}(r-y) +$
`PIDF2`	2-DOF proportional, integral, and derivative with first-order filter on derivative term	$u = K_p(br-y) + \frac{K_i}{s}($	$u = K_p(br-y) + K_i\frac{T_s}{z-1}(r-y) +$

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	$u = -K_p y + \frac{K_i}{s}(r$	$u = -K_p y + K_i \frac{T_s}{z-1}(r-y) - K$
`I-PDF`	2-DOF PIDF with b = 0, c = 0	$u = -K_p y + \frac{K_i}{s}(r$	$u = -K_p y + K_i \frac{T_s}{z-1}(r-y) - K$
`ID-P`	2-DOF PID with b = 0, c = 1	$u = -K_p y + \frac{K_i}{s}(r$	$u = -K_p y + K_i \frac{T_s}{z-1}(r-y) + K$
`IDF-P`	2-DOF PIDF with b = 0, c = 1	$u = -K_p y + \frac{K_i}{s}(r$	$u = -K_p y + K_i \frac{T_s}{z-1}(r-y) + K$
`PI-D`	2-DOF PID with b = 1, c = 0	$u = K_p(r-y) + \frac{K}{s}$	$u = K_p(r-y) + K_i \frac{T_s}{z-1}(r-y$
`PI-DF`	2-DOF PIDF with b = 1, c = 0	$u = K_p(r-y) + \frac{K}{s}$	$u = K_p(r-y) + K_i \frac{T_s}{z-1}(r-y$

See Also

`pidtune` | `pidTuner`

Related Examples

- “PID Controller Design at the Command Line” on page 9-2
- “PID Controller Design for Fast Reference Tracking”
- “Tune 2-DOF PID Controller (Command Line)” on page 9-16
- “Tune 2-DOF PID Controller (PID Tuner)” on page 9-22

More About

- “Designing PID Controllers with the PID Tuner”
- “Proportional-Integral-Derivative (PID) Controllers” on page 2-14
- “Two-Degree-of-Freedom PID Controllers” on page 2-17

10

Single-Input, Single-Output Control Design

Getting Started with the SISO Design Tool

This example shows how the SISO Design Tool facilitates the compensator design process by providing interactive and automated tools to tune compensators for a feedback control system.

Compensator Design Task and the SISO Design Tool

The SISO Design Tool allows:

1) The design problem to be setup graphically by defining the control design requirement on time, frequency, and pole/zero response plots.

2) Tuning the compensator with:

- automated design methods such as Ziegler Nichols, IMC, and LQG.
- graphically tuning poles and zeros on design plots such as Bode and root locus.
- optimization to meet time and frequency-domain requirements using Simulink® Design Optimization™.

3) While tuning the compensators, the closed-loop and open-loop responses are dynamically updated to display the performance of the control system.

The design process using the SISO Design Tool will be illustrated with an example problem.

Compensator Design Problem Example

For this example we will design a compensator for the system

$$G(s) = \frac{1}{s+1}$$

with the following design requirements:

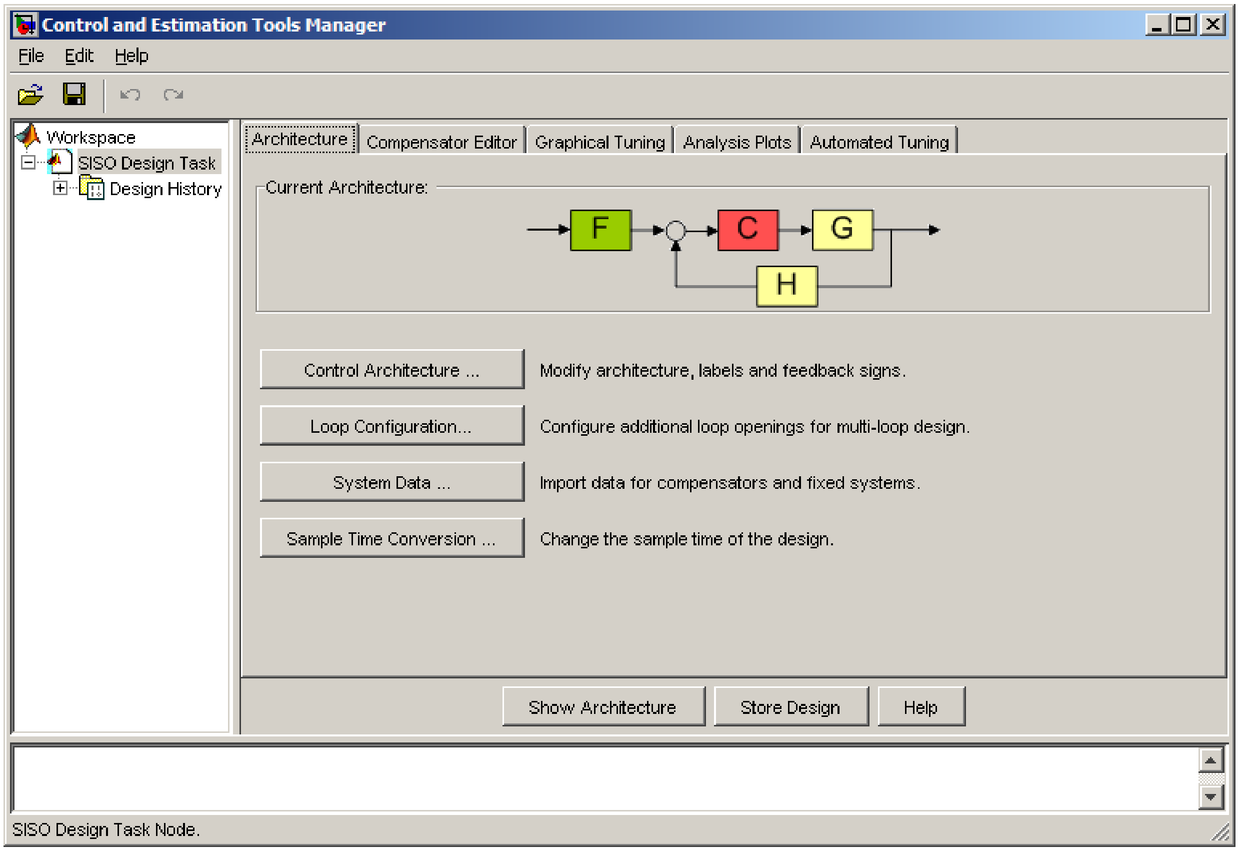
- Zero steady state error with respect to a step input.
- 80% rise time < 1 second.
- Settling time < 2 seconds.
- Maximum overshoot < 20%.
- Open-loop crossover constraint of less than 5 rad/s.

Launching the SISO Design Tool and Configuring Design Objectives

For this example we will use the standard feedback structure with the controller in the forward path which happens to be the default feedback structure when launching the SISO Design Tool. To launch the SISO Design Tool with the specified plant G type

```
>> sisotool(tf(1,[1,1]))
```

This will bring up two windows. The first window is Control and Estimation Tools Manager (CETM)



Modify architecture, labels and feedback signs.

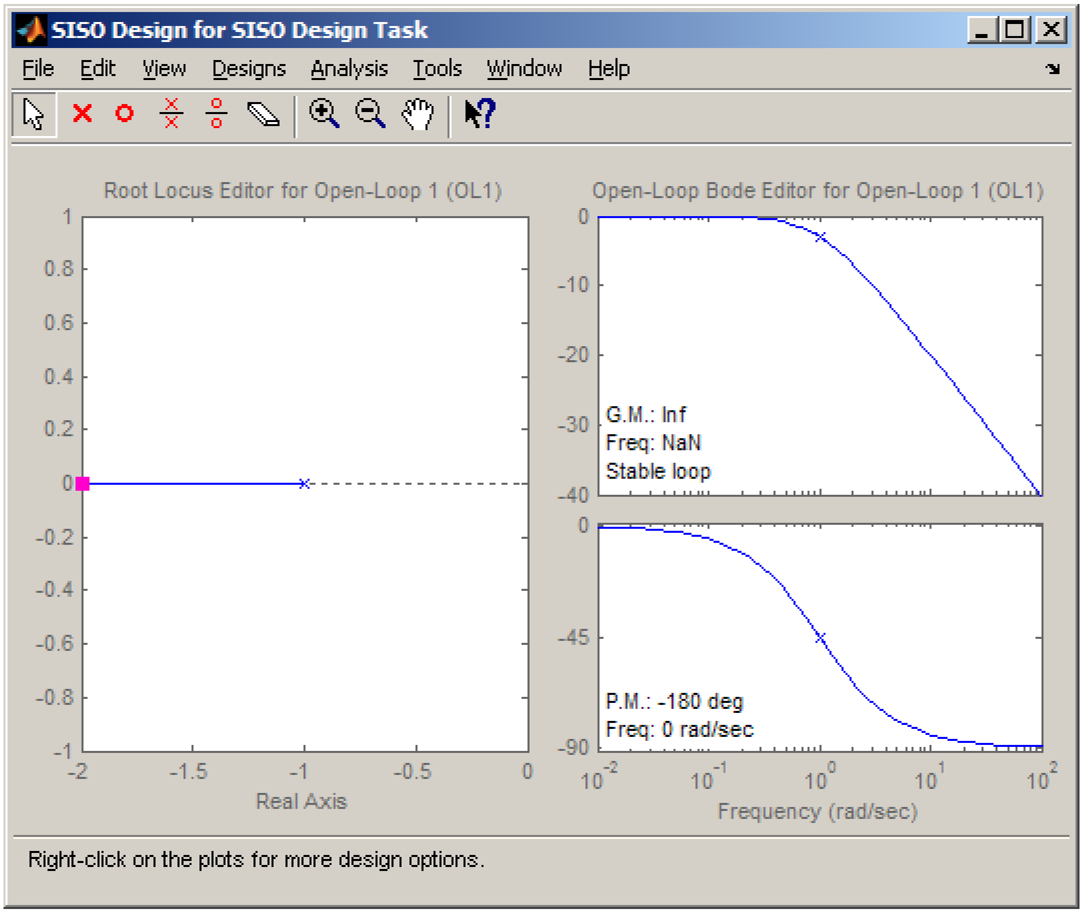
Configure additional loop openings for multi-loop design.

Import data for compensators and fixed systems.

Change the sample time of the design.

SISO Design Task Node.

and the second window is the SISO Design graphical editors

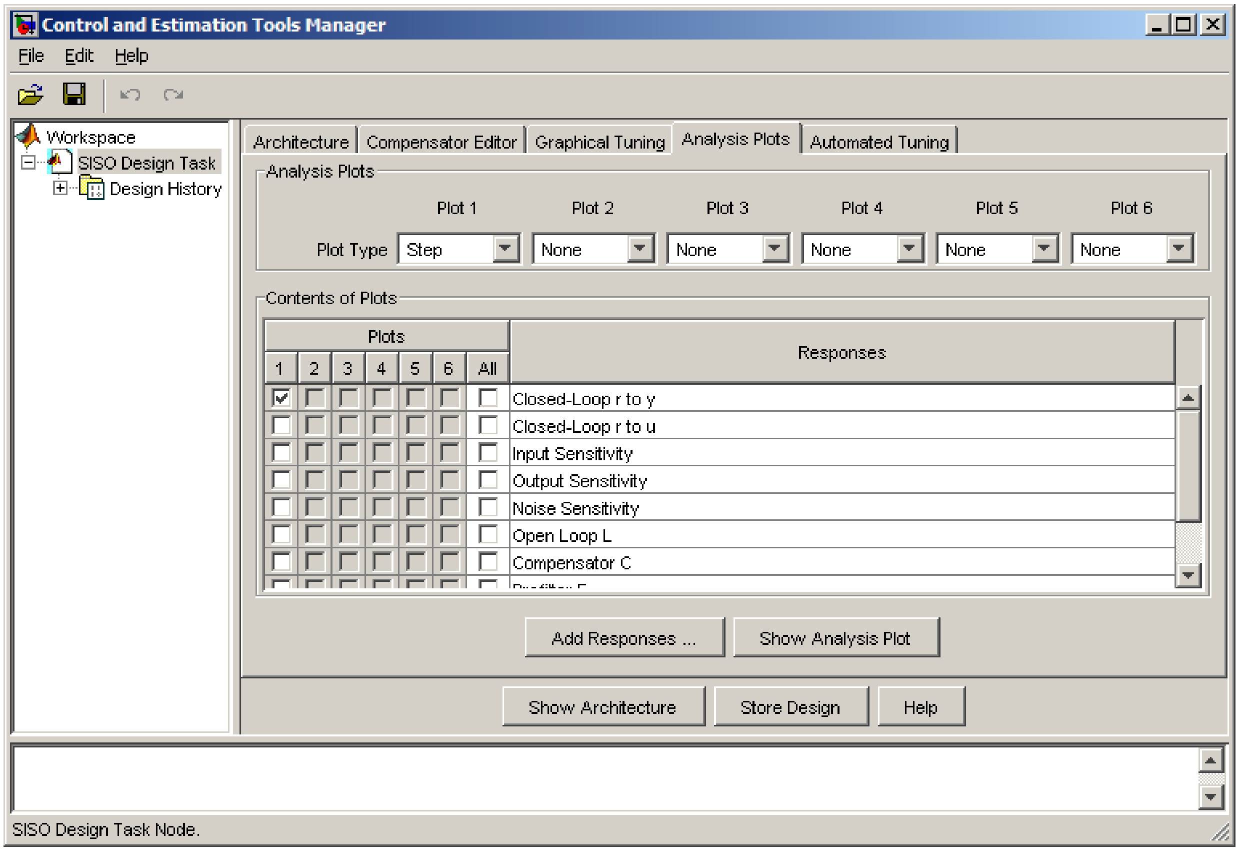

SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Root Locus Editor for Open-Loop 1 (OL1)
Open-Loop Bode Editor for Open-Loop 1 (OL1)
Real Axis
Frequency (rad/sec)
G.M.: Inf
Freq: NaN
Stable loop
P.M.: -180 deg
Freq: 0 rad/sec

Right-click on the plots for more design options.

In the CETM the **SISO Design Task** node contains tabbed panels which are used to configure the compensator design options as well as manipulate the compensators. For complete details of the functionality for each of the panels refer to the documentation.

For this design example we will use the root-locus plot and open-loop Bode plot for graphically tuning the compensator and validate the design by viewing the step response.

To view the closed-loop step response, click on the **Analysis Plot** tab in the CETM. Now configure the plot by selecting "Step" for the first plot and checking the first check box for the response "Closed-Loop r to y". This will bring up the SISO Tool Viewer.



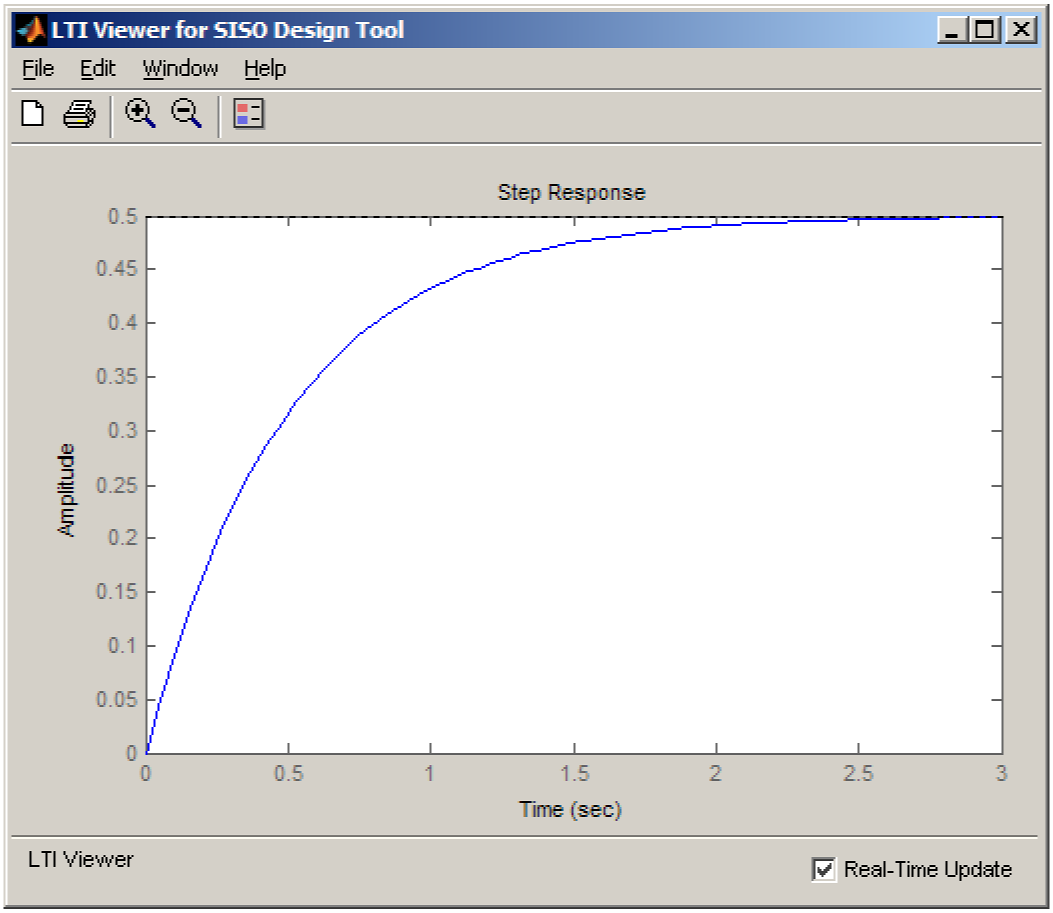
Analysis Plots

	Plot 1	Plot 2	Plot 3	Plot 4	Plot 5	Plot 6
Plot Type	Step	None	None	None	None	None

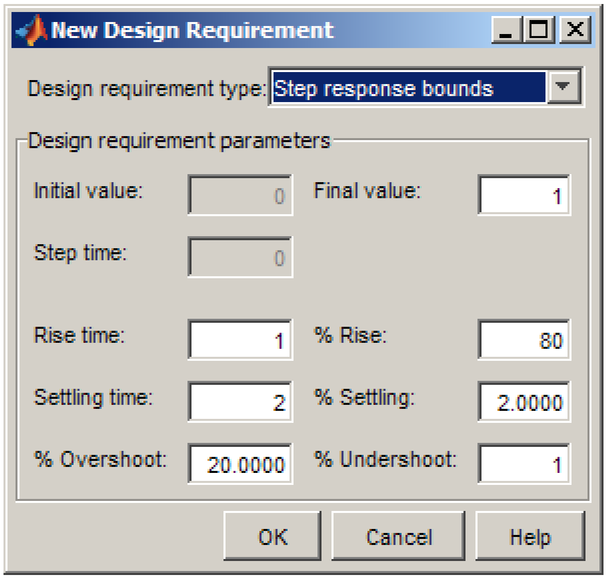
Contents of Plots

Plots							Responses
1	2	3	4	5	6	All	
✓							Closed-Loop r to y
							Closed-Loop r to u
							Input Sensitivity
							Output Sensitivity
							Noise Sensitivity
							Open Loop L
							Compensator C

SISO Design Task Node.



Now add the time domain design requirements to the step response plot by right clicking on the axis and selecting the **Design Requirements -> New** menu item. We will use the "Step response bounds" design requirement type to specify the rise time, settling time and overshoot requirements.

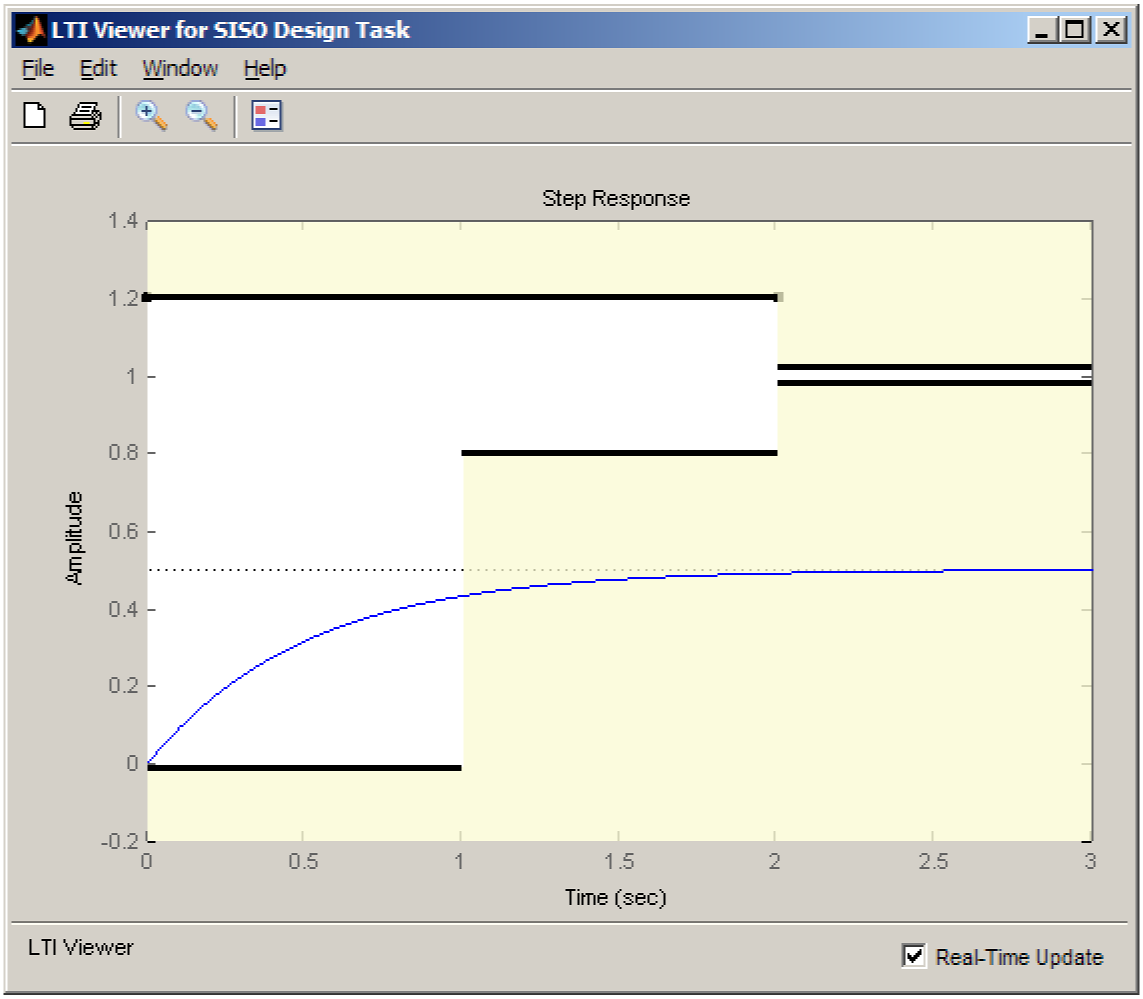


Design requirement type: Step response bounds

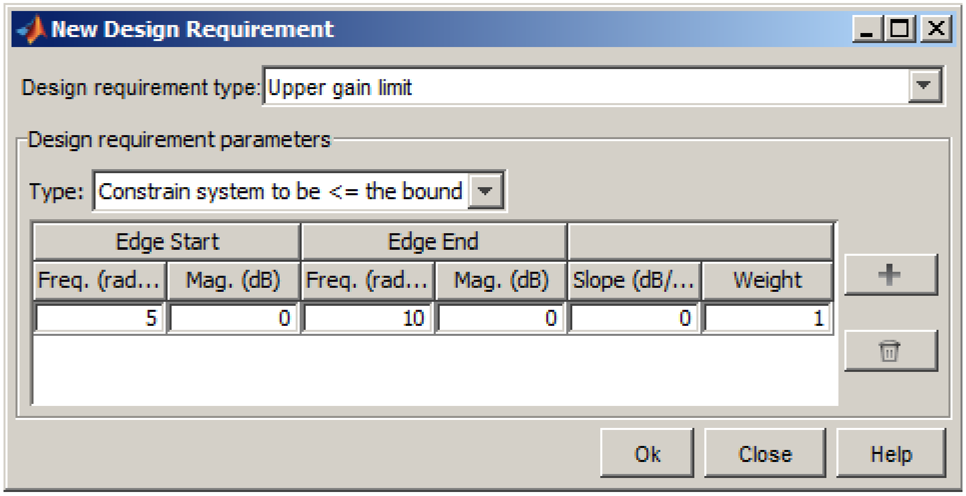
Design requirement parameters

Initial value:	0	Final value:	1
Step time:	0		
Rise time:	1	% Rise:	80
Settling time:	2	% Settling:	2.0000
% Overshoot:	20.0000	% Undershoot:	1

We can now use this time response with its requirements to view the performance of the compensator design.



To specify the frequency domain crossover requirement, right click the bode axis in the SISO Design window and select the **Design Requirement->New** menu item and specify an upper gain limit.



Design requirement type: Upper gain limit

Design requirement parameters

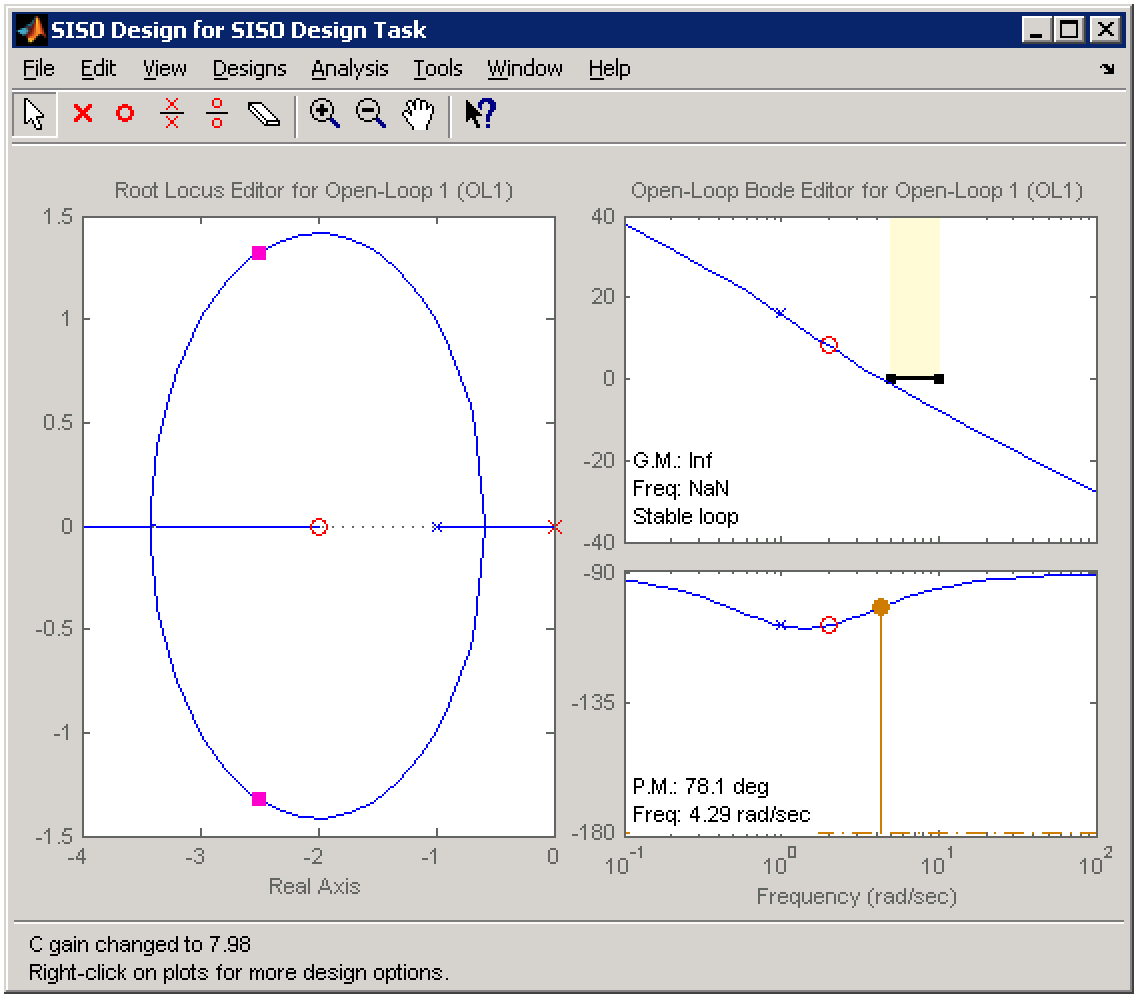
Type: Constrain system to be <= the bound

Edge Start		Edge End			
Freq. (rad...	Mag. (dB)	Freq. (rad...	Mag. (dB)	Slope (dB/...	Weight
5	0	10	0	0	1

Now that the problem has been set up we will begin to design the compensator to satisfy the problem specifications.

Tuning Compensators

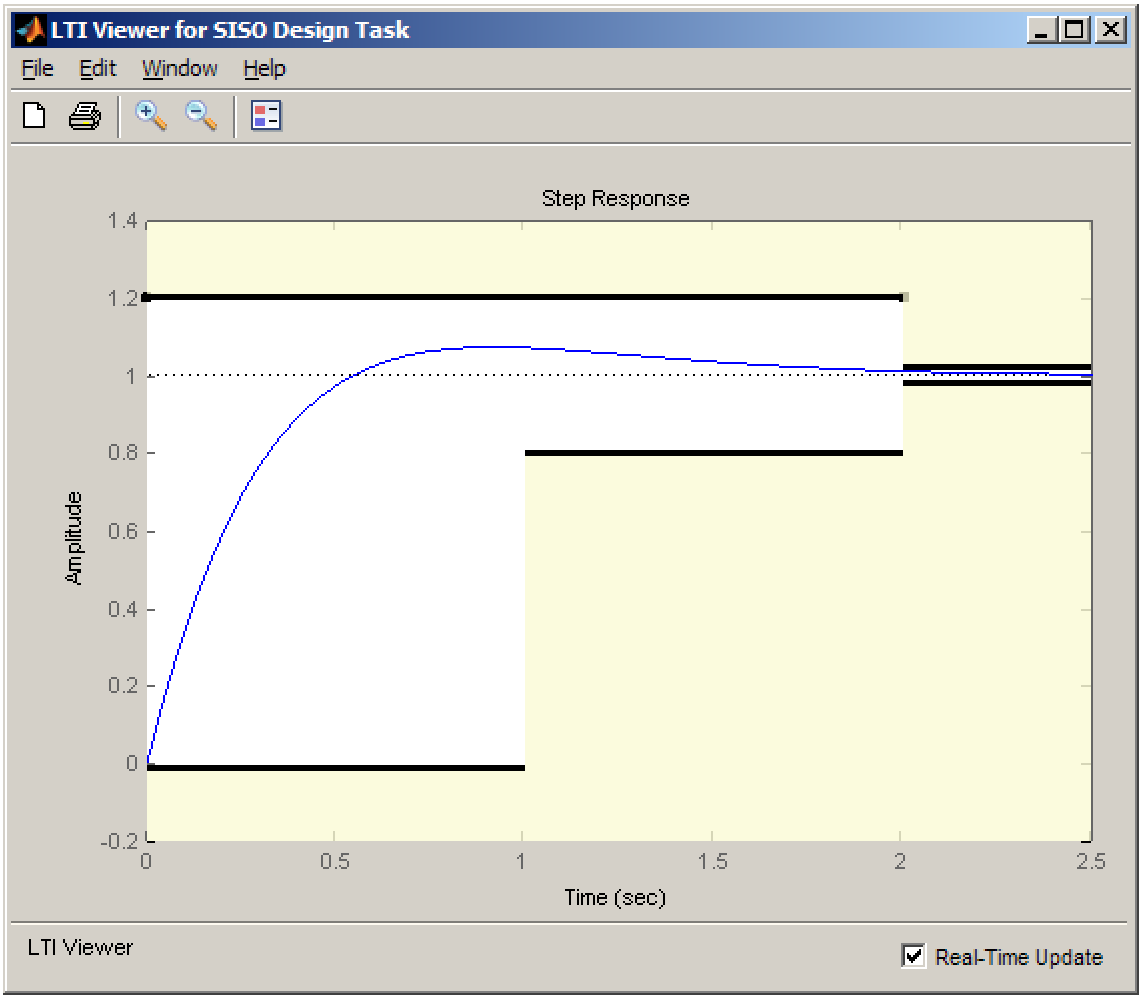
Compensators can be manually tuned from the graphical editors or the **Compensator Editor** tab of the CETM. For this example we will use the graphical editors to tune the compensator. To begin the design an integrator will be added to achieve zero steady state error to a step input. To add the integrator to the compensator use the right-click menu on the root-locus plot and select **Add Pole/Zero->Integrator**. To create a desirable shape for the root locus plot we will add a zero at approximately -2. To add the zero, use the right-click menu on the root-locus plot and select **Add Pole/Zero->Real Zero** menu item and then left-click at approximately -2 on the real axis of the root locus plot. Now in the bode plot adjust the open-loop gain by clicking and dragging the curve on the magnitude plot such that the cross-over and time domain constraints are satisfied.


SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Root Locus Editor for Open-Loop 1 (OL1)
Real Axis
Open-Loop Bode Editor for Open-Loop 1 (OL1)
Frequency (rad/sec)

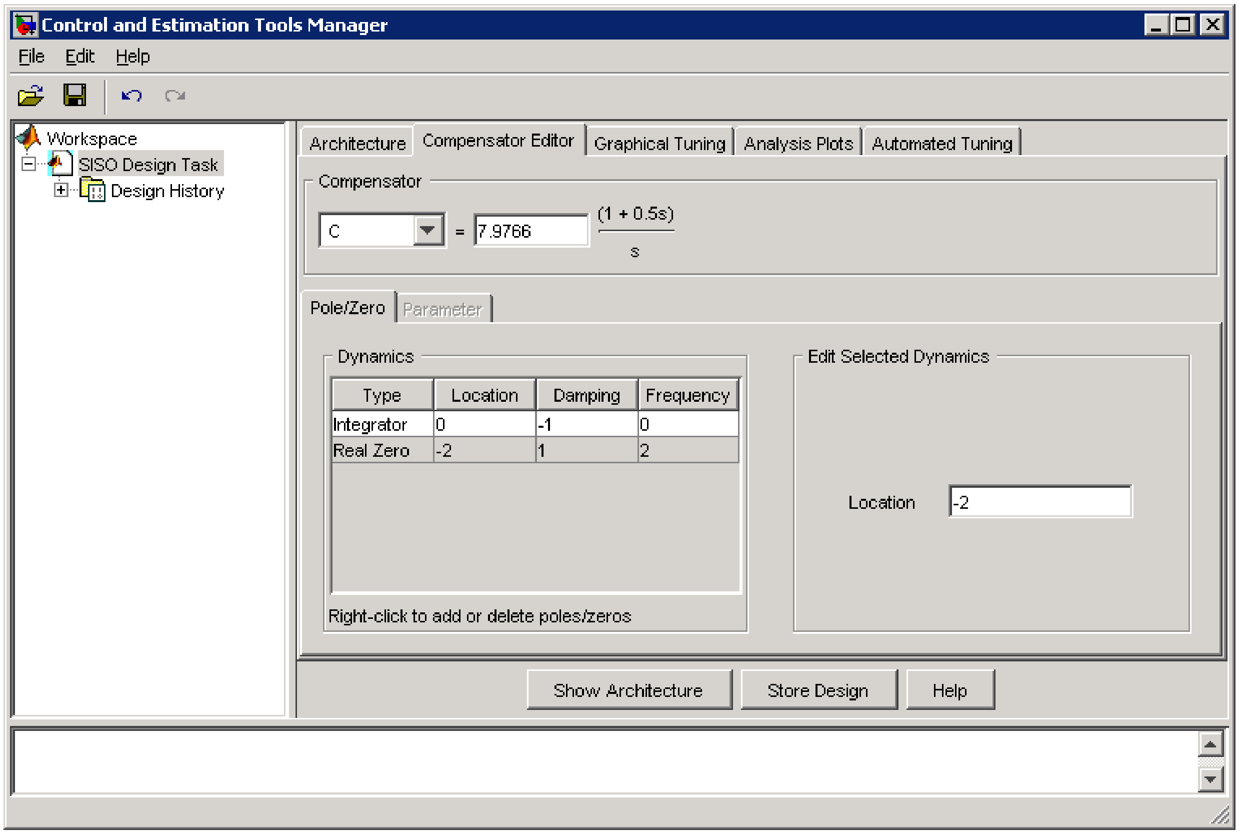
G.M.: Inf
Freq: NaN
Stable loop

P.M.: 78.1 deg
Freq: 4.29 rad/sec

C gain changed to 7.98
Right-click on plots for more design options.



To view the compensator go to **Compensator Editor** tab. Note that the steps performed in the graphical tuning plots to tune the compensator can also be accomplished from this panel.



Compensator: C = 7.9766 $\frac{(1 + 0.5s)}{s}$

Dynamics

Type	Location	Damping	Frequency
Integrator	0	-1	0
Real Zero	-2	1	2

Right-click to add or delete poles/zeros

Edit Selected Dynamics

Location: -2

Automated Tuning of Compensators

In addition to the manual tuning interfaces, the SISO Design Tool also provides the following automated tuning algorithms:

- Use the **PID tuning**, **IMC tuning**, and **LQG synthesis** options in the **Automated Tuning** panel to compute initial parameters for the compensators based on tuning parameters such as closed-loop time constants. See the example "Automated Controller Design in the SISO Design Tool".
- Use the **Optimization based tuning** option in the **Automated Tuning** panel (requires Simulink Design Optimization) to tune the compensators using both time and frequency domain design requirements. See the example "DC Motor Controller Tuning".

Summary

Using the SISO Design Tool we were able to successfully design a compensator such that all of the specified design requirements were satisfied. The tool facilitated the heuristic process of compensator design by providing an interactive and visual environment for

- Specifying the design requirements
- Tuning the compensator, and
- Evaluating the performance of the design.

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*.

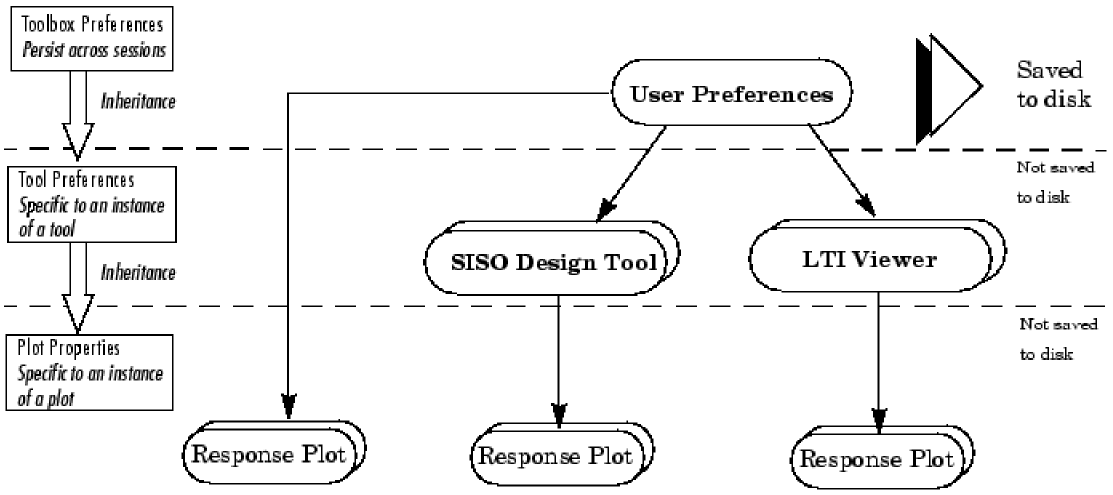
Property and Preferences Hierarchy

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 14-3 and "Customizing Response Plots Using Plot Tools" on page 14-20 for response plots • "Linear System Analyzer Preferences Editor" on page 13-2 for Linear System Analyzer plots • "Graphical Tuning Window Preferences Editor" on page 13-9 for Graphical Tuning Window plots
A single plot or many plots, programmatically from the command line	"Customizing Response Plots from the Command Line" on page 14-24
All Control System Toolbox plots (changes apply globally to all plot types and persist from session to session)	"Toolbox Preferences Editor" on page 12-2

Setting Toolbox Preferences

Toolbox Preferences Editor

In this section...
"Overview of the Toolbox Preferences Editor" on page 12-2
"Opening the Toolbox Preferences Editor" on page 12-2

Overview of the Toolbox Preferences Editor

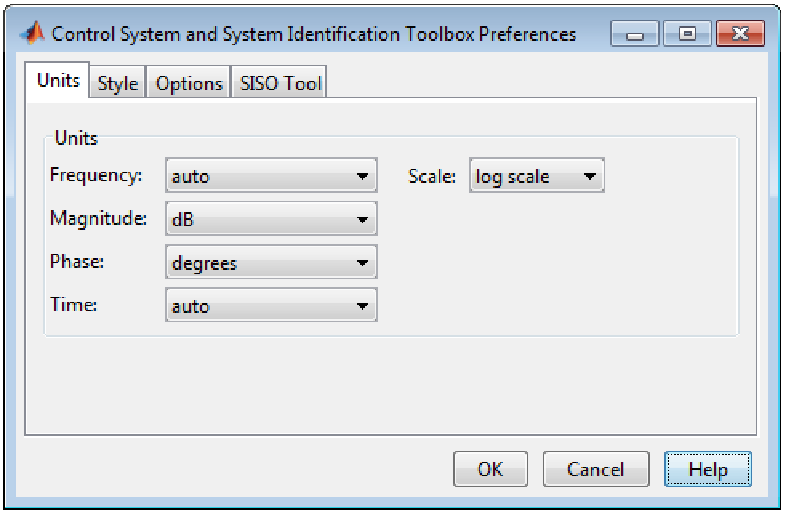
The Toolbox Preferences editor allows you to set plot preferences that will persist from session to session.

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 and System Identification Toolbox Preferences

Units

Frequency: auto

Scale: log scale

Magnitude: dB

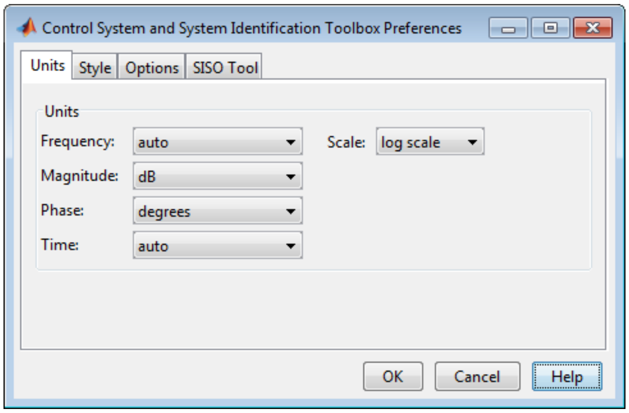
Phase: degrees

Time: auto

Control System Toolbox Preferences Editor

- “Units Pane” on page 12-4
- “Style Pane” on page 12-7
- “Options Pane” on page 12-8
- “SISO Tool Pane” on page 12-9

Units Pane


Control System and System Identification Toolbox Preferences
Units
Style
Options
SISO Tool
OK
Cancel
Help

Units

Frequency: auto

Scale: log scale

Magnitude: dB

Phase: degrees

Time: auto

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.

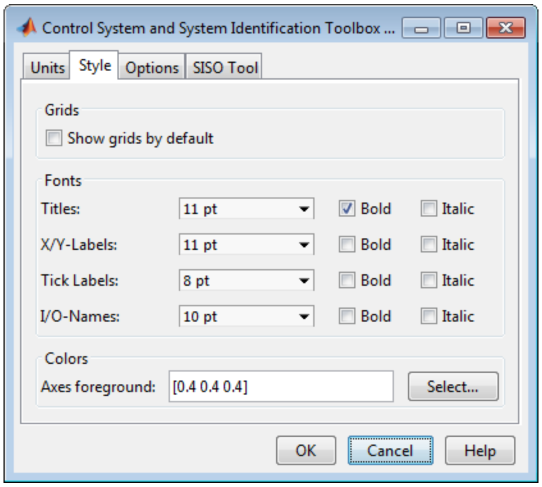
  Other Time Units Options

    - `'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.


Control System and System Identification Toolbox ...

Grids

☐ Show grids by default

Fonts

Titles:	11 pt	☑ Bold	☐ Italic
X/Y-Labels:	11 pt	☐ Bold	☐ Italic
Tick Labels:	8 pt	☐ Bold	☐ Italic
I/O-Names:	10 pt	☐ Bold	☐ Italic

Colors

Axes foreground: [0.4 0.4 0.4]

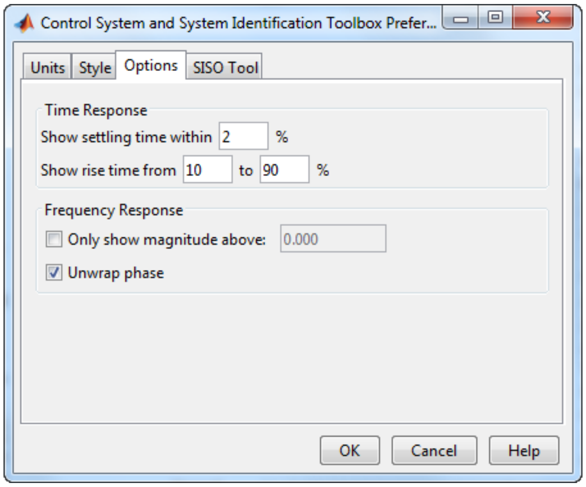
You have the following choices:

- **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. See "Select colors" on page 13-13 for more information.

Options Pane

The Options pane has selections for time responses and frequency responses. This figure shows the Options pane with default settings.



Time Response
Show settling time within 2 %
Show rise time from 10 to 90 %

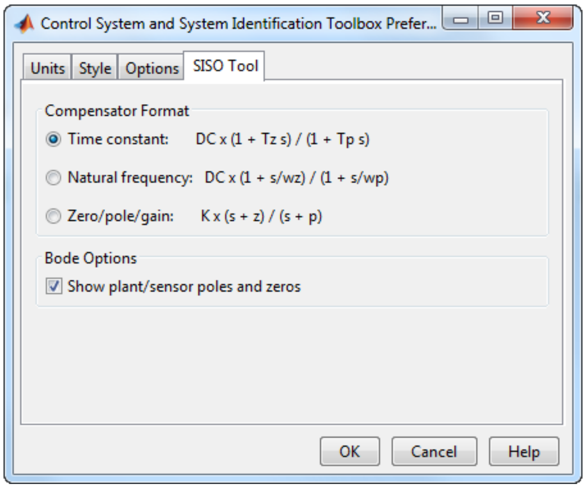
Frequency Response
☐ Only show magnitude above: 0.000
☑ Unwrap phase

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°).

SISO Tool Pane

The SISO Tool pane has settings for the SISO Design Tool. This figure shows the SISO Tool pane with default settings.



Control System and System Identification Toolbox Prefer...

Units | Style | Options | SISO Tool

Compensator Format

- (●) Time constant: DC x (1 + Tz s) / (1 + Tp s)
- ( ) Natural frequency: DC x (1 + s/wz) / (1 + s/wp)
- ( ) Zero/pole/gain: K x (s + z) / (s + p)

Bode Options

- [✓] Show plant/sensor poles and zeros

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

  $$DC \times \frac{(1+Tz_1 s)}{(1+Tp_1 s)} \cdots$$

  where DC 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

  $$DC \times \frac{\left(1+s/\omega_{z_1}\right)}{\left(1+s/\omega_{p_1}\right)} \cdots$$

where *DC* 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

$$K \times \frac{(s+z_1)}{(s+p_1)}$$

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 SISO Design Tool 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.

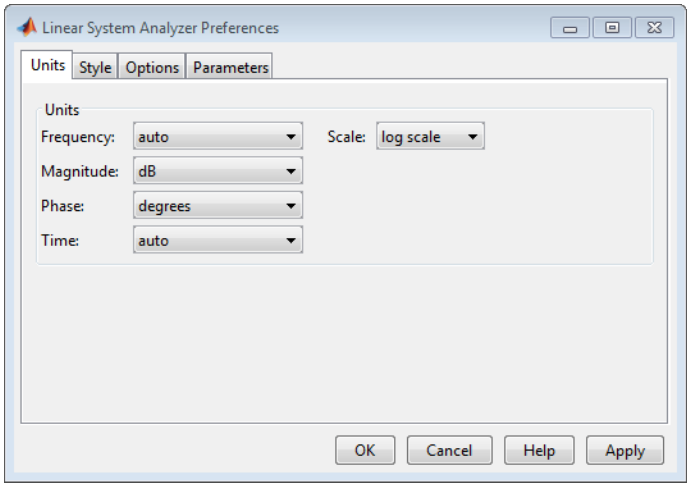
Setting Tool Preferences

Linear System Analyzer Preferences Editor

In this section...

Opening the Linear System Analyzer Preference Editor

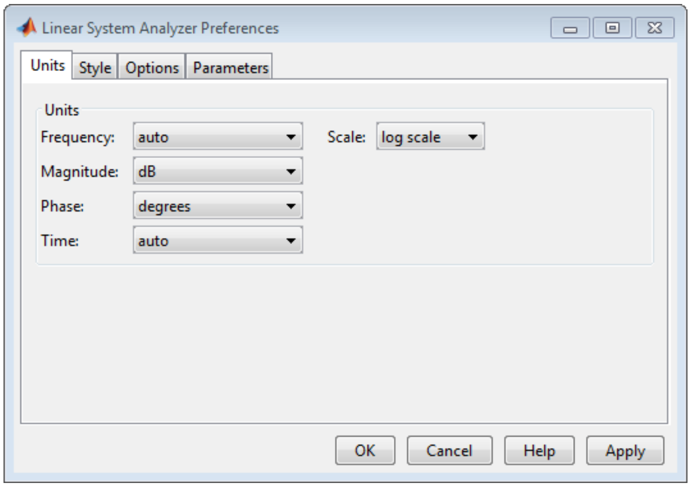
In the Linear System Analyzer, select **Edit** > **Linear System Analyzer Preferences**. The Linear System Analyzer Preferences dialog box let you customize various Linear System Analyzer properties, including units, fonts, and various other characteristics. This figure shows the editor open to its first pane.



Units
Frequency: auto
Scale: log scale
Magnitude: dB
Phase: degrees
Time: auto

- "Units Pane" on page 13-3
- "Style Pane" on page 13-5
- "Options Pane" on page 13-6
- "Parameters Pane" on page 13-7

Units Pane



Units

Frequency: auto

Scale: log scale

Magnitude: dB

Phase: degrees

Time: auto

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.

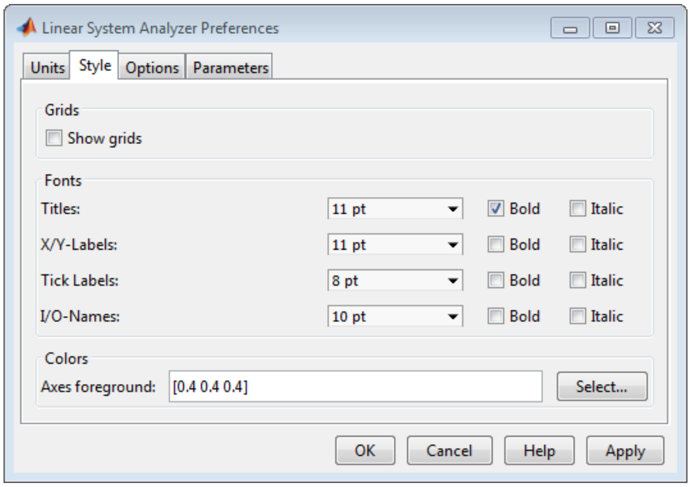
  Other Time Units Options

  - '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.



Grids

☐ Show grids

Fonts

Titles:	11 pt	☑ Bold	☐ Italic
X/Y-Labels:	11 pt	☐ Bold	☐ Italic
Tick Labels:	8 pt	☐ Bold	☐ Italic
I/O-Names:	10 pt	☐ Bold	☐ Italic

Colors

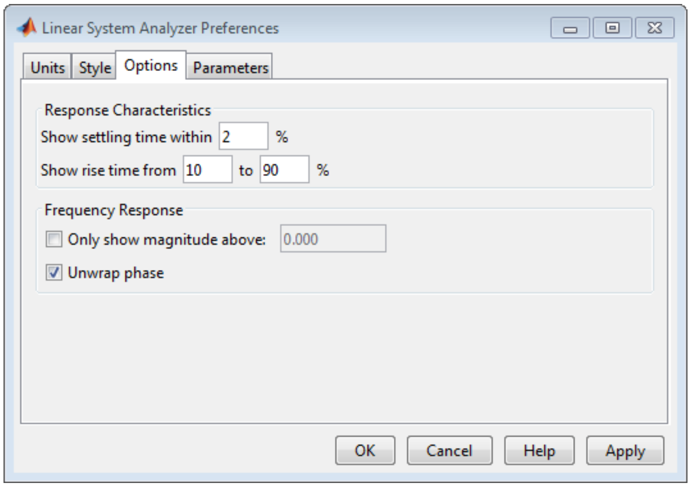
Axes foreground: [0.4 0.4 0.4] Select...

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. See “Select colors” on page 13-13 for more information.

Options Pane

The Options pane has selections for time responses and frequency responses.



Response Characteristics

Show settling time within 2 %

Show rise time from 10 to 90 %

Frequency Response

☐ Only show magnitude above: 0.000

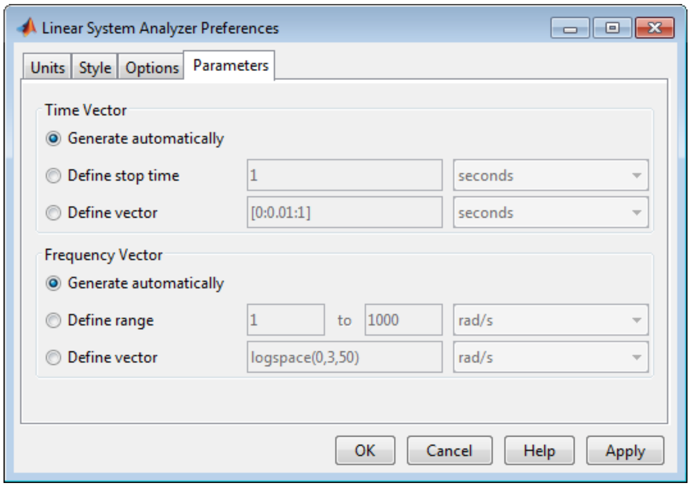
☑ Unwrap phase

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.



Time Vector

[● Generate automatically]

○ Define stop time 1 seconds

○ Define vector [0:0.01:1] seconds

Frequency Vector

[● Generate automatically]

○ Define range 1 to 1000 rad/s

○ Define vector logspace(0,3,50) rad/s

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

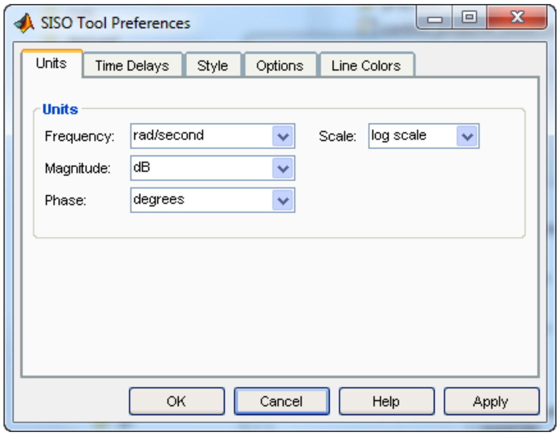
- "Linear System Analyzer Overview" on page 18-2

Graphical Tuning Window Preferences Editor

In this section...

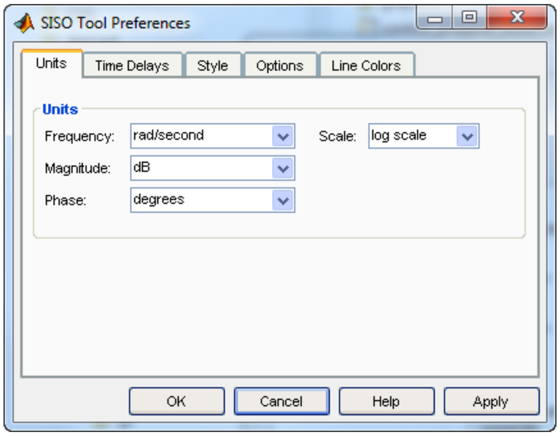
Opening the Graphical Tuning Window Preferences Editor

To open the **SISO Tool Preferences** editor, select **SISO Tool Preferences** from the **Edit** menu of the Graphical Tuning window. This window opens.



Units
Frequency: rad/second
Scale: log scale
Magnitude: dB
Phase: degrees

Units Pane



Units

Frequency: rad/second

Scale: log scale

Magnitude: dB

Phase: degrees

The **Units** pane has settings for the following units:

- **Frequency**

  The default units are `rad/second`.

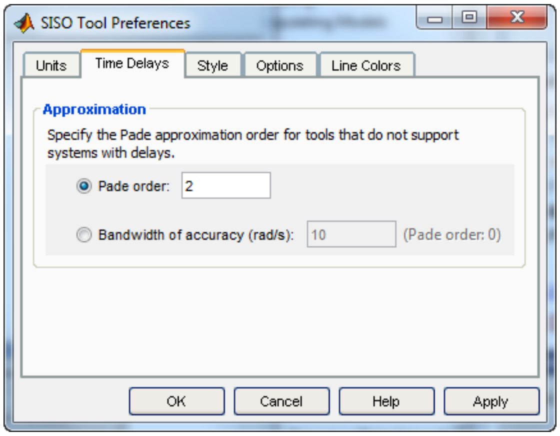
  ### Other Frequency Units Options

  - `'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

For frequency and magnitude axes, you can select logarithmic or linear scales.

Time Delays Pane



Approximation

Specify the Pade approximation order for tools that do not support systems with delays.

(●) Pade order: 2

( ) Bandwidth of accuracy (rad/s): 10 (Pade order: 0)

In the Time Delays pane, specify the order for Padé approximations of delays in your system as either:

- Actual Padé order

- Bandwidth of accuracy (rad/s) — The highest frequency at which the approximated response matches the actual system. The software computes and displays the corresponding Padé order.

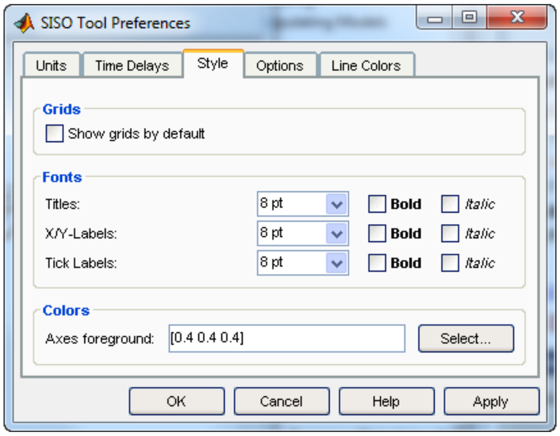
The following compensator design tools do not support systems with exact time delays. If your system has exact continuous-time delays, these tools automatically compute a Padé approximation of the delays. In this case, you receive a notification.

- Root locus
- Pole-zero
- PID automated tuning
- IMC automated tuning
- LQG automated tuning
- Loop shaping automated tuning

Tip To determine if a certain Padé order gives a good approximation of your system, use the `pade` command to approximate your system. Then, compare a plot of the two systems using the `bode` command.

Style Pane

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for all plots in the Graphical Tuning Window. This figure shows the Style pane.


SISO Tool Preferences

Units | Time Delays | Style | Options | Line Colors

Grids
- [ ] Show grids by default

Fonts

Titles:	8 pt	[ ] Bold	[ ] *Italic*
X/Y-Labels:	8 pt	[ ] Bold	[ ] *Italic*
Tick Labels:	8 pt	[ ] Bold	[ ] *Italic*

Colors
Axes foreground: [0.4 0.4 0.4] Select...

Grids Panel

Select the box to activate grids for all plots in the GRAPHICAL Tuning Window

Fonts Panel

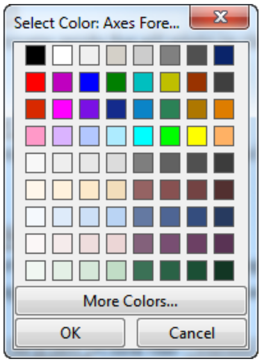
Set the font size, weight (bold), and angle (italic) by using the menus and check boxes.

Colors Panel

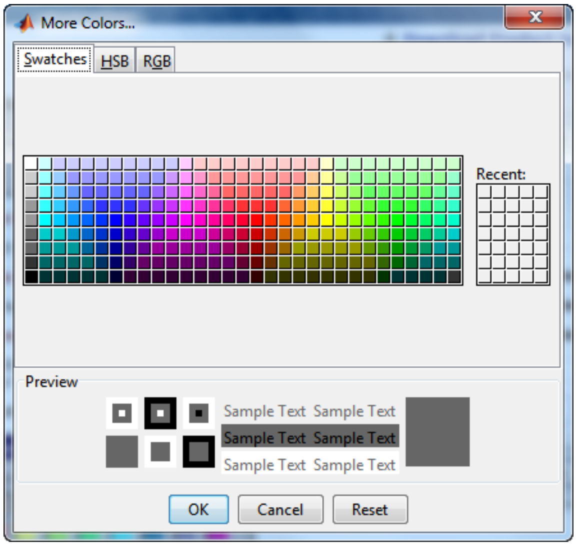
Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

Select colors

Click the **Select** button to open the Select Color window for the axes foreground.



You can use this window to choose axes foreground colors without having to set RGB (red-green-blue) values numerically. To make your selections, click on the colored rectangles and press OK. If you want a broader range of colors, click the **More Colors** button. This opens the More Colors window, as shown in the following figure.

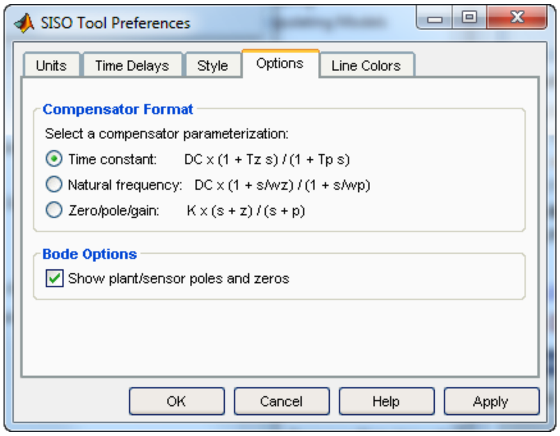


Click on the array of color swatches to select a color. Alternatively, click the HSB tab to use the Hue Saturation Brightness (HSB) color picker. Or, click the RGB tab to enter numeric RGB values.

When you select a color, the **Preview** section of the window changes to preview your selection. Click **OK** to accept the choice. Click **Reset** to reset the selection.

Options Pane

The Options pane, shown below, has selections for compensator format and Bode diagrams.



Compensator Format

Select a compensator parameterization:

(●) Time constant: DC x (1 + Tz s) / (1 + Tp s)

( ) Natural frequency: DC x (1 + s/wz) / (1 + s/wp)

( ) Zero/pole/gain: K x (s + z) / (s + p)

Bode Options

✓ Show plant/sensor poles and zeros

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

  $$DC \times \frac{(1+Tz_1 s)}{(1+Tp_1 s)} \cdots$$

  where DC 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

  $$DC \times \frac{\left(1+s/\omega_{z_1}\right)}{\left(1+s/\omega_{p_1}\right)} \cdots$$

  where DC 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

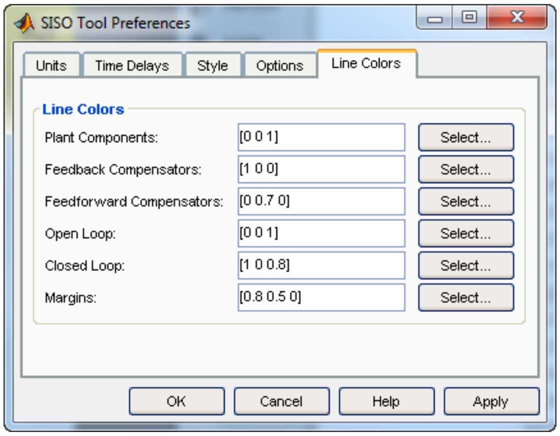
$$K \times \frac{(s+z_1)}{(s+p_1)}$$

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 GRAPHICAL Tuning Window shows the plant and sensor poles and zeros as blue x's and o's, respectively. Clear this check 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.

Line Colors Pane

The Line Colors pane, shown below, has selections for specify the colors of the lines in the response plots of the Graphical Tuning Window.



Line Colors

Plant Components:	[0 0 1]
Feedback Compensators:	[1 0 0]
Feedforward Compensators:	[0 0.7 0]
Open Loop:	[0 0 1]
Closed Loop:	[1 0 0.8]
Margins:	[0.8 0.5 0]

To change the colors of plot lines associated with parts of your model, specify 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, click the **Select** button to open the Select Color window. See “Select colors” on page 13-13 for more information.

See Also

`controlSystemDesigner`

Customizing Response Plot Properties

Introduction

The lowest level of the “Property and Preferences Hierarchy” on page 11-3 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

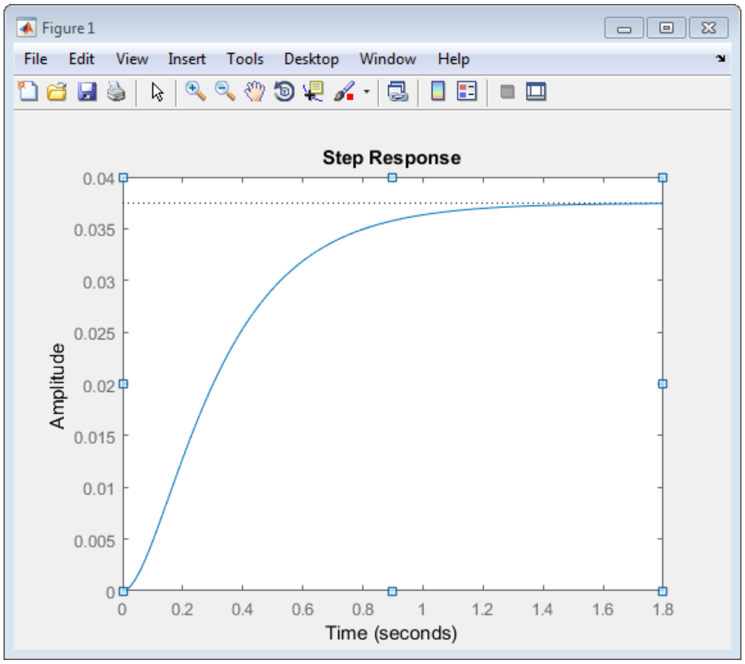
After you create a response plot, there are two ways to open the Property Editor:

- 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.

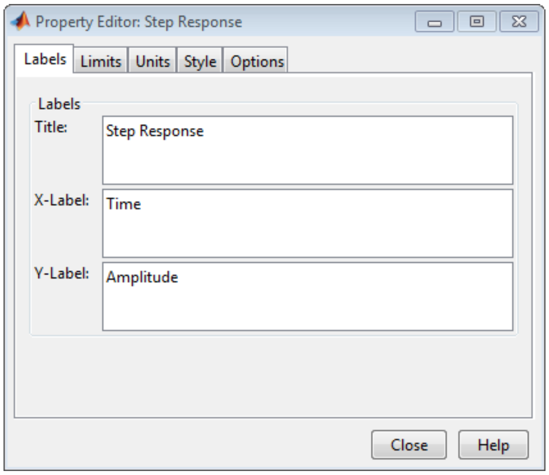
```
load ltiexamples
step(sys_dc)
```

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.



Labels

Title: Step Response

X-Label: Time

Y-Label: Amplitude

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.

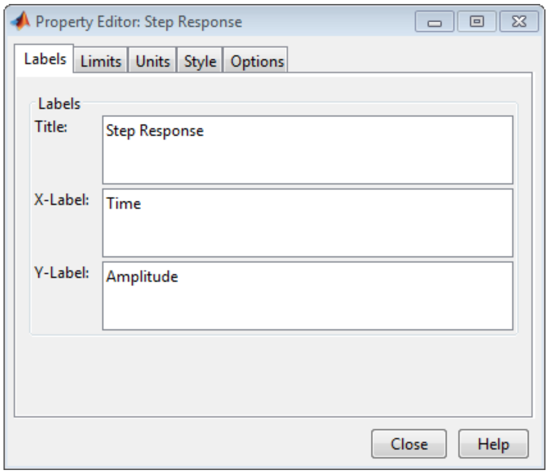
  These include peak response, settling time, phase and gain margins, etc. Plot options change with each plot response type. The Property Editor displays only the options that make sense for the selected response plot. For example, phase and gain margins are not available for step responses.

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.



Labels

Title: Step Response

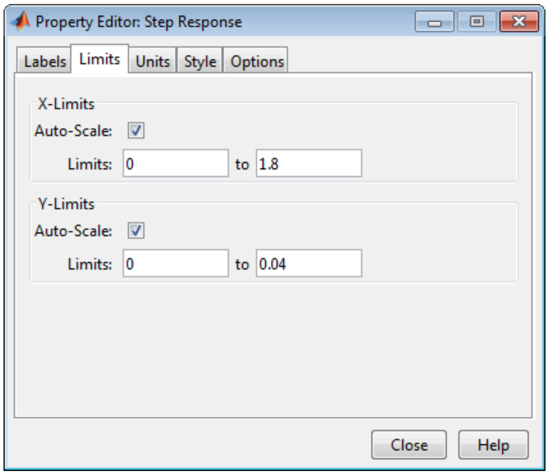
X-Label: Time

Y-Label: Amplitude

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.

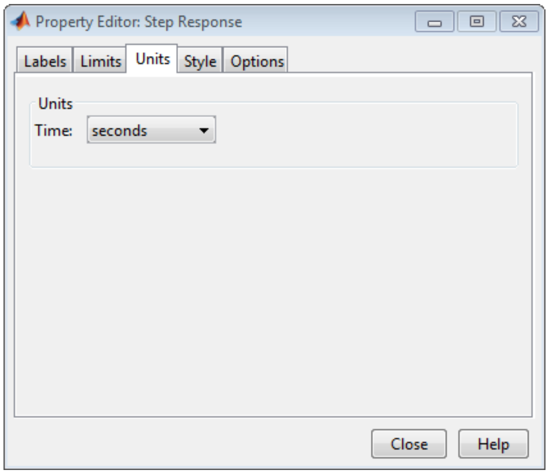


X-Limits
Auto-Scale: ✓
Limits: 0 to 1.8

Y-Limits
Auto-Scale: ✓
Limits: 0 to 0.04

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.



Units
Time: seconds

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'`

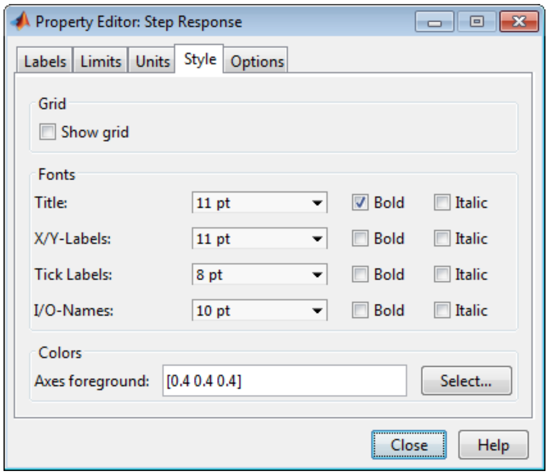
Response Plot	Unit Conversions
	• 'hours' • 'days' • 'weeks' • 'months' • 'years' • Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • '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.



Grid

☐ Show grid

Fonts

Title:	11 pt	✓ Bold	☐ Italic
X/Y-Labels:	11 pt	☐ Bold	☐ Italic
Tick Labels:	8 pt	☐ Bold	☐ Italic
I/O-Names:	10 pt	☐ Bold	☐ Italic

Colors

Axes foreground: [0.4 0.4 0.4]

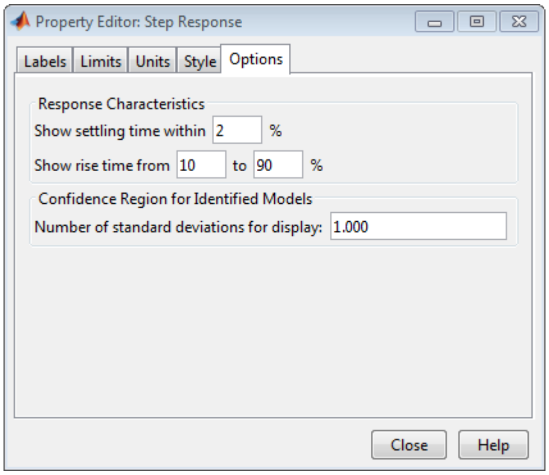
You have the following choices:

- **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. See “Select colors” on page 13-13 for more information.

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 Characteristics

Show settling time within 2 %

Show rise time from 10 to 90 %

Confidence Region for Identified Models

Number of standard deviations for display: 1.000

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 Δ`y`, plot the uncertainty ±Δ`y` as a function of time (default). If cleared, `y`±Δ`y` 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 14-20
"Opening and Working with Plot Tools" on page 14-21
"Example of Changing Line Color Using Plot Tools" on page 14-21

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 14-3 and "Customizing Response Plots from the Command Line" on page 14-24.

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

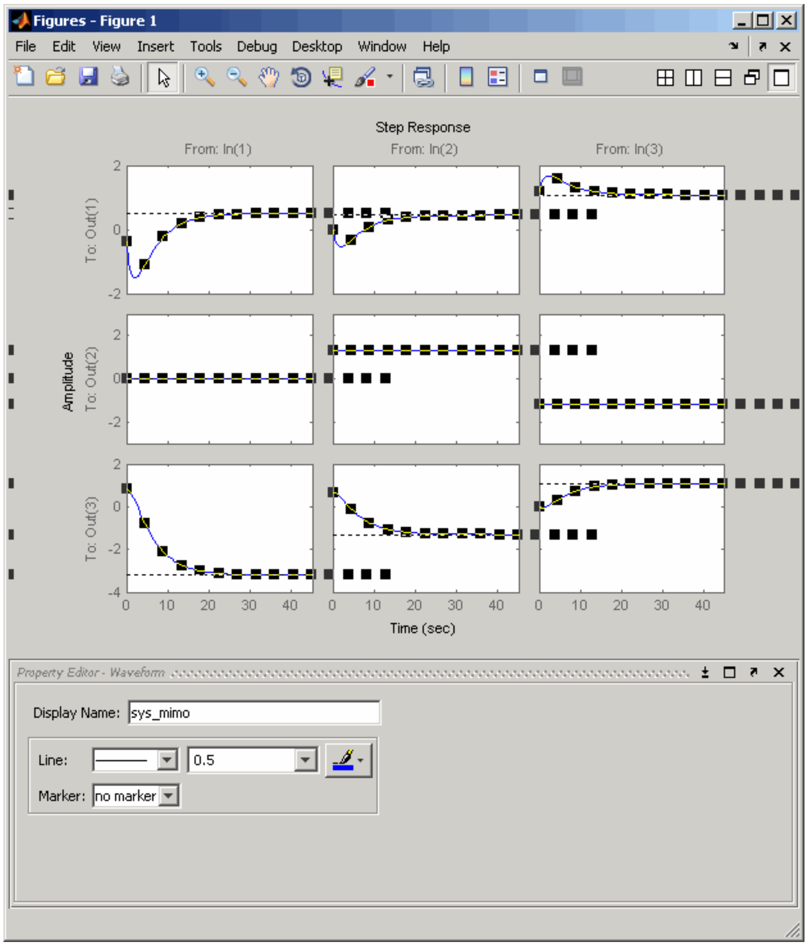
To change the line color of a MIMO system plot:

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.



Display Name: sys_mimo

Line: 0.5

Marker: no marker

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

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 14-27.
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 14-28.
3. Use `setoptions`, along with the plot handle and the plot options handle, to access and modify many plot options.

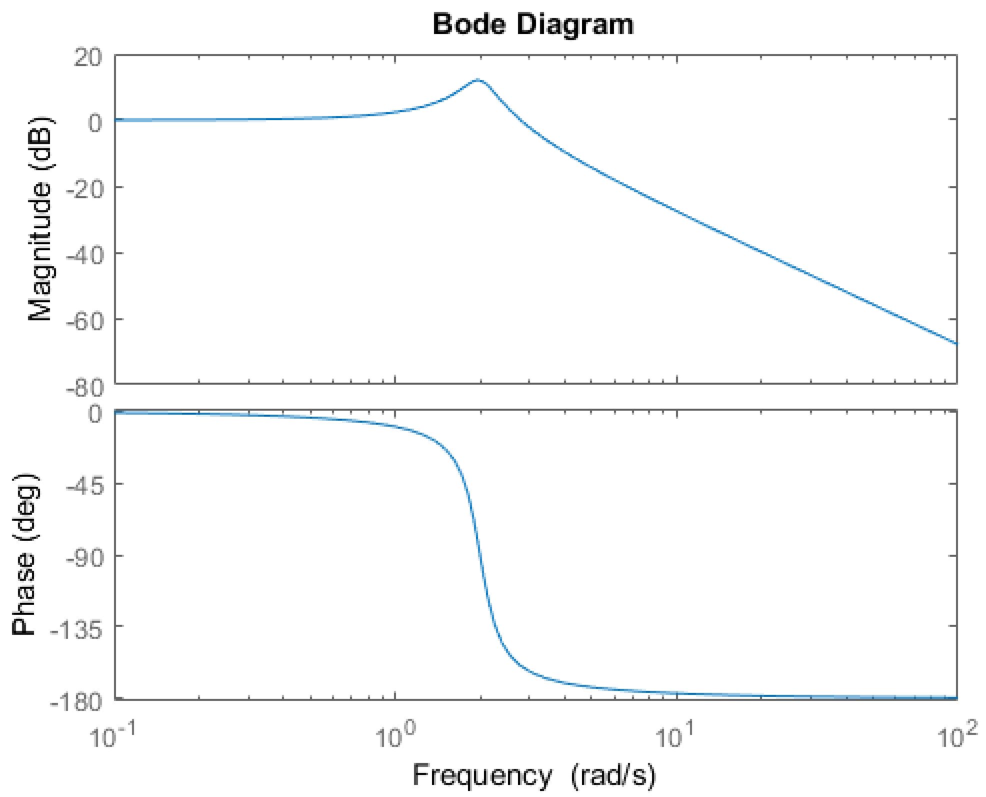
Note: You can also use `setoptions` to customize plots using property/value pairs instead of the plot options handle. Using property/value pairs shortens the procedure to one line of code.

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.

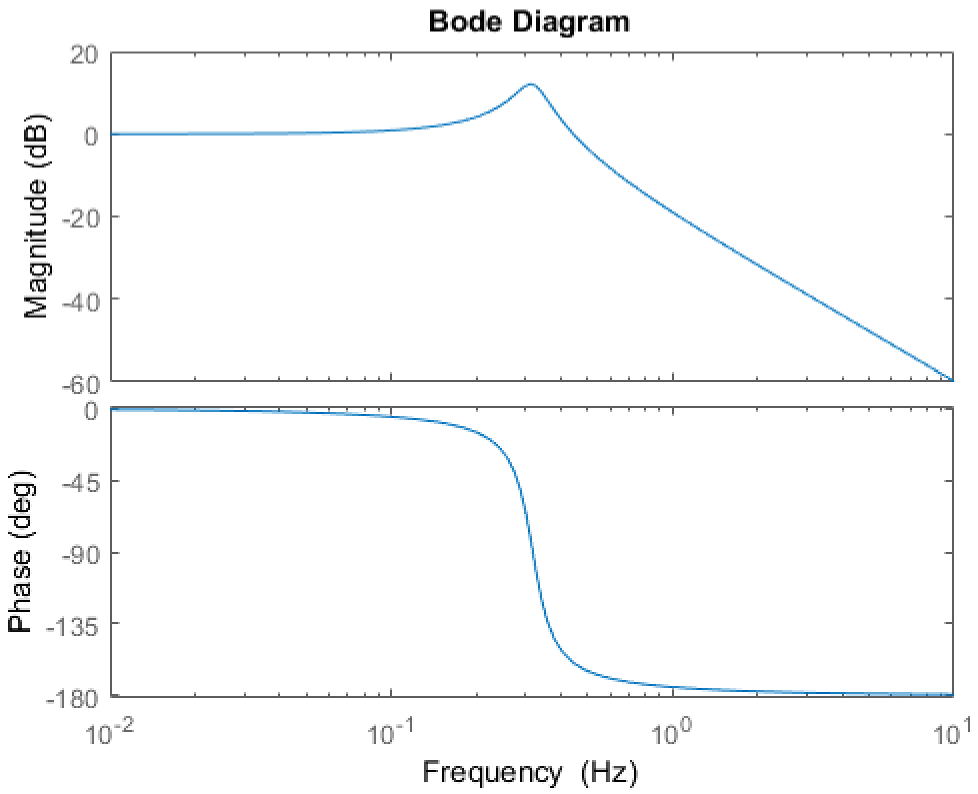
```
sys = tf(4,[1 0.5 4]);
h = bodeplot(sys);
```



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 14-28
- “Retrieving a Handle” on page 14-29
- “Creating a Handle” on page 14-29
- “Which Properties Can You Modify?” on page 14-30

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 14-34 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 14-34 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 14-30
- "Changing Plot Units" on page 14-31
- "Create Plots Using Existing Plot Options Handle" on page 14-32
- "Creating a Default Plot Options Handle" on page 14-32
- "Using Dot Notation Like a Structure" on page 14-33
- "Setting Property Pairs in setoptions" on page 14-33

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.

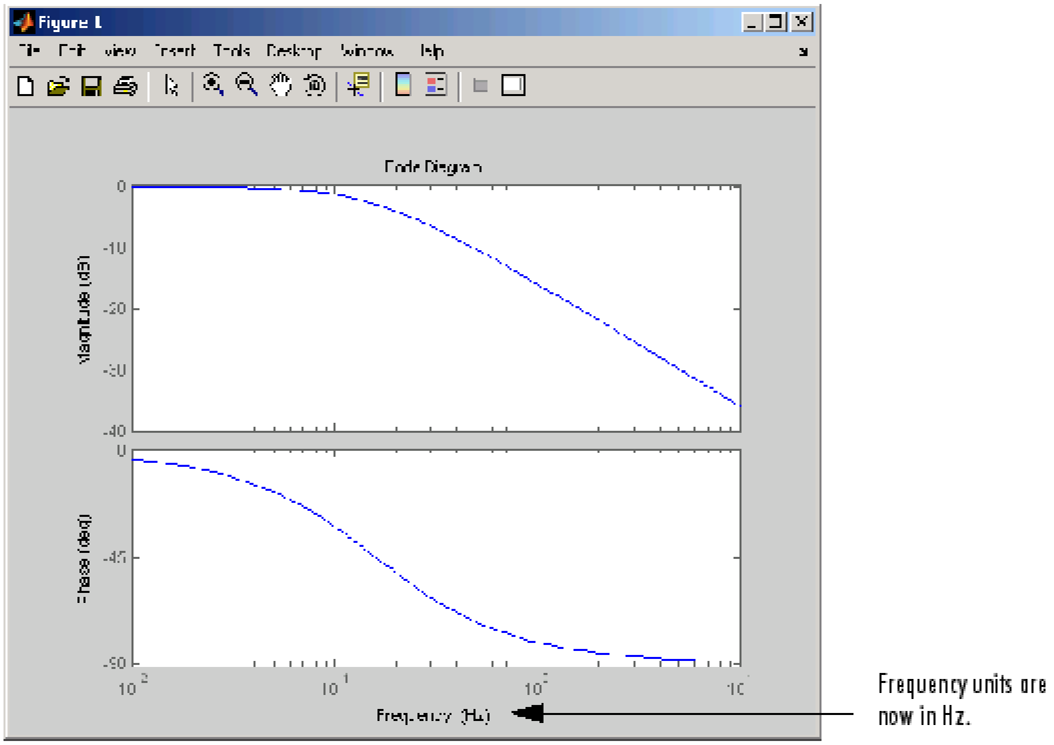
Changing Plot Units

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')
```



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;
```

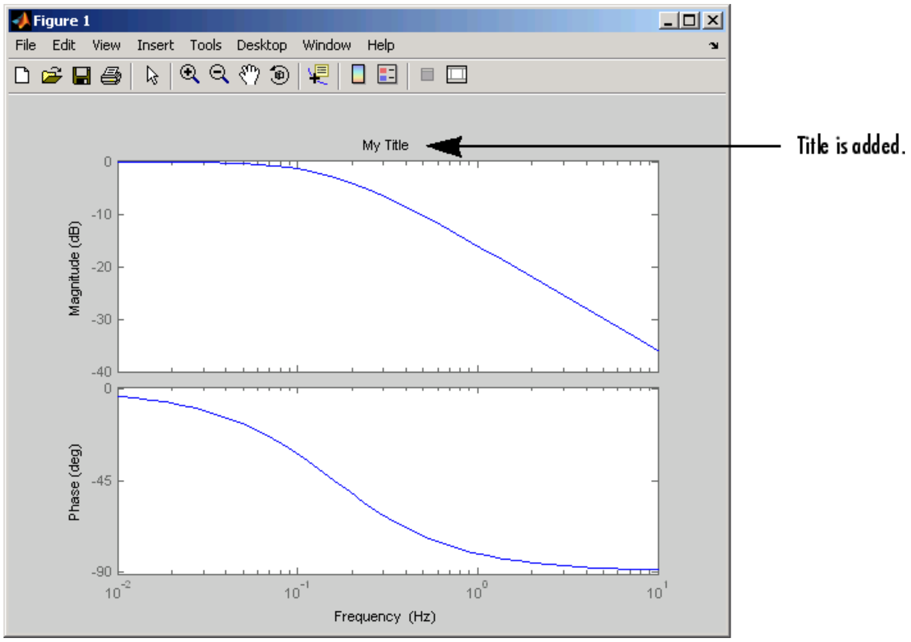
Change the frequency units and apply the changes to `sys`.

```
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'

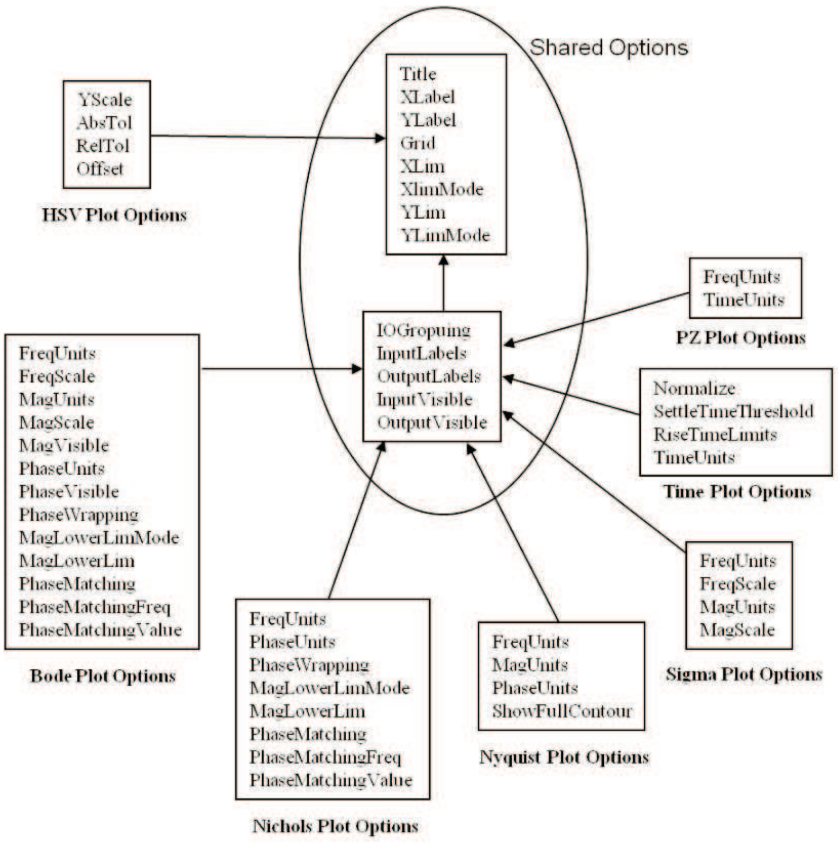
Sigma Plots

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



Customizing Plots Inside the SISO Design Tool

In this section...

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

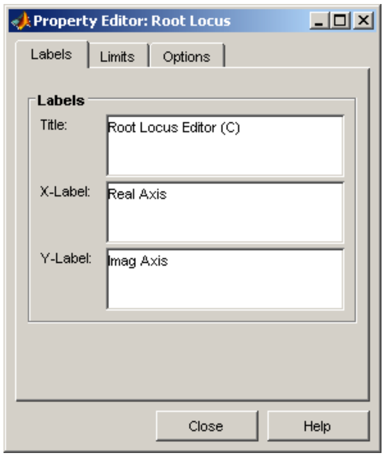
You can use each of these property editors to create the customized plots within the SISO Design tool.

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

Title: Root Locus Editor (C)

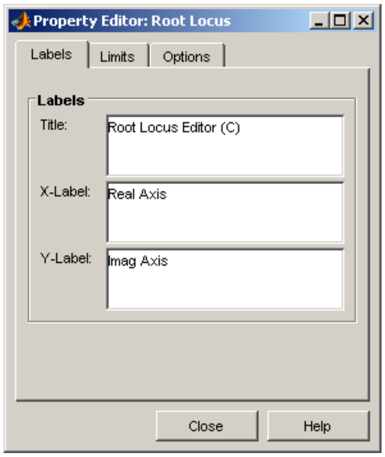
X-Label: Real Axis

Y-Label: Imag Axis

- “Labels Pane” on page 14-47
- “Limits Pane” on page 14-48
- “Options Pane” on page 14-49

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.



Labels

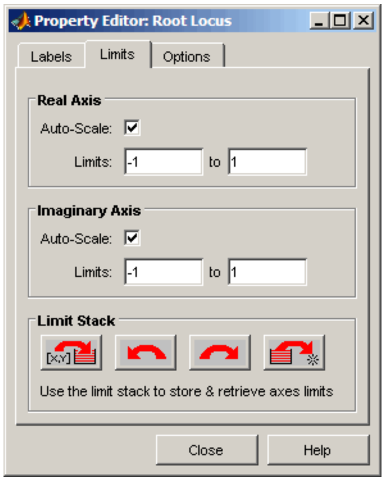
Title: Root Locus Editor (C)

X-Label: Real Axis

Y-Label: Imag Axis

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.



Real Axis
Auto-Scale: ✓
Limits: -1 to 1

Imaginary Axis
Auto-Scale: ✓
Limits: -1 to 1

Limit Stack
Use the limit stack to store & retrieve axes limits

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

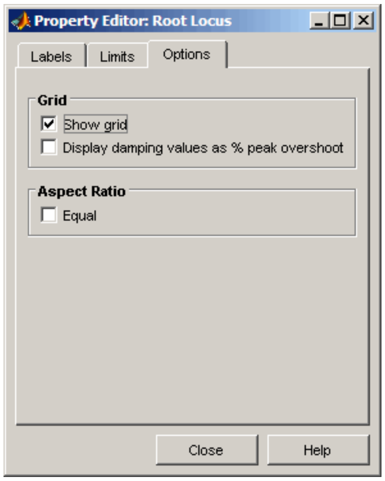
— Retrieve the next stack entry

— 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.


Property Editor: Root Locus

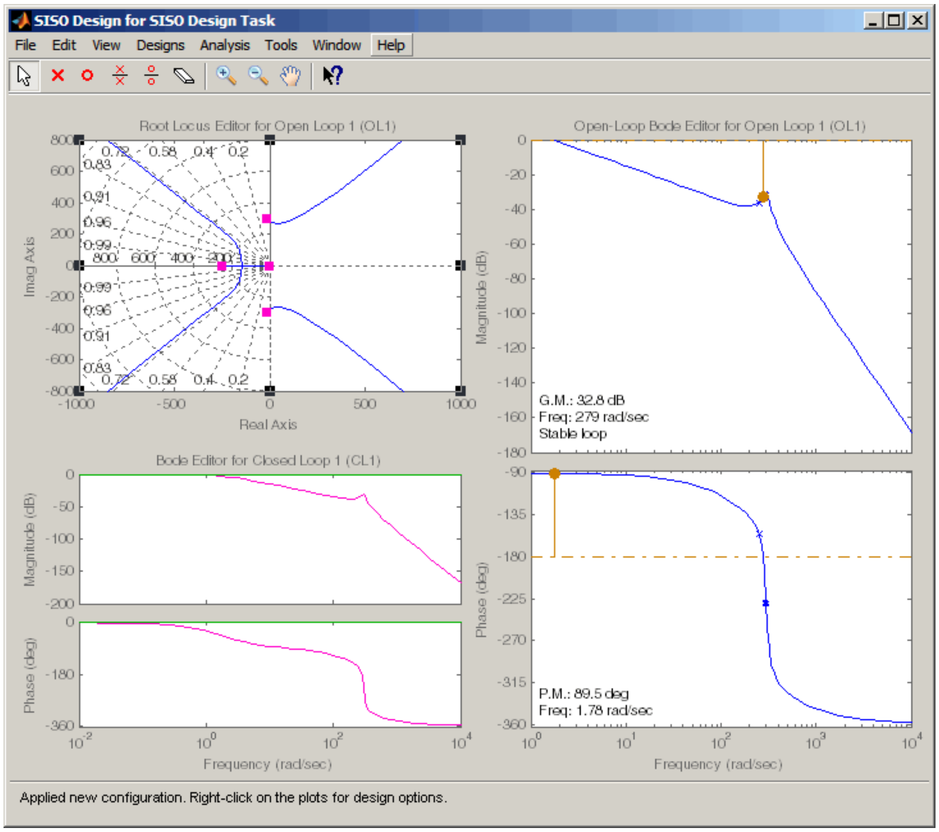
Grid
- [✓] Show grid
- [ ] Display damping values as % peak overshoot

Aspect Ratio
- [ ] Equal

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.


SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Root Locus Editor for Open Loop 1 (OL1)
Imag Axis
Real Axis
Open-Loop Bode Editor for Open Loop 1 (OL1)
Magnitude (dB)
G.M.: 32.8 dB
Freq: 279 rad/sec
Stable loop
Bode Editor for Closed Loop 1 (CL1)
Magnitude (dB)
Phase (deg)
Frequency (rad/sec)
Phase (deg)
P.M.: 89.5 deg
Freq: 1.78 rad/sec
Frequency (rad/sec)

Applied new configuration. Right-click on the plots for design options.

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.

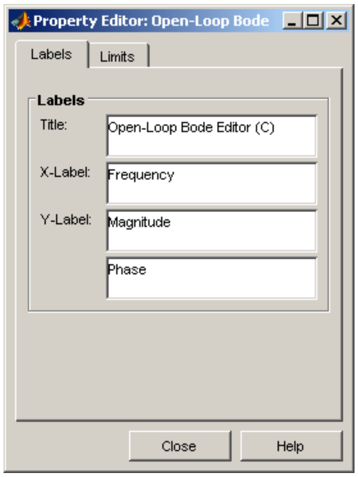
Open-Loop Bode Property Editor

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.

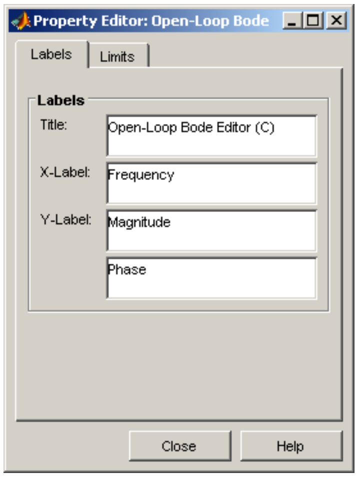


Title: Open-Loop Bode Editor (C)
X-Label: Frequency
Y-Label: Magnitude
Phase

- “Labels Pane” on page 14-51
- “Limits Pane” on page 14-52

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.



Labels

Title: Open-Loop Bode Editor (C)

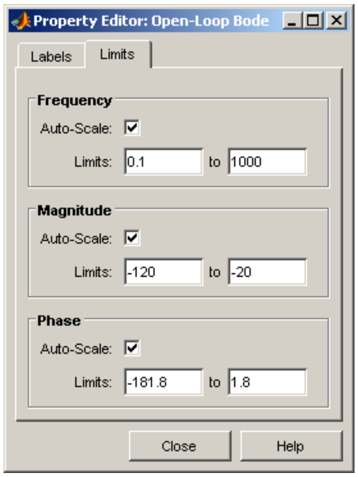
X-Label: Frequency

Y-Label: Magnitude

Phase

Limits Pane

You can use the Limits pane to override the default limits for the frequency, magnitude, and phase scales for your plots.



Frequency

Auto-Scale: ✓

Limits: 0.1 to 1000

Magnitude

Auto-Scale: ✓

Limits: -120 to -20

Phase

Auto-Scale: ✓

Limits: -181.8 to 1.8

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.

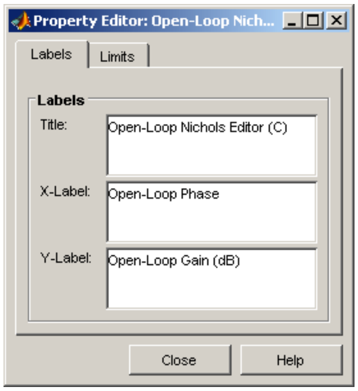
To restore the default settings, select the **Auto-Scale** boxes again.

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.



Title: Open-Loop Nichols Editor (C)

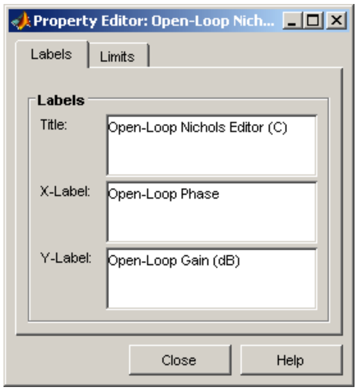
X-Label: Open-Loop Phase

Y-Label: Open-Loop Gain (dB)

- “Labels Pane” on page 14-53
- “Limits Pane” on page 14-54

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.



Labels

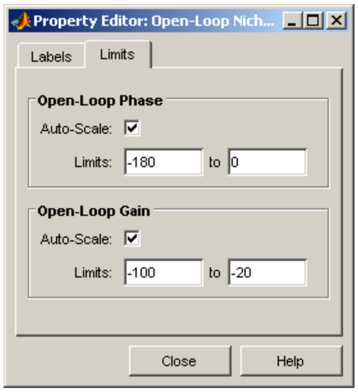
Title: Open-Loop Nichols Editor (C)

X-Label: Open-Loop Phase

Y-Label: Open-Loop Gain (dB)

Limits Pane

You can use the Limits pane to override the default limits for the frequency, magnitude, and phase scales for your plots.



Open-Loop Phase

Auto-Scale: ✓

Limits: -180 to 0

Open-Loop Gain

Auto-Scale: ✓

Limits: -100 to -20

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 14-50 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.

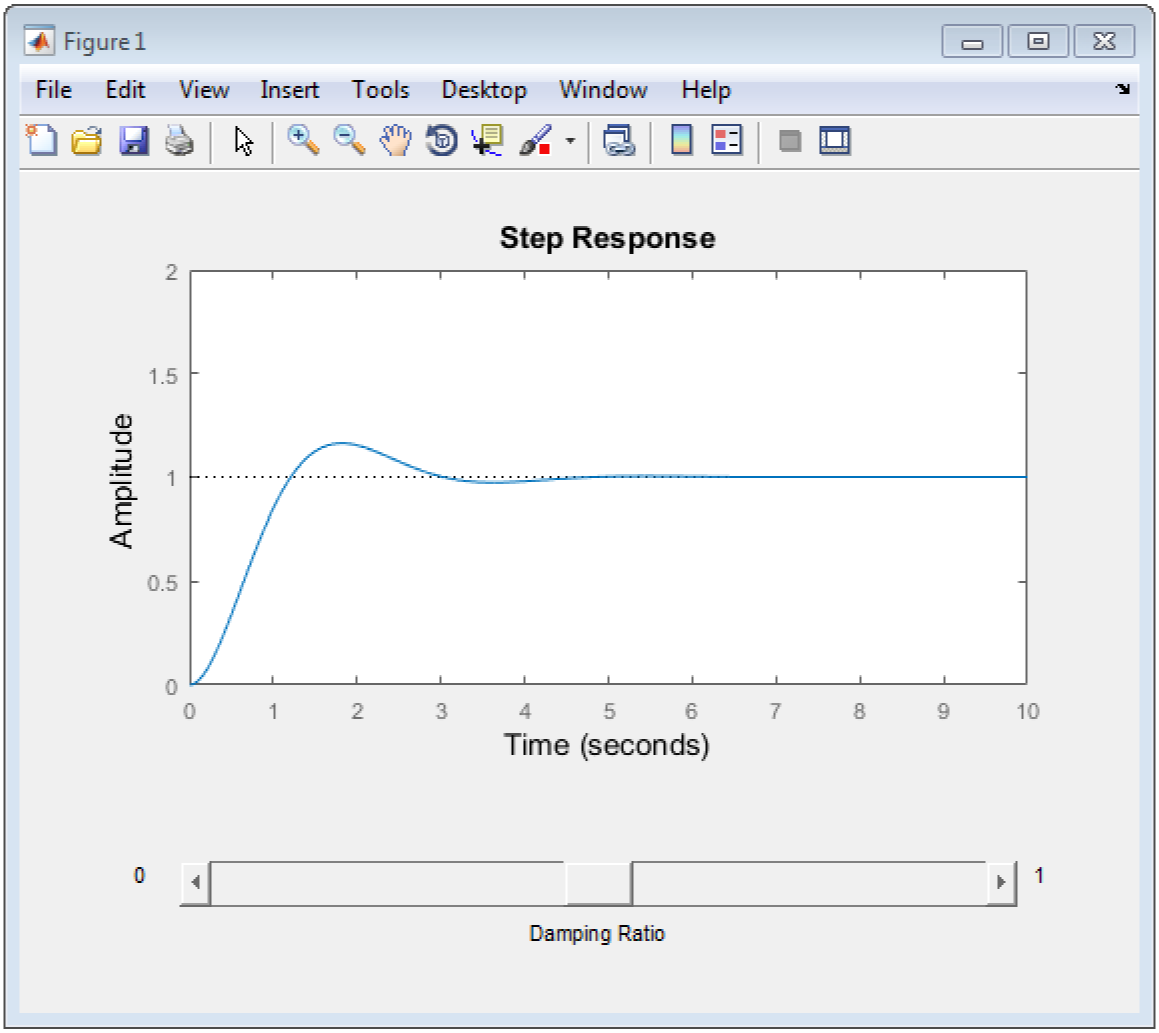
```
zeta = .5;                                    % Damping Ratio
wn = 2;                                       % Natural Frequency
sys = tf(wn^2,[1,2*zeta*wn,wn^2]);
```

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);
```



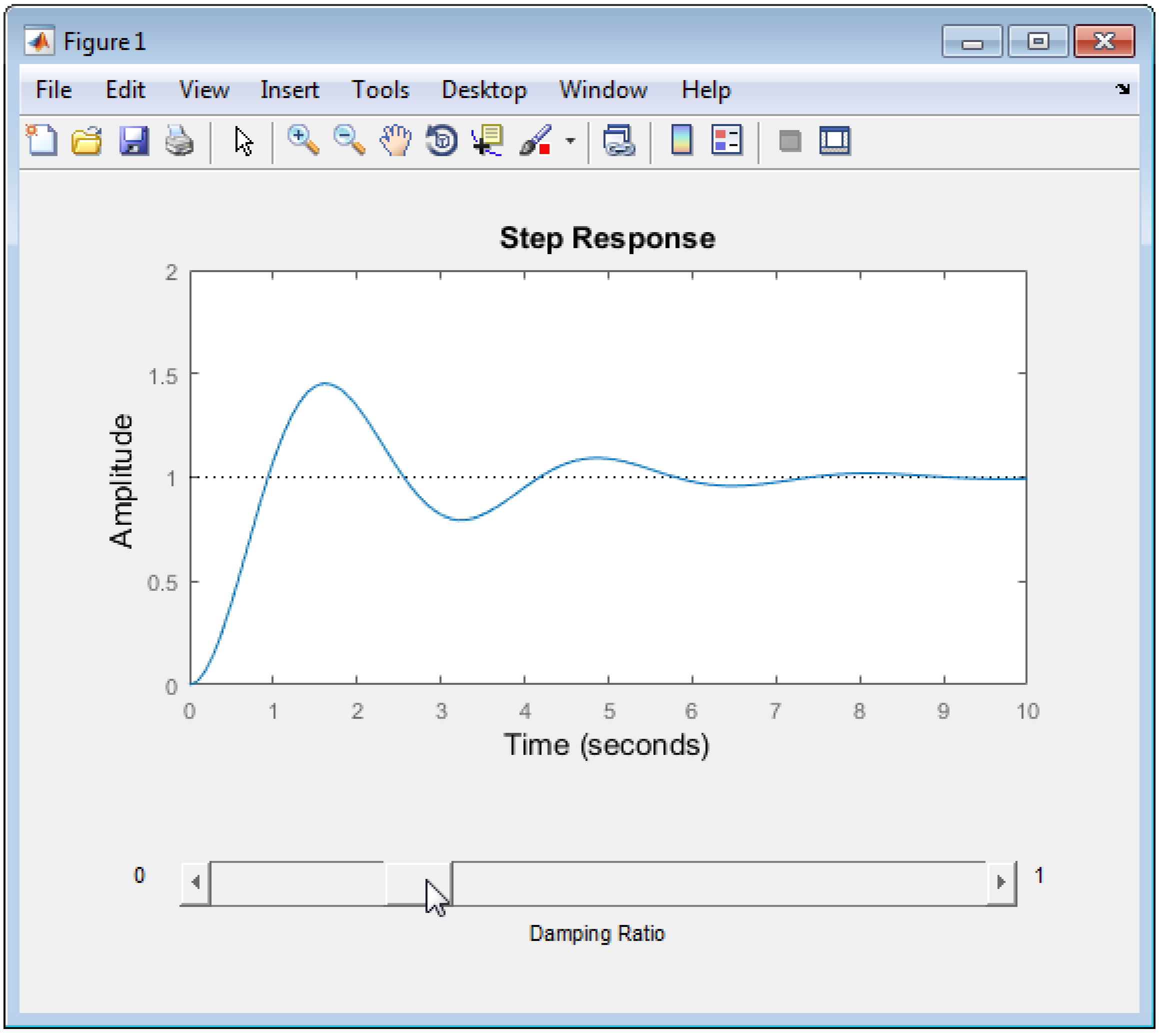
Set the callback that updates the step response plot as the damping ratio slider is moved.

```
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 UIs 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.

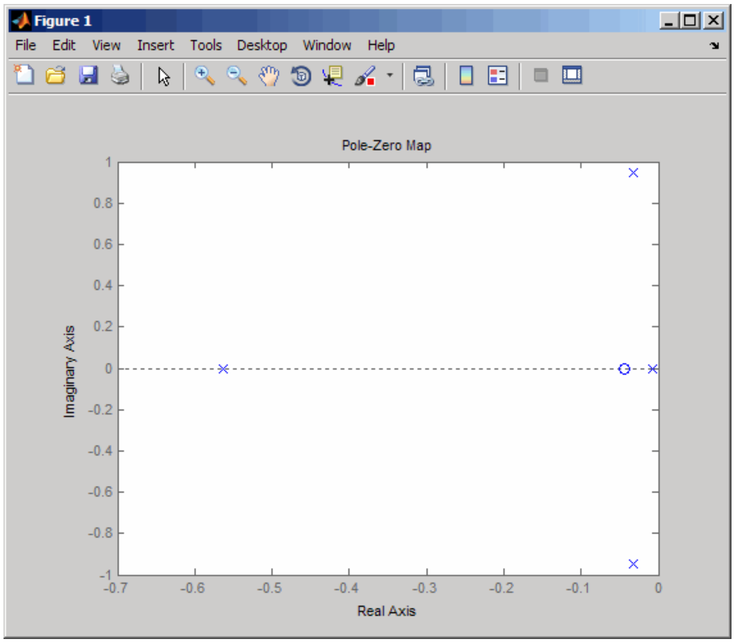
Computing Open-Loop Poles

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)
```



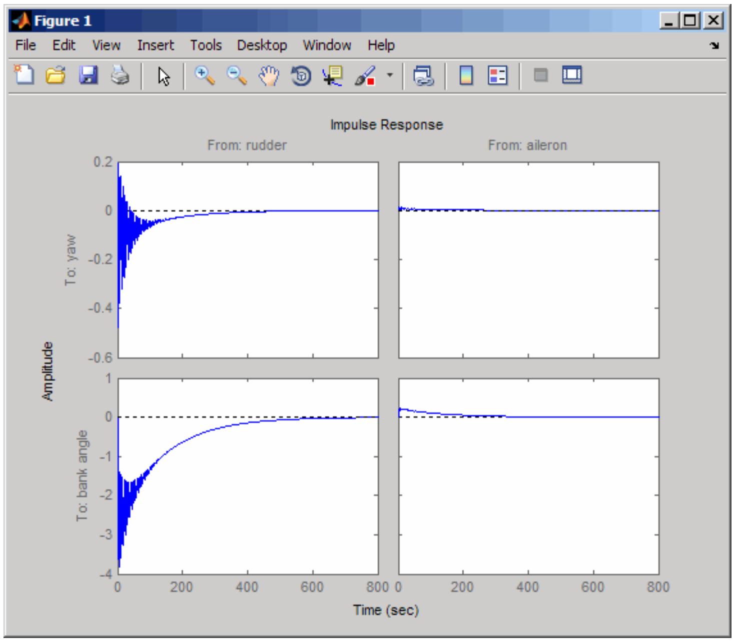
This model has one pair of lightly damped poles. They correspond to what is called the "Dutch roll mode."

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

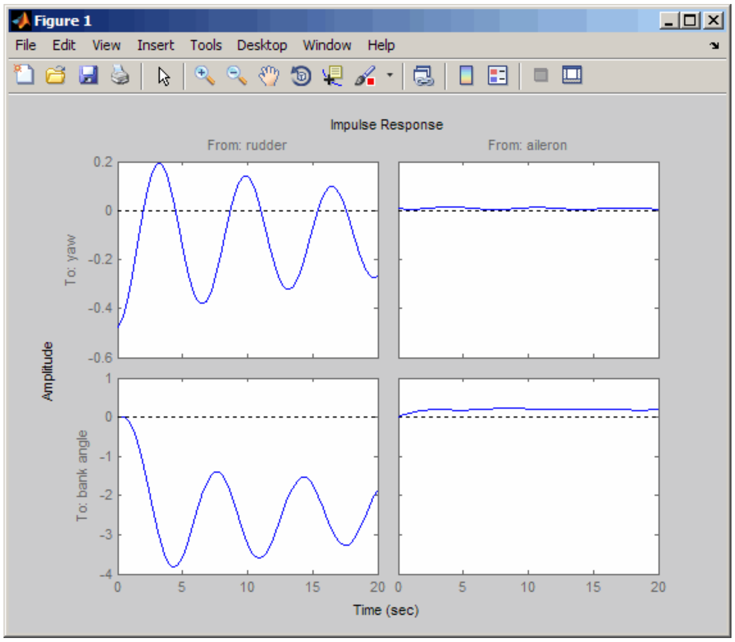
First, perform some open-loop analysis to determine possible control strategies. Start with the time response (you could use `step` or `impulse` here).

```
impulse(sys)
```

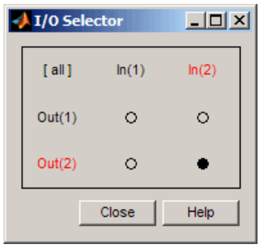


The impulse response confirms that the system is lightly damped. But the time frame is much too long because the passengers and the pilot are more concerned about the behavior during the first few seconds rather than the first few minutes. Next look at the response over a smaller time frame of 20 seconds.

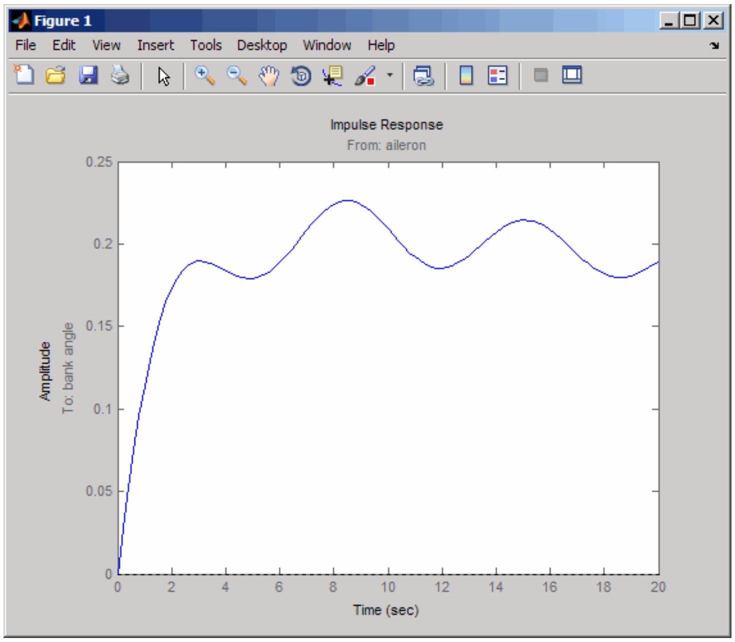
```
impulse(sys,20)
```



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.



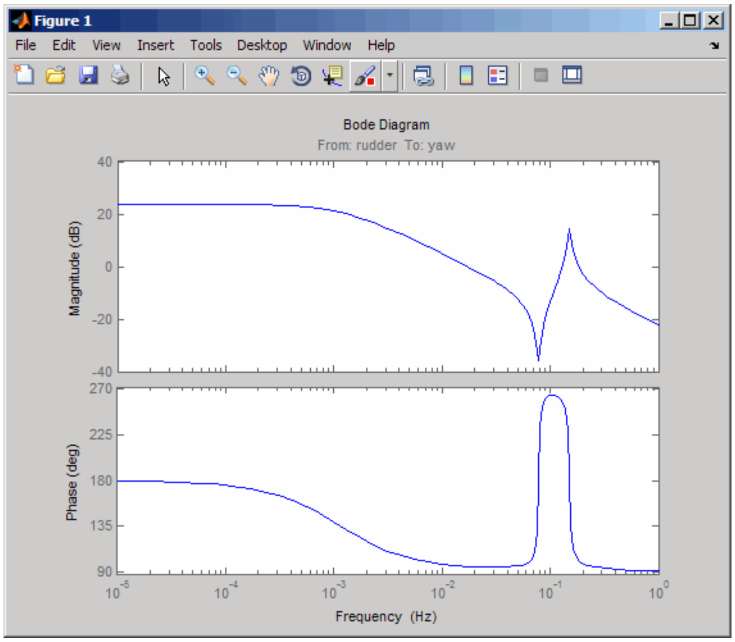
The new figure is shown below.



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)
```

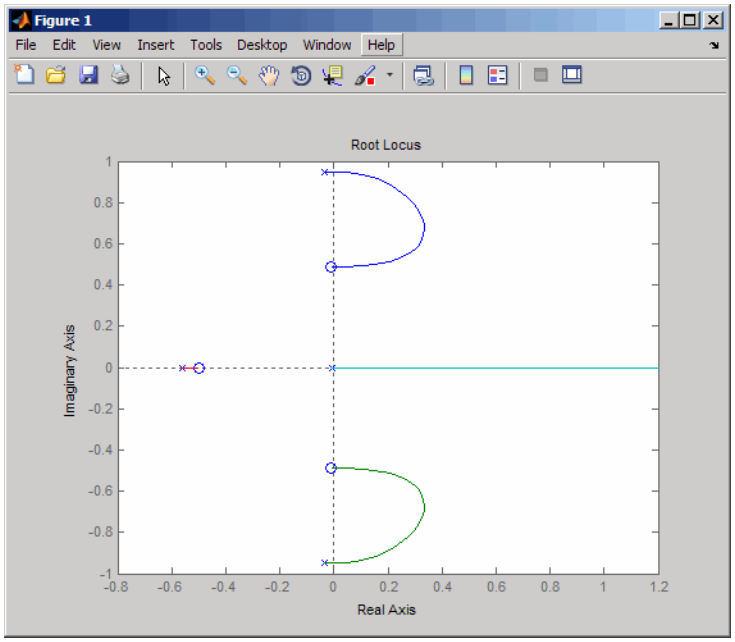


From this Bode diagram, you can see that the rudder has significant effect around the lightly damped Dutch roll mode (that is, near ω = 1 rad/sec).

Root Locus Design

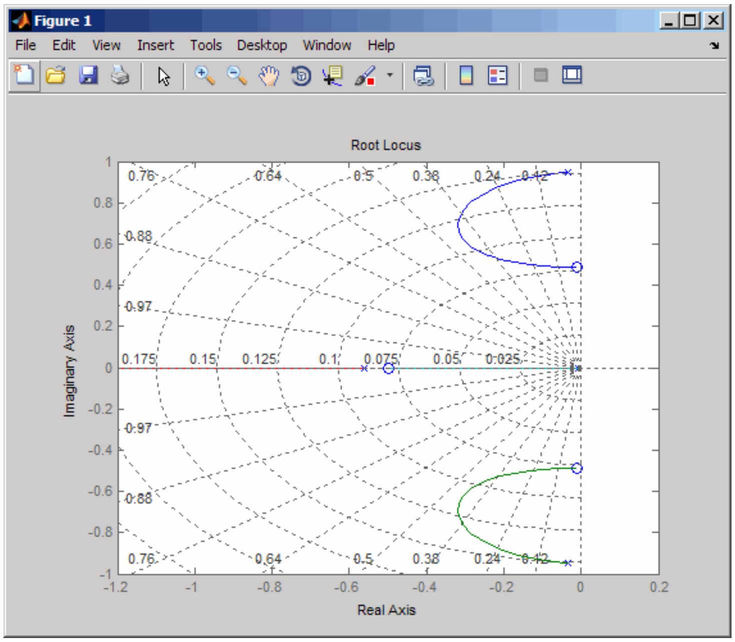
A reasonable design objective is to provide a damping ration $\zeta > 0.35$ with a natural frequency $\omega_n < 1.0$ rad/sec. Since the simplest compensator is a static gain, first try to determine appropriate gain values using the root locus technique.

```
% Plot the root locus for the rudder to yaw channel
rlocus(sys11)
```

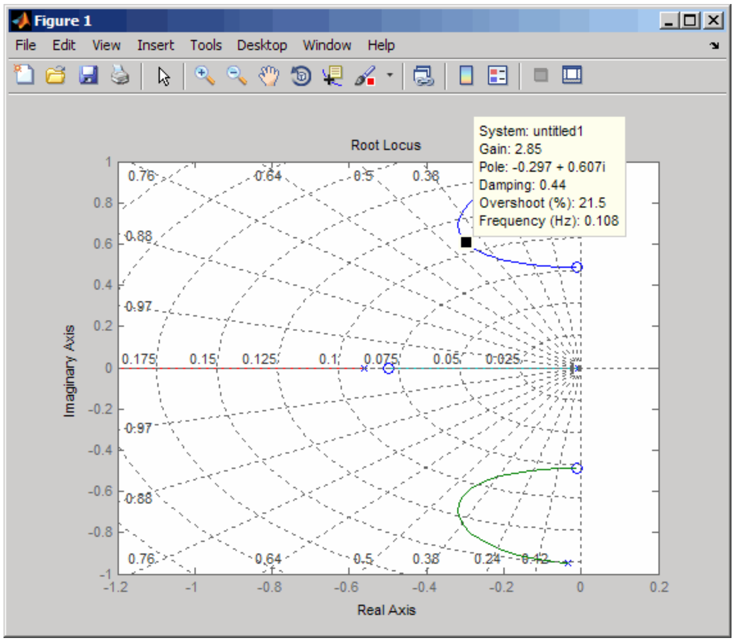


This is the root locus for negative feedback and shows that the system goes unstable almost immediately. If, instead, you use positive feedback, you may be able to keep the system stable.

```
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.



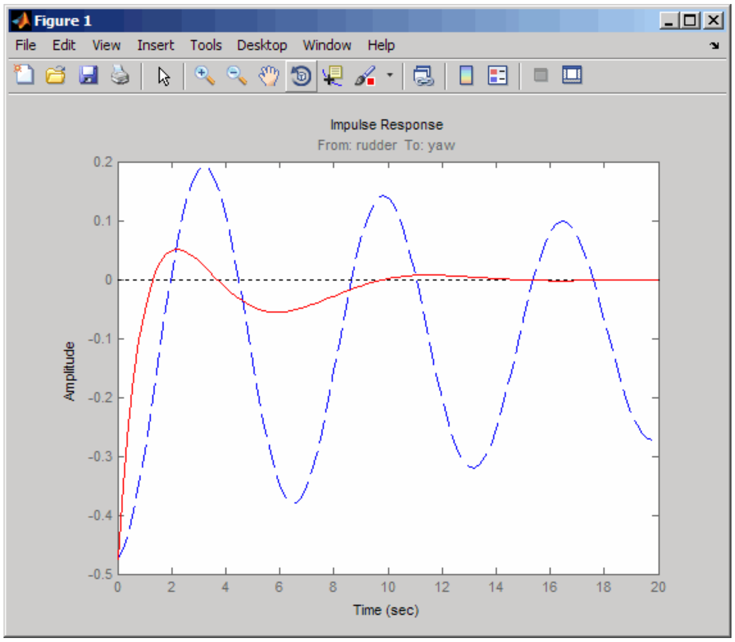
System: untitled1
Gain: 2.85
Pole: -0.297 + 0.607i
Damping: 0.44
Overshoot (%): 21.5
Frequency (Hz): 0.108

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.

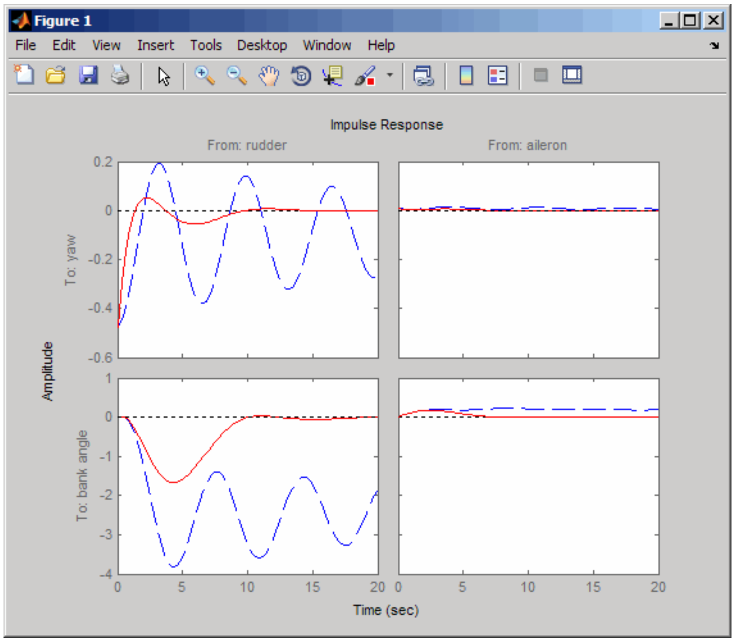
Now close the loop on the full MIMO model and see how the response from the aileron looks. The feedback loop involves input 1 and output 1 of the plant (use `feedback` with index vectors selecting this input/output pair). At the MATLAB prompt, type

```
cloop = feedback(sys,-K,1,1);
damp(cloop)    % closed-loop poles
```

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

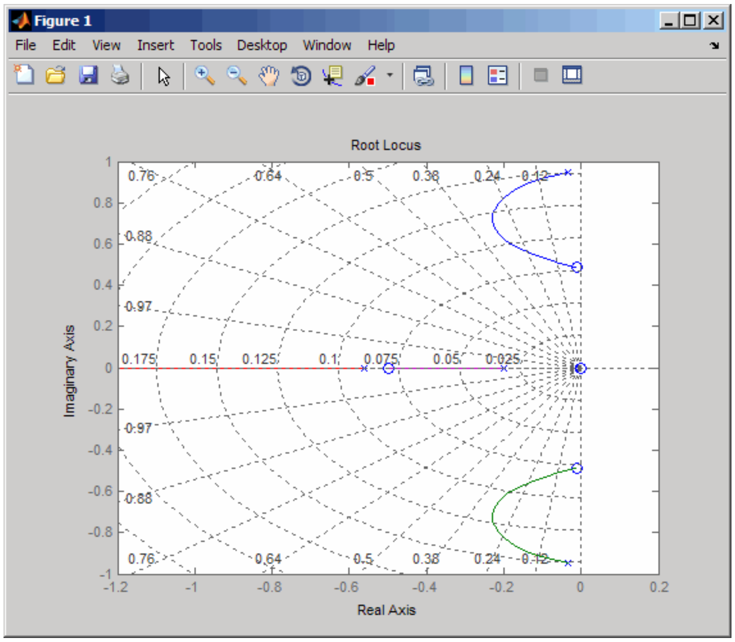
```
H = zpk(0,-0.2,1);
```

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;
```

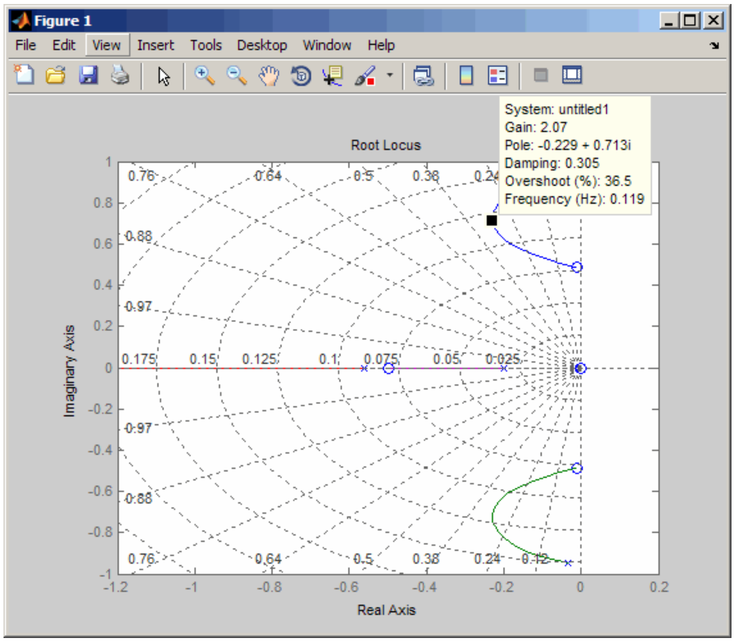
and draw another root locus for this open-loop model.

```
rlocus(-oloop)
sgrid
```



Create and drag a data marker around the upper curve to locate the maximum damping, which is about $\zeta = 0.3$.

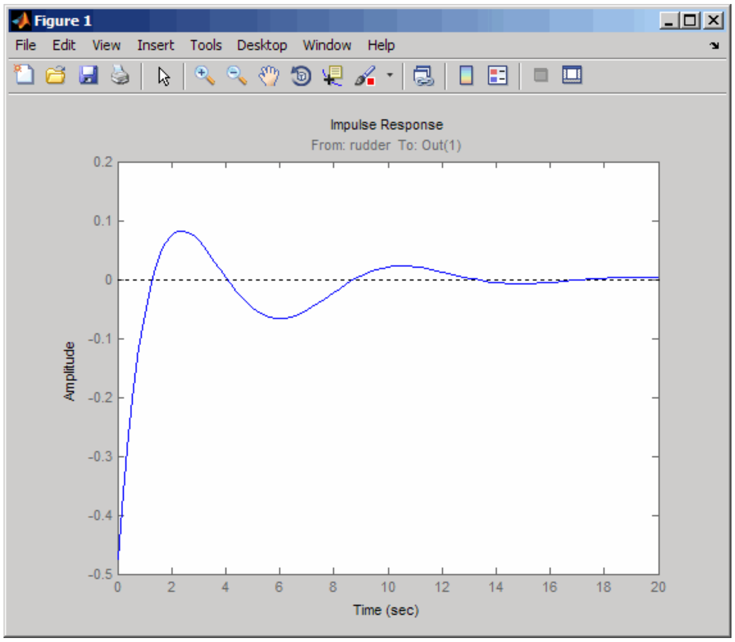
This figure shows a data marker at the maximum damping ratio; the gain is approximately 2.07.



System: untitled1
Gain: 2.07
Pole: -0.229 + 0.713i
Damping: 0.305
Overshoot (%): 36.5
Frequency (Hz): 0.119

Look at the closed-loop response from rudder to yaw rate.

```
K = 2.07;
cl11 = feedback(oloop,-K);
impulse(cl11,20)
```

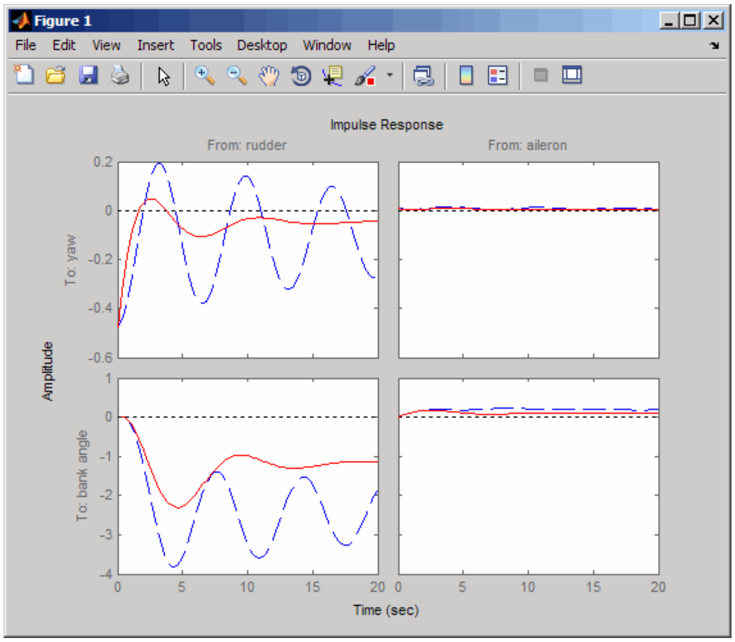


The response settles nicely but has less damping than your previous design. Finally, you can verify that the washout filter has fixed the spiral mode problem. First form the complete washout filter $kH(s)$ (washout + gain).

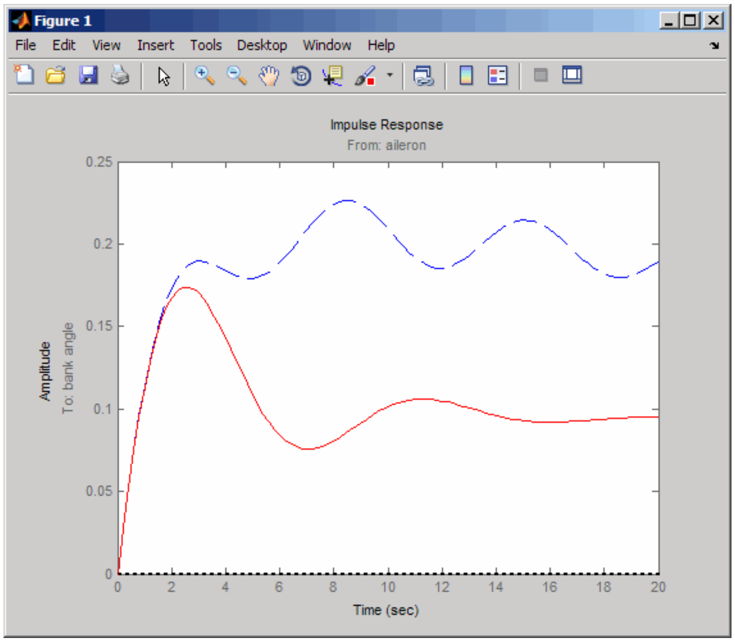
```
WOF = -K * H;
```

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.

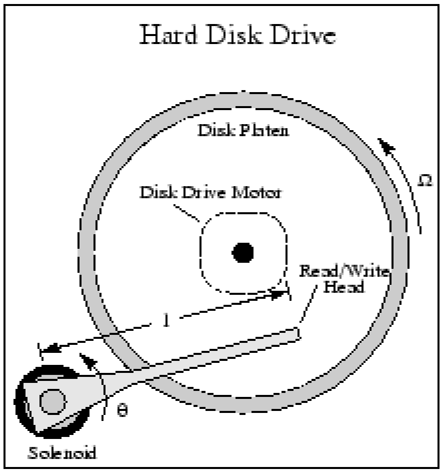


Although you did not quite meet the damping specification, your design has increased the damping of the system substantially and now allows the pilot to fly the aircraft normally.

Hard-Disk Read/Write Head Controller

In this section...

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

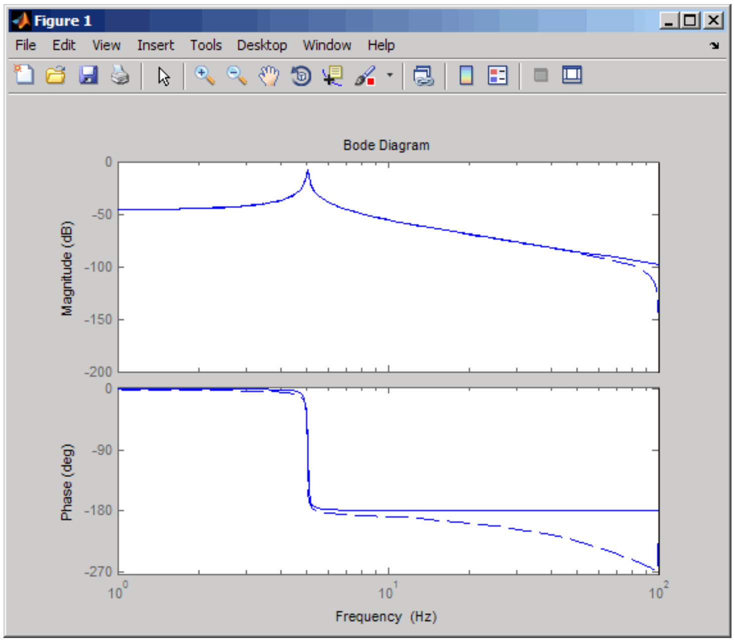
The task here is to design a digital controller that provides accurate positioning of the read/write head. The design is performed in the digital domain. First, discretize the continuous plant. Because our plant will be equipped with a digital-to-analog converter (with a zero-order hold) connected to 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
```

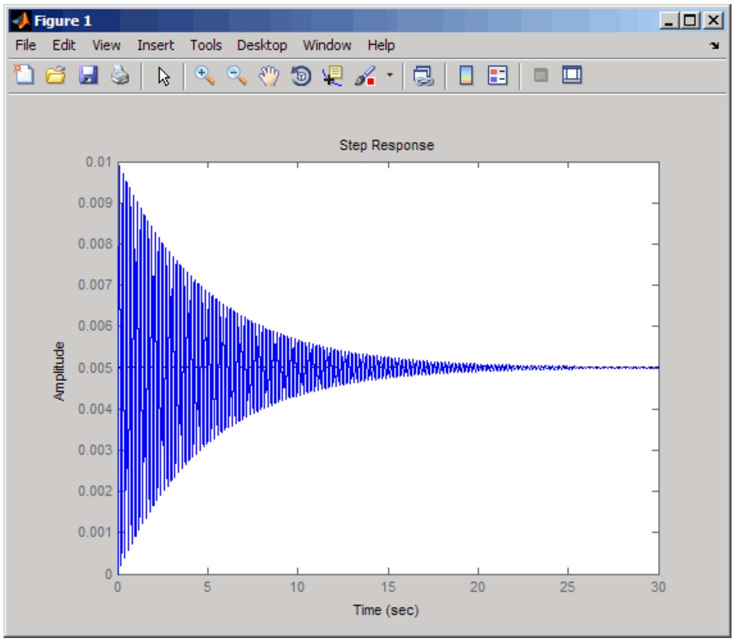
You can compare the Bode plots of the continuous and discretized models with

```
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

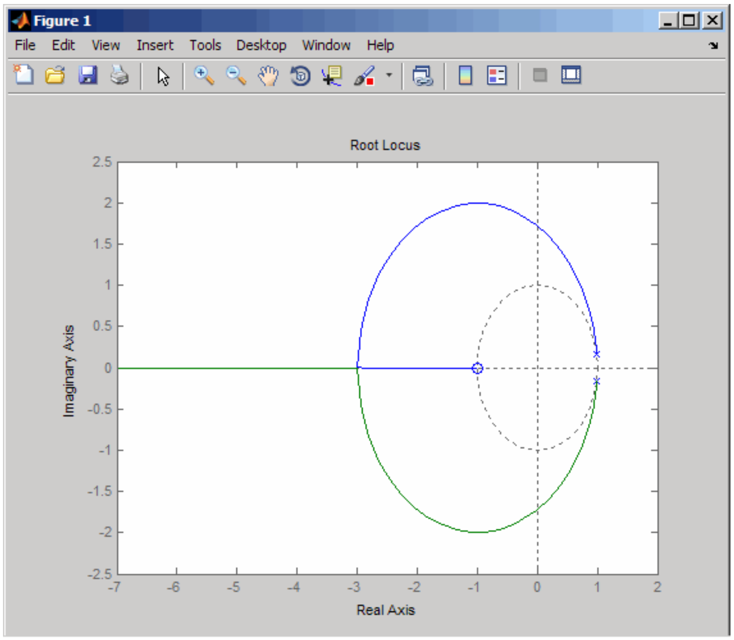
```
% Open-loop poles of discrete model
damp(Hd)
         Pole                Magnitude     Damping       Frequency       Time Constant
                                                       (rad/seconds)       (seconds)

  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.

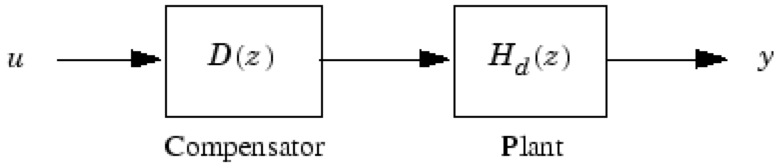
Adding a Lead Network

Try the compensator

$$D(z) = \frac{z+\alpha}{z+b}$$

with $a = -0.85$ and $b = 0$.

The corresponding open-loop model



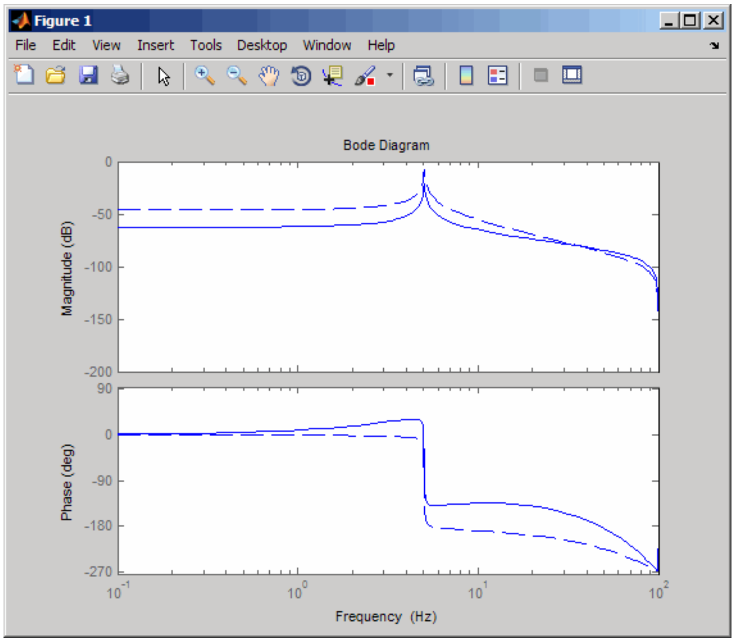
is obtained by the series connection

```
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.

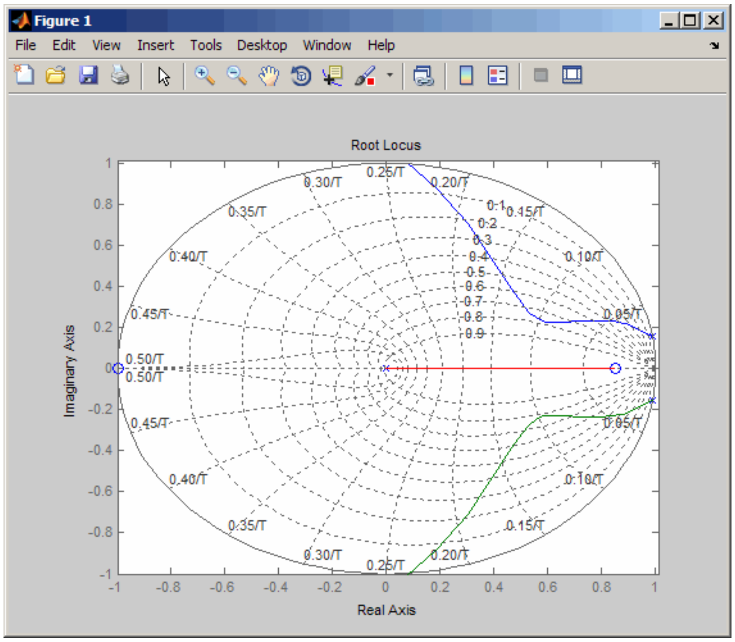


The plot above shows that the compensator has shifted up the phase plot (added lead) in the frequency range ω > 10 rad/sec.

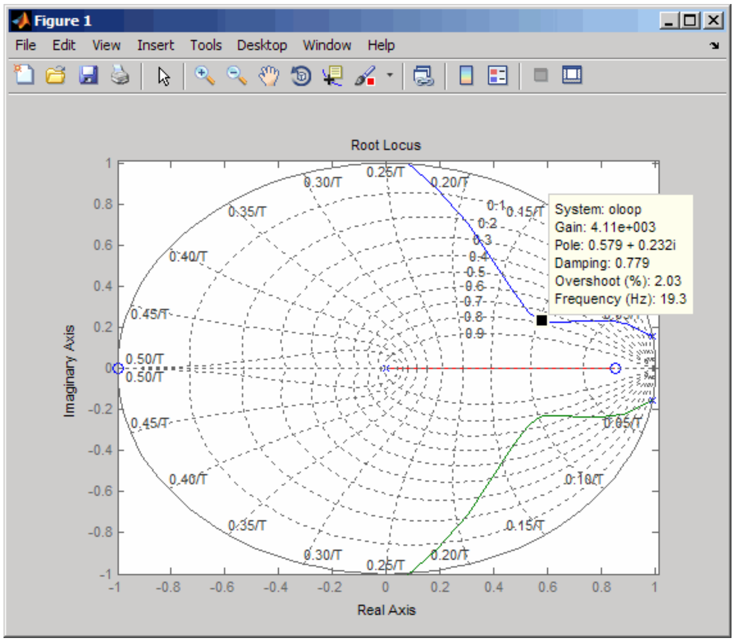
Now try the root locus again with the plant and compensator as open loop.

```
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 ζ = 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.

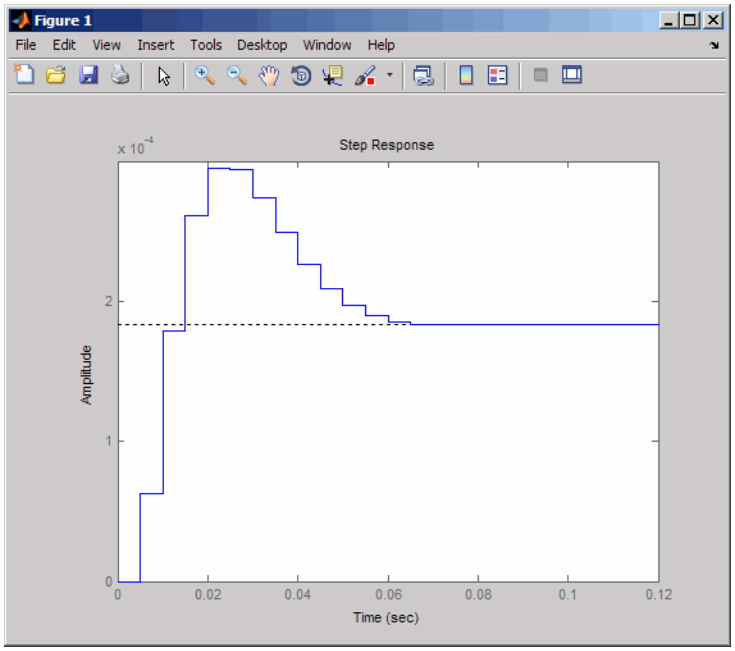


System: oloop
Gain: 4.11e+003
Pole: 0.579 + 0.232i
Damping: 0.779
Overshoot (%): 2.03
Frequency (Hz): 19.3

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)
```

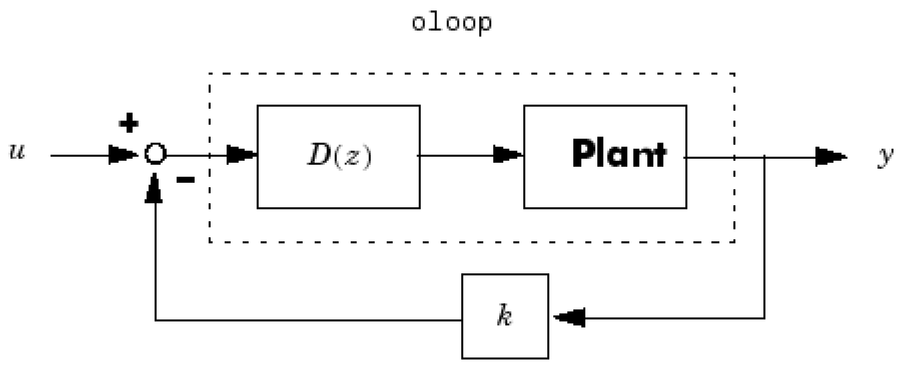


This response depends on your closed loop set point. The one shown here is relatively fast and settles in about 0.07 seconds. Therefore, this closed loop disk drive system has a seek time of about 0.07 seconds. This is slow by today's standards, but you also started with a very lightly damped system.

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
```

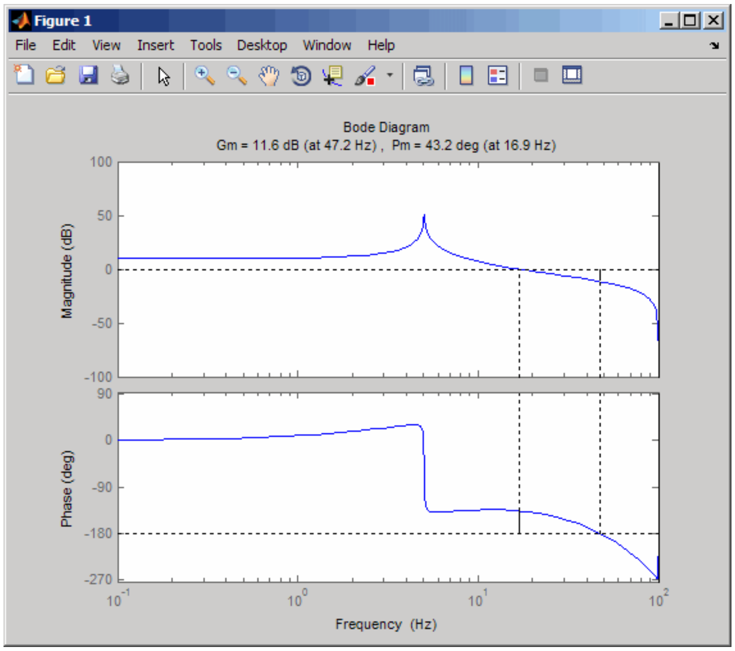
To obtain the gain margin in dB, type

```
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.

LQG Regulation: Rolling Mill Case Study

In this section...

Overview of this Case Study

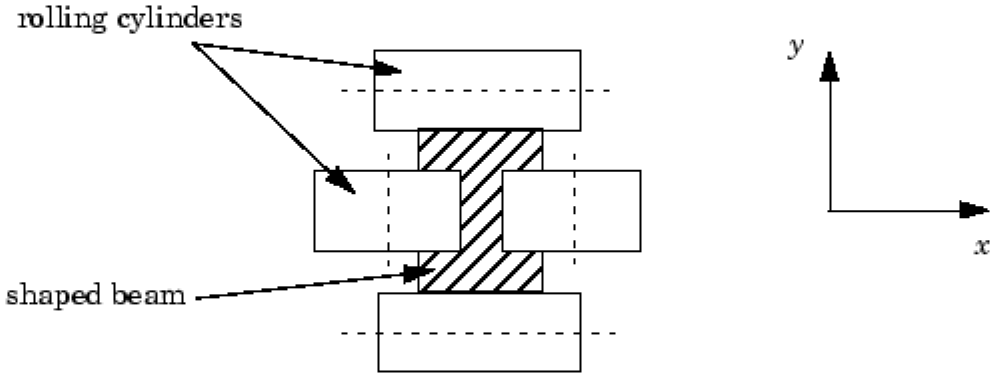
This case study demonstrates the use of the LQG design tools in a process control application. The goal is to regulate the horizontal and vertical thickness of the beam produced by a hot steel rolling mill. This example is adapted from [1]. The full plant model is MIMO and the example shows the advantage of direct MIMO LQG design over separate SISO designs for each axis. Type

```
milldemo
```

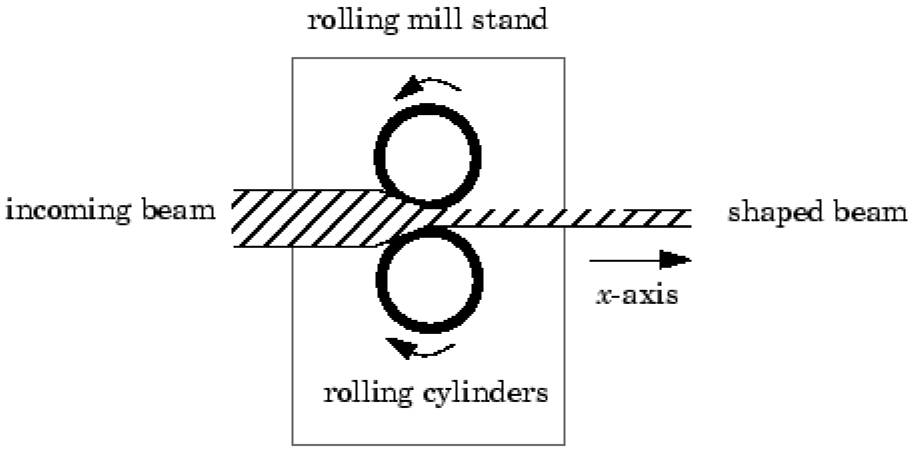
at the command line to run this demonstration interactively.

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*.



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:

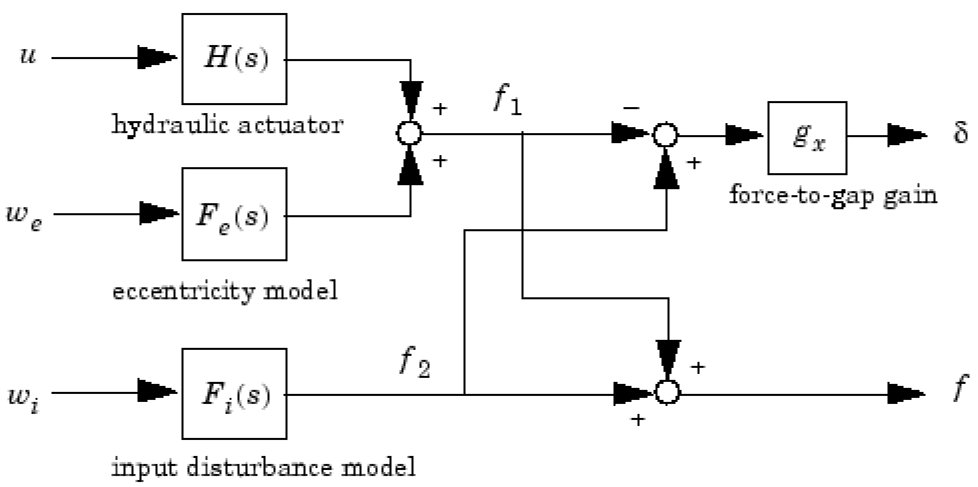
- Variations in the thickness/hardness of the incoming beam
- Eccentricity in the rolling cylinders

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 15-30. 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)
```

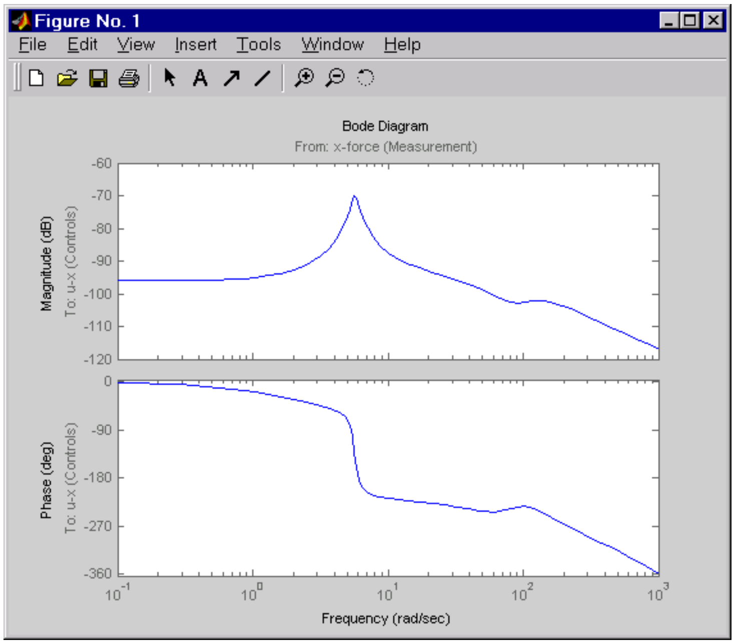
Finally, connect the state-feedback gain `kx` and state estimator `estx` to form the LQG regulator.

```
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^o 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`.

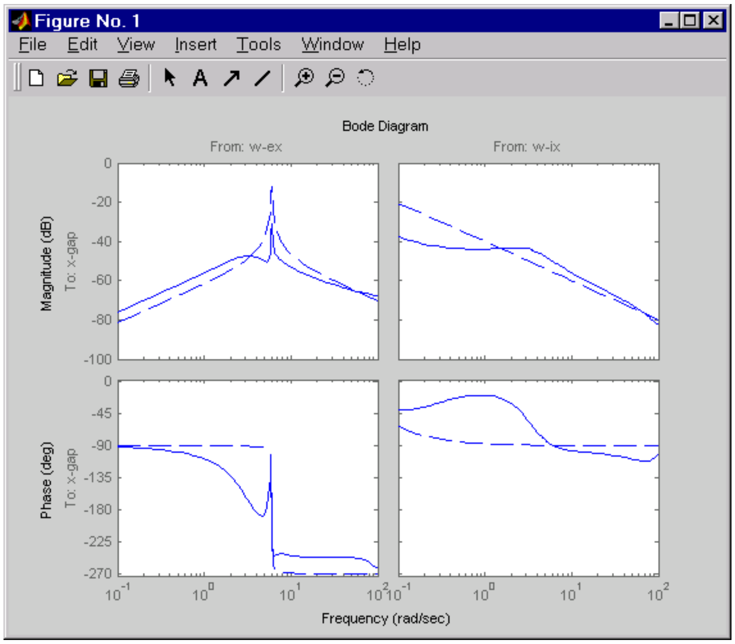
```
Px.inputname
ans =
    'u-x'
    'w-ex'
    'w-ix'

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')
```

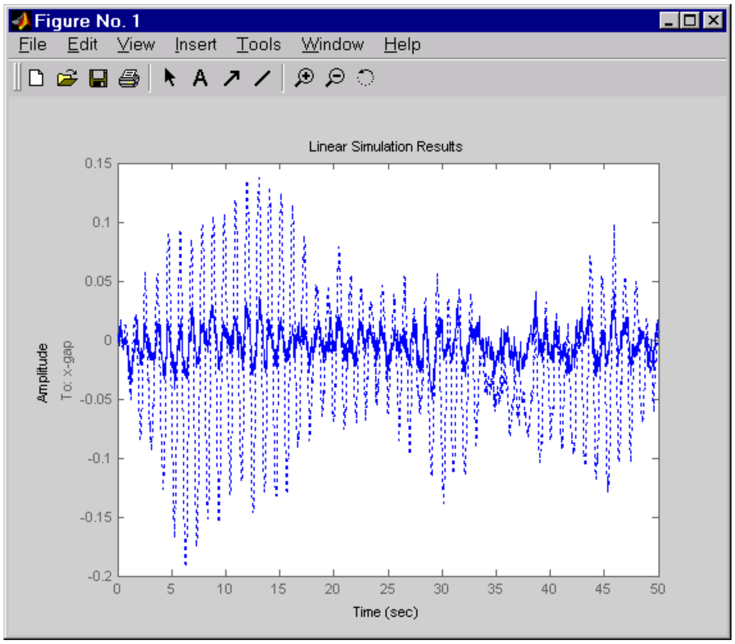


The dashed lines show the open-loop response. Note that the peak gain of the eccentricity-to-gap response and the low-frequency gain of the input-thickness-to-gap response have been reduced by about 20 dB.

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.

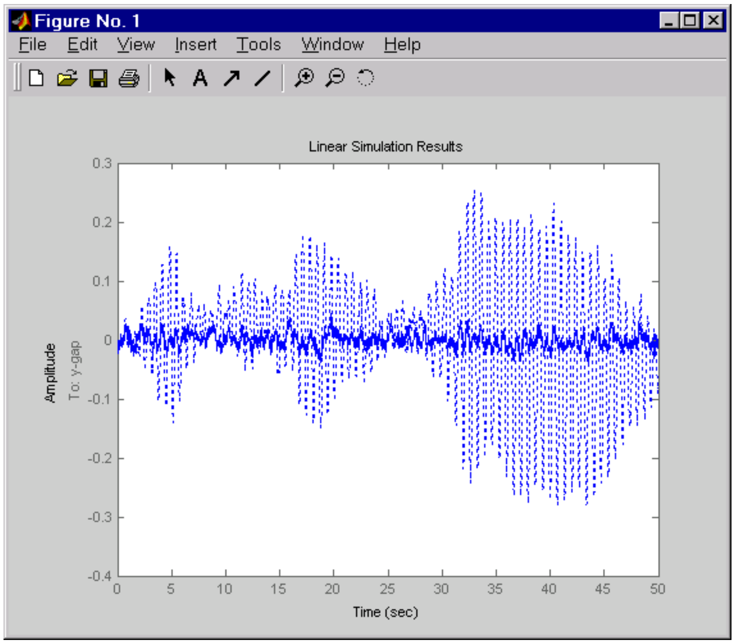
```
% Specify model components
Hy = tf(7.8e8,[1 71 88^2],'inputn','u-y')
Fiy = tf(2e4,[1 0.05],'inputn','w-iy')
Fey = tf([1e5 0],[1 0.19 9.4^2],'inputn','w-ey')
gy = 0.5e-6 % force-to-gap gain

% 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.



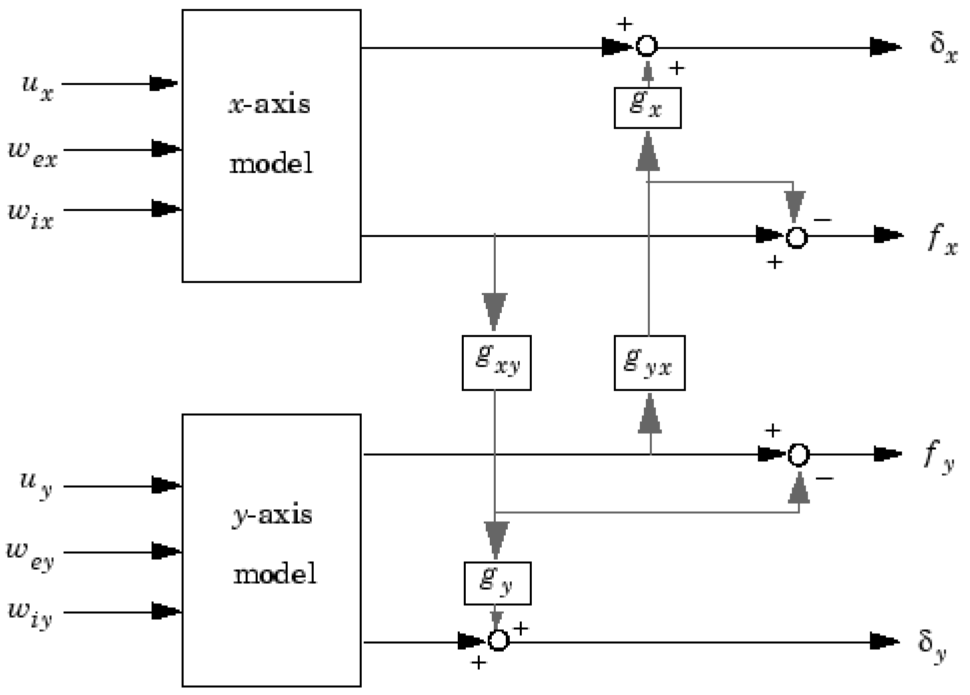
The dotted lines correspond to the open-loop response. The simulation results are comparable to those for the *x*-axis.

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
```

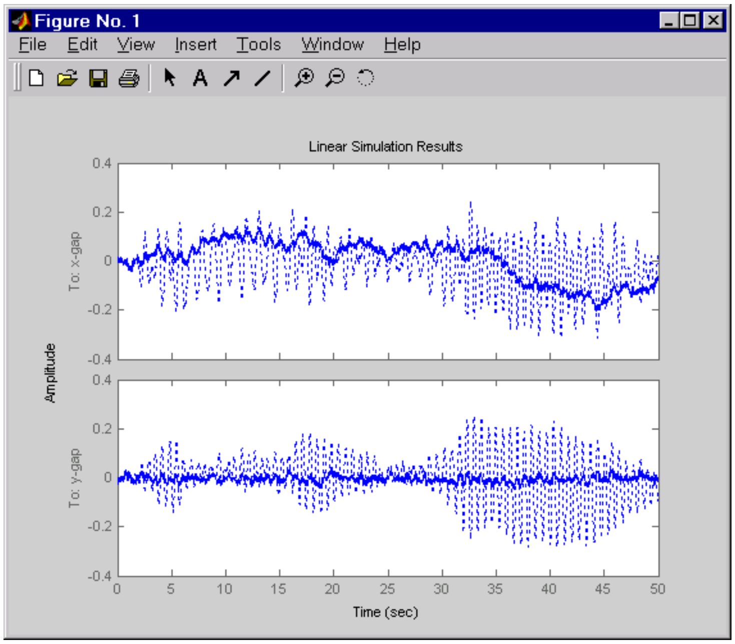
To simulate the closed-loop response, also form the closed-loop model by

```
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 15-41. 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).

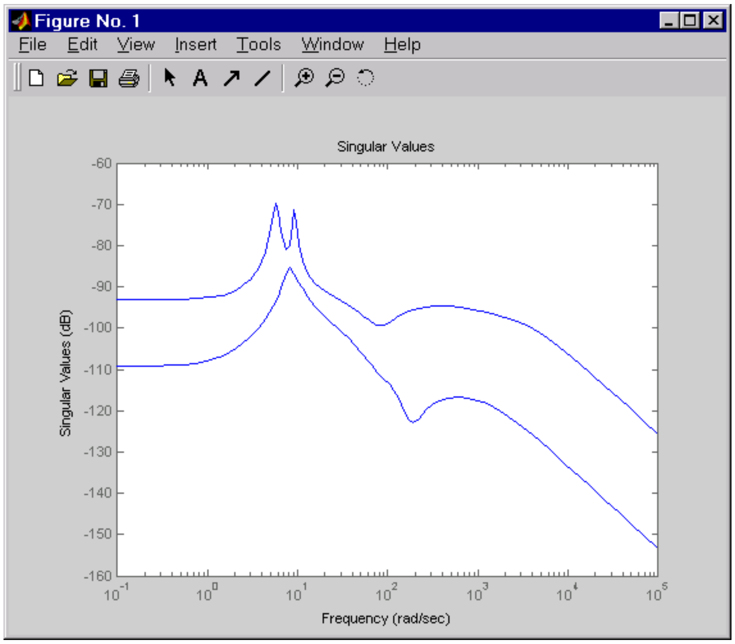
```
k = lqry(Pdes(1:2,1:2),eye(2),1e-4*eye(2))      % LQ gain
est = kalman(Pdes(3:4,:),eye(4),1e3*eye(2))     % Kalman estimator

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).

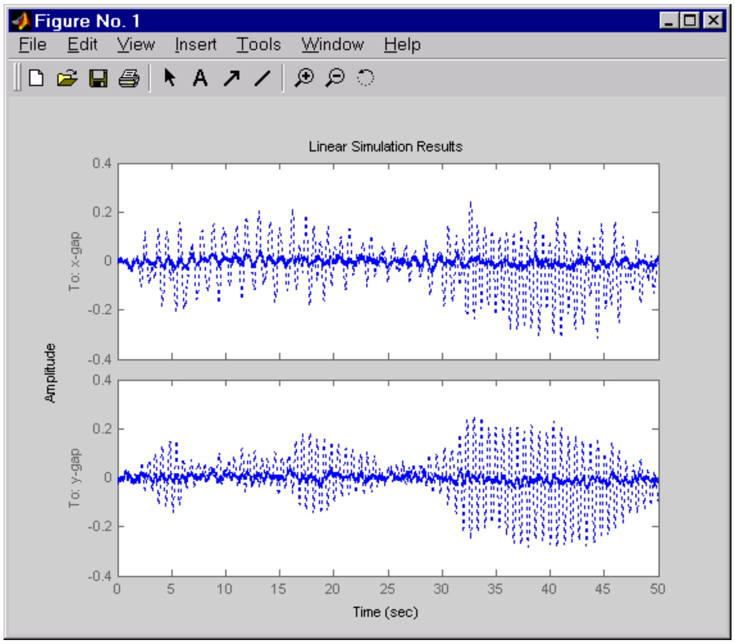
```
sigma(RegMIMO)
```



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:

$$\begin{aligned}\hat{x}[n+1|n] &= A(I-MC)\,\hat{x}[n|n-1] + \begin{bmatrix} B & AM \end{bmatrix}\begin{bmatrix} u[n] \\ y_v[n] \end{bmatrix}\\ \hat{y}[n|n] &= C(I-MC)\,\hat{x}[n|n-1] + CMy_v[n].\end{aligned}$$

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:

$$\begin{aligned}x[n+1] &= Ax[n] + Bu[n] + Bw[n]\\ y[n] &= Cx[n]\end{aligned}$$

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.

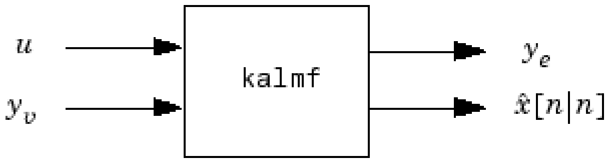
```
Q = 1;
R = 1;
[kalmf,L,P,M] = kalman(Plant,Q,R);
```

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 $\hat{x}[n|n]$.



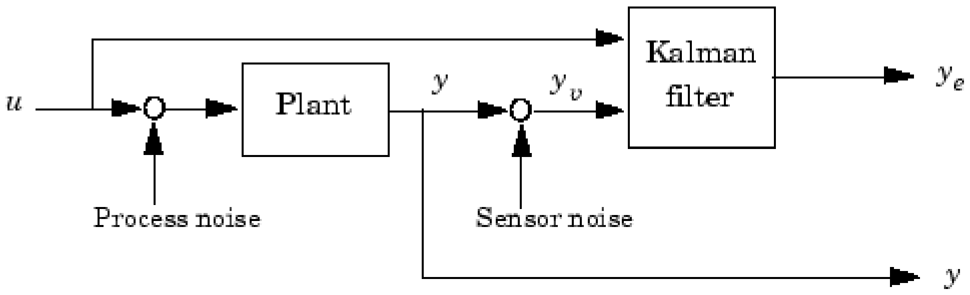
Kalman filter

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.

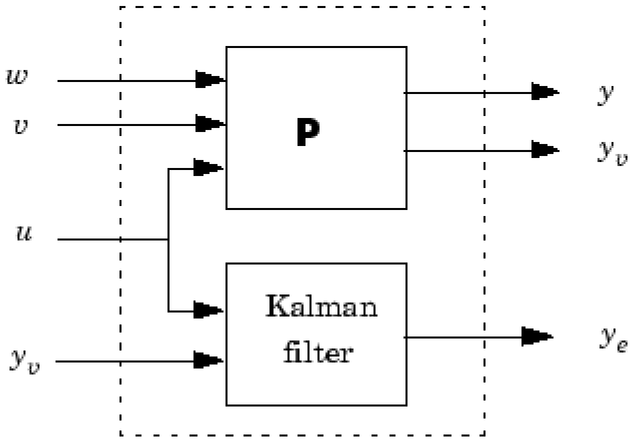
The block diagram below shows how to generate both true and filtered outputs.



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.

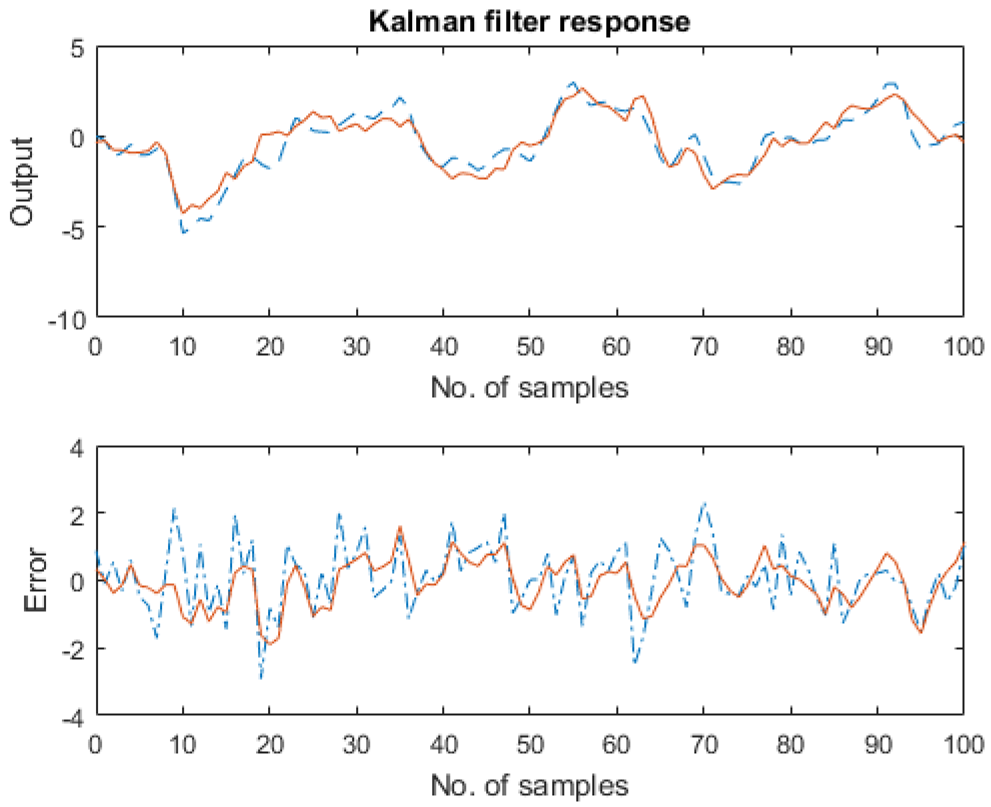
```
t = [0:100]';
u = sin(t/5);

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.

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] + Gw[n] \\ y_v[n] &= Cx[n] + v[n]\,. \end{aligned}$$

The time-varying Kalman filter is given by the following recursions:

- Measurement update:

$$\begin{aligned} \hat{x}[n|n] &= \hat{x}[n|n-1] + M[n]\left(y_v[n] - C\hat{x}[n|n-1]\right) \\ M[n] &= P[n|n-1]\,C^T\left(R[n] + CP[n|n-1]\,C^T\right)^{-1} \\ P[n|n] &= (I - M[n]\,C)P[n|n-1]\,. \end{aligned}$$

- Time update:

$$\begin{aligned} \hat{x}[n+1|n] &= A\hat{x}[n|n] + Bu[n] \\ P[n+1|n] &= AP[n|n]\,A^T + GQ[n]\,G^T. \end{aligned}$$

Here, $\hat{x}[n|n-1]$ and $\hat{x}[n|n]$ are as described previously. Additionally:

$$\begin{aligned}
&Q[n] = E(w[n]\,w[n]^T) \\
&R[n] = E(v[n]\,v[n]^T) \\
&P[n|n] = E(\{x[n] - x[n|n]\}\{x[n] - x[n|n]\}^T) \\
&P[n|n-1] = E(\{x[n] - x[n|n-1]\}\{x[n] - x[n|n-1]\}^T).
\end{aligned}$$

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:

$$x[1|0] = 0, \quad P[1|0] \; = BQB^T$$

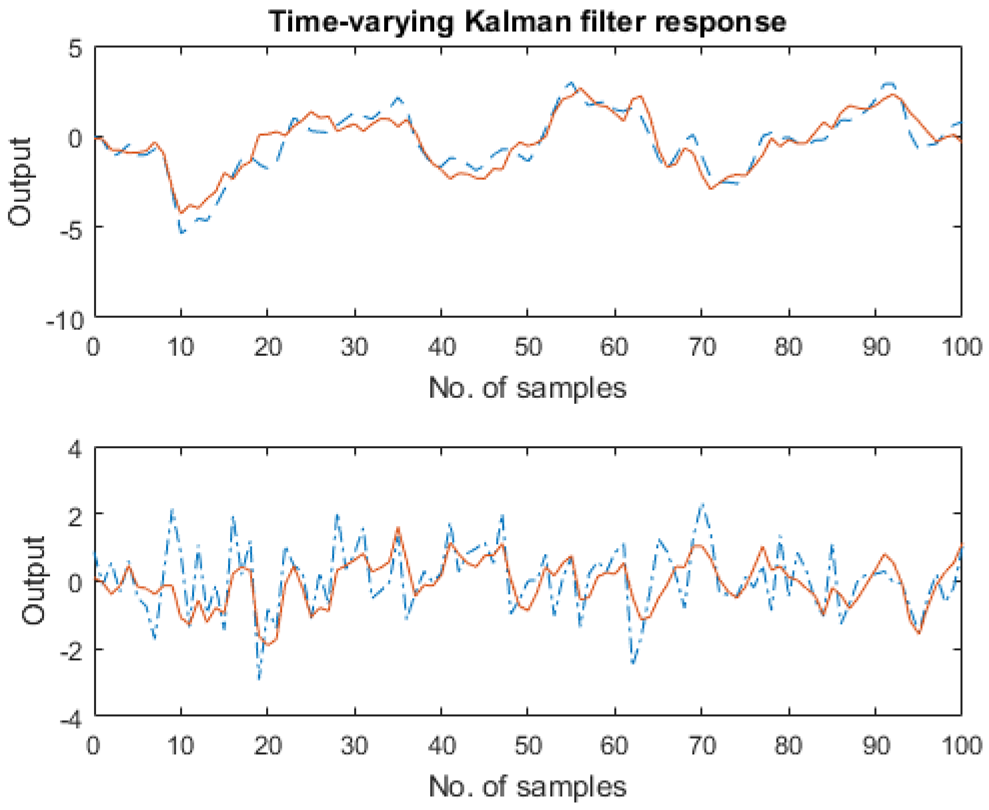
Implement the time-varying filter with a `for` loop.

```
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.

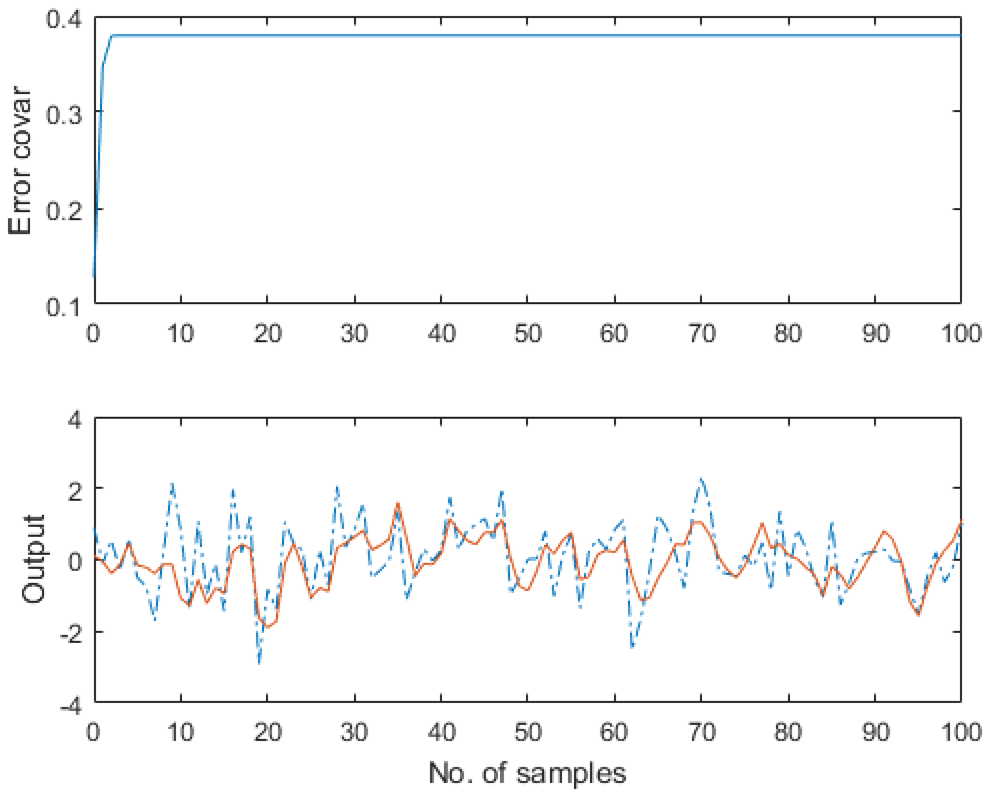
```
subplot(211), plot(t,y,'--',t,ye,'-')
title('Time-varying Kalman filter response')
xlabel('No. of samples'), ylabel('Output')
subplot(212), plot(t,y-yv,'-.',t,y-ye,'-')
xlabel('No. of samples'), ylabel('Output')
```



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.

16

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 16-3.

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 16-3
- "Specifying the Frequency Axis Limits in the Scaling Tool GUI" on page 16-5
- "Specifying the Frequency Band for Maximum Accuracy in the Scaling Tool GUI" on page 16-5
- "Saving the Scaling in the Scaling Tool GUI" on page 16-6

For an example of using the Scaling Tool GUI on a real model, see Scaling Models to Maximize Accuracy.

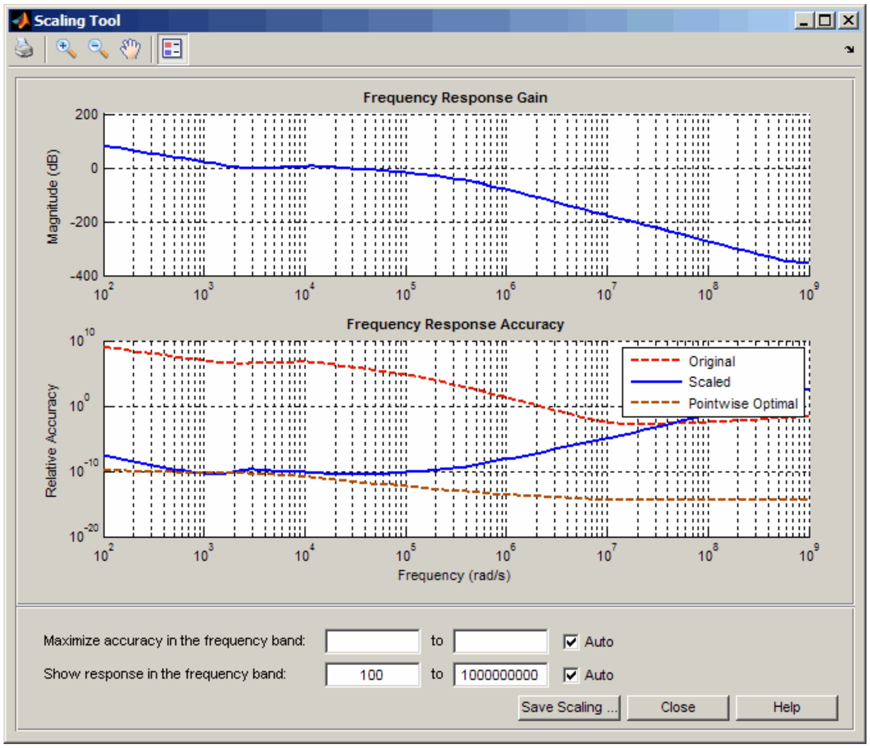
For more information about scaling models from the command line, see the `prescale` reference page.

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.



Maximize accuracy in the frequency band: [ ] to [ ] ☑ Auto

Show response in the frequency band: 100 to 1000000000 ☑ Auto

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.

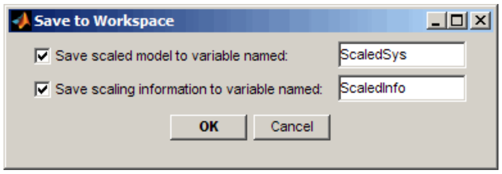
Tip To return to the default scaling, select the **Auto** check box.

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.

   ☑ Save scaled model to variable named: ScaledSys

   ☑ Save scaling information to variable named: ScaledInfo

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.

Using the SISO Design Tool and the Linear System Analyzer

SISO Design Tool

SISO Design Tool Overview

The SISO Design Tool is a graphical-user interface (GUI) to design compensators.

The SISO Design Tool has the following components:

- The SISO Design Task Node in the Control and Estimation Tools Manager, a user interface (UI) that facilitates the design of compensators for single-input, single-output feedback loops through a series of interactive panes.
- The Graphical Tuning Window, a graphical user interface (GUI) for displaying and manipulating the Bode, root locus, and Nichols plots for the controller currently being designed. This window is titled SISO Design for *Design Name*. The Graphical Tuning Window by default displays the root locus and Bode diagrams for your imported systems. The two are dynamically linked; for example, if you change the gain in the root locus, it immediately affects the Bode diagrams as well.
- The SISO Design Task-associated Linear System Analyzer.
- A tool that automatically generates compensators using PID, internal model control (IMC), or linear-quadratic-Gaussian (LQG) methods.
- Optimization-based tuning methods that automatically tune the system to satisfy design requirements (available if you have Simulink Design Optimization software installed).

For more information about SISO Design Tool menus and options, see:

- "Opening the SISO Design Tool" on page 17-3
- "Using the SISO Design Task Node" on page 17-4
- "Using the SISO Design Task in the Controls & Estimation Tools Manager" on page 17-11
- "SISO Design Task Graphical Tuning Window" on page 17-40
- "Using the Graphical Tuning Window Menu Bar" on page 17-42
- "Using the Graphical Tuning Window Toolbar" on page 17-53
- "Using the Right-Click Menus in the Graphical Tuning Window" on page 17-54
- "Linear System Analyzer for SISO Design Task Design Requirements" on page 17-73

Opening the SISO Design Tool

To open the SISO Design Task node in the Control and Estimation Tools Manager and the Graphical Tuning Window, type

```
controlSystemDesigner
```

Load the `Gservo` model that you want to control in the MATLAB workspace by typing

```
load ltiexamples
```

In the **Architecture** tab of the Control and Estimation Tools Manager GUI, click **System Data** to import the model into the SISO Design Tool.

Tip You can import LTI models and arrays of LTI models for the plant **G** and sensor **H** into the SISO Design Tool. For more information, see "Importing Models into the SISO Design Tool".

Alternatively, to open the SISO Design Tool with the `Gservo` system, type

```
controlSystemDesigner(Gservo)
```

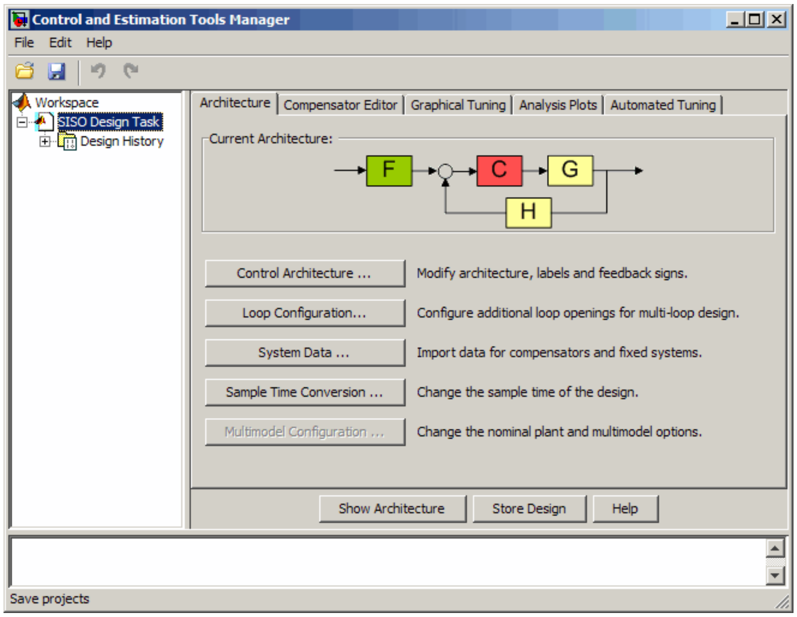
Note: You can also open the SISO Design Tool from the MATLAB desktop. To do so, in the **Apps** tab, click **Control System Designer**. Then, in the **Control and Estimation Tools Manager**, click **System Data** to specify your plant model.

Using the SISO Design Task Node

In this section...
"The SISO Design Task Node" on page 17-4
"SISO Design Task Node Menu Bar" on page 17-4

The SISO Design Task Node

The following figure shows the SISO Design Task node in the Control and Estimation Tools Manager.



Control Architecture ... Modify architecture, labels and feedback signs.
Loop Configuration... Configure additional loop openings for multi-loop design.
System Data ... Import data for compensators and fixed systems.
Sample Time Conversion ... Change the sample time of the design.
Multimodel Configuration ... Change the nominal plant and multimodel options.

SISO Design Task Node on the Control and Estimation Tools Manager

SISO Design Task Node Menu Bar

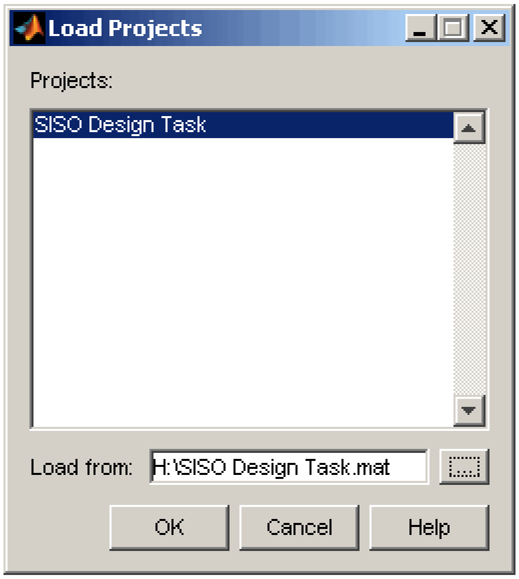
The SISO Design Task node menu bar contains the following menus:

File Menu Options
Load
Save
Export
Close

- **Load**

  To load a saved SISO Design Tool project, select **Load** from the **File** menu. This opens the Load Projects window.

  Projects are saved as MAT-files. Select the project you want to load from the list, or click **...** to browse for projects you can select from, and click **OK**.

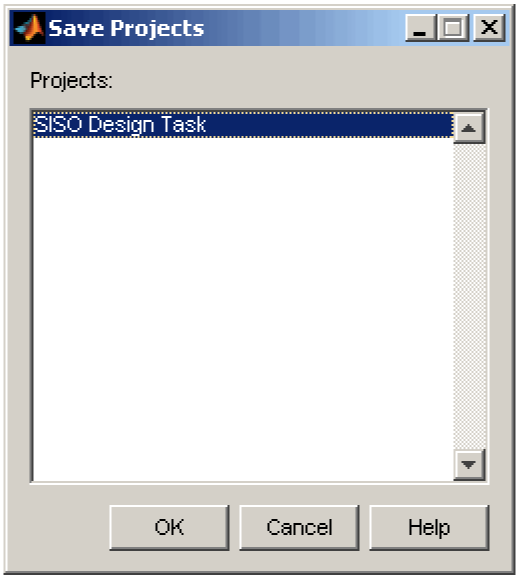


Projects:
SISO Design Task

Load from: H:\SISO Design Task.mat

- **Save**

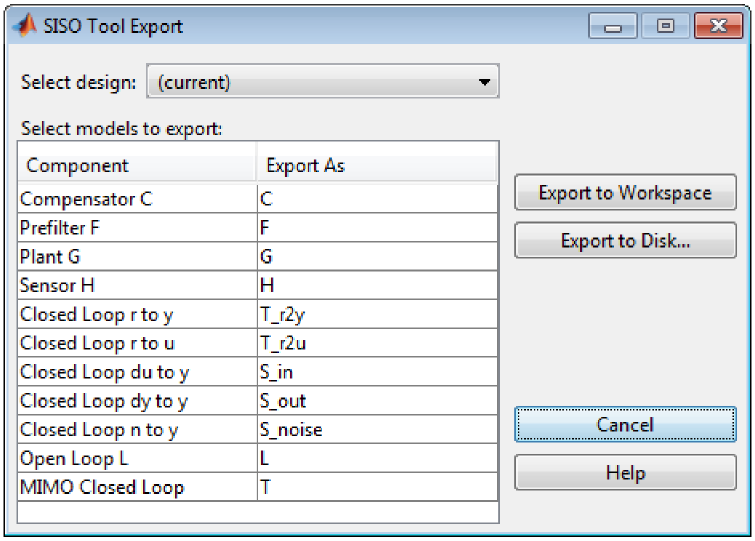
  You can exit the MATLAB technical computing environment and later restore the SISO Design Tool to the state you left it in by saving the project. Select **Save** from the **File** menu. This opens the **Save Projects** window.



To save a project, specify a file name and click **OK**. The current state and configuration of your SISO Design Tool are saved as a MAT-file. To load a saved project, select **Load** from the **File** menu (see previous bullet).

- **Export**

  Selecting **Export** from the **File** menu opens the SISO Tool Export window.



Component	Export As
Compensator C	C
Prefilter F	F
Plant G	G
Sensor H	H
Closed Loop r to y	T_r2y
Closed Loop r to u	T_r2u
Closed Loop du to y	S_in
Closed Loop dy to y	S_out
Closed Loop n to y	S_noise
Open Loop L	L
MIMO Closed Loop	T

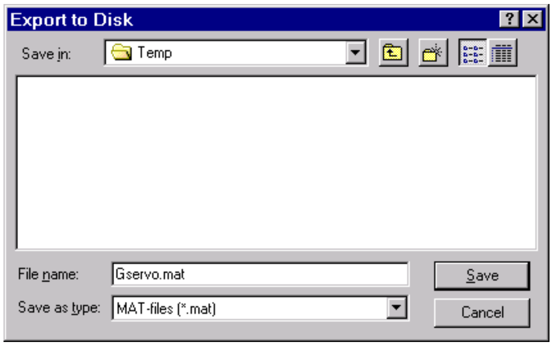
You can perform the following tasks in this window:

- Export models to the MATLAB workspace or to a disk. The exported models are:
  - LTI objects if the plant and sensor are LTI models
  - Arrays of LTI objects if the plant or sensor are arrays of LTI models
- Rename models when exporting
- Save variations on models, including open and closed loop models, sensitivity transfer functions, and state-space representations

To export models to the MATLAB workspace, follow these steps:

1. Select the model you want to export from the Component list by left-clicking the model name. To select more than one model, hold down the **Shift** key if they are adjacent on the list. If you want to save nonadjacent models, hold down the **Ctrl** key while selecting the models.
2. For each model you want to save, specify a name in the model's cell in the Export As list. A default name exists if you do not want to assign a new name.
3. Click **Export to Workspace**.

If you want to save your models as a MAT-file, follow steps 1 and 2 and click **Export to Disk**, which opens this window.



Choose where you want to save the file in the **Save in** field and specify the name of the MAT-file in the **File name** field. Click **Save** to save the file.

- **Close**

  Use **Close** to close the SISO Design Tool. This closes all components of the SISO Design Tool.

Edit Menu Options

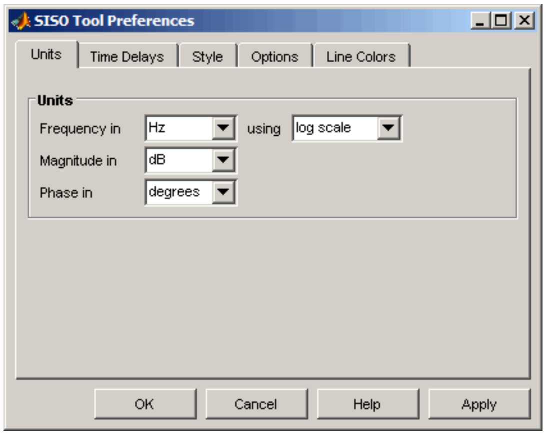
Undo
Redo
SISO Tool Preferences

- **Undo**

  Use **Undo** to go back in design steps. Note that the **Undo** menu changes when the task you have just performed changes. For example, if you change the compensator gain, the **Undo** menu item now reads **Undo Edit Gain**.
- **Redo**

  Use **Redo** to go forward in the design steps. You can only use **Redo** if you have previously used **Undo**. Like the **Undo** menu, the **Redo** menu changes when the task you have just performed changes. For example, if you change the compensator gain, and then select **Undo Edit Gain**, the Redo menu item becomes **Redo Edit Gain**.
- **SISO Tool Preferences**

Select **SISO Tool Preferences** from the **Edit** menu to open the **SISO Tool Preferences** dialog box.



Units

Frequency in: Hz using: log scale

Magnitude in: dB

Phase in: degrees

You can use this window to do the following:

- Change units
- Add plot grids, change font styles for titles, labels, etc., and change axes foreground colors
- Change the compensator format
- Show or hide system poles and zeros in Bode diagrams

For a discussion of this window's features, see ??? online in the Control System Toolbox documentation.

Help

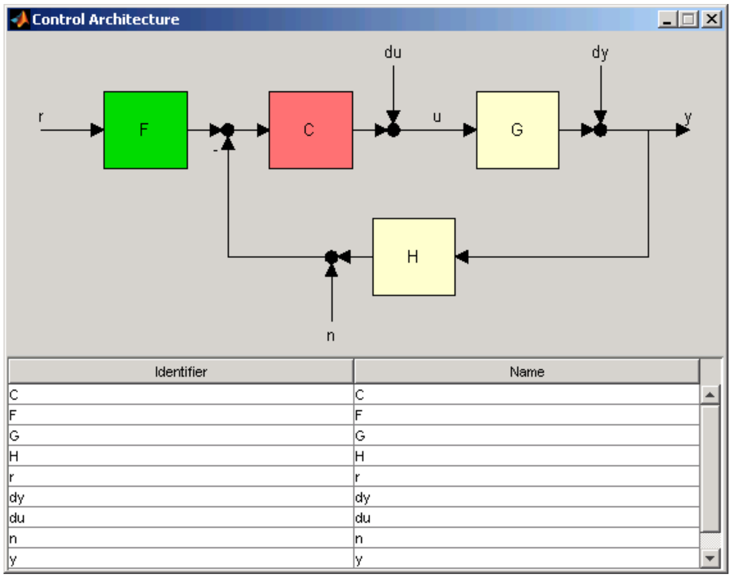
Selecting **About the Control and Estimation Tools Manager** in the **Help** menu opens a window with the version number and a copyright notice for this product.

Buttons Available from Any Pane

- "Showing the Control Architecture" on page 17-10
- "Store Design" on page 17-10

Showing the Control Architecture

Click **Show Architecture** to open a window that displays the block diagram for your model. For example,



Identifier	Name
C	C
F	F
G	G
H	H
r	r
dy	dy
du	du
n	n
y	y

Below the block diagram is a table that shows the default names for each part of the block diagram and the assigned name, if you have one.

Store Design

Click **Store Design** to save your design to your SISO Design Task node. Click on **Design** under the node to see a snapshot summary of your design. Click on the **Design History** node to show a list of all stored designs.

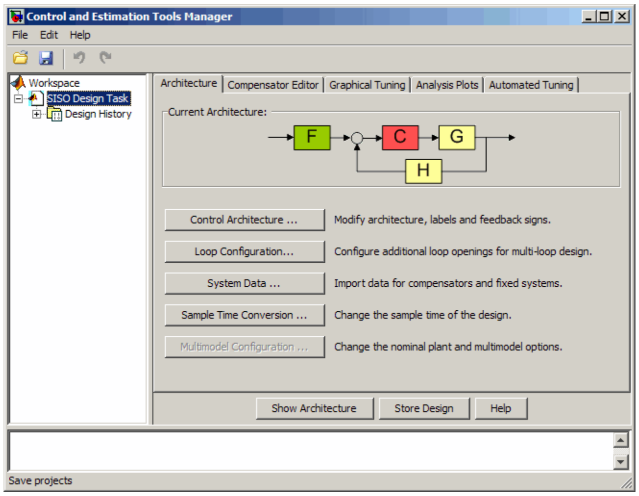
Using the SISO Design Task in the Controls & Estimation Tools Manager

In this section...

Architecture

Use the **Architecture** tab for:

- "Block Diagram Structure Modifications" on page 17-12
- "Loop Configuration" on page 17-13
- "Model Import" on page 17-14
- "Sample Times for Continuous/Discrete Conversions" on page 17-16
- "Multimodel Configuration" on page 17-17

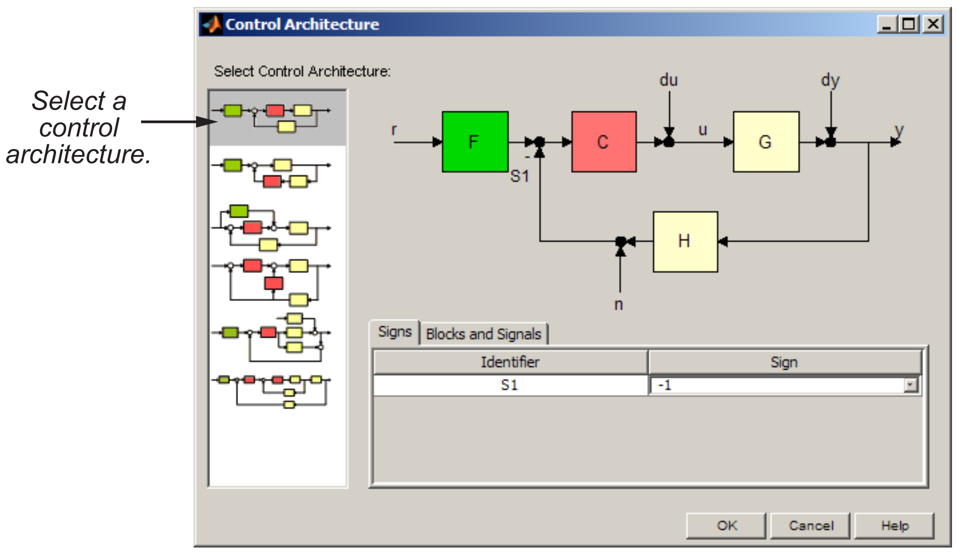


Modify architecture, labels and feedback signs.
Configure additional loop openings for multi-loop design.
Import data for compensators and fixed systems.
Change the sample time of the design.
Change the nominal plant and multimodel options.

Architecture Pane on the SISO Design Task Node

Block Diagram Structure Modifications

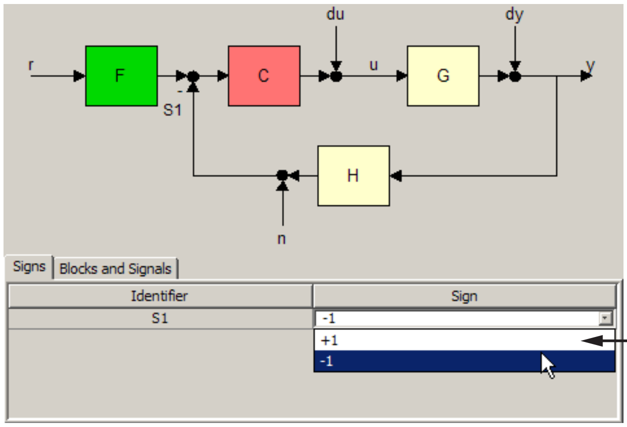
Click **Control Architecture** to change the feedback structure and label signals and blocks. The following pane appears:



Identifier	Sign
S1	-1

Select an architecture from the list of block configurations. These include compensator in the forward path, compensator in the feedback path, feedforward controller, and various multi-loop configurations. The window automatically updates to show the selected configuration.

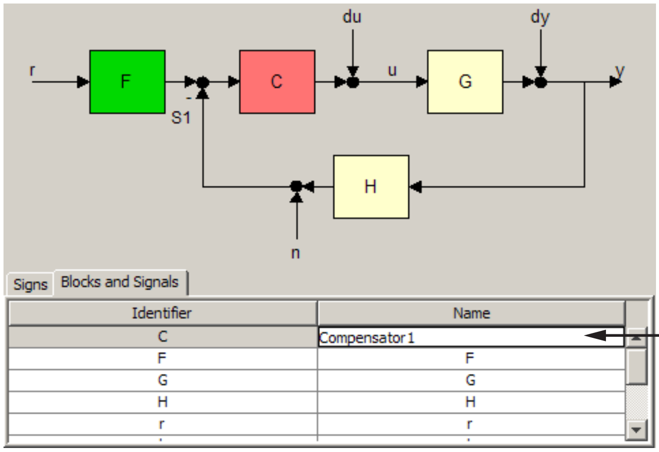
Each configuration has associated Signs and Blocks and Signals panes. This figure shows the Signs pane.



Identifier	Sign
S1	-1

Use menu to toggle between + and - for feedback signals at summing junction.

The Blocks and Signals pane displays the generic identifier, for example F for the prefilter block, and a default name.



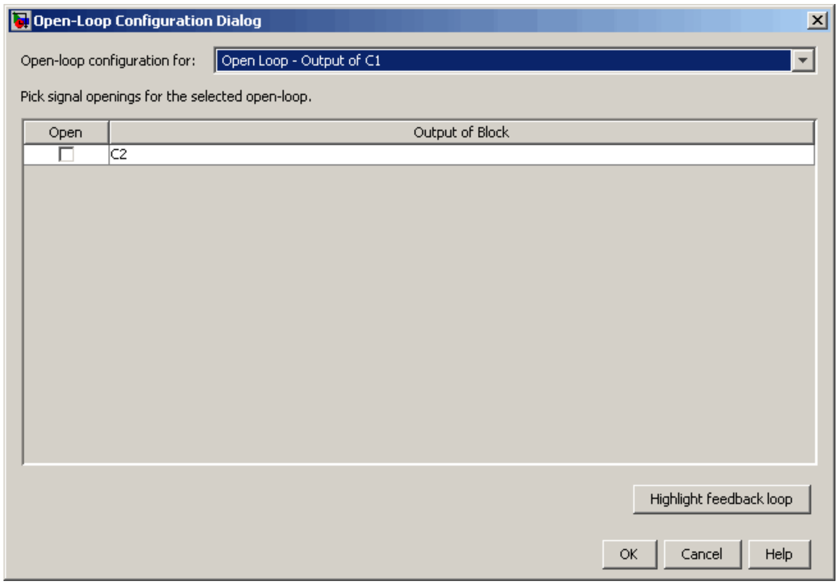
Identifier	Name
C	Compensator1
F	F
G	G
H	H
r	r

Double-click to change name.

On the Signs pane, use the menu to toggle between.

Loop Configuration

Click **Loop Configuration** to configure loops for multi-loop design by opening signals to remove the effects of other feedback loops.


Open-Loop Configuration Dialog
Open-loop configuration for:
Open Loop - Output of C1
Pick signal openings for the selected open-loop.
Highlight feedback loop
OK
Cancel
Help

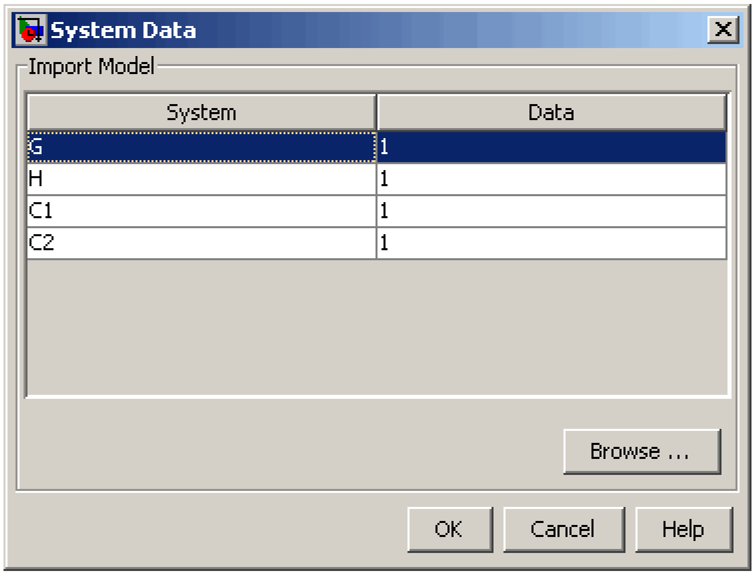
Open	Output of Block
☐	C2

To specify openings for a given open loop, select the loop in the combo box. Click **Highlight Feedback Loop** to see the effects of the selected openings.

For an example of how to use this window in design, see Multi-Loop Compensator Design.

Model Import

Click **System Data** on the Architecture pane to import models into your system. This opens the System Data dialog box, shown below.

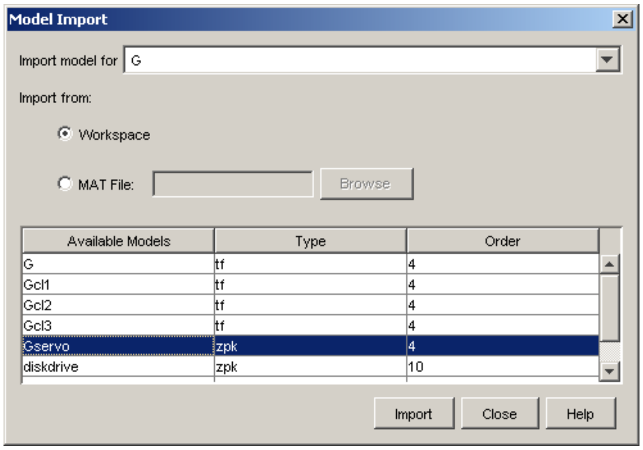


System	Data
G	1
H	1
C1	1
C2	1

You can import models for the plant (**G**), compensator (**C**), prefilter (**F**), and/or sensor (**H**). **G** or **H** or both are LTI models or row or column arrays of LTI models. If both **G** and **H** are arrays, their sizes must match.

To import a model:

1. Select a system in the **System** column and click **Browse**. The Model Import dialog box opens, as shown in the next figure.


Model Import

Import model for: G

Import from:

(● Workspace) ○ MAT File:

Available Models	Type	Order
G	tf	4
Gcl1	tf	4
Gcl2	tf	4
Gcl3	tf	4
Gservo	zpk	4
diskdrive	zpk	10

2 Select a model from the **Available Models** list. You can import models from:

- The MATLAB workspace
- A MAT-file

3 Click **Import**, then click **Close**. You can now see the model loaded into the system selected in the **System Data** dialog.

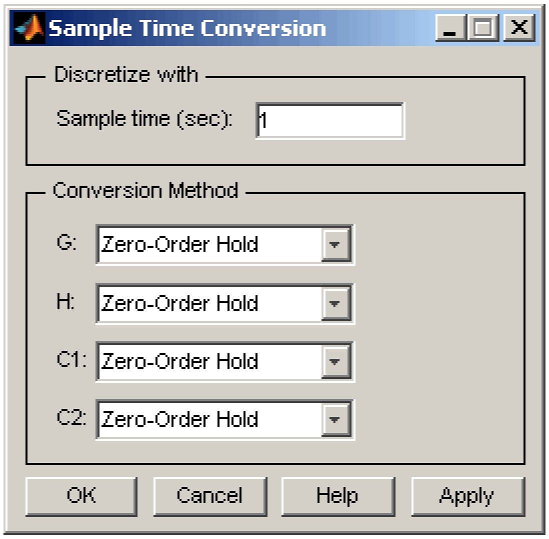
4 Click **OK**. The Graphical Tuning window is updated with the model you loaded.

Alternatively, you can import by entering a valid expression or variable (double, LTI object or row or column array of LTI objects) in the Data column in the System Data window.

For more information, see “Importing Models into the SISO Design Tool”.

Sample Times for Continuous/Discrete Conversions

Click **Sample Time Conversion** to convert the sample time of the system or switch between different sample times to design different compensators.


Sample Time Conversion

Discretize with

Sample time (sec): 1

Conversion Method

G: Zero-Order Hold

H: Zero-Order Hold

C1: Zero-Order Hold

C2: Zero-Order Hold

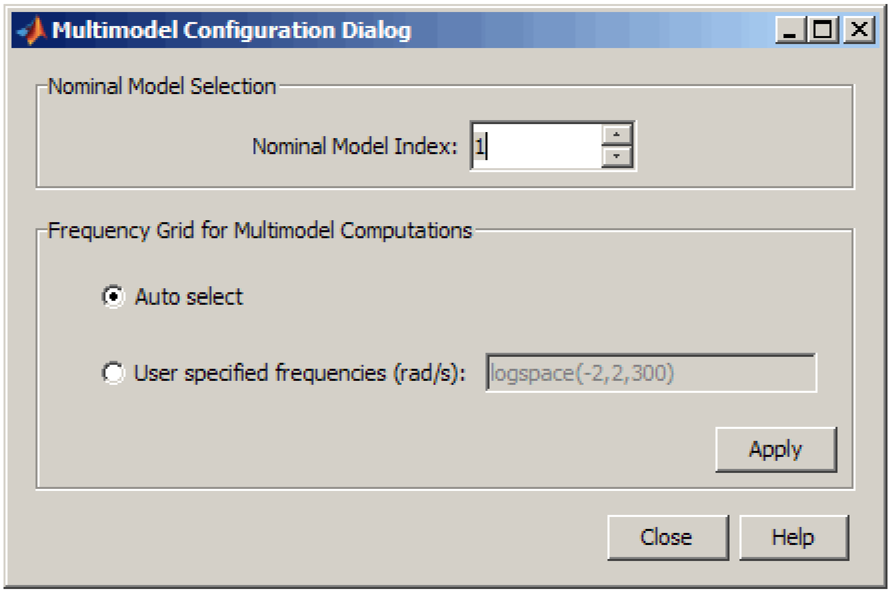
Choose from Zero-Order Hold, First-Order Hold, Impulse Variant, Tustin, Tustin w/ Prewarping, and Matched Pole-Zero.

For a full description, see “Continuous/Discrete Conversions Using the Sample Time Conversion Dialog Box” on page 17-47.

Multimodel Configuration

The **Multimodel Configuration** button is enabled only when you import or open the SISO Design Tool GUI with a row or column arrays of LTI models for the plant **G** or sensor **H** or both. The LTI arrays model system variations in the plant and sensor. If both **G** and **H** are arrays, their sizes must match.

Click **Multimodel Configuration** to specify the nominal model and frequency grid for multimodel computations. This action opens the Multimodel Configuration Dialog window, as shown in the next figure.



Nominal Model Selection

Nominal Model Index: 1

Frequency Grid for Multimodel Computations

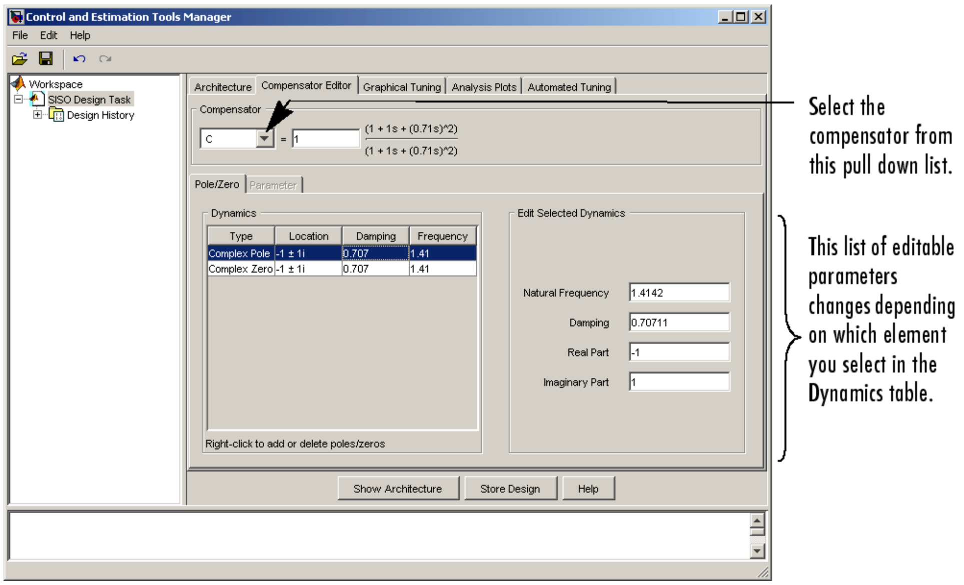
[✓ Auto select]

User specified frequencies (rad/s): logspace(-2,2,300)

For more information, see “Control Design Analysis of Multiple Models”.

Compensator Editor

Use the **Compensator Editor** for adding or editing gains, poles, and zeros.



Compensator: C = 1 × (1 + 1s + (0.71s)^2) / (1 + 1s + (0.71s)^2)

Dynamics

Type	Location	Damping	Frequency
Complex Pole	-1 ± 1i	0.707	1.41
Complex Zero	-1 ± 1i	0.707	1.41

Right-click to add or delete poles/zeros

Edit Selected Dynamics

Natural Frequency	1.4142
Damping	0.70711
Real Part	-1
Imaginary Part	1

Select the compensator from this pull down list.

This list of editable parameters changes depending on which element you select in the Dynamics table.

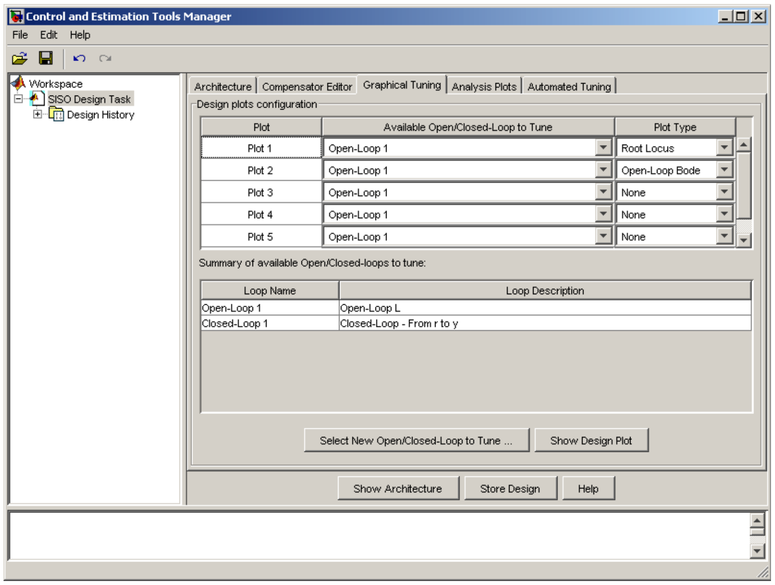
Compensator Editor Pane on the SISO Design Task Node

1. Enter the compensator gain in the text box in the top part of the pane.
2. Add or remove compensator poles and zeros by right-clicking in the **Dynamics** table.
3. Adjust pole and zero settings by entering values directly in the **Edit Selected Dynamics** group box.

Graphical Tuning

Use the Graphical Tuning pane for

- "Configuring Design Plots for the Graphical Tuning Window" on page 17-20
- "Selecting New Loops to Tune" on page 17-22
- "Refocusing on the Graphical Tuning Window" on page 17-22



Design plots configuration

Plot	Available Open/Closed-Loop to Tune	Plot Type
Plot 1	Open-Loop 1	Root Locus
Plot 2	Open-Loop 1	Open-Loop Bode
Plot 3	Open-Loop 1	None
Plot 4	Open-Loop 1	None
Plot 5	Open-Loop 1	None

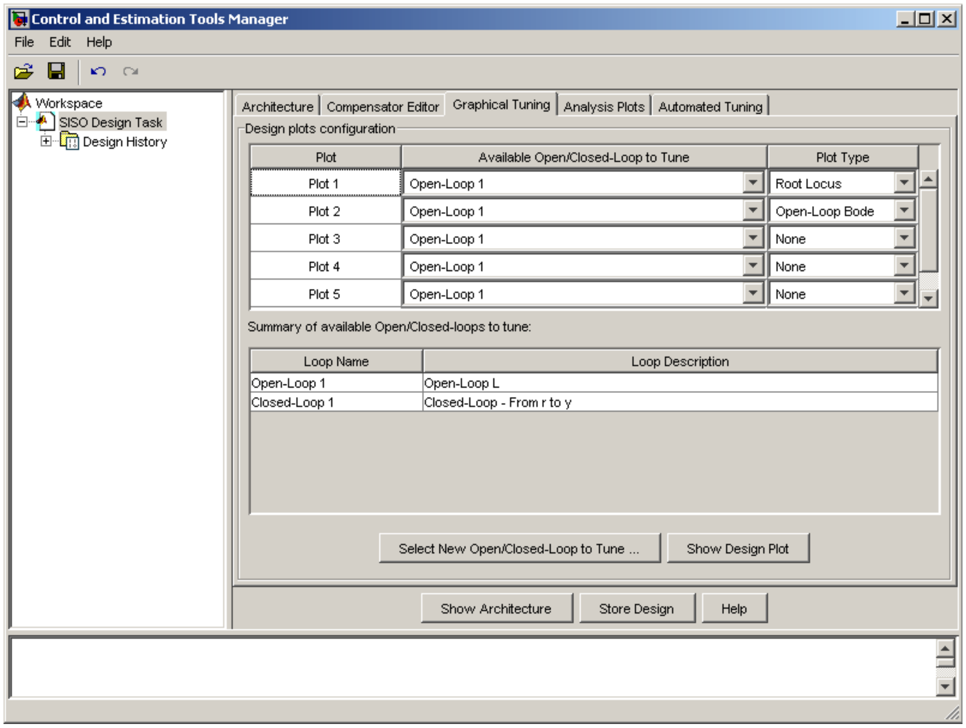
Summary of available Open/Closed-loops to tune:

Loop Name	Loop Description
Open-Loop 1	Open-Loop L
Closed-Loop 1	Closed-Loop - From r to y

Graphical Tuning Pane on the SISO Design Task Node

Configuring Design Plots for the Graphical Tuning Window

Click the **Graphical Tuning** tab to configure design plots displayed in the Graphical Tuning Window.



Plot	Available Open/Closed-Loop to Tune	Plot Type
Plot 1	Open-Loop 1	Root Locus
Plot 2	Open-Loop 1	Open-Loop Bode
Plot 3	Open-Loop 1	None
Plot 4	Open-Loop 1	None
Plot 5	Open-Loop 1	None

Summary of available Open/Closed-loops to tune:

Loop Name	Loop Description
Open-Loop 1	Open-Loop L
Closed-Loop 1	Closed-Loop - From r to y

In the Graphical Tuning window, use design plots to graphically manipulate system response. These design plots are dynamically linked to the SISO Design Task. When you change the dynamics of your compensator in either the SISO Design Task or the Graphical Tuning window, the design updates in both places.

For open-loop responses, the available plot types are:

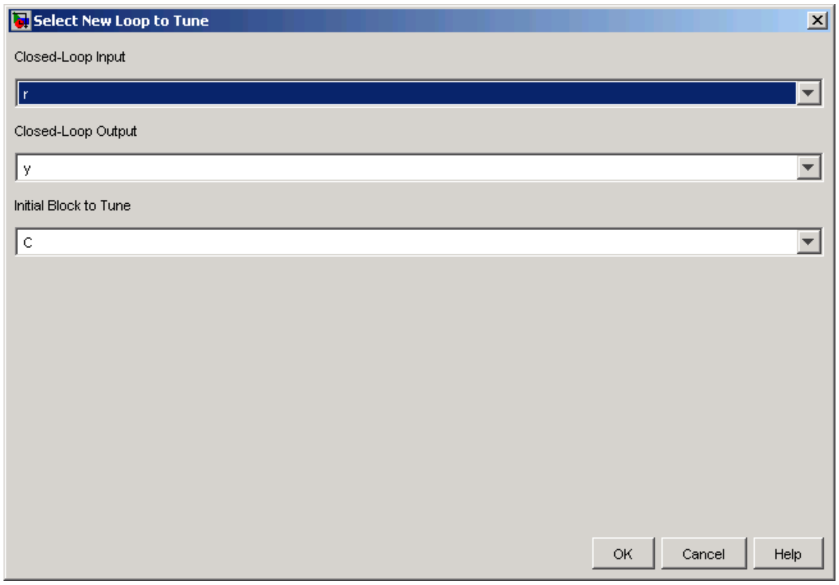
- Root locus
- Nichols
- Bode

For closed-loop responses, the available plot type is Bode.

For row or column arrays of LTI models, the design plots show the individual response of all models in the array by default. For more information, see “Using the Graphical Tuning Window” in the Getting Started Guide.

Selecting New Loops to Tune

Click **Select New Open/Closed Loops to Tune** to open a window for specifying new loops to tune.



Use the pull down menus to select the desired closed loop to tune by specifying the input, output, and blocks for tuning. Using the dialog box, you can select additional closed loops to tune.

Any loop you specify is displayed in the **Summary of Available Loops to Tune** in the Graphical Tuning pane. The list is also available in the **Design plots configuration** table of the same pane. You can use the latter for configuring design plots.

Refocusing on the Graphical Tuning Window

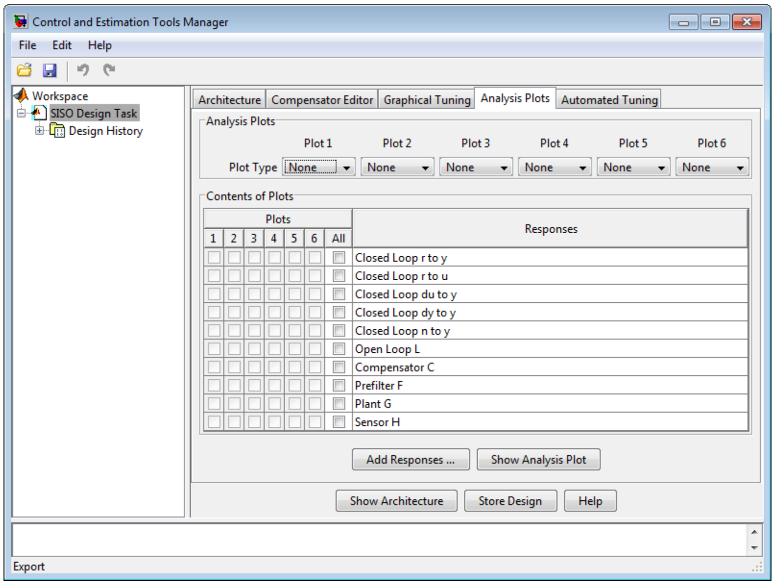
Click **Show Design Plot** to change the focus to the Graphical Tuning window.

Analysis Plots

Use the **Analysis Plots** pane for

- "Customizing Loop Responses" on page 17-23

- "Adding New Response Plots" on page 17-24
- "Opening or Changing the Focus to the Linear System Analyzer" on page 17-24



Analysis Plots

	Plot 1	Plot 2	Plot 3	Plot 4	Plot 5	Plot 6
Plot Type	None	None	None	None	None	None

Contents of Plots

1	2	3	4	5	6	All	Responses
☐	☐	☐	☐	☐	☐	☐	Closed Loop r to y
☐	☐	☐	☐	☐	☐	☐	Closed Loop r to u
☐	☐	☐	☐	☐	☐	☐	Closed Loop du to y
☐	☐	☐	☐	☐	☐	☐	Closed Loop dy to y
☐	☐	☐	☐	☐	☐	☐	Closed Loop n to y
☐	☐	☐	☐	☐	☐	☐	Open Loop L
☐	☐	☐	☐	☐	☐	☐	Compensator C
☐	☐	☐	☐	☐	☐	☐	Prefilter F
☐	☐	☐	☐	☐	☐	☐	Plant G
☐	☐	☐	☐	☐	☐	☐	Sensor H

Analysis Plots Pane on the SISO Design Task Node

Customizing Loop Responses

The following sections describe the main components of the **Analysis Plots** pane.

Analysis Plots

You can have up to six plots in one Linear System Analyzer. To add a plot, start by selecting "Plot 1" from the list of plots. Then select a new plot type from the pull down menu. You can choose any of the plots available in the Linear System Analyzer. Select "None" to remove a plot.

Contents of plots

Once you have selected a plot type, you can include several open- and closed-loop transfer function responses for display. You can plot open-loop responses for each of the components of your system, including your compensator (**C**), plant (**G**), prefilter (**F**), or sensor (**H**). In addition, various closed loop and sensitivity response plots are available.

For row or column arrays of LTI models, the analysis plots show the response of the nominal model only by default. For more information, see “Analysis Plots for Loop Responses” in the Getting Started Guide.

Adding New Response Plots

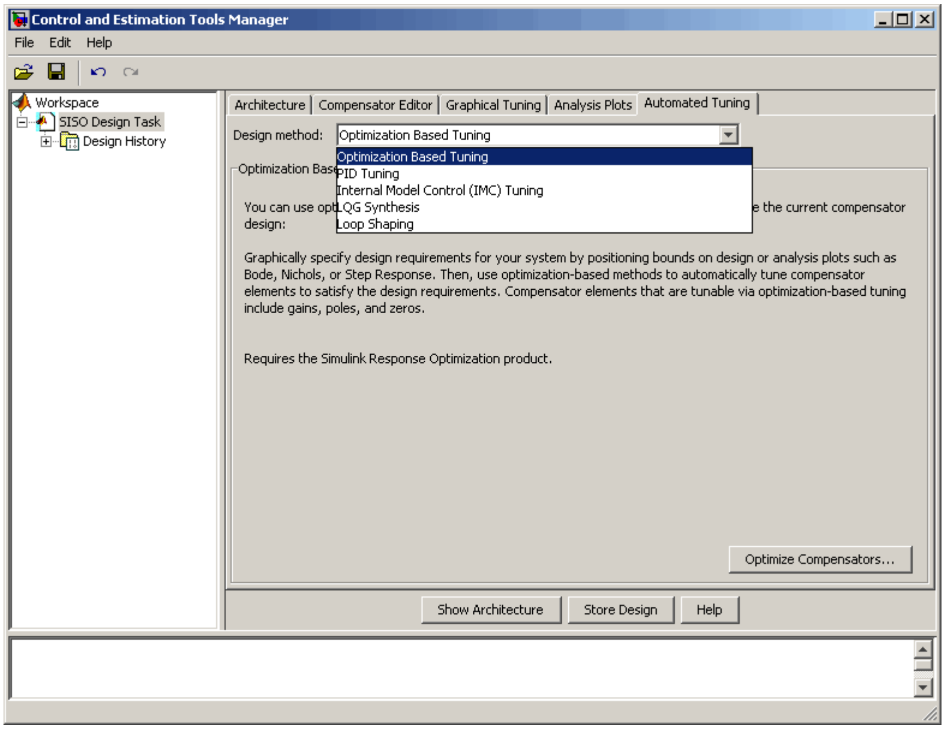
Click **Add Responses** to open a window with three drop-down menus for selecting open and closed loop responses for various input and output nodes in the control architecture block diagram. This allows you to select additional responses for viewing. The **Response** table updates automatically to include the selected response.

Opening or Changing the Focus to the Linear System Analyzer

Click **Show Analysis Plot** to open a new Linear System Analyzer for SISO Design with the response plots that you selected. All the plots open in one instance of the Linear System Analyzer.

Automated Tuning

Use the **Automated Tuning** pane to select a method for automatic tuning of your compensator design. Automated tuning methods help you design an initial compensator for a SISO loop that satisfies your design specifications.



Design method: Optimization Based Tuning

Optimization Base

You can use opt ... e the current compensator design:

Graphically specify design requirements for your system by positioning bounds on design or analysis plots such as Bode, Nichols, or Step Response. Then, use optimization-based methods to automatically tune compensator elements to satisfy the design requirements. Compensator elements that are tunable via optimization-based tuning include gains, poles, and zeros.

Requires the Simulink Response Optimization product.

You can choose among the following design methods:

- "Optimization-Based Tuning" on page 17-27 — Optimize compensator parameters using design requirements implemented in graphical tuning and analysis plots
- "PID Tuning" on page 17-28 — Tune PID controller parameters using the Robust response time tuning algorithm or classic tuning formulas
- "Internal Model Control (IMC) Tuning" on page 17-34 — Obtain a full-order stabilizing feedback controller using the IMC design method
- "LQG Synthesis" on page 17-35 — Design a full-order stabilizing feedback controller as a Linear-Quadratic-Gaussian (LQG) tracker
- "Loop Shaping" on page 17-37 — Find a full-order stabilizing feedback controller with a desired open loop bandwidth or shape

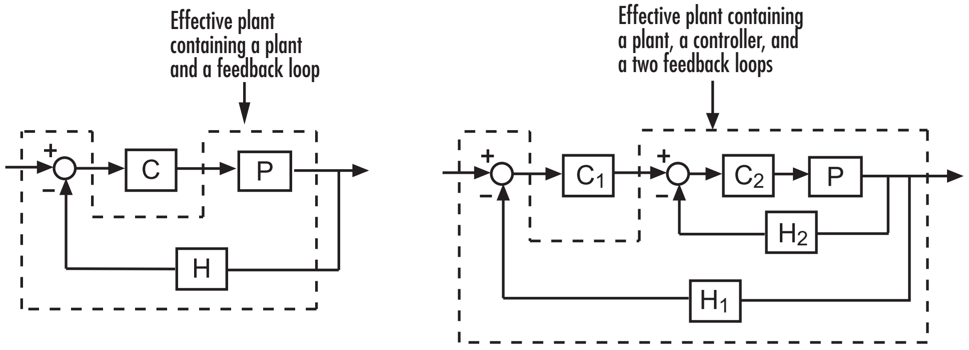
After you select a design method, the pane updates to display the corresponding options.

Note If the particular design method you are using does not apply or fails, try selecting different tuning specifications or switch to a different design method.

Stability of an Effective Plant for Automated Tuning

Knowing the stability of the effective plant in your model may help you understand which automated tuning methods work for your model. Some of the automated tuning methods only apply to compensators whose open loops ($L = C\hat{P}$) have stable effective plants ($\hat{P}$).

An *effective plant* is the system controlled by the compensator you design and contains all elements of the open loop in your model other than this compensator. The following figure shows two examples of effective plants.


Effective plant containing a plant and a feedback loop
+
−
C
P
H
Effective plant containing a plant, a controller, and a two feedback loops
+
−
C_1
+
−
C_2
P
H_2
H_1

Generic Work Flow

For each method, follow these steps to do your design:

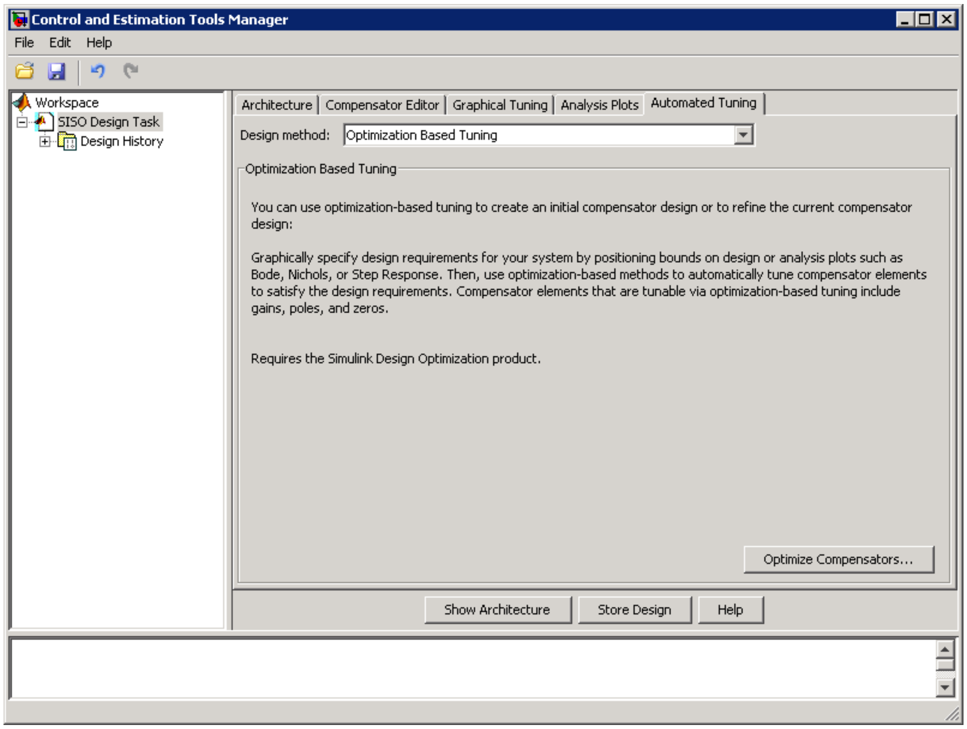
1 Select an automated tuning algorithm from the **Design method** drop-down menu.
2 If you select `Optimization-Based Tuning`, stop here and see "Optimization-Based Tuning" on page 17-27.
3 Select a compensator from the drop-down menu.
4 Determine how you want the compensator to perform and set the tuning specifications.
5 Click **Update Compensator** and notice the changes in the associated design and analysis plots.

Note: If you encounter a disabled **Update Compensator** button, try selecting different tuning specifications (Step 4) or switch to a different tuning algorithm (Step 1). The disabled button means that the current method does not work for your model.

Optimization-Based Tuning

Optimization-based tuning is available only if you have Simulink Design Optimization software installed. You can use this method to either:

- Directly tune response signals within Simulink models.
- Tune responses of LTI systems using a SISO Design Task.



Design method: Optimization Based Tuning

Optimization Based Tuning

You can use optimization-based tuning to create an initial compensator design or to refine the current compensator design:

Graphically specify design requirements for your system by positioning bounds on design or analysis plots such as Bode, Nichols, or Step Response. Then, use optimization-based methods to automatically tune compensator elements to satisfy the design requirements. Compensator elements that are tunable via optimization-based tuning include gains, poles, and zeros.

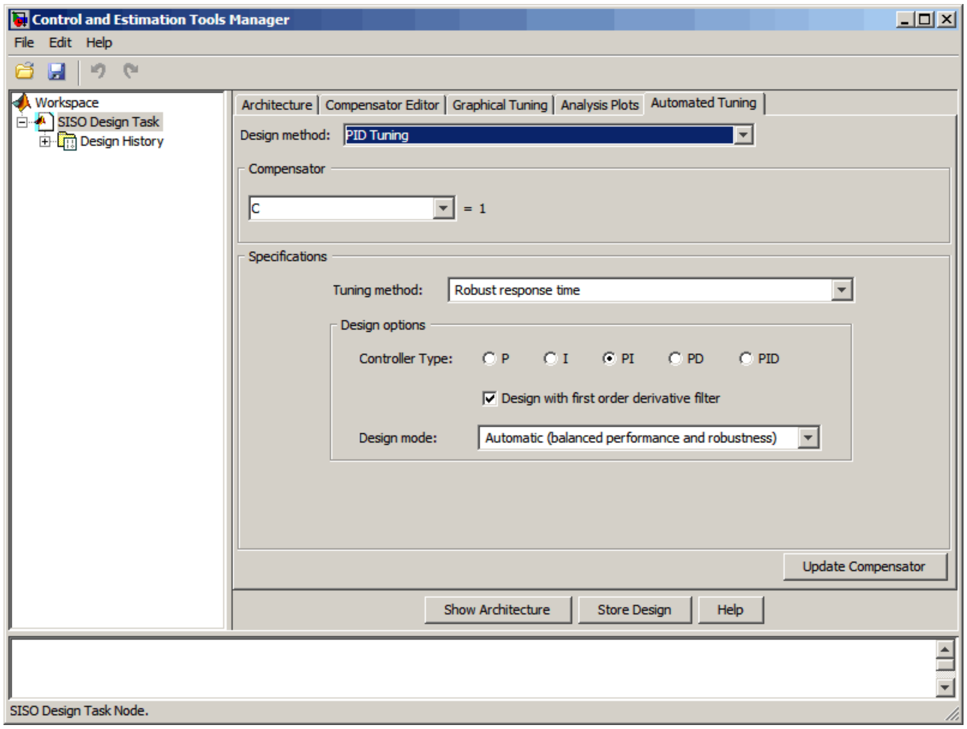
Requires the Simulink Design Optimization product.

See “Frequency Domain Response Optimization Example” in the Simulink Design Optimization documentation for more details.

PID Tuning

PID (proportional-integral-derivative) control is the most popular control technique used in modern industry.

To use automatic PID tuning, select `PID Tuning` from the Automated Tuning pane of the Control and Estimation Tools Manager. In most cases, the PID controllers resulting from PID tuning provide acceptable performance. Use the "Analysis Plots" on page 17-22 to verify design results.



Design method: PID Tuning

C = 1

Tuning method: Robust response time

Controller Type: P I [✓ PI] PD PID

[✓] Design with first order derivative filter

Design mode: Automatic (balanced performance and robustness)

Types of PID Controllers

SISO Design Tool provides automated tuning for the following PID controller types.

- P — Proportional-only control
- I — Integral-only control (available only for the `Robust response time` tuning method)

- PI — Proportional-integral control
- PD — Proportional-derivative control (available only for the `Robust response time` tuning method)
- PDF — Proportional-derivative control with a low-pass filter on the derivative term (available only for the `Robust response time` tuning method)
- PID — Proportional-integral-derivative control
- PIDF — Proportional-integral-derivative control with a low-pass filter on the derivative term

PID Tuning Methods

SISO Design Tool provides a Robust response time algorithm for interactive tuning, as well as six well-known classical tuning methods.

Robust response time. This method computes PID parameters to robustly stabilize your system based on the bandwidth and phase margin that you specify. Using the robust response time method you can:

- Tune interactively, adjusting bandwidth and phase margin to achieve your desired balance between performance and robustness
- Tune any type of PID controller (P, I, PI, PD, PDF, PID, or PIDF)
- Tune all PID parameters, including the derivative filter
- Design for plants that are stable, unstable, or integrating

Classical design formulas. SISO Design Tool includes the following well-known PID design formulas:

- `Approximate MIGO frequency response` — Closed-loop frequency-domain approximate M-constrained integral gain approximation (see [1], Section 7.5).
- `Approximate MIGO step response` — Open-loop time-domain approximate M-constrained integral gain approximation (see [1], Sections 7.3–7.4).
- `Chien-Hrones-Reswick` — Approximates the plant as a first-order model with a time delay and computes PID parameters using a Chien-Hrones-Reswick look-up table for zero overshoot and disturbance rejection (see [1], Section 6.2).
- `Skogestad IMC` — Approximates the plant as a first-order model with a time delay and computes PID parameters using Skogestad design rules (see [2]).

  (This method is different from selecting “Internal Model Control (IMC) Tuning” on page 17-34 as the full-order compensator tuning method).

- `Ziegler-Nichols frequency response` — Computes controller parameters from a Ziegler-Nichols lookup table, based on the ultimate gain and frequency of the system (see [1], Section 6.2).
- `Ziegler-Nichols step response` — Approximates the plant as a first-order model with a time delay that and computes PID parameters using the Ziegler-Nichols design method (see [1], Section 6.2).

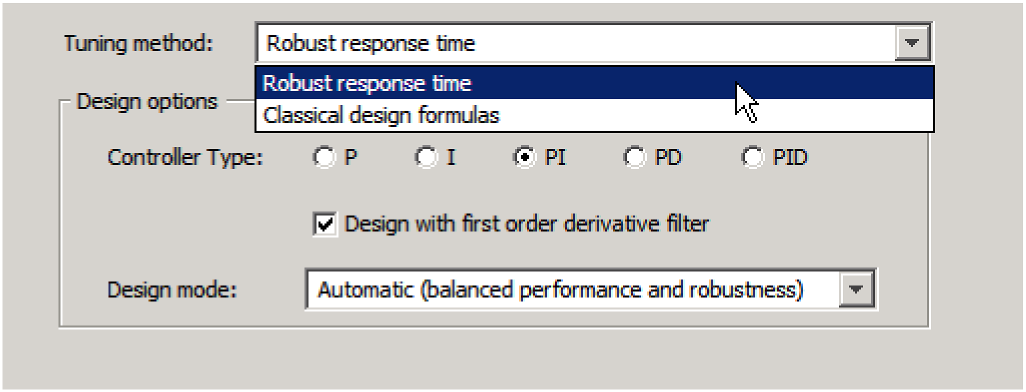
The classical design formulas:

- Require a stable or integrating plant.
- Can design for P, PI, PID, or PID with derivative filter.
- Cannot tune the derivative filter. If you select PID with derivative filter, classical design formulas set the filter time constant to *Td*/10, where *Td* is the tuned derivative time.

Automated Tuning using the Robust Response Time Method

To use the robust response time tuning method:

1 Select `Robust response time` from the **Tuning method** menu.


Tuning method: Robust response time

Design options

Controller Type: P I [✓ PI] PD PID

[✓] Design with first order derivative filter

Design mode: Automatic (balanced performance and robustness)

2 Choose a controller type by clicking the corresponding radio button. To include a first-order filter on the derivative action for PD or PID controller type, check the **Design with first order derivative filter** checkbox.

Note: If you are tuning a `PID Controller` block in a Simulink model and your block is type P, I, or PI, select the same controller type as the block. If your block is type PD or PID, select the corresponding button and check the **Design with first order derivative filter** checkbox.

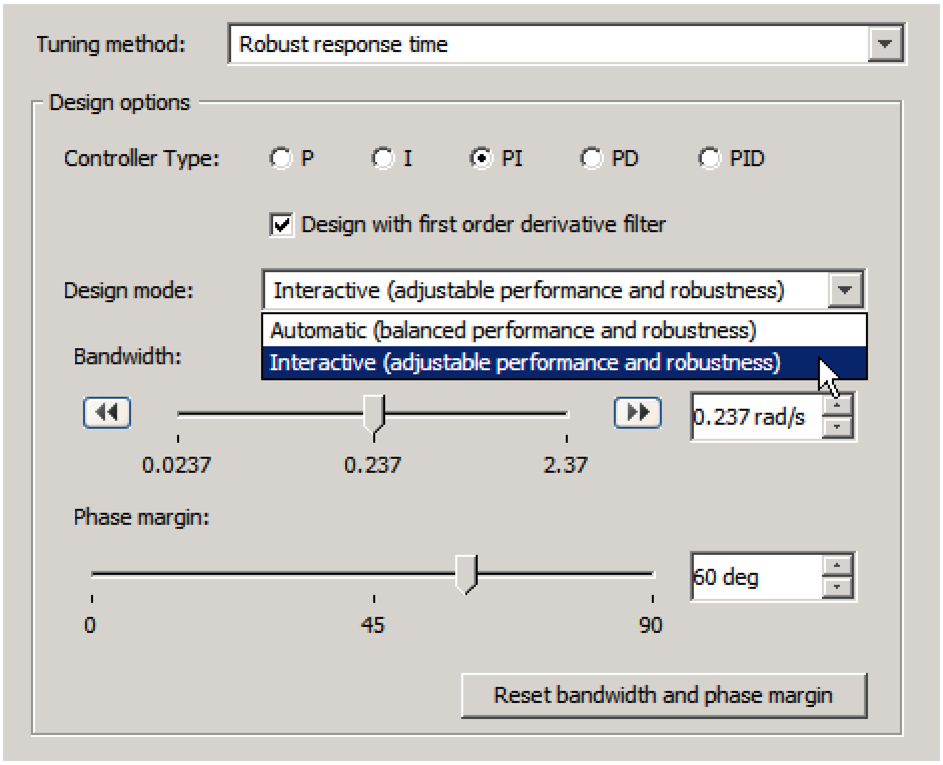
---

3 Click **Update Compensator** to design a controller of the selected type.

By default, the SISO Design Tool automatically computes controller parameters for balanced performance and robustness.

4 Analyze the response using the analysis plots you select on the Analysis Plots pane.

5 To design a controller interactively, select `Interactive (adjustable performance and robustness)` from the **Design mode** menu. This selection activates the **Bandwidth** and **Phase Margin** sliders.


Tuning method:
Robust response time
Design options
Controller Type:
P
I
PI
PD
PID
Design with first order derivative filter
Design mode:
Interactive (adjustable performance and robustness)
Automatic (balanced performance and robustness)
Interactive (adjustable performance and robustness)
Bandwidth:
0.237 rad/s
0.0237
0.237
2.37
Phase margin:
60 deg
0
45
90
Reset bandwidth and phase margin

Adjust the bandwidth and phase margin to achieve your desired controller performance. For example:

- Increase the bandwidth for a more aggressive controller.

- Increase the phase margin to reduce overshoot.

You can adjust the bandwidth and phase margin values by:

- Moving the sliders
- Entering values in the text field
- Incrementally adjusting the values in the text field using the up and down arrows.

To increase or decrease the bandwidth by a factor of 10, click the right or left double arrows, respectively.

Note: Click the **Reset bandwidth and phase margin** button at any time to return the sliders to the bandwidth and phase margin of the default compensator design for your plant.

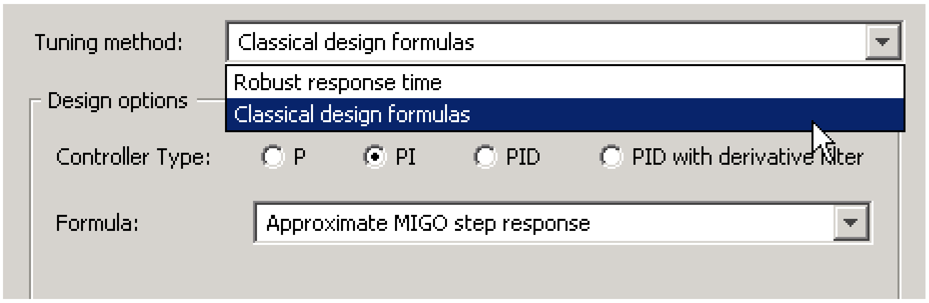
6 Click **Update Compensator** again. SISO Design Tool computes new controller parameters for the specified target bandwidth and phase margin. Repeat steps 4-6 as necessary to achieve your desired performance.

Note: SISO Design Tool displays the tuned compensator in `zpk` form in the **Compensator** section of the Automated Tuning Pane. To convert the compensator to parallel or standard PID form, export the compensator to the MATLAB workspace, as described in "SISO Design Task Node Menu Bar" on page 17-4. Then use the `pid` or `pidstd` commands to convert the exported compensator to parallel or standard PID parameters, respectively.

Automated Tuning using Classical Design Formulas

To use one of the classical design formulas:

1 Select `Classical design formulas` from the **Tuning method** menu.

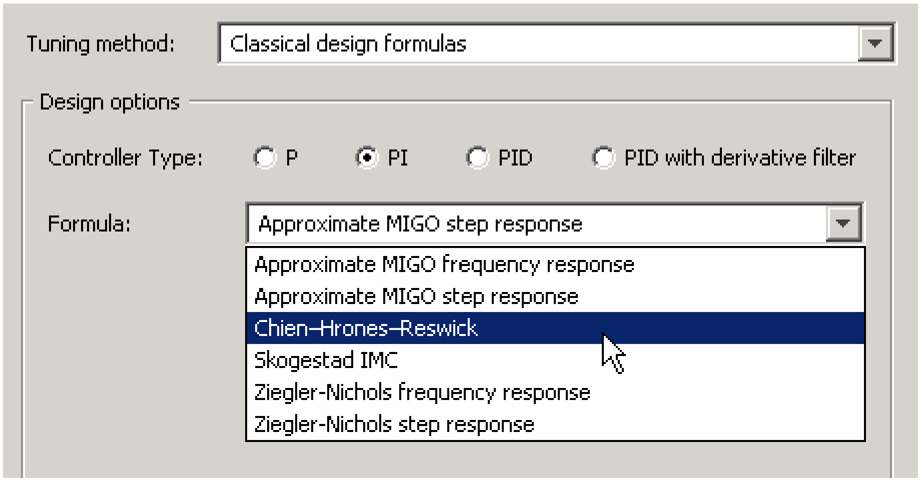

Tuning method:
Classical design formulas
Robust response time
Classical design formulas
Design options
Controller Type:
P
PI
PID
PID with derivative filter
Formula:
Approximate MIGO step response

2 Select a controller type (P, PI, PID, or PID with derivative filter) by clicking the corresponding radio button.

Note: If you are tuning a `PID Controller` block in a Simulink model and your block is type P or PI, select the same controller type as the block. If your block is type PID, select **PID with derivative filter**. If your block is type PD or I, use the Robust response time tuning method.

---

3 Select the classical design formula you want to use from the **Formula** menu.


Tuning method:
Classical design formulas
Design options
Controller Type:
P
PI
PID
PID with derivative filter
Formula:
Approximate MIGO step response
Approximate MIGO frequency response
Approximate MIGO step response
Chien–Hrones–Reswick
Skogestad IMC
Ziegler-Nichols frequency response
Ziegler-Nichols step response

4 Click the **Update Compensator** button to design a controller using the selected formula.

Note: SISO Design Tool displays the tuned compensator in `zpk` form in the **Compensator** section of the Automated Tuning Pane. To convert the compensator to parallel or standard PID form, export the compensator to the MATLAB workspace, as described in "SISO Design Task Node Menu Bar" on page 17-4. Then use the `pid` or `pidstd` commands to convert the exported compensator to parallel or standard PID parameters, respectively.

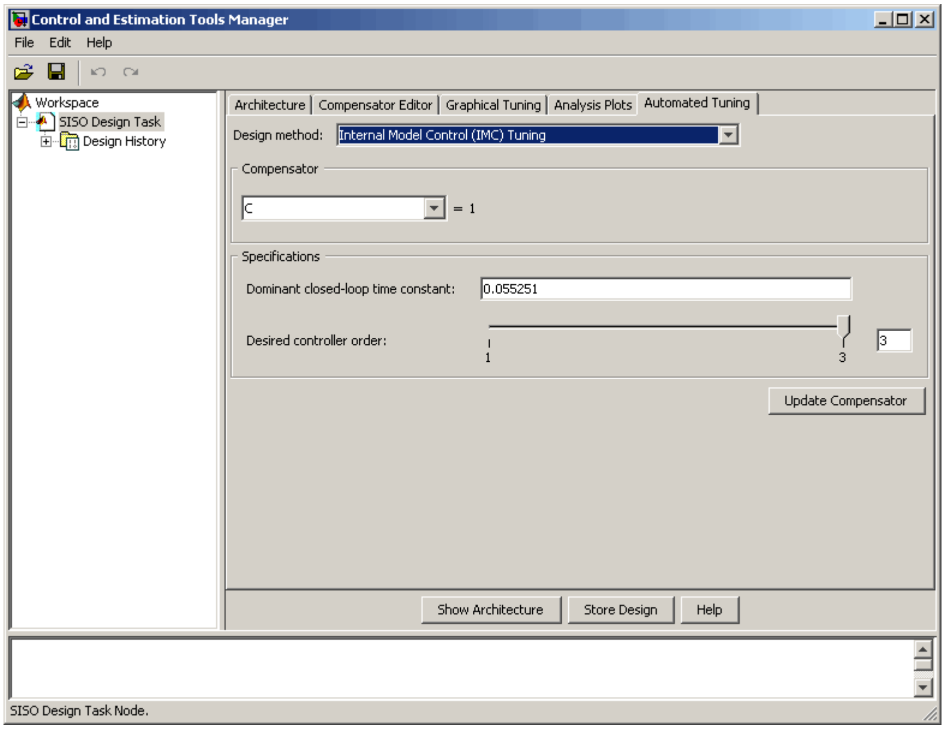
References

[1] Åström, K. J. and Hägglund, T. *Advanced PID Control*, Research Triangle Park, NC: Instrumentation, Systems, and Automation Society, 2006.

[2] Skogestad, S., "Simple analytic rules for model reduction and PID controller tuning." *Journal of Process Control*, Vol. 13, No. 4, 2003, pp. 291–309.

Internal Model Control (IMC) Tuning

IMC design generates a full-order feedback controller that guarantees closed-loop stability when there is no model error. It 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.



Design method: Internal Model Control (IMC) Tuning

Compensator

C = 1

Specifications

Dominant closed-loop time constant: 0.055251

Desired controller order: 3

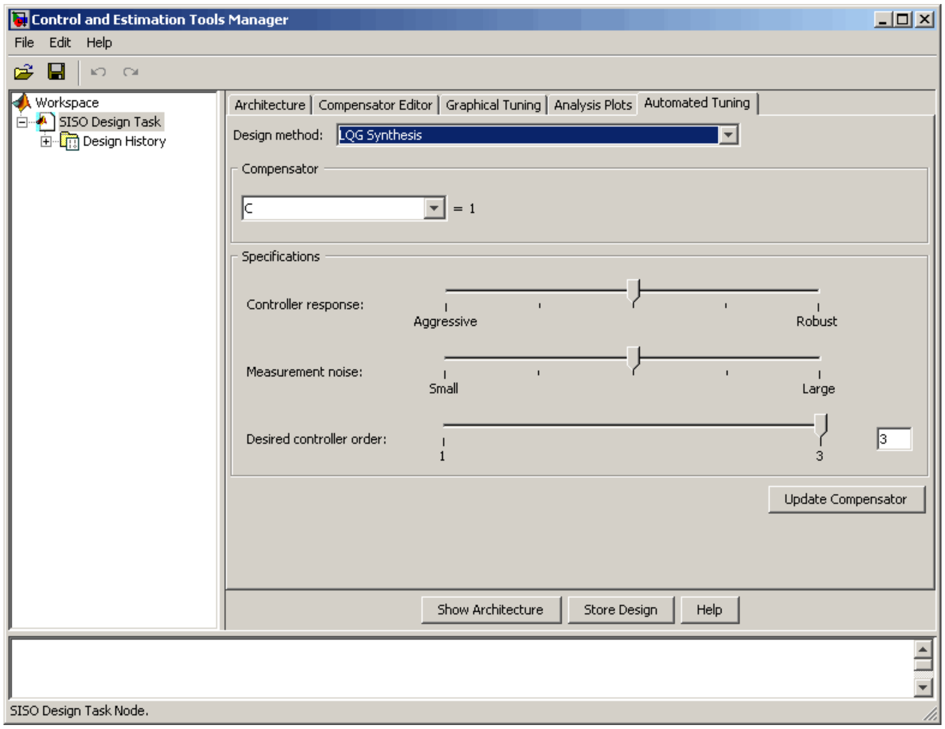
SISO Design Task Node.

To design an IMC controller:

1. Specify a value in the **Dominant closed-loop time constant** field. The initial value is set as 5% of the open-loop settling time. In general, increasing this value slows down the closed system and makes it more robust.
2. Specify a value in the **Desired controller order** field using the slider. After you obtain a full-order feedback controller, you can try to reduce its order. You may lose performance and closed-loop stability if you reduce the order.
3. Click **Update Compensator**.

LQG Synthesis

LQG tracker design generates a full-order feedback controller that guarantees closed-loop stability. It also contains an integrator, which guarantees zero steady-state error for plants without a free differentiator.



Design method: LQG Synthesis

Compensator

C = 1

Specifications

Controller response: Aggressive — Robust

Measurement noise: Small — Large

Desired controller order: 1 — 3 [3]

SISO Design Task Node.

To design an LQG controller:

1 Specify your preference for controller response using the **Controller response** slider.

- Move the slider to the left for aggressive control response.

  This means that large overshoot is more heavily penalized so that the controller acts more aggressively. If you believe your model is accurate and that the manipulated variable has a large enough range, an aggressive controller is more desirable.

- Move the slider to the right for robust control response.

2 Specify your estimation of the level of measurement noise using the **Measurement noise** slider.

- Move the slider to the left for small measurement noise.

This means that you expect low noise from the process output measurement. Because this measurement is used by the Kalman estimator, process disturbances are picked up more accurately by the estimated states. In this case, the controller is freer from robustness considerations.

- Move the slider to the right for large measurement noise. This results in a controller that is more robust to measurement noise.

3 Specify your preference for controller order using the **Desired controller order** slider.

4 Click **Update Compensator**.

Loop Shaping

Loop shaping generates a stabilizing feedback controller to match as closely as possible to a desired loop shape. You can specify this loop shape as a bandwidth or an open loop frequency response. If you have Robust Control Toolbox software installed, you can use loop shaping for SISO systems. For more information see the section on H-Infinity Loop Shaping in the *Robust Control Toolbox User's Guide*.

To design a controller using loop shaping:

1 Select a tuning preference by clicking one of these option buttons:

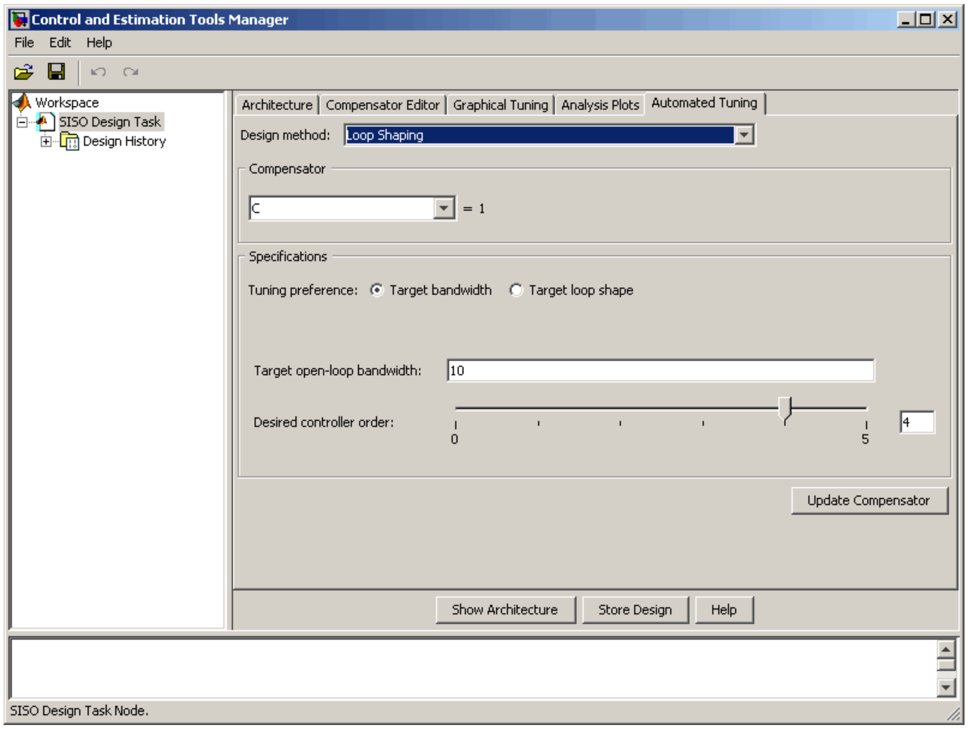
- **Target bandwidth** — Allows you to specify a target loop shape bandwidth (ω_b).

  This results in a loop shape of your specified bandwidth over an integrator ($\frac{\omega_b}{s}$).

- **Target loop shape** — Allows you to specify the target open loop shape in one of the following representations: state-space, zero-pole-gain, or transfer functions.

2 Set the tuning options available for your selected tuning preference as follows:

- If you chose **Target bandwidth**, specify the desired **Target open-loop bandwidth** in the editable box.



Design method: Loop Shaping

Compensator

C = 1

Specifications

Tuning preference: [✓ Target bandwidth] Target loop shape

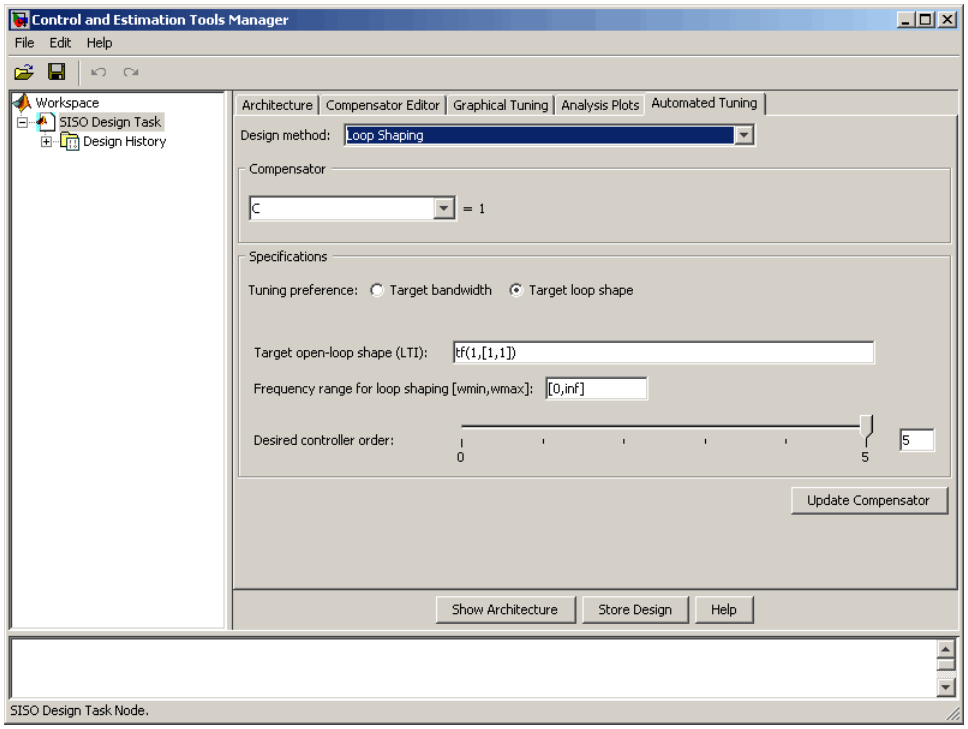
Target open-loop bandwidth: 10

Desired controller order: 4

SISO Design Task Node.

- If you chose **Target loop shape**, do the following:
  - Enter the desired **Target open-loop shape (LTI)**.

    This can be a state-space representation, a zero-pole-gain representation, or a transfer function.
  - Enter the desired **Frequency range for loop shaping [wmin,wmax]**.



Design method: Loop Shaping

Compensator

C = 1

Specifications

Tuning preference: ○ Target bandwidth ● Target loop shape

Target open-loop shape (LTI): tf(1,[1,1])

Frequency range for loop shaping [wmin,wmax]: [0,inf]

Desired controller order: 0 5 — 5

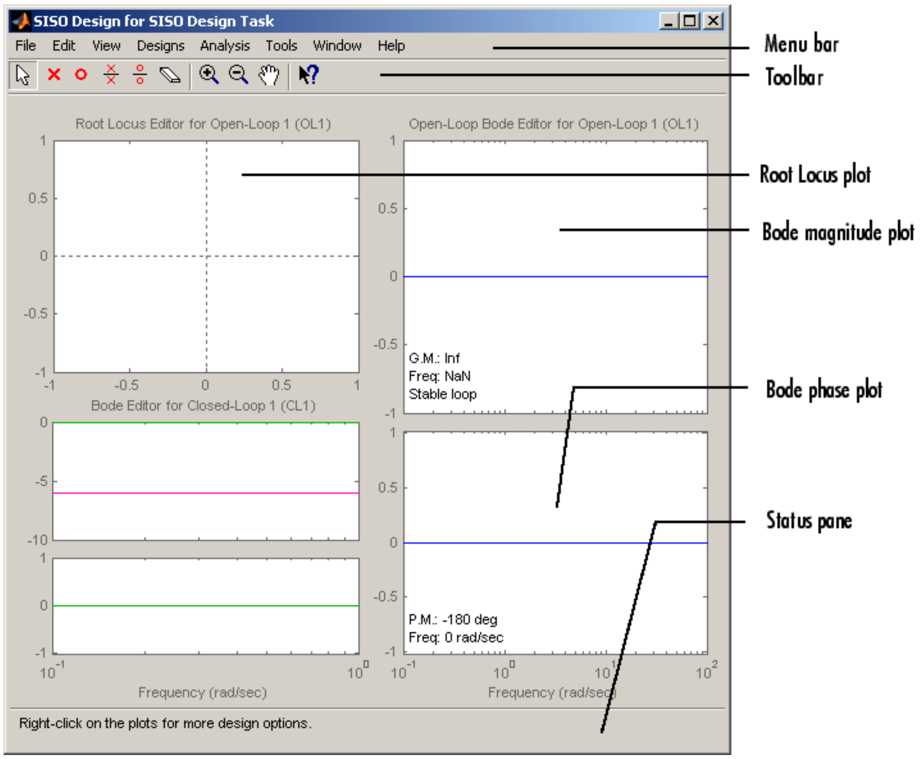
SISO Design Task Node.

3 Specify your preference for controller order using the **Desired controller order** slider.

4 Click **Update Compensator**.

SISO Design Task Graphical Tuning Window

The following figure shows the Graphical Tuning window and introduces some terminology.


SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Root Locus Editor for Open-Loop 1 (OL1)
Open-Loop Bode Editor for Open-Loop 1 (OL1)
Bode Editor for Closed-Loop 1 (CL1)
G.M.: Inf
Freq: NaN
Stable loop
P.M.: -180 deg
Freq: 0 rad/sec
Frequency (rad/sec)
Right-click on the plots for more design options.
Menu bar
Toolbar
Root Locus plot
Bode magnitude plot
Bode phase plot
Status pane

Graphical Tuning Window

Bode and Nichols plots in the graphical tuning window automatically display the following information:

- Gain margin and the -180 degree phase crossing frequency where it is measured
- Phase margin and the 0 dB gain crossing frequency where it is measured
- Whether the characteristic equation 1+L is stable (**Stable loop**) or unstable (**Unstable loop**). L is the open loop plotted in the figure.

For row or column arrays of LTI models, the plots display the characteristics of the nominal model only.

The following topics describe the Graphical Tuning window features:

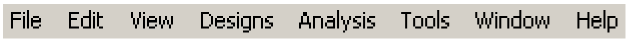
- "Using the Graphical Tuning Window Menu Bar" on page 17-42
- "Using the Graphical Tuning Window Toolbar" on page 17-53
- "Using the Right-Click Menus in the Graphical Tuning Window" on page 17-54

Using the Graphical Tuning Window Menu Bar

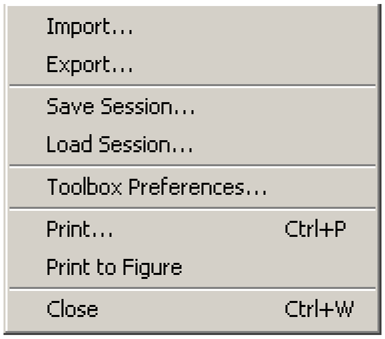
In this section...

Overview of the Graphical Tuning Window Menu Bar

Several of the tasks you can do in the SISO Design Tool can be done from the menu bar, shown below.



File



Using the **File** menu, you can:

- Import and export models

- Save and reload sessions
- Set toolbox preferences
- Print and print to figure
- Close the Graphical Tuning Window

The following sections describe the **File** menu options in turn.

Import

Selecting **Import** opens the same System Data dialog box that clicking **System Data** on the **Architecture** pane does. See "Model Import" on page 17-14.

Export

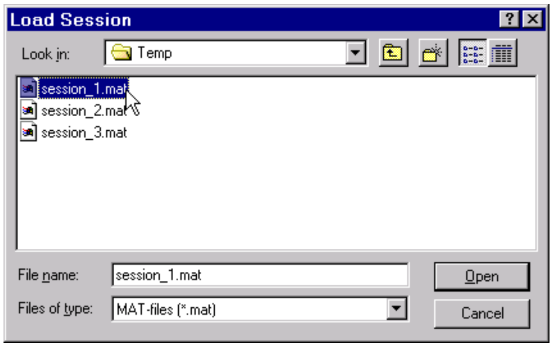
Selecting **Export** from the Graphical Tuning window **File** menu opens the same SISO Tool Export window that selecting **Export** from the SISO Design Task node **File** menu does. See Export.

Save Session

Selecting **Save Session** from the Graphical Tuning window **File** menu opens the same Save Projects window that selecting **Save** from the SISO Design Task node **File** menu does. See Save.

Load Session

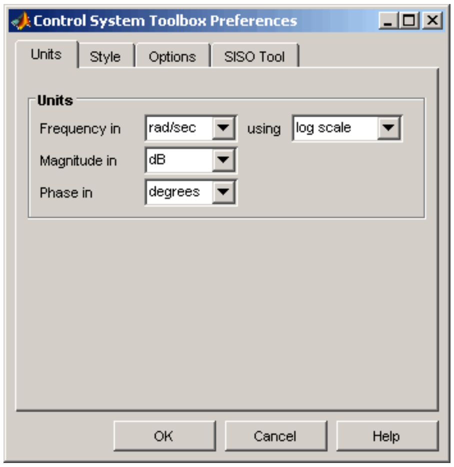
To load a saved SISO Design Tool session, select **Load Session** from the **File** menu. This opens the **Load Session** dialog.



Sessions are saved as MAT-files. Select the session you want to load from the list, and click **Open**. See “Save Session” on page 17-43 for information on saving **SISO Design Tool** sessions.

Toolbox Preferences

Select **Toolbox Preferences** from the **File** menu to open the **Control System Toolbox Preferences** window.



Units

Frequency in rad/sec using log scale

Magnitude in dB

Phase in degrees

The Control System Toolbox Preferences Window

For a discussion of this window's features, see "??? online in the Control System Toolbox documentation.

Print

Use **Print** to send a picture of the Graphical Tuning window to your printer.

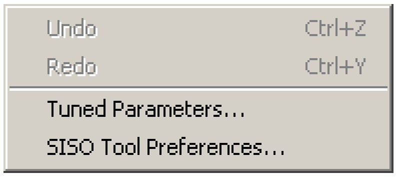
Print to Figure

Print to Figure opens a separate figure window containing the design views in your current Graphical Tuning window.

Close

Use **Close** to close the Graphical Tuning window.

Edit



Undo and Redo

Selecting **Undo** and **Redo perform the same actions as selecting Undo and Redo from the SISO Design Task Node Edit menu. See "Edit Menu Options" on page 17-8.**

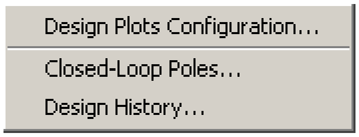
Tuned Parameters

Selecting **Tuned Parameters** opens the SISO Tool Preferences dialog box on the **Options** page.

SISO Tool Preferences

Selecting the SISO Tool Preferences option opens the same dialog box that selecting SISO Tool Preferences from the Edit menu on the SISO Design Task Node. See "Edit Menu Options" on page 17-8.

View

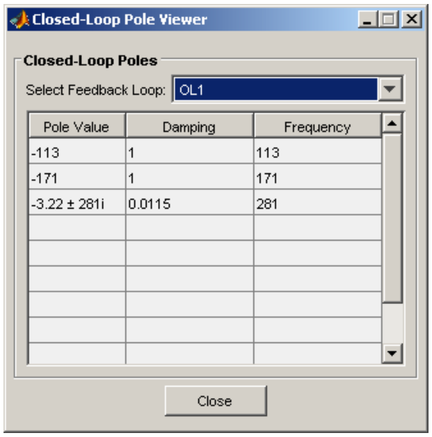


Design Plots Configuration

Select **Design Plots Configuration** to open the Graphical Tuning pane. See "Graphical Tuning" on page 17-19.

Closed-Loop Poles

Select **Closed-Loop Poles** from **View** to open the **Closed-Loop Pole Viewer**.



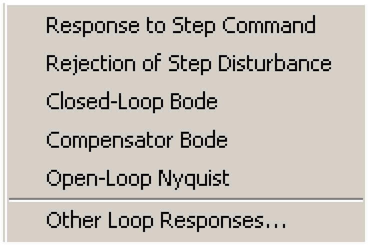
Pole Value	Damping	Frequency
-113	1	113
-171	1	171
-3.22 ± 281i	0.0115	281

This window displays all the closed-loop pole values of the selected feedback loop and the associated damping and frequency values.

Design History

Select **Design History** from the **View** menu to open the **Design History** window, which displays all the actions you've performed during a design session. You can save the history to an ASCII flat text file by clicking **Save to Text File**.

Analysis



Common Response Plots

Each of the top group of items in the Analysis menu opens a Linear System Analyzer that is dynamically linked to your SISO Design Tool. You have the following response plot choices:

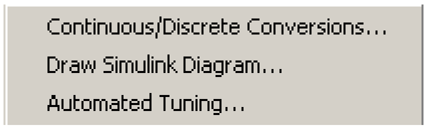
- **Response to Step Command** — The closed-loop step response of your system
- **Rejection of Step Disturbance** — The open-loop step response of your system
- **Closed-Loop Bode** — The closed-loop Bode diagram for your system
- **Compensator Bode** — The open-loop Bode diagram for your compensator
- **Open-Loop Nyquist** — The open-loop Nyquist plot for your system

When you make changes to the design via the Graphical Tuning window, the Compensator Editor pane, or the Automated Tuning pane, the response plots in the Linear System Analyzer automatically change to reflect the new design's responses.

Other Loop Responses

If you choose **Other Loop Responses**, the **Analysis Plots** pane opens. See “Analysis Plots” on page 17-22.

Tools



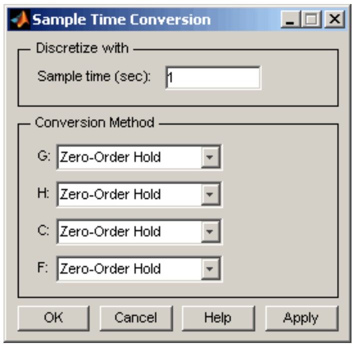
Continuous/Discrete Conversions Using the Sample Time Conversion Dialog Box

- “Converting Continuous-Time Models to Discrete-Time Models” on page 17-47
- “Converting Discrete-Time Models to Continuous-Time” on page 17-48
- “Changing the Sample Time of a Discrete-Time Model” on page 17-49

Converting Continuous-Time Models to Discrete-Time Models

To convert a continuous-time model to a discrete-time model, perform these steps:

1 In the SISO Design Tool, select **Tools** > **Continuous/Discrete Conversions** to open the **Sample Time Conversion** window.



2 Specify a positive number for the sample time in the **Sample time (sec)** field.

3 Select a continuous-to-discrete conversion method for each component of your model. The components include the plant (**G**), the compensator (**C**), the prefilter (**F**), or the sensor (**H**). You can choose from the following conversion methods:

- Zero-order hold
- First-order hold
- Tustin
- Tustin with prewarping

Note: If you choose Tustin with prewarping, you must specify the critical frequency in radians per second.

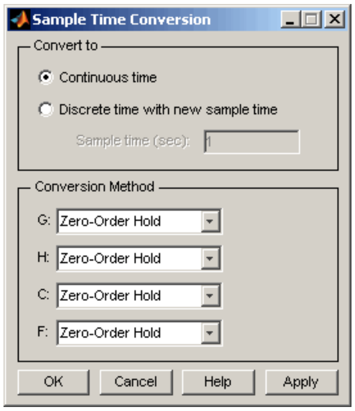
- Matched pole/zero

For more information on each of these conversion methods, see "Continuous-Discrete Conversion Methods" on page 5-23 in the Control System Toolbox documentation.

Converting Discrete-Time Models to Continuous-Time

To convert a discrete-time model to a continuous-time model, perform these steps:

1 In the SISO Design Tool, select **Tools** > **Continuous/Discrete Conversions** to open the **Sample Time Conversion** window.



2. Select a discrete-to-continuous conversion method for each component of your model. The components include the plant (**G**), the compensator (**C**), the prefilter (**F**), or the sensor (**H**). You can choose from the following conversion methods:

   - Zero-order hold
   - First-order hold
   - Tustin
   - Tustin with prewarping

     Note: If you choose Tustin with prewarping, you must specify the critical frequency in radians per second.

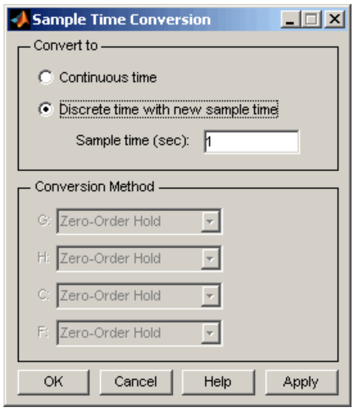
   - Matched pole/zero

   For more information on each of these conversion methods, see "Continuous-Discrete Conversion Methods" on page 5-23 in the Control System Toolbox documentation.

Changing the Sample Time of a Discrete-Time Model

To change the sample time of (resample) a discrete system, perform these steps:

1. In the SISO Design Tool, select **Tools** > **Continuous/Discrete Conversions** to open the **Sample Time Conversion** window.
2. Click the **Discrete time with new sample time** option button.



Sample time (sec): 1

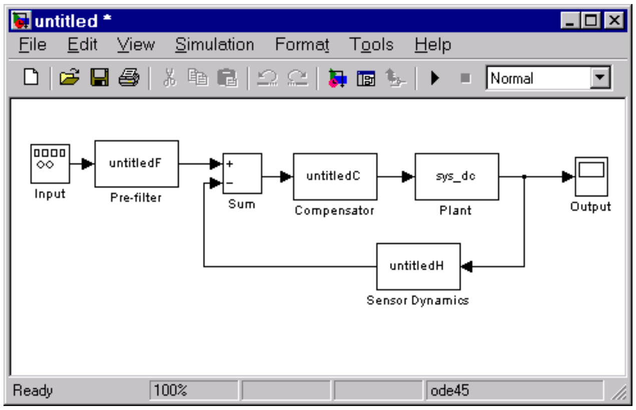
G: Zero-Order Hold
H: Zero-Order Hold
C: Zero-Order Hold
F: Zero-Order Hold

3 Specify a positive number for the sample time in the **Sample time (sec)** field.

Draw Simulink Diagram

Note You must have a license for Simulink to use this feature. If you do not have Simulink, this option does not appear under the **Tools** menu.

Select **Draw Simulink Diagram** from the Tools menu to draw a block diagram of your system (plant, compensator, prefilter, and sensor). The following diagram shows how the tool would render the DC motor example described in the *Control System Toolbox Getting Started Guide*.



Automated Tuning

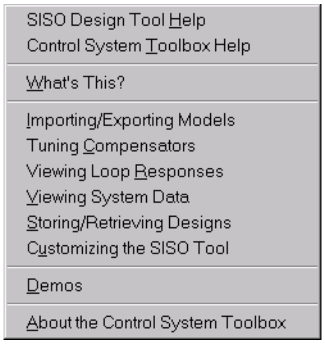
Select **Automated Tuning** from the Tools menu in the SISO Design Tool to open the **Automated Tuning** tab of the Control and Estimation Tools Manager. You can use this tab to perform automated tuning of compensators. For more information, see “Automated Tuning Design”.

Window

The **Window** menu item lists all of the windows open in the MATLAB technical computing environment. The first item is always the MATLAB Command Window. After that, the windows you have opened are listed in the order in which you opened them. Select any window from the list to make it the active window.

Help

Help brings you to various places in the Control System Toolbox help system. This figure shows the menu.


SISO Design Tool Help
Control System Toolbox Help
What's This?
Importing/Exporting Models
Tuning Compensators
Viewing Loop Responses
Viewing System Data
Storing/Retrieving Designs
Customizing the SISO Tool
Demos
About the Control System Toolbox

Each topics takes you to brief discussions of basic information about the SISO Design Tool and the Control System Toolbox software:

- **SISO Design Tool Help** — An overview of the SISO Design Tool
- **Control System Toolbox Help** — A roadmap for the Control System Toolbox help
- **What's This?** — Activates the "What's This?" cursor, which appears as a question mark. Click in various regions of the SISO Design Tool to see brief descriptions of the tool's features.

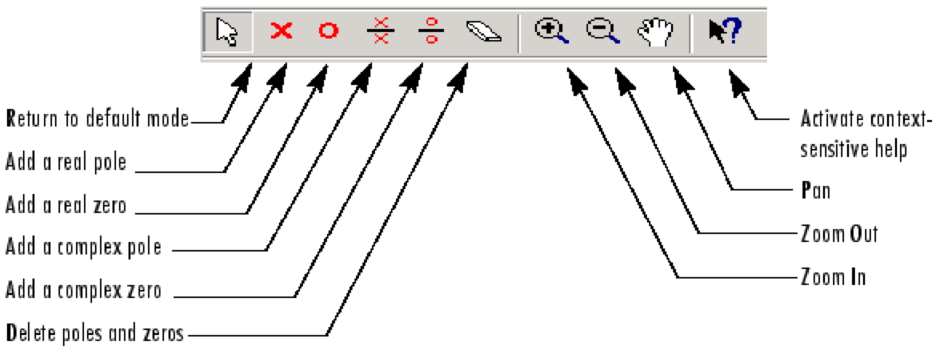
- **Importing/Exporting Models** — How to import models into the SISO Design Tool and how to export completed designs
- **Tuning Compensators** — Basic information about adjusting gains and adding dynamics to your prefilter (**F**) and compensator (**C**)
- **Viewing Loop Responses** — How to open a Linear System Analyzer containing loop responses for your system. Many response types are available.
- **Viewing System Data** — How to see information about your model
- **Storing/Retrieving Designs** — How to store and retrieve designed systems
- **Customizing the SISO Tool** — How to open the SISO Tool Preferences editor, which allows you to customize plot displays in the tool
- **Examples** — A link to Control System Toolbox featured examples
- **About the Control System Toolbox software** — The version number of your Control System Toolbox software

Using the Graphical Tuning Window Toolbar

The toolbar performs the following operations:

- Add and delete real and complex poles and zeros
- Zoom in and out
- Invoke the SISO Design Tool's context-sensitive help

This picture shows the toolbar.



Options Available from the Toolbar

You can use the tool tips feature to find out what a particular icon does. Just place your mouse over the icon in question, and you will see a brief description of what it does.

Once you've selected an icon, your mouse stays in that mode until you press the icon again.

You can reach all of these options from the right-click menus.

Using the Right-Click Menus in the Graphical Tuning Window

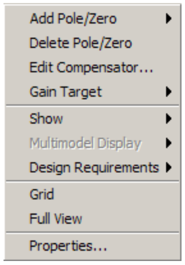
In this section...

Overview of the Right-Click Menus

The Graphical Tuning window provides right-click menus for all the views available. These views include the root-locus, open-loop Bode diagrams, Nichols plot, and the closed-loop Bode diagrams. The menu items in each of these views are identical. The design requirements, however, differ, depending on which view you are accessing the menus from.

You can use the right-click menu to design a compensator by adding poles, zeros, lead, lag, and notch filters. In addition, you can use this menu to add grids and zoom in on selected regions. Also, you can open each view's **Property Editor** to customize units and other elements of the display.

Note: Click items on the menu bar pictured below to get help contents.



Open-Loop Right-Click Menu

Note that if you have a closed-loop response, the Gain Target menu item is replaced by “Select Compensator” on page 17-72.

Add Pole/Zero

The **Add Pole/Zero** menu options give you the ability to add dynamics to your compensator design, including poles, zeros, lead and lag networks, and notch filters. The following pole/zero configurations are available:

- **Real Pole**
- **Complex Pole**
- **Integrator**
- **Real Zero**
- **Complex Zero**
- **Differentiator**
- **Lead**
- **Lag**
- **Notch**

In all but the integrator and differentiator, once you select the configuration, your cursor changes to an `x'. To add the item to your compensator design, place the x at the desired location on the plot and left-click your mouse. You will see the root locus design automatically update to include the new compensator dynamics.

The notch filter has three adjustable parameters. For a discussion about how to add and adjust notch filters, see "Adding a Notch Filter" in the *Control System Toolbox Getting Started Guide.*

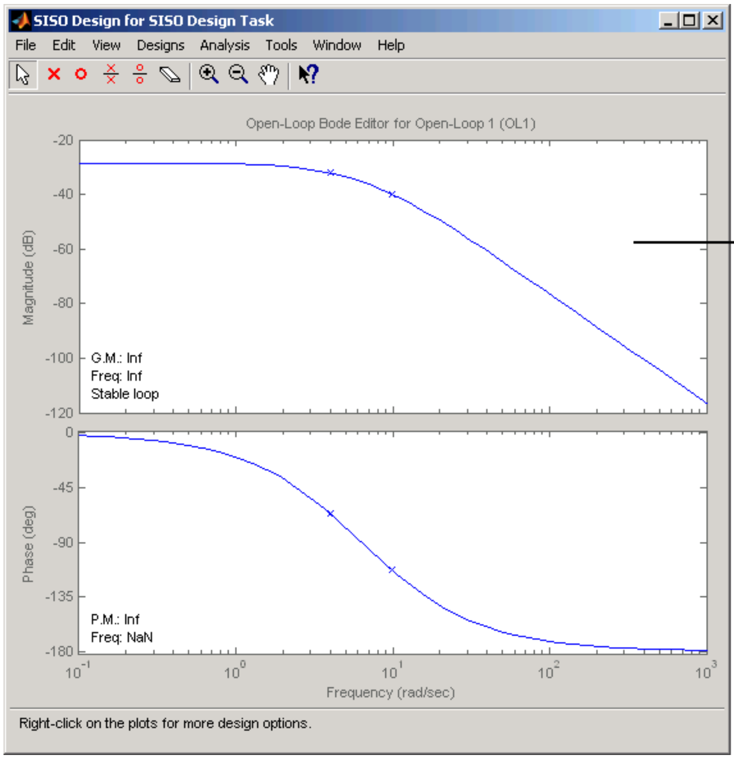
Note: For systems with FRD plants, you cannot add or modify poles and zeros outside the plotted frequency range on Bode and Nichols plots. Instead, you can make such modifications using the Compensator Editor pane of the Control and Estimation Tools Manager. For more information, see "Compensator Editor" on page 17-18.

Example: Adding a Complex Pair of Poles

This example shows you how to add a complex pair of poles to the open-loop Bode diagram. First, type

```
load ltiexamples
controlSystemDesigner('bode',sys_dc)
```

at the MATLAB prompt. This opens the SISO Design Tool with the DC motor example loaded and the open-loop Bode diagram displayed in the Graphical Tuning window.

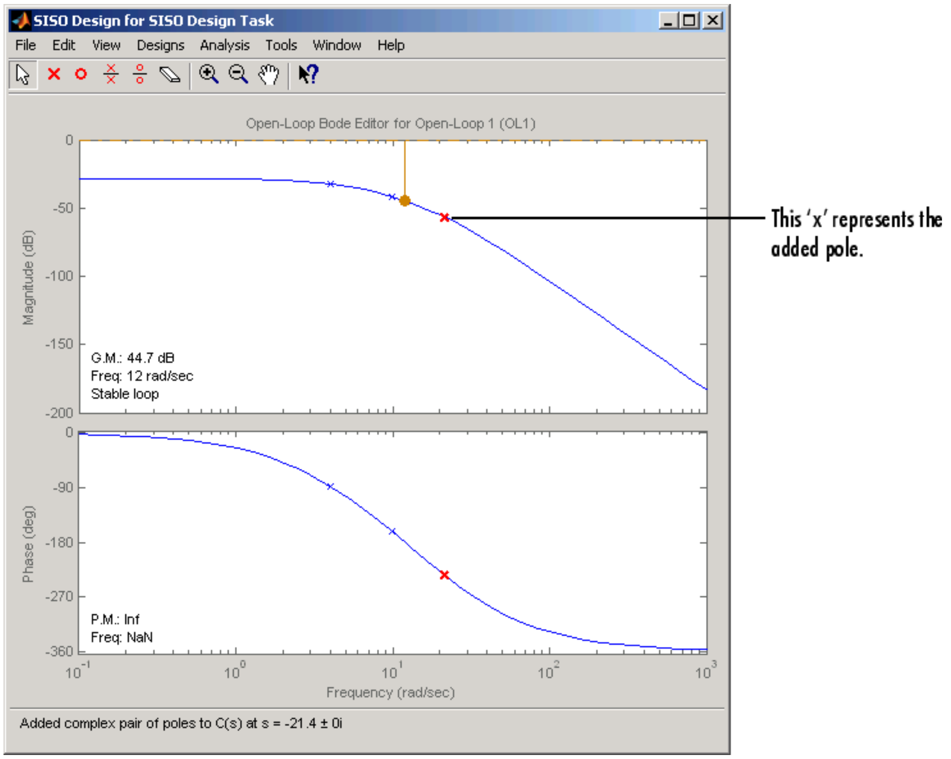

SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Open-Loop Bode Editor for Open-Loop 1 (OL1)
Magnitude (dB)
Phase (deg)
Frequency (rad/sec)
G.M.: Inf
Freq: Inf
Stable loop
P.M.: Inf
Freq: NaN
Right-click on the plots for more design options.

After selecting **Add Pole/Zero->Complex Pole** from the right-click menu, use the mouse cursor to specify the frequency of the complex pole pair.

To add a complex pair of poles:

1 Select **Add Pole/Zero->Complex Pole** from the right-click menu
2 Place the mouse cursor where you want the pole to be located
3 Left-click to add the pole

Your Graphical Tuning window should look similar to this.


SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Open-Loop Bode Editor for Open-Loop 1 (OL1)
Magnitude (dB)
Phase (deg)
Frequency (rad/sec)
G.M.: 44.7 dB
Freq: 12 rad/sec
Stable loop
P.M.: Inf
Freq: NaN
This 'x' represents the added pole.

Added complex pair of poles to C(s) at s = -21.4 ± 0i

In the case of Bode diagrams, when you place a complex pole, the default damping value is 1, which means you have a double real pole. To change the damping, grab the red `x' by left-clicking on it and drag it upward with your mouse. You will see damping ratio change in the Status pane at the bottom of the SISO Design Tool.

Delete Pole/Zero

Select **Delete Pole/Zero** to delete poles and zeros from your compensator design. When you make this selection, your cursor changes to an eraser. Place the eraser over the pole or zero you want to delete and left-click your mouse.

Note the following:

- You can only delete compensator poles and zeros. Plant (**G** in the feedback structure pane) poles and zeros cannot be altered.
- If you delete one of a pair of poles or zeros, the other member of the pair is also removed.

Edit Compensator

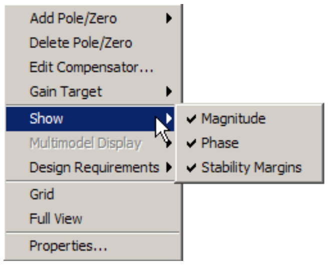
Edit Compensator opens the **Compensator Editor** pane in the SISO Design Task. You can use this pane to adjust the compensator gain and add or remove compensator poles and zeros from your compensator (**C**) or prefilter (**F**) design. See "Compensator Editor" on page 17-18 for a discussion of this pane.

Gain Target

This feature is intended for users of the Simulink Control Design software. It is nonfunctional in the Control System Toolbox software.

Show

Use **Show** to select/deselect the display of characteristics relevant to which view you are working with. This figure displays the Show submenu for the open-loop Bode diagram.

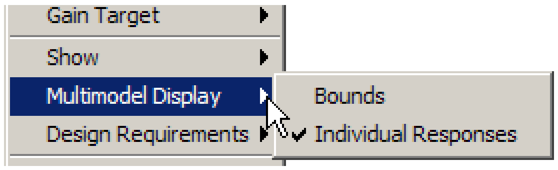


For this particular view, the options available are magnitude, phase, and stability margins. Selecting any of these toggles between showing and hiding the feature. A check next to the feature means that it is currently displayed on the Bode diagram plots. Although the characteristics are different for each view in the Graphical Tuning window, they all toggle on and off in the same manner.

Multimodel Display

This menu is enabled only when you import or open the SISO Design Tool with row or column arrays of LTI models.

Use **Multimodel Display** to view the responses of models in an LTI array as individual responses or as an envelope encompassing all responses on the Bode and Nichols plots.



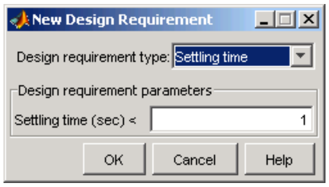
On the Root Locus plot, use this menu to show or hide the pole/zero locations of all models in the array, except the nominal one.

For more information on control design analysis for multiple models, see “Control Design Analysis of Multiple Models” in the Getting Started Guide.

Design Requirements

When designing compensators, it is common to have design specifications that call for specific settling times, damping ratios, and other characteristics. The Graphical Tuning window provides tools for design requirements that can help make the task of meeting design specifications easier.

The New Design Requirement dialog box lets you create design requirements by creating graphical representations for feasible and nonfeasible regions, automatically changes to reflect which design requirements are available for the view in which you are working. Select **Design Requirements > New** to open the New Design Requirement dialog box.


New Design Requirement

Design requirement type: Settling time

Design requirement parameters

Settling time (sec) < 1

Since each view has a different set of design requirements, click the following links to go to the appropriate descriptions:

- “Design Requirements for the Root Locus” on page 17-60
- “Design Requirements for Open- and Closed-Loop Bode Diagrams” on page 17-63
- “Design Requirements for Open-Loop Nichols Plots” on page 17-66
- “Linear System Analyzer for SISO Design Task Design Requirements” on page 17-73

For row or column arrays of LTI models, the design requirements are for the nominal plant that you are designing the controller for. You can analyze the effects of this controller on the remaining models. See “Control Design Analysis of Multiple Models” in the Getting Started Guide.

Design Requirements for the Root Locus

For the root locus, you can use the following design requirements:

- “Settling Time” on page 17-60
- “Percent Overshoot” on page 17-60
- “Damping Ratio” on page 17-61
- “Natural Frequency” on page 17-61
- “Region Constraint” on page 17-61

Use the **Design requirement type** drop-down list to select a design requirement. In each case, to specify the design requirement, enter the value in the **Design requirement parameters** pane. You can select any or all of them, or have more than one of each.

Settling Time

If you specify a settling time in the continuous-time root locus, a vertical line appears on the root locus plot at the pole locations associated with the settling time value provided (using a first-order approximation). This vertical line is exact for a second order system and is only an approximation for higher order systems. In the discrete-time case, the design requirement boundary is a curved line.

Percent Overshoot

Specifying percent overshoot in the continuous-time root locus causes two rays, starting at the root locus origin, to appear. These rays are the locus of poles associated with the percent value (using a second-order approximation). In the discrete-time case, the design requirement appears as two curves originating at (1,0) and meeting on the real axis in the left-hand plane.

Note that the percent overshoot (p.o.) design requirement can be expressed in terms of the damping ratio, as in this equation:

$$p.o. = 100\exp\left(-\frac{\pi\zeta}{\sqrt{1-\zeta^2}}\right)$$

where ζ is the damping ratio.

Damping Ratio

Specifying a damping ratio in the continuous-time root locus causes two rays, starting at the root locus origin, to appear. These rays are the locus of poles associated with the damping ratio. In the discrete-time case, the design requirement boundary appears as curved lines originating at (1,0) and meeting on the real axis in the left-hand plane.

Natural Frequency

If you specify a natural frequency lower bound, a semicircle centered around the root locus origin appears. If you specify a natural frequency upper bound, the inverse of this semicircle appears. The radius equals the natural frequency.

Region Constraint

Specifying a region constraint at given locations causes black lines and a yellow area to appear. The vertices of this free-form piecewise region are defined by the specified real and imaginary values.

Example: Adding Damping Ratio Design Requirements

This example adds a damping ratio design requirement of 0.707.

1 At the MATLAB prompt, type the following:

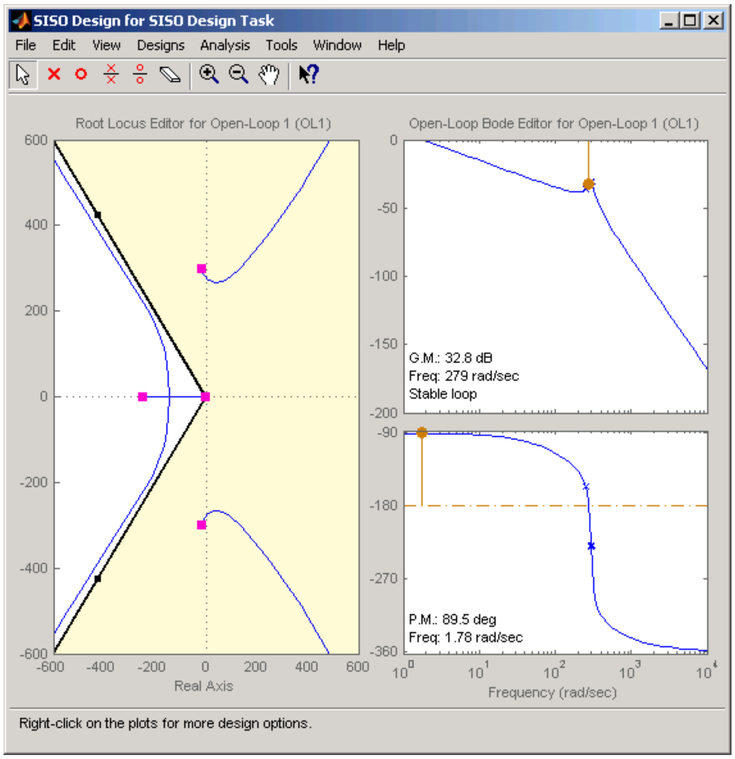
```
load ltiexamples
controlSystemDesigner(sys_dc)
```

This opens the SISO Design Tool with the DC motor example imported.

2 From the root locus right-click menu, select **Design Requirement > New** to open the New Design Requirement dialog box.

3 To add the design requirement, select **Damping Ratio** as the design requirement. Click **OK** to accept the default damping ratio of 0.707.

The Graphical Tuning window should now look similar to this figure.



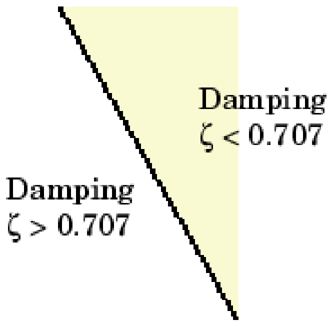
G.M.: 32.8 dB
Freq: 279 rad/sec
Stable loop

P.M.: 89.5 deg
Freq: 1.78 rad/sec

Right-click on the plots for more design options.

Damping Ratio Requirements in the Root Locus

The two rays centered at (0,0) represent the damping ratio boundaries. The dark edge is the region boundary, and the shaded area outlines the exclusion region. This figure explains what this means for this design requirement.



You can, for example, use this design requirement to ensure that the closed-loop poles, represented by the red squares, have some minimum damping. Try adjusting the gain until the damping ratio of the closed-loop poles is 0.7.

Design Requirements for Open- and Closed-Loop Bode Diagrams

For both the open- and closed-loop Bode diagrams, you have the following options:

- "Upper Gain Limit" on page 17-63
- "Lower Gain Limit" on page 17-63
- "Gain and Phase Margin" on page 17-63

Specifying any of these design requirements causes lines to appear in the Bode magnitude curve. To specify an upper or lower gain limit, enter the frequency range, the magnitude limit, and/or the slope in decibels per decade, in the appropriate fields of the New design requirement dialog box. You can have as many gain limit design requirements as you like in your Bode magnitude plots.

Upper Gain Limit

You can specify one or multiple piecewise linear upper gain limits over a frequency range, which appear as straight lines on the Bode magnitude curve. You must select frequency limits, the upper gain limit in decibels, and the slope in dB/decade.

Lower Gain Limit

You can specify one or multiple lower gain limit in the same fashion as the upper gain limit.

Gain and Phase Margin

You can specify a lower bound for the gain, the phase margin, or both. The specified bounds appear in text on the Bode magnitude plot.

Example: Adding Upper Gain Limits

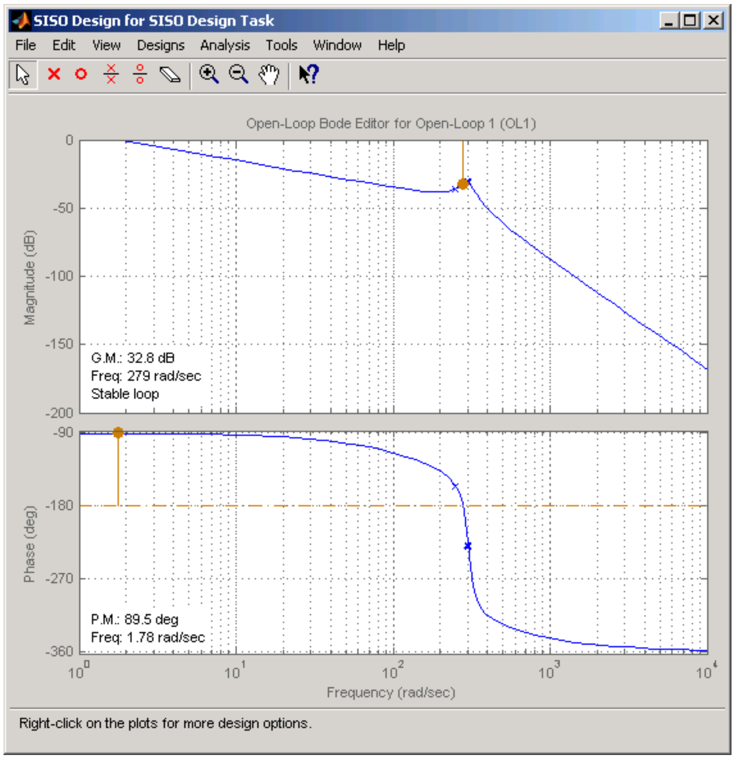
This example shows you how to add two upper gain limit requirements to the open-loop Bode diagram.

1 At the MATLAB prompt, type the following:

```
load ltiexamples
controlSystemDesigner('bode',Gservo)
```

This opens the SISO Design Tool with the servomechanism model loaded.

2 Use the right-click menu to add a grid.

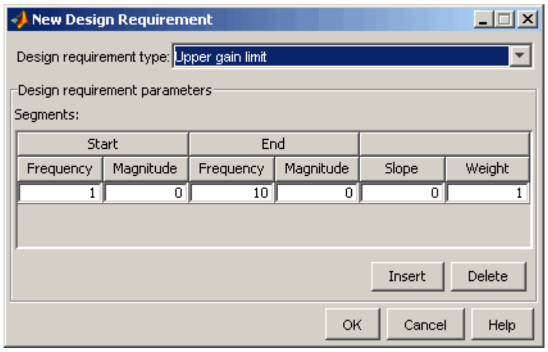

SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Open-Loop Bode Editor for Open-Loop 1 (OL1)
Magnitude (dB)
0
-50
-100
-150
-200
Phase (deg)
-90
-180
-270
-360
10^0
10^1
10^2
10^3
10^4
Frequency (rad/sec)

G.M.: 32.8 dB
Freq: 279 rad/sec
Stable loop

P.M.: 89.5 deg
Freq: 1.78 rad/sec

Right-click on the plots for more design options.

3 To add an upper gain limit requirement of 0 dB from 10 rad/sec to 100 rad/sec, open the New Design Requirement dialog box and select **Upper gain limit** from the pull-down menu. Fill in the dialog box fields as shown in the following figure.



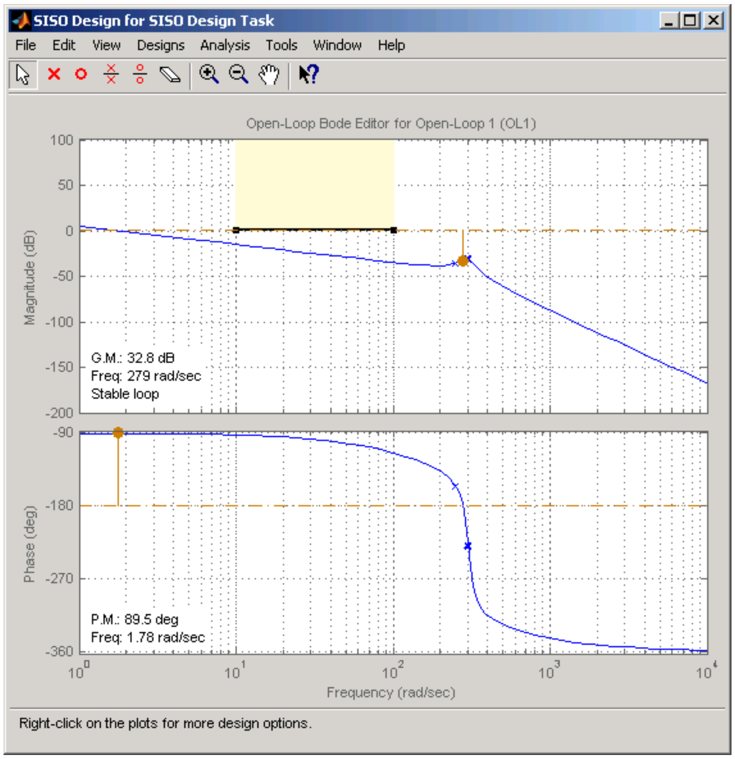
Design requirement type: Upper gain limit

Design requirement parameters

Segments:

Start		End			
Frequency	Magnitude	Frequency	Magnitude	Slope	Weight
1	0	10	0	0	1

Your Graphical Tuning window should now look like this (you may have to adjust some axis limits).

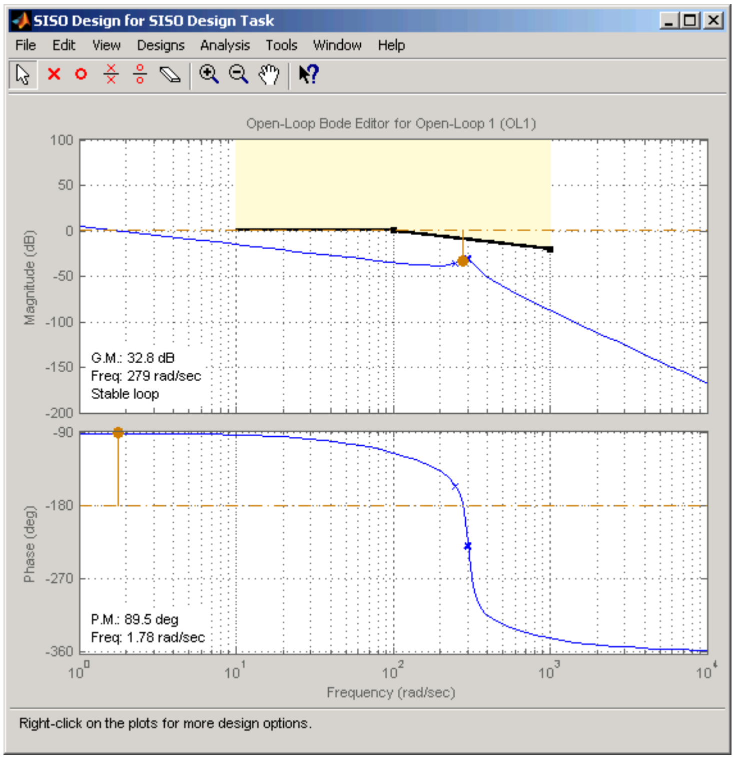


G.M.: 32.8 dB
Freq: 279 rad/sec
Stable loop

P.M.: 89.5 deg
Freq: 1.78 rad/sec

Right-click on the plots for more design options.

4 To constrain the roll off, open the New Design Requirement dialog box and add an upper gain limit from 100 rad/sec to 1000 rad/sec with a slope of -20 db/decade. This figure shows the result.



With these design requirements in place, you can see how much you can increase the compensator gain and still meet design specifications.

Note that you can change the design requirements by moving them with your mouse. See “Editing Design Requirements” on page 17-69 for more information.

Design Requirements for Open-Loop Nichols Plots

For open-loop Nichols plots, you have the following design requirement options:

- “Phase Margin” on page 17-67
- “Gain Margin” on page 17-67

- “Closed-Loop Peak Gain” on page 17-67
- “Gain-Phase Design Requirement” on page 17-67

Specifying any of these design requirements causes lines or curves to appear in the Nichols plot. In each case, to specify the design requirement, enter the value in the **Design requirement parameters** pane. You can select any or all of them, or have more than one of each.

Phase Margin

Specify a minimum phase margin at a given location. For example, you can require a minimum of 30 degrees at the -180 degree crossover. The phase margin specified should be a number greater than 0. The location must be a -180 plus a multiple of 360 degrees. If you enter an invalid location point, the closest valid location is selected.

Gain Margin

Specify a gain margin at a given location. For example, you can require a minimum of 20 dB at the -180 degree crossover. The location must be -180 plus a multiple of 360 degrees. If you enter an invalid location point, the closest valid location is selected.

Closed-Loop Peak Gain

Specify a peak closed-loop gain at a given location. The specified dB value can be positive or negative. The design requirement follows the curves of the Nichols plot grid, so it is recommended that you have the grid on when using this feature.

Gain-Phase Design Requirement

Specify both a gain and phase design requirement at a given location. The vertices of this free-form piecewise region are defined by the specified open-loop phase and open-loop gain values.

Example: Adding a Closed-Loop Peak Gain Design Requirement

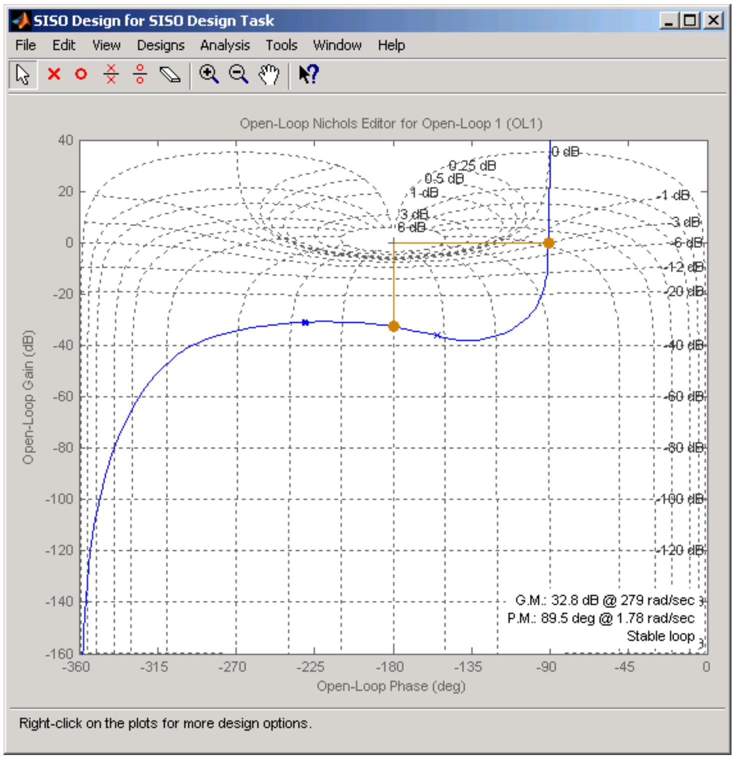
This example shows how to add a closed-loop peak gain design requirement to the Nichols plot.

1. At the MATLAB prompt, type the following:

   ```
   load ltiexamples
   controlSystemDesigner('nichols',Gservo)
   ```

This opens the SISO Design Tool with `Gservo` imported as the plant.

2 Use the right-click menu to add a grid, as this figure shows.

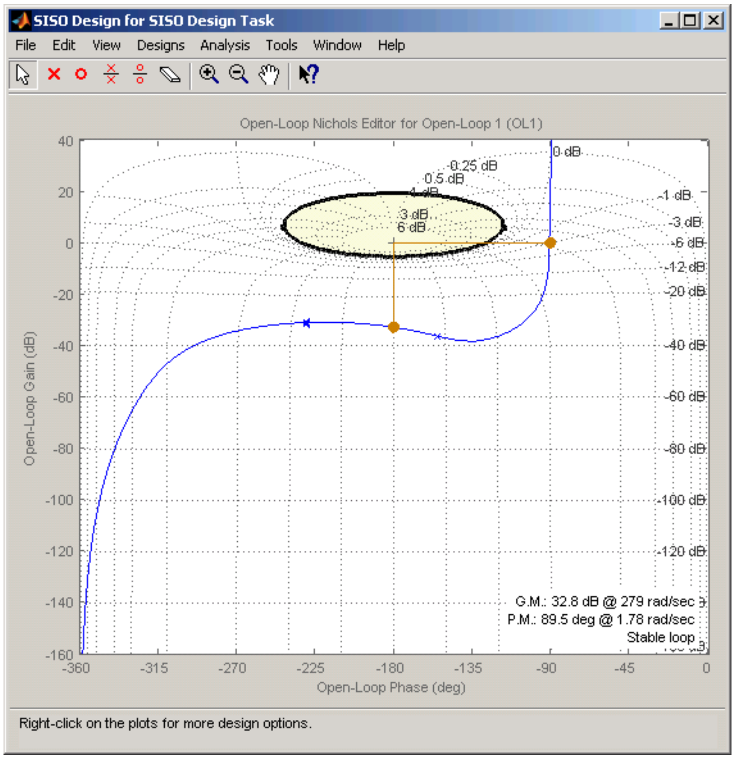


G.M.: 32.8 dB @ 279 rad/sec
P.M.: 89.5 deg @ 1.78 rad/sec
Stable loop

Right-click on the plots for more design options.

3 To add closed-loop peak gain of 1 dB at -180 degrees, open the New Design Requirement dialog box and select **Closed-Loop Peak Gain** from the pull-down menu. Set the peak gain field to 1 dB.

The figure shows the resulting design requirement.


SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Open-Loop Nichols Editor for Open-Loop 1 (OL1)
Open-Loop Gain (dB)
Open-Loop Phase (deg)
0 dB
0.25 dB
0.5 dB
1 dB
3 dB
6 dB
-1 dB
-3 dB
-6 dB
-12 dB
-20 dB
-40 dB
-60 dB
-80 dB
-100 dB
-120 dB
G.M.: 32.8 dB @ 279 rad/sec
P.M.: 89.5 deg @ 1.78 rad/sec
Stable loop
Right-click on the plots for more design options.

As long as the curve is outside of the gray region, the closed-loop gain is guaranteed to be less than 1 dB. Note that this is equivalent, up to second order, to specifying the peak overshoot in the time domain. In this case, a 1 dB closed-loop peak gain corresponds to an overshoot of 15%.

Editing Design Requirements

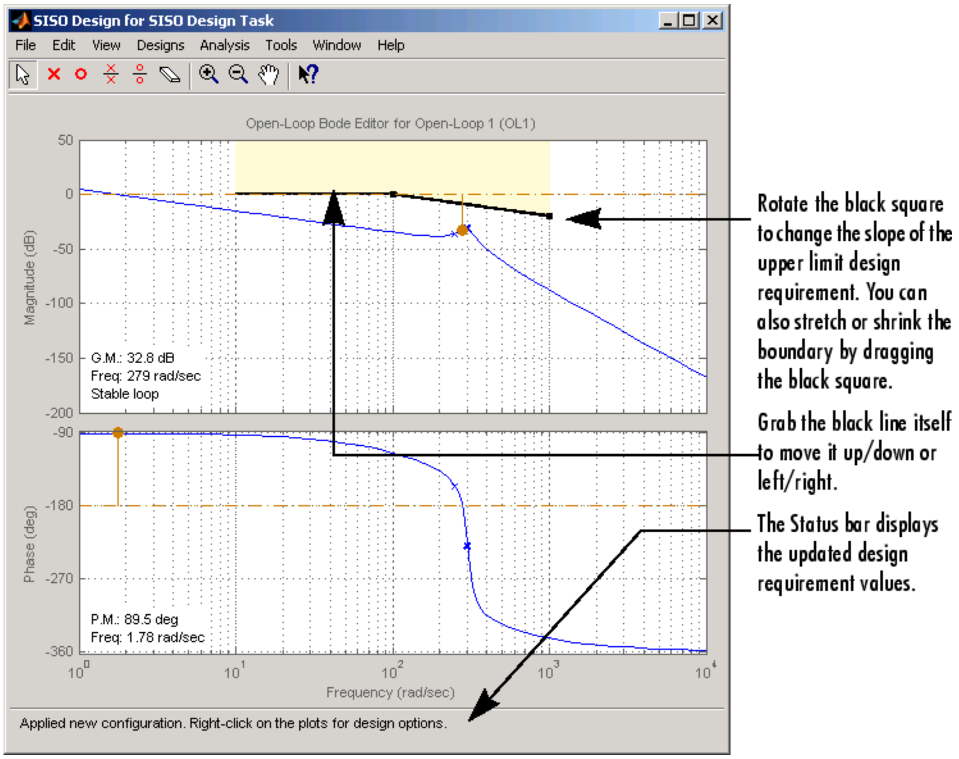
To edit an existing design requirement, left-click on the design requirement boundary to select it. Two black squares appear on the design requirement when it is selected. In general, there are two ways to adjust a design requirement:

- Click on the design requirement boundary and drag it. Generally, this does not change the shape of the boundary. That is, the adjustment is strictly a translation of the design requirement.

- Grab a black square and drag it. In this case, you can rotate, expand, and/or contract the design requirement.

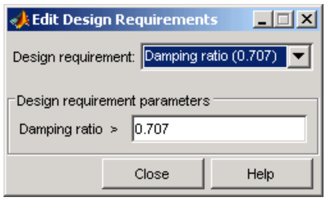
For example, in Bode diagrams you can move an upper gain limit by clicking on it and moving it anywhere in the plot region. As long as you haven't grabbed a black square, the length and slope of the gain limit will not change as you move the line. On the other hand, you can change the slope of the upper gain limit by grabbing one of the black squares and rotating the line. In all cases, the Status pane at the bottom of the Graphical Tuning window displays the design requirement values as they change.

This figure shows the process of editing an upper gain limit in the open-loop Bode diagram.


SISO Design for SISO Design Task
File Edit View Designs Analysis Tools Window Help
Open-Loop Bode Editor for Open-Loop 1 (OL1)
50
0
-50
-100
-150
-200
Magnitude (dB)
G.M.: 32.8 dB
Freq: 279 rad/sec
Stable loop
-90
-180
-270
-360
Phase (deg)
P.M.: 89.5 deg
Freq: 1.78 rad/sec
10^0
10^1
10^2
10^3
10^4
Frequency (rad/sec)
Rotate the black square to change the slope of the upper limit design requirement. You can also stretch or shrink the boundary by dragging the black square.
Grab the black line itself to move it up/down or left/right.
The Status bar displays the updated design requirement values.

Applied new configuration. Right-click on the plots for design options.

An alternative way to adjust a design requirement is to select **Design Requirements->Edit** from the right-click menu. The **Edit Design Requirement** window opens.



Design requirement: Damping ratio (0.707)

Design requirement parameters

Damping ratio > 0.707

To adjust a design requirement, select the boundary by clicking on it and change the values in the fields of the Design requirement parameters pane. If you have additional design requirement in, for example, the Bode diagram, you can edit them directly from this window by selecting **Open-Loop Bode** from the **Editor** menu.

Deleting Design Requirements

To delete a design requirement, place your cursor directly over the design requirement yellow region. Right-click to open a menu containing **Edit** and **Delete**. Select **Delete** from the menu list; this eliminates the design requirement. You can also delete design requirements by left-clicking on a design requirement boundary and then pressing the **BackSpace** or **Delete** key on your keyboard.

Finally, you can delete design requirements by selecting **Undo Add Design Requirement** from the **Edit** menu, or pressing **Ctrl+Z** if adding design requirements was the last action you took.

Grid

Grid adds a grid to the selected plot.

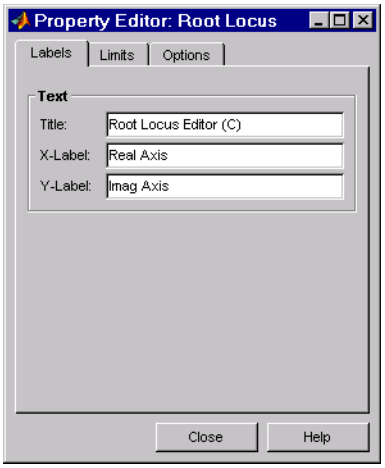
Full View

Selecting **Full View** causes the plot to scale limits so that the entire curve is visible.

Properties

Properties opens the **Property Editor**, which is a GUI for customizing root locus, Bode diagrams, and Nichols plots inside the Graphical Tuning window. The Property Editor automatically reconfigures as you select among the different plots open.

This picture shows the open window for the root locus.



Text
Title: Root Locus Editor (C)
X-Label: Real Axis
Y-Label: Imag Axis

You can use this window to change titles and axis labels, reset axes limits, add grid lines, and change the aspect ratio of the plot. Note that you can also activate this menu by double-clicking anywhere in the root locus away from the curve.

The are only three panes in the Property Editor: Labels, Limits, and Options. The configuration of each page differs, depending on whether you're working with the root-locus, Bode diagrams, or the open-loop Nichols plot. Click the **Help** button on the Property Editor you have open to view information specific to that editor, or click on the links below:

- ???
- ???
- ???

Select Compensator

This option allows you to select which compensator to edit for closed-loop Bode response.

Status Pane

The Status pane is located at the bottom of the Graphical Tuning window. It displays the most recent action you have performed, occasionally provides advice on how to use the window, and tracks key parameters when moving objects in the design views.

Linear System Analyzer for SISO Design Task Design Requirements

In this section...

Overview of Linear System Analyzer Design Requirements

You can use the Linear System Analyzer for SISO Design Tasks to specify both time and frequency domain requirements in analysis plots. Adding and editing design requirements is similar to those illustrated in the Graphical Tuning window.

Note: To add design requirements, you must open the Linear System Analyzer from the SISO Design Task in the Control and Estimation Tools Manager. Design requirements are not available from a Linear System Analyzer that is opened using the `linearSystemAnalyzer` command.

For row or column arrays of LTI models, the design requirements are for the nominal model that you are designing the controller for. You can analyze the effects of this controller on the remaining models. See "Control Design Analysis of Multiple Models" in the Getting Started Guide.

Available Design Requirements in the Linear System Analyzer

The design requirements for Bode, Root Locus, and Nichols plots can be applied to both graphical tuning windows and the Linear System Analyzer. See "Design Requirements" on page 17-59 for information on graphical tuning design requirements.

You can also specify the following design requirements for both step and impulse response plots:

- **Upper time response bounds** — Creates an upper amplitude bound for a specified time duration.
- **Lower time response bounds** — Creates a lower amplitude bound for a specified time duration.

If you are using a step response plot, you can also specify the following design requirement:

- **Step response bounds** — Creates a group of upper and lower time response bounds, in the shape of a step response envelope, to encompass your specified design requirement parameters.

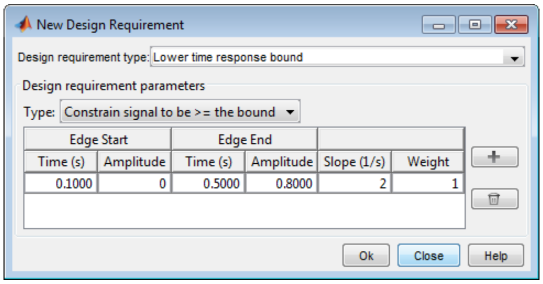
Example: Time Domain Requirement

This example shows you how to create a lower bound time response design requirement.

1. At the MATLAB prompt, type the following:

   ```
   load ltiexamples
   controlSystemDesigner(Gservo)
   ```

2. From the **Analysis Plot** pane, select a step response for the closed-loop system. See “Analysis Plots” on page 17-22 if you are unfamiliar with this task.
3. Right-click on the plot and select **Design Requirements** > **New**.
4. Select `Lower time response bound` from the **Design requirement type** menu.
5. Set Time from 0.1 to 0.5 s.
6. Set Amplitude from 0 to 0.8. Your New Design Requirement window should look like this.



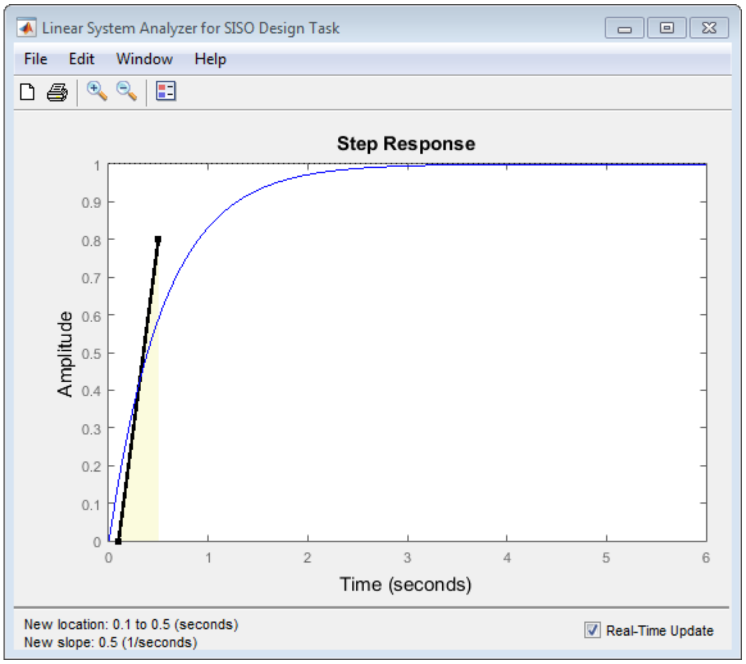
Design requirement type: Lower time response bound

Design requirement parameters

Type: Constrain signal to be >= the bound

Edge Start		Edge End			
Time (s)	Amplitude	Time (s)	Amplitude	Slope (1/s)	Weight
0.1000	0	0.5000	0.8000	2	1

7. Click **OK**. This adds the design requirement to the plot. Your step response should look like this.



New location: 0.1 to 0.5 (seconds)
New slope: 0.5 (1/seconds)

Now, when you adjust or tune the controller, you can use the plot to observe how well the response meets your design requirements.

See Also

`controlSystemDesigner`

Related Examples

- “Getting Started with the SISO Design Tool” on page 10-2

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.

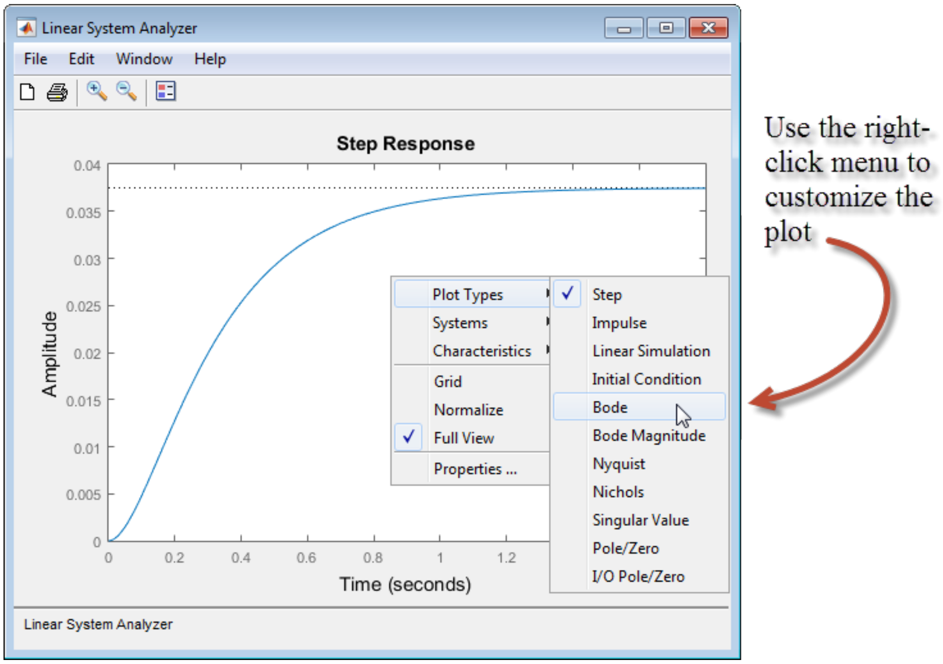
You can launch the Linear System Analyzer in two ways:

- Use the `linearSystemAnalyzer` command.
- Create analysis plots or design requirements within SISO Design Tool.

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 18-4
- “Importing, Exporting, and Deleting Models in the Linear System Analyzer” on page 18-12
- “Selecting Response Types” on page 18-15
- “Analyzing MIMO Models” on page 18-20
- “Customizing the Linear System Analyzer” on page 18-26

Using the Right-Click Menu in the Linear System Analyzer

In this section...
"Overview of the Right-Click Menu" on page 18-4
"Setting Characteristics of Response Plots" on page 18-4
"Adding Design Requirements" on page 18-8

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
- **Design Requirements** — Opens the New Design Requirement window for adding step response design requirements to your plot (available only for Linear System Analyzers linked to the Graphical Tuning window of the SISO Design Tool)

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.

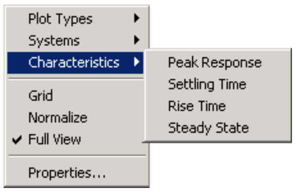
Setting Characteristics of Response Plots

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.



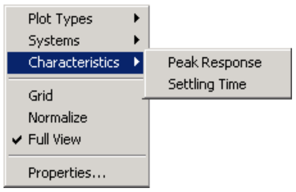
You can display the following information in the step response:

- **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 13-2, or the ???.

Impulse Response

Impulse Response plots the model's response to an impulse.

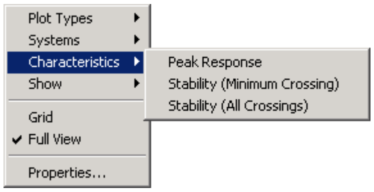


The Linear System Analyzer can display the following information in the impulse response:

- **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

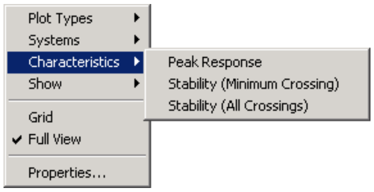
Bode plots the open-loop Bode phase and magnitude diagrams for the model.



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



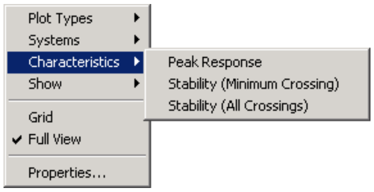
Bode Magnitude plots the Bode magnitude diagram for the model.

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



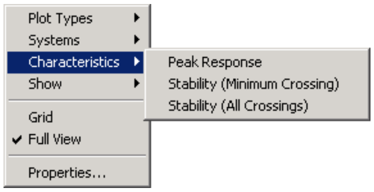
Nyquist plots the Nyquist diagram for the model.

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.

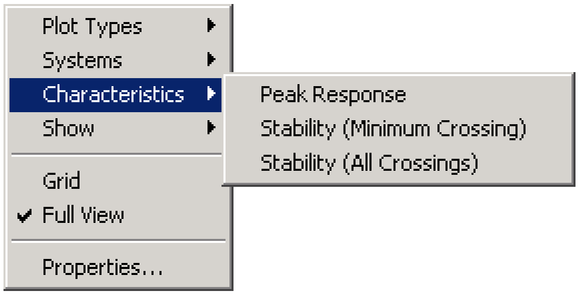


The Linear System Analyzer can display the following types of information in the Nichols chart:

- **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.

Adding Design Requirements

If you open a Linear System Analyzer for the Graphical Tuning window of the SISO Design Tool, you have plots linked to your compensator design. In this environment, the Linear System Analyzer provides access to design requirements, a set of graphical tools for creating constraints in your design plots.

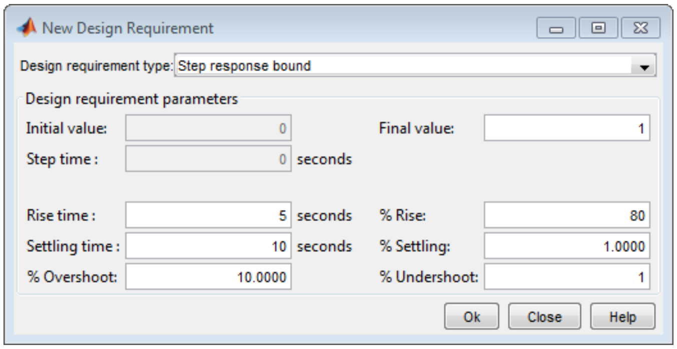
In addition to all the design requirements available in the Graphical Tuning window, the Linear System Analyzer has the step response design requirements described in “Choosing Step Response Specifications” on page 18-9.

For more information on adding design requirements to Linear System Analyzer plots, see "Linear System Analyzer for SISO Design Task Design Requirements" on page 17-73.

Note Design requirements are not available from a Linear System Analyzer that is opened using the `linearSystemAnalyzer` command or the **Apps** tab of the MATLAB desktop.

Choosing Step Response Specifications

To specify step response characteristics select **Design Requirements > New** in the right-click menu. This action opens the **New Design Requirements** editor. Select **step response bounds** from the **Design requirement type** pull down menu to display the step response specifications as shown below.



Design requirement type: Step response bound

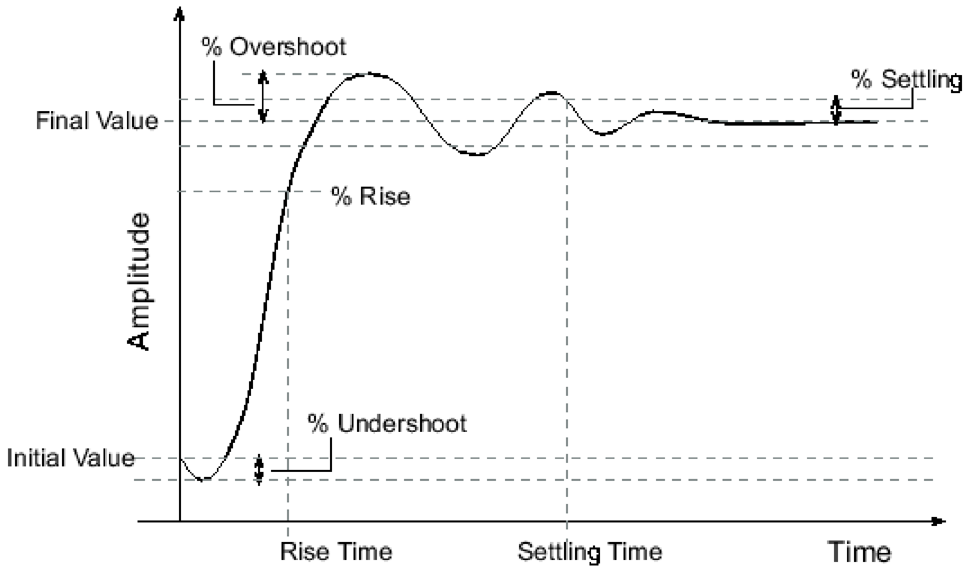
Design requirement parameters

Initial value:	0		Final value:	1
Step time :	0	seconds		
Rise time :	5	seconds	% Rise:	80
Settling time :	10	seconds	% Settling:	1.0000
% Overshoot:	10.0000		% Undershoot:	1

The top three options specify the details of the step input:

- **Initial value**: input level before the step occurs. This option is grayed out because LTI systems always have initial value equal to 0.
- **Step time**: time at which the step takes place. This option is grayed out since LTI systems always have an initial time equal to 0.
- **Final value**: input level after the step occurs

The remaining options specify the characteristics of the response signal. Each of the step response characteristics is described in the figure below.



- **Rise time**: The time taken for the response signal to reach a specified percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Rise**: The percentage used in the rise time.
- **Settling time**: The time taken until the response signal settles within a specified region around the final value. This settling region is defined as the final step value plus or minus the specified percentage of the final value.
- **% Settling**: The percentage used in the settling time.
- **% Overshoot**: The amount by which the response signal can exceed the final value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.
- **% Undershoot**: The amount by which the response signal can undershoot the initial value. This amount is specified as a percentage of the step's range. The step's range is the difference between the final and initial values.

Enter values for the response specifications in the **Design Requirements** editor, based on the requirements of your model, and then click **OK**. The constraint edges will now reflect the constraints specified.

See Also

`linearSystemAnalyzer`

Related Examples

- “Joint Time- and Frequency-Domain Analysis” on page 6-20

More About

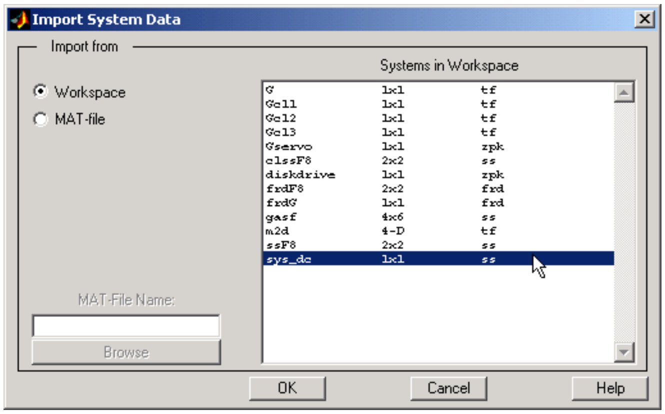
- “Linear System Analyzer Overview” on page 18-2

Importing, Exporting, and Deleting Models in the Linear System Analyzer

In this section...
"Importing Models" on page 18-12
"Exporting Models" on page 18-13
"Deleting Models" on page 18-13

Importing Models

To import models into the Linear System Analyzer, select **File** > **Import**. The **Import System Data** dialog box opens, as shown below.



Systems in Workspace

```
G           1x1     tf
Gcl1        1x1     tf
Gcl2        1x1     tf
Gcl3        1x1     tf
Gservo      1x1     zpk
clssF8      2x2     ss
diskdrive   1x1     zpk
frdF8       2x2     frd
frdG        1x1     frd
gasf        4x6     ss
m2d         4-D     tf
ssF8        2x2     ss
sys_dc      1x1     ss
```

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.

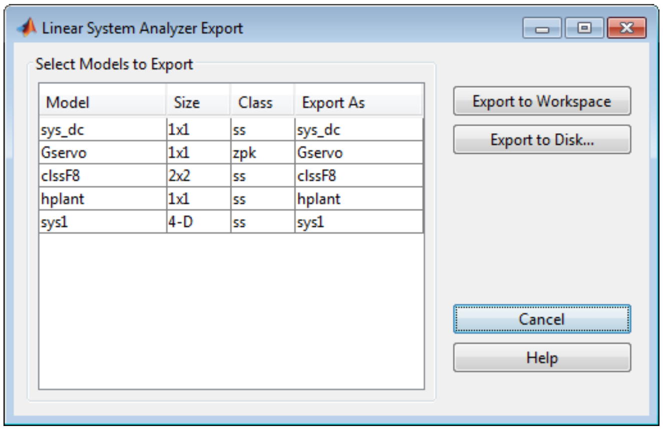
Alternatively, you can directly import a model into the Linear System Analyzer using the linearSystemAnalyzer function, as in

```
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.



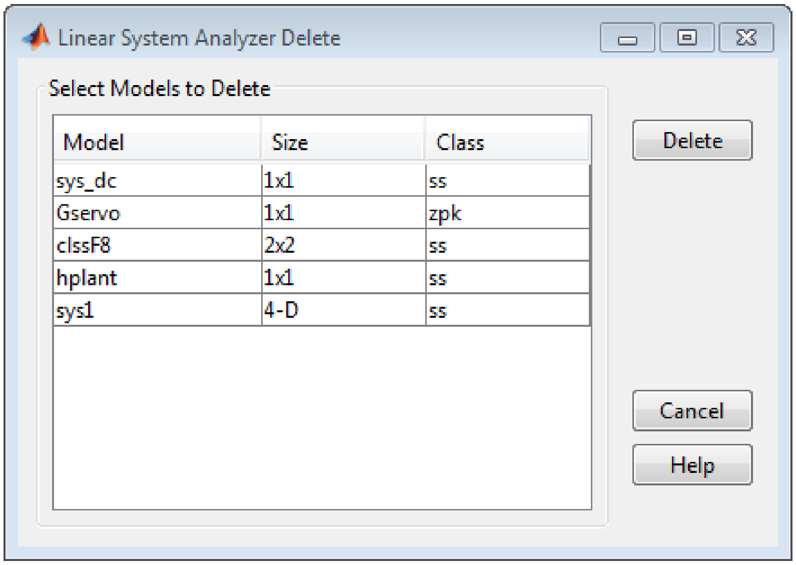
Model	Size	Class	Export As
sys_dc	1x1	ss	sys_dc
Gservo	1x1	zpk	Gservo
clssF8	2x2	ss	clssF8
hplant	1x1	ss	hplant
sys1	4-D	ss	sys1

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.



Model	Size	Class
sys_dc	1x1	ss
Gservo	1x1	zpk
clssF8	2x2	ss
hplant	1x1	ss
sys1	4-D	ss

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 18-2

Selecting Response Types

In this section...

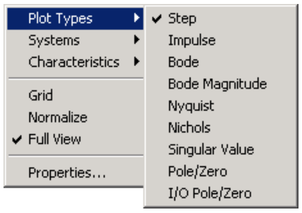
Methods for Selecting Response Types

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

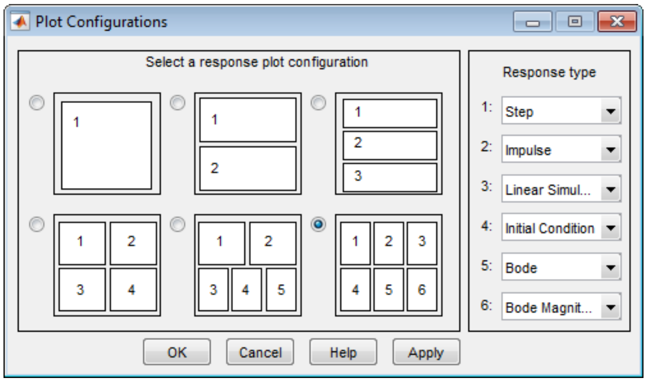
If you have a plot open in the Linear System Analyzer, you can switch to any other response plot available by selecting **Plot Type** from the right click menu.



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.



Response type

1: Step

2: Impulse

3: Linear Simul...

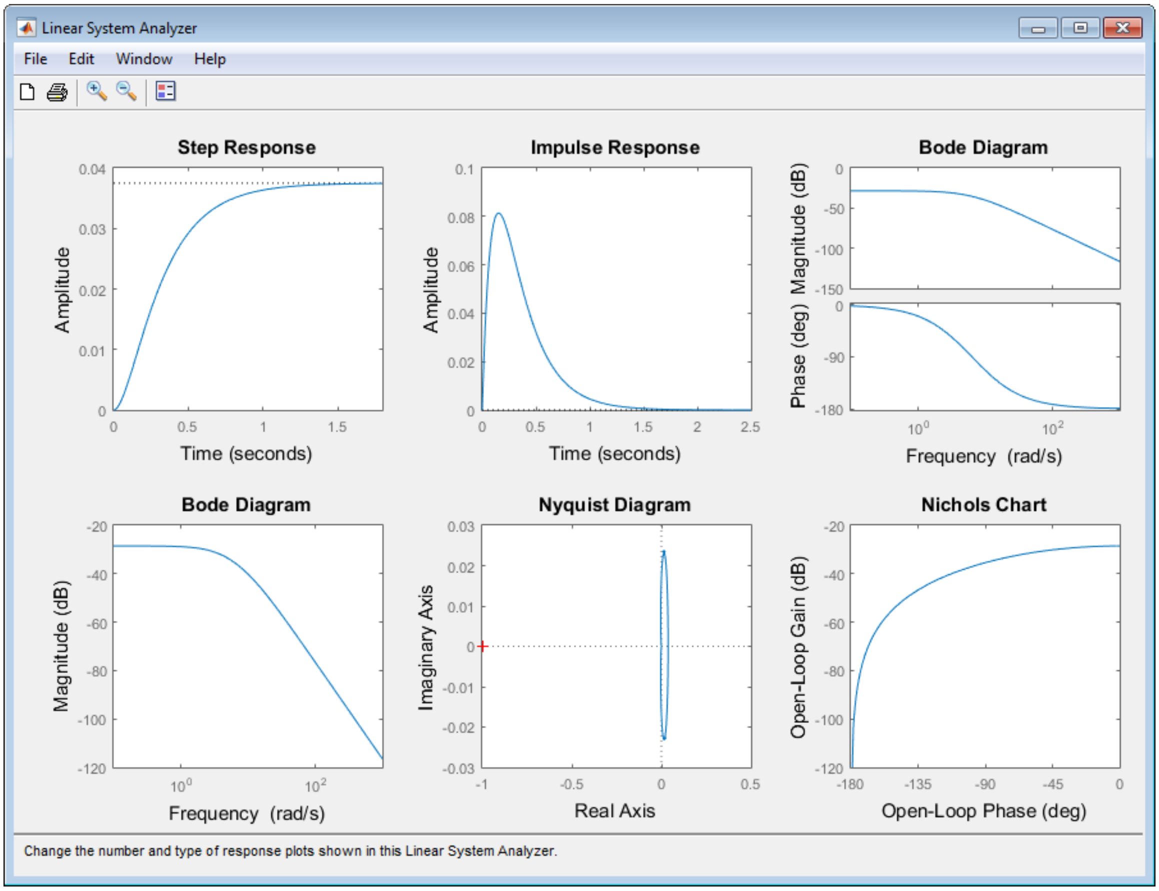
4: Initial Condition

5: Bode

6: Bode Magnit...

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.

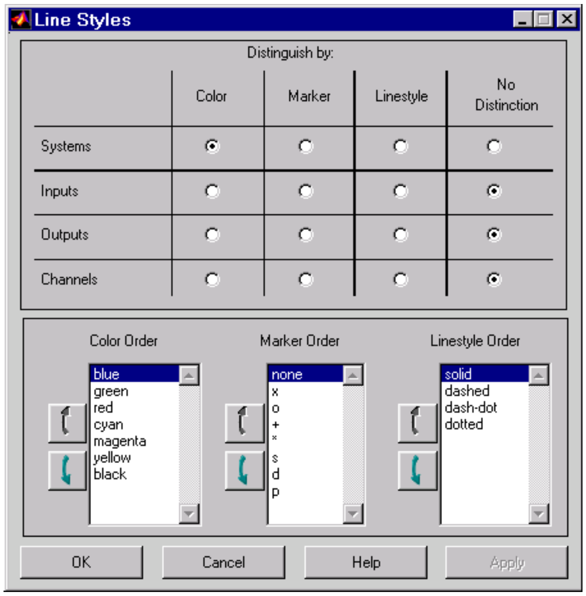


Change the number and type of response plots shown in this Linear System Analyzer.

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.



Distinguish by:

	Color	Marker	Linestyle	No Distinction
Systems	✓			
Inputs				✓
Outputs				✓
Channels				✓

Color Order	Marker Order	Linestyle Order
blue	none	solid
green	x	dashed
red	o	dash-dot
cyan	+	dotted
magenta	*	
yellow	s	
black	d	
	p	

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 6-20

More About

- "Linear System Analyzer Overview" on page 18-2

Analyzing MIMO Models

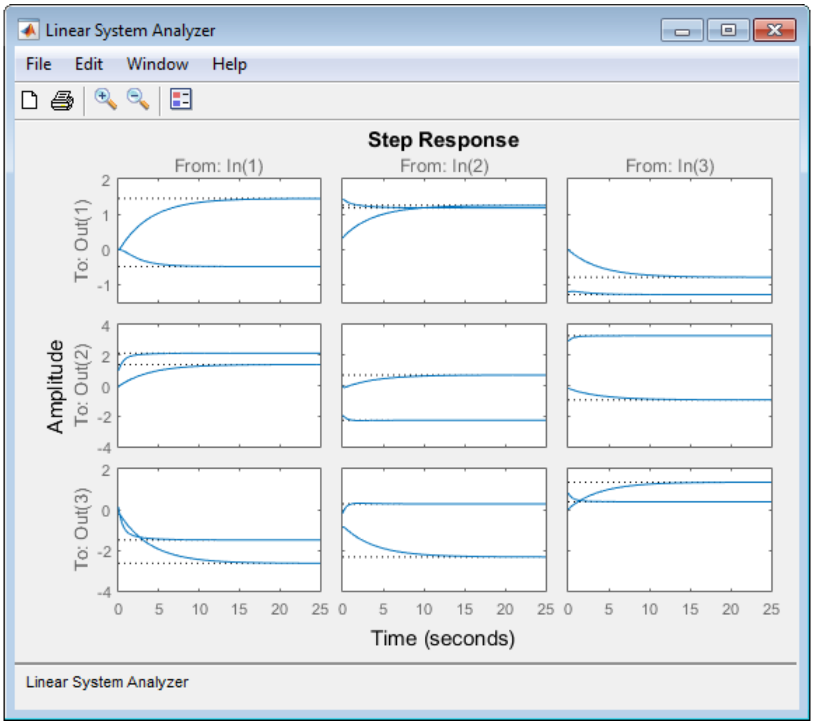
In this section...

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:

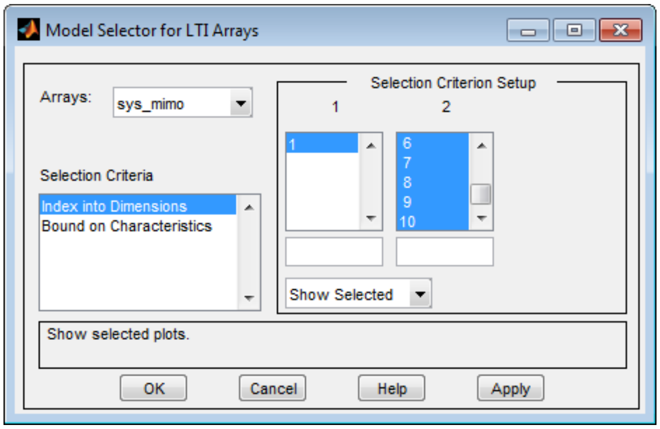
```
sys_mimo=stack(1,rss(3,3,3),rss(3,3,3));
linearSystemAnalyzer(sys_mimo);
```

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.

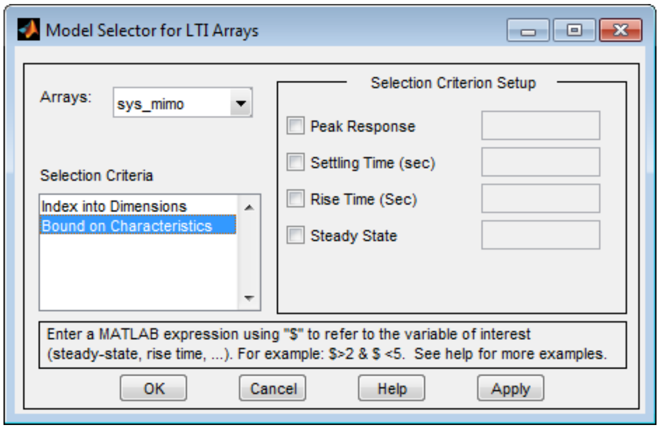
Arrays

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



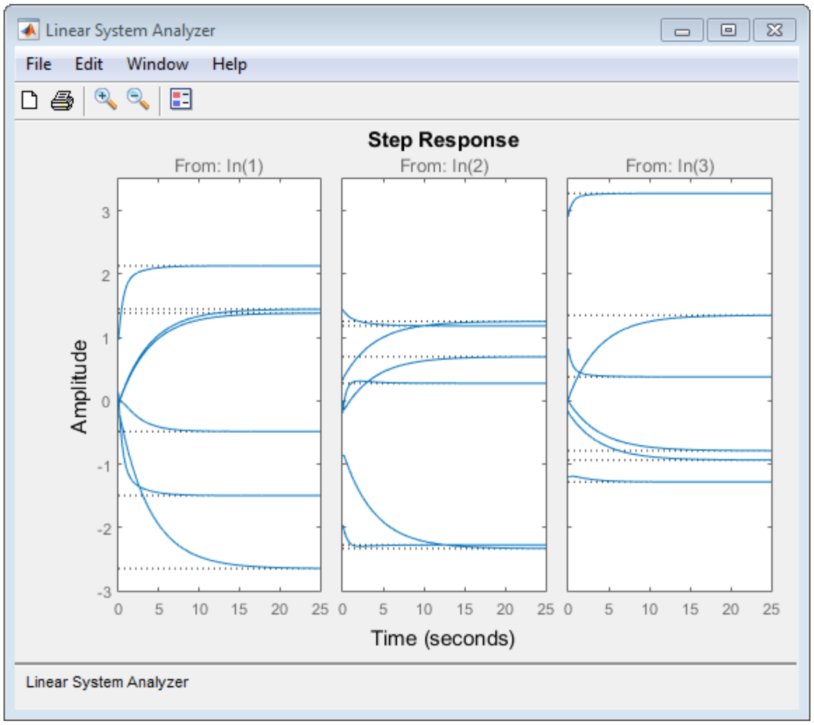
Enter a MATLAB expression using "$" to refer to the variable of interest (steady-state, rise time, ...). For example: $>2 & $ <5. See help for more examples.

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**.

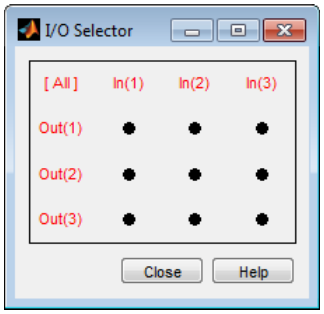
For example, if you select **Outputs**, the step plot reconfigures into 3 plots, grouping all the outputs together on each plot. Each plot now displays the responses from one of the inputs to all of the MIMO system's outputs, for all of the models in the array.



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

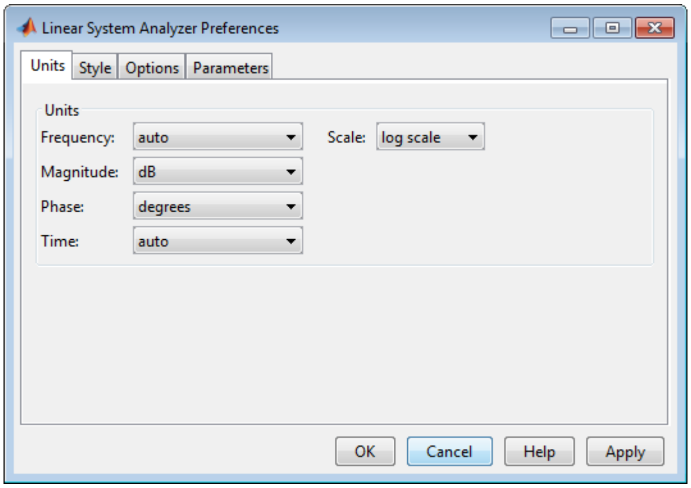
In this section...
"Overview of Customizing the Linear System Analyzer" on page 18-26
"Linear System Analyzer Preferences Editor" on page 18-26

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.



Units
Frequency: auto
Scale: log scale
Magnitude: dB
Phase: degrees
Time: auto

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 13-2.

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 18-2