Control System Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

Using the Control System Toolbox

Version 5

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax

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

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

Using the Control System Toolbox

Printing History:

November 2000	Online only	Revised for Version 5 (Release 12) (Reorganized and a name change)
June 2001	Online only	Revised for Version 5.1 (Release 12.1)
July 2002	Online only	Revised for Version 5.2 (Release 13)

Contents

1 LTI Models

2 Operations on LTI Models

3 Model Analysis Tools

4 Arrays of LTI Models

5 Customization

6 Setting Toolbox Preferences

7 Setting Tool Preferences

8 Customizing Response Plot Properties

9 Design Case Studies

10 Reliable Computations

11 Tool and Viewer Quick Start

Index

1

LTI Models

The Control System Toolbox offers extensive tools to manipulate and analyze linear time-invariant (LTI) models. It supports both continuous- and discrete-time systems. Systems can be single-input/single-output (SISO) or multiple-input/multiple-output (MIMO). In addition, you can store several LTI models in an array under a single variable name. See Chapter 4, "Arrays of LTI Models" for information on LTI arrays.

This section introduces key concepts about the MATLAB representation of LTI models, including LTI objects, precedence rules for operations, and an analogy between LTI systems and matrices. In addition, it summarizes the basic commands you can use on LTI objects.

LTI Models

You can specify LTI models as:

- Transfer functions (TF), for example,

$$P(s) = \frac{s+2}{s^2+s+10}$$

- Zero-pole-gain models (ZPK), for example,

$$H(z) = \begin{bmatrix} \frac{2(z-0.5)}{z(z+0.1)} & \frac{(z^2+z+1)}{(z+0.2)(z+0.1)} \end{bmatrix}$$

- State-space models (SS), for example,

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du$$

  where A, B, C, and D are matrices of appropriate dimensions, x is the state vector, and u and y are the input and output vectors.

- Frequency response data (FRD) models

  FRD models consist of sampled measurements of a system's frequency response. For example, you can store experimentally collected frequency response data in an FRD.

Using LTI Models in the Control System Toolbox

You can manipulate TF, SS, and ZPK models using the arithmetic and model interconnection operations described in Chapter 2, "Operations on LTI Models" and analyze them using the model analysis functions, such as `bode` and `step`. FRD models can be manipulated and analyzed in much the same way you analyze the other model types, but analysis is restricted to frequency-domain methods.

Using a variety of design techniques, you can design compensators for systems specified with TF, ZPK, SS, and FRD models. These techniques include root locus analysis, pole placement, LQG optimal control, and frequency domain loop-shaping. For FRD models, you can either:

- Obtain an identified TF, SS, or ZPK model using system identification techniques.
- Use frequency-domain analysis techniques.

Other Uses of FRD Models

FRD models are unique model types available in the Control System Toolbox collection of LTI model types, in that they don't have a parametric representation. In addition to the standard operations you may perform on FRD models, you can also use them to:

- Perform frequency-domain analysis on systems with nonlinearities using describing functions.
- Validate identified models against experimental frequency response data.

LTI Objects

Depending on the type of model you use, the data for your model may consist of a simple numerator/denominator pair for SISO transfer functions, four matrices for state-space models, and multiple sets of zeros and poles for MIMO zero-pole-gain models or frequency and response vectors for FRD models. For convenience, the Control System Toolbox provides customized data structures (*LTI objects*) for each type of model. These are called the TF, ZPK, SS, and FRD objects. These four LTI objects encapsulate the model data and enable you to manipulate LTI systems as single entities rather than collections of data vectors or matrices.

Creating an LTI Object: An Example

An LTI object of the type TF, ZPK, SS, or FRD is created whenever you invoke the corresponding constructor function, `tf`, `zpk`, `ss`, or `frd`. For example,

```
P = tf([1 2],[1 1 10])
```

creates a TF object, `P`, that stores the numerator and denominator coefficients of the transfer function

$$P(s) = \frac{s+2}{s^2+s+10}$$

See "Creating LTI Models" on page 1-8 for methods for creating all of the LTI object types.

LTI Properties and Methods

The LTI object implementation relies on MATLAB object-oriented programming capabilities. Objects are MATLAB structures with an additional flag indicating their class (TF, ZPK, SS, or FRD for LTI objects) and have pre-defined fields called *object properties*. For LTI objects, these properties include the model data, sample time, delay times, input or output names, and input or output groups (see "LTI Properties" on page 1-25 for details). The functions that operate on a particular object are called the object *methods*. These may include customized versions of simple operations such as addition or multiplication. For example,

```
P = tf([1 2],[1 1 10])
Q = 2 + P
```

performs transfer function addition.

$$Q(s) = 2 + P(s) = \frac{2s^2+3s+22}{s^2+s+10}$$

The object-specific versions of such standard operations are called *overloaded* operations. For more details on objects, methods, and object-oriented programming, see Classes and Objects"in the MATLAB documentation. For details on operations on LTI objects, see Chapter 2, "Operations on LTI Models."

Precedence Rules

Operations like addition and commands like `feedback` operate on more than one LTI model at a time. If these LTI models are represented as LTI objects of different types (for example, the first operand is TF and the second operand is SS), it is not obvious what type (for example, TF or SS) the resulting model should be. Such type conflicts are resolved by *precedence rules*. Specifically, TF, ZPK, SS, and FRD objects are ranked according to the precedence hierarchy.

FRD > SS > ZPK > TF

Thus ZPK takes precedence over TF, SS takes precedence over both TF and ZPK, and FRD takes precedence over all three. In other words, any operation involving two or more LTI models produces:

- An FRD object if at least one operand is an FRD object
- An SS object if no operand is an FRD object and at least one operand is an SS object
- A ZPK object if no operand is an FRD or SS object and at least one is an ZPK object
- A TF object only if all operands are TF objects

Operations on systems of different types work as follows: the resulting type is determined by the precedence rules, and all operands are first converted to this type before performing the operation.

Viewing LTI Systems As Matrices

In the frequency domain, an LTI system is represented by the linear input/output map

$$y = Hu$$

This map is characterized by its transfer matrix H, a function of either the Laplace or Z-transform variable. The transfer matrix H maps inputs to outputs, so there are as many columns as inputs and as many rows as outputs.

If you think of LTI systems in terms of (transfer) matrices, certain basic operations on LTI systems are naturally expressed with a matrix-like syntax. For example, the parallel connection of two LTI systems `sys1` and `sys2` can be expressed as

```
sys = sys1 + sys2
```

because parallel connection amounts to adding the transfer matrices. Similarly, subsystems of a given LTI model `sys` can be extracted using matrix-like subscripting. For instance,

```
sys(3,1:2)
```

provides the I/O relation between the first two inputs (column indices) and the third output (row index), which is consistent with

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} H(1,1) & H(2,1) \\ H(2,1) & H(2,2) \\ H(3,1) & H(3,2) \end{bmatrix} \begin{bmatrix} u_1 & u_2 \end{bmatrix}$$

for $y = Hu$.

Command Summary

The next two tables give an overview of the main commands you can apply to LTI models.

Table 1-1: Creating LTI Models

Command	Description
`drss`	Generate random discrete state-space model.
`dss`	Create descriptor state-space model.
`filt`	Create discrete filter with DSP convention.
`frd`	Create an FRD model.
`frdata`	Retrieve FRD model data.
`get`	Query LTI model properties.
`set`	Set LTI model properties.
`rss`	Generate random continuous state-space model.
`ss`	Create a state-space model.

Table 1-1: Creating LTI Models (Continued)

Command	Description
`ssdata`, `dssdata`	Retrieve state-space data (respectively, descriptor state-space data).
`tf`	Create a transfer function.
`tfdata`	Retrieve transfer function data.
`zpk`	Create a zero-pole-gain model.
`zpkdata`	Retrieve zero-pole-gain data.

Table 1-2: Converting LTI Models

Command	Description
`c2d`	Continuous- to discrete-time conversion.
`d2c`	Discrete- to continuous-time conversion.
`d2d`	Resampling of discrete-time models.
`frd`	Conversion to an FRD model.
`pade`	Padé approximation of input delays.
`ss`	Conversion to state space.
`tf`	Conversion to transfer function.
`zpk`	Conversion to zero-pole-gain.

Creating LTI Models

The functions `tf`, `zpk`, `ss`, and `frd` create transfer function models, zero-pole-gain models, state-space models, and frequency response data models, respectively. These functions take the model data as input and produce TF, ZPK, SS, or FRD objects that store this data in a single MATLAB variable. This section shows how to create continuous or discrete, SISO or MIMO LTI models with `tf`, `zpk`, `ss`, and `frd`.

Note You can only specify TF, ZPK, and SS models for systems whose transfer matrices have real-valued coefficients.

Transfer Function Models

This section explains how to specify continuous-time SISO and MIMO transfer function models. The specification of discrete-time transfer function models is a simple extension of the continuous-time case (see "Discrete-Time Models" on page 1-19). In this section you can also read about how to specify transfer functions consisting of pure gains.

SISO Transfer Function Models

A continuous-time SISO transfer function

$$h(s) = \frac{n(s)}{d(s)}$$

is characterized by its numerator $n(s)$ and denominator $d(s)$, both polynomials of the Laplace variable s.

There are two ways to specify SISO transfer functions:

- Using the `tf` command
- As rational expressions in the Laplace variable s

To specify a SISO transfer function model $h(s) = n(s)/d(s)$ using the `tf` command, type

```
h = tf(num,den)
```

where `num` and `den` are row vectors listing the coefficients of the polynomials $n(s)$ and $d(s)$, respectively, when these polynomials are ordered in *descending* powers of s. The resulting variable `h` is a TF object containing the numerator and denominator data.

For example, you can create the transfer function $h(s) = s/(s^2 + 2s + 10)$ by typing

```
h = tf([1 0],[1 2 10])
```

MATLAB responds with

```
Transfer function:
      s
--------------
s^2 + 2 s + 10
```

Note the customized display used for TF objects.

You can also specify transfer functions as rational expressions in the Laplace variable s by:

1 Defining the variable `s` as a special TF model

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

2 Entering your transfer function as a rational expression in `s`

For example, once `s` is defined with `tf` as in **1**,

```
H = s/(s^2 + 2*s +10);
```

produces the same transfer function as

```
h = tf([1 0],[1 2 10]);
```

Note You need only define the variable `s` as a TF model once. All of the subsequent models you create using rational expressions of `s` are specified as TF objects, unless you convert the variable `s` to ZPK. See "Model Conversion" on page 1-39 for more information.

MIMO Transfer Function Models

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

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

Consider the rational transfer matrix

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

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. See "Model Interconnection Functions" on page 2-16 for more details on concatenation operations for LTI systems.

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 Structures and 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

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

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)
```

MATLAB responds with

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

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

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

Pure Gains

You can use `tf` with only one argument to specify simple gains or gain matrices as TF objects. For example,

```
G = tf([1 0;2 1])
```

produces the gain matrix

$$G = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$$

while

```
E = tf
```

creates an empty transfer function.

Zero-Pole-Gain Models

This section explains how to specify continuous-time SISO and MIMO zero-pole-gain models. The specification for discrete-time zero-pole-gain models is a simple extension of the continuous-time case. See "Discrete-Time Models" on page 1-19.

SISO Zero-Pole-Gain Models

Continuous-time SISO zero-pole-gain models are of the form

$$h(s) = k\frac{(s-z_1)\ldots(s-z_m)}{(s-p_1)\ldots(s-p_n)}$$

where k is a real-valued scalar (the *gain*), and $z_1,\ldots,z_m$ and $p_1,\ldots,p_n$ are the real or complex conjugate pairs of zeros and poles of the transfer function $h(s)$. This model is closely related to the transfer function representation: the zeros are simply the numerator roots, and the poles, the denominator roots.

There are two ways to specify SISO zero-pole-gain models:

- Using the `zpk` command
- As rational expressions in the Laplace variable *s*

The syntax to specify ZPK models directly using `zpk` is

```
h = zpk(z,p,k)
```

where `z` and `p` are the vectors of zeros and poles, and `k` is the gain. This produces a ZPK object `h` that encapsulates the `z`, `p`, and `k` data. For example, typing

```
h = zpk(0, [1 i 1+i 2],  2)
```

produces

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

You can also specify zero-pole-gain models as rational expressions in the Laplace variable *s* by:

1 Defining the variable s as a ZPK model

```
s = zpk('s')
```

2 Entering the transfer function as a rational expression in s.

For example, once s is defined with zpk,

```
H =  2s/((s   2)*(s^2 + 2*s + 2));
```

returns the same ZPK model as

```
h = zpk([0], [2  1 i  1+i ],  2);
```

Note You need only define the ZPK variable s once. All subsequent rational expressions of s will be ZPK models, unless you convert the variable s to TF. See "Model Conversion" on page 1-39 for more information on conversion to other model types.

MIMO Zero-Pole-Gain Models

Just as with TF models, you can also specify a MIMO ZPK model by concatenation of its SISO entries (see "Model Interconnection Functions" on page 2-16).

You can also use the command zpk to specify MIMO ZPK models. The syntax to create a p-by-m MIMO zero-pole-gain model using zpk is

```
H = zpk(Z,P,K)
```

where

- Z is the p-by-m cell array of zeros (Z{i,j} = zeros of $H_{ij}(s)$)
- P is the p-by-m cell array of poles (P{i,j} = poles of $H_{ij}(s)$)
- K is the p-by-m matrix of gains (K(i,j) = gain of $H_{ij}(s)$)

For example, typing

```
Z = {[], 5;[1 i 1+i] []};

P = {0,[ 1  1];[1 2 3],[]};
```

```
K = [ 1  3;2  0];

H = zpk(Z,P,K)
```

creates the two-input/two-output zero-pole-gain model

$$H(s) = \begin{bmatrix} \frac{-1}{s} & \frac{3(s+5)}{(s+1)^2} \\ \frac{2(s^2-2s+2)}{(s-1)(s-2)(s-3)} & 0 \end{bmatrix}$$

Notice that you use `[]` as a place-holder in `Z` (or `P`) when the corresponding entry of $H(s)$ has no zeros (or poles).

State-Space Models

State-space models rely on linear differential or difference equations to describe the system dynamics. Continuous-time models are of the form

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

where x is the state vector and u and y are the input and output vectors. Such models may arise from the equations of physics, from state-space identification, or by state-space realization of the system transfer function.

Use the command `ss` to create state-space models

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

For a model with `Nx` states, `Ny` outputs, and `Nu` inputs

- `A` is an `Nx`-by-`Nx` real-valued matrix.
- `B` is an `Nx`-by-`Nu` real-valued matrix.
- `C` is an `Ny`-by-`Nx` real-valued matrix.
- `D` is an `Ny`-by-`Nu` real-valued matrix.

This produces an SS object `sys` that stores the state-space matrices A, B, C, and D. For models with a zero D matrix, you can use `D = 0` (zero) as a shorthand for a zero matrix of the appropriate dimensions.

As an illustration, consider the following simple model of an electric motor.

$$\frac{d^2\theta}{dt^2} + 2\frac{d\theta}{dt} + 5\theta = 3I$$

where θ is the angular displacement of the rotor and I the driving current. The relation between the input current $u = I$ and the angular velocity $y = d\theta/dt$ is described by the state-space equations

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

where

$$x = \begin{bmatrix} \theta \\ \frac{d\theta}{dt} \end{bmatrix} \qquad A = \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ 3 \end{bmatrix} \qquad C = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

This model is specified by typing

```
sys = ss([0 1; 5  2],[0;3],[0 1],0)
```

to which MATLAB responds

```
a =
                    x1           x2
           x1        0      1.00000
           x2  5.00000      2.00000

b =
                    u1
           x1        0
           x2  3.00000

c =
                    x1           x2
           y1        0      1.00000

d =
                    u1
           y1        0
```

In addition to the A, B, C, and D matrices, the display of state-space models includes state names, input names, and output names. Default names (here, `x1`, `x2`, `u1`, and `y1`) are displayed whenever you leave these unspecified. See "LTI Properties" on page 1-25 for more information on how to specify state, input, or output names.

Descriptor State-Space Models

Descriptor state-space (DSS) models are a generalization of the standard state-space models discussed above. They are of the form

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

The Control System Toolbox supports only descriptor systems with a nonsingular E matrix. While such models have an equivalent explicit form

$$\frac{dx}{dt} = (E^{-1}A)x + (E^{-1}B)u$$
$$y = Cx + Du$$

it is often desirable to work with the descriptor form when the E matrix is poorly conditioned with respect to inversion.

The function `dss` is the counterpart of `ss` for descriptor state-space models. Specifically,

```
sys = dss(A,B,C,D,E)
```

creates a continuous-time DSS model with matrix data `A,B,C,D,E`. For example, consider the dynamical model

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

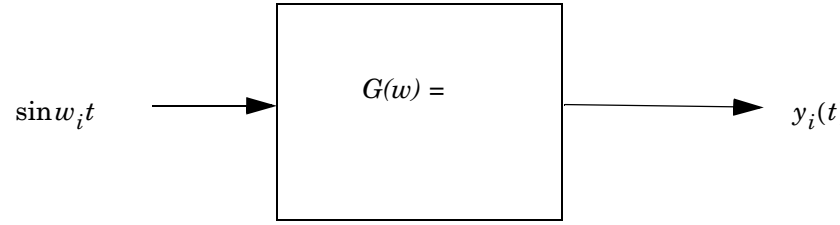
with vector ω of angular velocities. If the inertia matrix J is poorly conditioned with respect to inversion, you can specify this system as a descriptor model by

```
sys = dss( F,eye(n),eye(n),0,J)   % n = length of vector ω
```

Frequency Response Data (FRD) Models

In some instances, you may only have sampled frequency response data, rather than a transfer function or state-space model for the system you want to analyze or control. For information on frequency response analysis of linear systems, see Chapter 8 of [1].

For example, suppose the frequency response function for the SISO system you want to model is *G(w)*. Suppose, in addition, that you perform an experiment to evaluate *G(w)* at a fixed set of frequencies, $w_1, w_2, \ldots, w_n$. You can do this by driving the system with a sequence of sinusoids at each of these frequencies, as depicted below.



Here w_i is the input frequency of each sinusoid, $i = 1 \ldots n$, and $G(w) = |G(w)|\exp(j\angle G(w))$. The steady state output response of this system satisfies

$$y_i(t) = |G(w_i)|\sin(w_i t + \angle G(w_i)); \quad i = 1 \ldots n$$

A *frequency response data* (FRD) *object* is a model form you can use to store frequency response data (complex frequency response, along with a corresponding vector of frequency points) that you obtain either through simulations or experimentally. In this example, the frequency response data is obtained from the set of response pairs: $\{(G(w_i),w_i)\}, i = 1 \ldots n$.

Once you store your data in an FRD model, you can treat it as an LTI model, and manipulate an FRD model in most of the same ways you manipulate TF, SS, and ZPK models.

The basic syntax for creating a SISO FRD model is

```
sys = frd(response,frequencies,units)
```

where

- `frequencies` is a real vector of length `Nf`.

- `response` is a vector of length `Nf` of complex frequency response values for these frequencies.
- `units` is an optional string for the units of frequency: either `'rad/s'` (default) or `'Hz'`

For example, the MAT-file `LTIexamples.mat` contains a frequency vector `freq`, and a corresponding complex frequency response data vector `respG`. To load this frequency-domain data and construct an FRD model, type

```
load LTIexamples
sys = frd(respG,freq)

Continuous-time frequency response with 1 output and 1 input
at 5 frequency points.

From input 1 to:
Frequency(rad/s)        output 1
----------------        --------
        1         0.812505  0.000312i
        2         0.092593  0.462963i
        4         0.075781  0.001625i
        5          0.043735  0.000390i
```

The syntax for creating a MIMO FRD model is the same as for the SISO case, except that `response` is a *p*-by-*m*-by-*Nf* multidimensional array, where *p* is the number of outputs, *m* is the number of inputs, and *Nf* is the number of frequency data points (the length of `frequency`).

The following table summarizes the complex-valued response data format for FRD models.

Table 1-3: Data Format for the Argument response in FRD Models

Model Form	Response Data Format
SISO model	Vector of length `Nf` for which `response(i)` is the frequency response at the frequency `frequency(i)`

Table 1-3: Data Format for the Argument response in FRD Models (Continued)

Model Form	Response Data Format
MIMO model with `Ny` outputs and `Nu` inputs	`Ny`-by-`Nu`-by-`Nf` multidimensional array for which `response(i,j,k)` specifies the frequency response from input `j` to output `i` at frequency `frequency(k)`
`S1`-by-`...`-by-`Sn` array of models with `Ny` outputs and `Nu` inputs	`Ny`-by-`Nu`-by-`S1`-by-`...`-by-`Sn` multidimensional array, for which `response(i,j,k,:)` specifies the array of frequency response data from input `j` to output `i` at frequency `frequency(k)`

Discrete-Time Models

Creating discrete-time models is very much like creating continuous-time models, except that you must also specify a sampling period or *sample time* for discrete-time models. The sample time value should be scalar and expressed in seconds. You can also use the value –1 to leave the sample time unspecified.

To specify discrete-time LTI models using `tf`, `zpk`, `ss`, or `frd`, simply append the desired sample time value `Ts` to the list of inputs.

```
sys1 = tf(num,den,Ts)
sys2 = zpk(z,p,k,Ts)
sys3 = ss(a,b,c,d,Ts)
sys4 = frd(response,frequency,Ts)
```

For example,

```
h = tf([1  1],[1  0.5],0.1)
```

creates the discrete-time transfer function $h(z) = (z-1)/(z-0.5)$ with sample time 0.1 seconds, and

```
sys = ss(A,B,C,D,0.5)
```

specifies the discrete-time state-space model

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

with sampling period 0.5 second. The vectors $x[n], u[n], y[n]$ denote the values of the state, input, and output vectors at the nth sample.

By convention, the sample time of continuous-time models is `Ts = 0`. Setting `Ts = 1` leaves the sample time of a discrete-time model unspecified. For example,

```
h = tf([1  0.2],[1 0.3], 1)
```

produces

```
Transfer function:
z   0.2
-------
z + 0.3

Sampling time: unspecified
```

Note Do not simply omit `Ts` in this case. This would make `h` a continuous-time transfer function.

If you forget to specify the sample time when creating your model, you can still set it to the correct value by reassigning the LTI property `Ts`. See "Sample Time" on page 1-33 for more information on setting this property.

Discrete-Time TF and ZPK Models

You can specify discrete-time TF and ZPK models using `tf` and `zpk` as indicated above. Alternatively, it is often convenient to specify such models by:

1. Defining the variable z as a particular discrete-time TF or ZPK model with the appropriate sample time
2. Entering your TF or ZPK model directly as a rational expression in z.

This approach parallels the procedure for specifying continuous-time TF or ZPK models using rational expressions. This procedure is described in "SISO Transfer Function Models" on page 1-8 and "SISO Zero-Pole-Gain Models" on page 1-12.

For example,

```
z = tf('z', 0.1);
H = (z+2)/(z^2 + 0.6*z + 0.9);
```

creates the same TF model as

```
H = tf([1 2], [1 0.6 0.9], 0.1);
```

Similarly,

```
z = zpk('z', 0.1);
H = [z/(z+0.1)/(z+0.2) ; (z^2+0.2*z+0.1)/(z^2+0.2*z+0.01)]
```

produces the single-input, two-output ZPK model

```
Zero/pole/gain from input to output...
             z
 #1:  ---------------
      (z+0.1) (z+0.2)

      (z^2 + 0.2z + 0.1)
 #2:  ------------------
          (z+0.1)^2

Sampling time: 0.1
```

Note that:

- The syntax `z = tf('z')` is equivalent to `z = tf('z', 1)` and leaves the sample time unspecified. The same applies to `z = zpk('z')`.
- Once you have defined `z` as indicated above, any rational expressions in `z` creates a discrete-time model of the same type and with the same sample time as `z`.

Discrete Transfer Functions in DSP Format

In digital signal processing (DSP), it is customary to write discrete transfer functions as rational expressions in z^{-1} and to order the numerator and denominator coefficients in *ascending powers of* z^{-1}. For example, the numerator and denominator of

$$H(z^{-1}) = \frac{1 + 0.5z^{-1}}{1 + 2z^{-1} + 3z^{-2}}$$

would be specified as the row vectors `[1 0.5]` and `[1 2 3]`, respectively. When the numerator and denominator have different degrees, this convention

clashes with the "*descending powers of z*" convention assumed by `tf` (see "Transfer Function Models" on page 1-8, or `tf`). For example,

```
h = tf([1 0.5],[1 2 3])
```

produces the transfer function

$$\frac{z + 0.5}{z^2 + 2z + 3}$$

which differs from $H(z^{-1})$ by a factor z.

To avoid such convention clashes, the Control System Toolbox offers a separate function `filt` dedicated to the DSP-like specification of transfer functions. Its syntax is

```
h = filt(num,den)
```

for discrete transfer functions with unspecified sample time, and

```
h = filt(num,den,Ts)
```

to further specify the sample time `Ts`. This function creates TF objects just like `tf`, but expects `num` and `den` to list the numerator and denominator coefficients in *ascending powers of* z^{-1}. For example, typing

```
h = filt([1 0.5],[1 2 3])
```

produces

```
Transfer function:
   1 + 0.5 z^ 1
------------------
1 + 2 z^ 1 + 3 z^ 2

Sampling time: unspecified
```

You can also use `filt` to specify MIMO transfer functions in z^{-1}. Just as for `tf`, the input arguments `num` and `den` are then cell arrays of row vectors of appropriate dimensions (see "Transfer Function Models" on page 1-8 for details). Note that each row vector should comply with the "ascending powers of z^{-1}" convention.

Data Retrieval

The functions `tf`, `zpk`, `ss`, and `frd` pack the model data and sample time in a single LTI object. Conversely, the following commands provide convenient data retrieval for any type of TF, SS, or ZPK model `sys`, or FRD model `sysfr`.

```
[num,den,Ts] = tfdata(sys)     % Ts = sample time
[z,p,k,Ts] = zpkdata(sys)
[a,b,c,d,Ts] = ssdata(sys)
[a,b,c,d,e,Ts] = dssdata(sys)
[response,frequency,Ts] = frdata(sysfr)
```

Note that:

- `sys` can be any type of LTI object, *except* an FRD model
- `sysfr`, the input argument to `frdata`, can only be an FRD model

You can use any variable names you want in the output argument list of any of these functions. The ones listed here correspond to the model property names described in Tables 2-2 – 2.5.

The output arguments `num` and `den` assigned to `tfdata`, and `z` and `p` assigned to `zpkdata`, are cell arrays, even in the SISO case. These cell arrays have as many rows as outputs, as many columns as inputs, and their ijth entry specifies the transfer function from the jth input to the ith output. For example,

```
H = [tf([1  1],[1 2 10]) , tf(1,[1 0])]
```

creates the one-output/two-input transfer function

$$H(s) = \left[\frac{s-1}{s^2+2s+10} \quad \frac{1}{s}\right]$$

Typing

```
[num,den] = tfdata(H);
num{1,1}, den{1,1}
```

displays the coefficients of the numerator and denominator of the first input channel.

```
ans =
     0     1     1
```

```
ans =
     1     2    10
```

Note that the same result is obtained using

```
H.num{1,1}, H.den{1,1}
```

See "Direct Property Referencing" on page 1-31 for more information about this syntax.

To obtain the numerator and denominator of SISO systems directly as row vectors, use the syntax

```
[num,den,Ts] = tfdata(sys,'v')
```

For example, typing

```
sys = tf([1 3],[1 2 5]);
[num,den] = tfdata(sys,'v')
```

produces

```
num =

     0     1     3

den =

     1     2     5
```

Similarly,

```
[z,p,k,Ts] = zpkdata(sys,'v')
```

returns the zeros, `z`, and the poles, `p`, as *vectors* for SISO systems.

LTI Properties

The previous section shows how to create LTI objects that encapsulate the model data and sample time. You also have the option to attribute additional information, such as the input names or notes on the model history, to LTI objects. This section gives a complete overview of the *LTI properties*, i.e., the various pieces of information that can be attached to the TF, ZPK, SS, and FRD objects. Type `help ltiprops` for online help on available LTI properties.

From a data structure standpoint, the LTI properties are the various fields in the TF, ZPK, SS, and FRD objects. These fields have names (the *property names*) and are assigned values (the *property values*). We distinguish between *generic properties*, common to all four types of LTI objects, and *model-specific properties* that pertain only to one particular type of model.

Generic Properties

The generic properties are those shared by all four types of LTI models (TF, ZPK, SS, and FRD objects). They are listed in the table below.

Table 1-4: Generic LTI Properties

Property Name	Description	Property Value
`ioDelay`	I/O delay(s)	Matrix
`InputDelay`	Input delay(s)	Vector
`InputGroup`	Input channel groups	Cell array
`InputName`	Input channel names	Cell vector of strings
`Notes`	Notes on the model history	Text
`OutputDelay`	Output delay(s)	Vector
`OutputGroup`	Output channel groups	Cell array
`OutputName`	Output channel names	Cell vector of strings
`Ts`	Sample time	Scalar
`Userdata`	Additional data	Arbitrary

The sample time property `Ts` keeps track of the sample time (in seconds) of discrete-time systems. By convention, `Ts` is `0` (zero) for continuous-time systems, and `Ts` is `-1` for discrete-time systems with unspecified sample time. `Ts` is always a scalar, even for MIMO systems.

The `InputDelay`, `OutputDelay`, and `ioDelay` properties allow you to specify time delays in the input or output channels, or for each input/output pair. Their default value is zero (no delay). See "Time Delays" on page 1-42 for details on modeling delays.

The `InputName` and `OutputName` properties enable you to give names to the *individual* input and output channels. The value of each of these properties is a cell vector of strings with as many cells as inputs or outputs. For example, the `OutputName` property is set to

```
{ 'temperature' ; 'pressure' }
```

for a system with two outputs labeled `temperature` and `pressure`. The default value is a cell of empty strings.

Using the `InputGroup` and `OutputGroup` properties of LTI objects, you can create different groups of input or output channels, and assign names to the groups. For example, you may want to designate the first four inputs of a five-input model as `controls`, and the last input as `noise`. See "Input Groups and Output Groups" on page 1-36 for more information.

Finally, `Notes` and `Userdata` are available to store additional information on the model. The `Notes` property is dedicated to any text you want to supply with your model, while the `Userdata` property can accommodate arbitrary user-supplied data. They are both empty by default.

For more detailed information on how to use LTI properties, see "Additional Insight into LTI Properties" on page 1-32.

Model-Specific Properties

The remaining LTI properties are specific to one of the four model types (TF, ZPK, SS, or FRD). For single LTI models, these are summarized in the

following four tables. The property values differ for LTI arrays. See `set` for more information on these values.

Table 1-5: TF-Specific Properties

Property Name	Description	Property Value
`den`	Denominator(s)	Real cell array of row vectors
`num`	Numerator(s)	Real cell array of row vectors
`Variable`	Transfer function variable	String `'s'`, `'p'`, `'z'`, `'q'`, or `'z^ 1'`

Table 1-6: ZPK-Specific Properties

Property Name	Description	Property Value
`k`	Gains	Two-dimensional real matrix
`p`	Poles	Cell array of column vectors
`Variable`	Transfer function variable	String `'s'`, `'p'`, `'z'`, `'q'`, or `'z^ 1'`
`z`	Zeros	Cell array of column vectors

Table 1-7: SS-Specific Properties

Property Name	Description	Property Value
`a`	State matrix A	2-D real matrix
`b`	Input-to-state matrix B	2-D real matrix
`c`	State-to-output matrix C	2-D real matrix
`d`	Feedthrough matrix D	2-D real matrix
`e`	Descriptor E matrix	2-D real matrix
`Nx`	Number of states	Scalar integer
`StateName`	State names	Cell vector of strings

Table 1-8: FRD-Specific Properties

Property Name	Description	Property Value
`Frequency`	Frequency data points	Real-valued vector
`ResponseData`	Frequency response	Complex-valued multidimensional array
`Units`	Units for frequency	String `'rad/s'` or `'Hz'`

Most of these properties are dedicated to storing the model data. Note that the E matrix is set to `[]` (the empty matrix) for standard state-space models, a storage-efficient shorthand for the true value $E = I$.

The `Variable` property is only an attribute of TF and ZPK objects. This property defines the frequency variable of transfer functions. The default values are `'s'` (Laplace variable s) in continuous time and `'z'` (Z-transform variable z) in discrete time. Alternative choices include `'p'` (equivalent to s) and `'q'` or `'z^ 1'` for the reciprocal $q = z^{-1}$ of the z variable. The influence of the variable choice is mostly limited to the display of TF or ZPK models. One exception is the specification of discrete-time transfer functions with `tf` (see `tf` for details).

Note that `tf` produces the same result as `filt` when the `Variable` property is set to `'z^ 1'` or `'q'`.

Finally, the `StateName` property is analogous to the `InputName` and `OutputName` properties and keeps track of the state names in state-space models.

Setting LTI Properties

There are three ways to specify LTI property values:

- You can set properties when creating LTI models with `tf`, `zpk`, `ss`, or `frd`.
- You can set or modify the properties of an existing LTI model with `set`.
- You can also set property values using structure-like assignments.

This section discusses the first two options. See “Direct Property Referencing” on page 1-31 for details on the third option.

The function `set` for LTI objects follows the same syntax as its Handle Graphics counterpart. Specifically, each property is updated by a pair of arguments

```
PropertyName,PropertyValue
```

where

- *`PropertyName`* is a string specifying the property name. You can type the property name without regard for the case (upper or lower) of the letters in the name. Actually, you need only type any abbreviation of the property name that uniquely identifies the property. For example, `'user'` is sufficient to refer to the `Userdata` property.
- `PropertyValue` is the value to assign to the property (see `set` for details on admissible property values).

As an illustration, consider the following simple SISO model for a heating system with an input delay of 0.3 seconds, an input called “energy,” and an output called “temperature.”

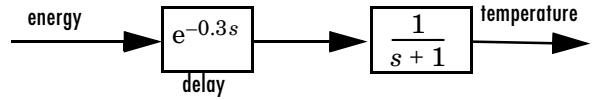


Figure 1-1: A Simple Heater Model

You can use a TF object to represent this delay system, and specify the time delay, the input and output names, and the model history by setting the corresponding LTI properties. You can either set these properties directly when you create the LTI model with `tf`, or by using the `set` command.

For example, you can specify the delay directly when you create the model, and then use the `set` command to assign `InputName`, `OutputName`, and `Notes` to `sys`.

```
sys = tf(1,[1 1],'Inputdelay',0.3);
set(sys,'inputname','energy','outputname','temperature',...
        'notes','A simple heater model')
```

Finally, you can also use the `set` command to obtain a listing of all settable properties for a given LTI model type, along with valid values for these properties. For the transfer function `sys` created above

```
set(sys)
```

produces

```
num:  Ny-by-Nu cell of row vectors (Nu = no. of inputs)
den:  Ny-by-Nu cell of row vectors (Ny = no. of outputs)
Variable:  [ 's' | 'p' | 'z' | 'z^-1' | 'q' ]
Ts:  scalar
InputDelay:  Nu-by-1 vector
OutputDelay:  Ny-by-1 vector
ioDelay:  Ny-by-Nu array (I/O delays)
InputName:  Nu-by-1 cell array of strings
OutputName:  Ny-by-1 cell array of strings
InputGroup:  M-by-2 cell array if M input groups
OutputGroup:  P-by-2 cell array if P output groups
Notes:  array or cell array of strings
UserData:  arbitrary
```

Accessing Property Values Using get

You access the property values of an LTI model `sys` with `get`. The syntax is

```
PropertyValue = get(sys,PropertyName)
```

where the string *`PropertyName`* is either the full property name, or any abbreviation with enough characters to identify the property uniquely. For example, typing

```
h = tf(100,[1 5 100],'inputname','voltage',...
                         'outputn','current',...
                         'notes','A simple circuit')
get(h,'notes')
```

produces

```
ans =

    'A simple circuit'
```

To display all of the properties of an LTI model `sys` (and their values), use the syntax `get(sys)`. In this example,

```
get(h)
```

produces

```
num = {[0 0 100]}
den = {[1 5 100]}
Variable = 's'
Ts = 0
InputDelay = 0
OutputDelay = 0
ioDelay = 0
InputName = {'voltage'}
OutputName = {'current'}
InputGroup = {0x2 cell}
OutputGroup = {0x2 cell}
Notes = {'A simple circuit'}
UserData = []
```

Notice that default (output) values have been assigned to any LTI properties in this list that you have not specified.

Finally, you can also access property values using direct structure-like referencing. This topic is explained in the next section.

Direct Property Referencing

An alternative way to query/modify property values is by structure-like referencing. Recall that LTI objects are basic MATLAB structures except for the additional flag that marks them as TF, ZPK, SS, or FRD objects (see “LTI Objects” on page 1-3). The field names for LTI objects are the property names, so you can retrieve or modify property values with the structure-like syntax.

```
PropertyValue = sys.PropertyName% gets property value
sys.PropertyName = PropertyValue% sets property value
```

These commands are respectively equivalent to

```
PropertyValue = get(sys,'PropertyName')
set(sys,'PropertyName',PropertyValue)
```

For example, type

```
sys = ss(1,2,3,4,'InputName','u');
sys.a
```

and you get the value of the property "a" for the state-space model `sys`.

```
ans =
     1
```

Similarly,

```
sys.a =  1;
```

resets the state transition matrix for `sys` to –1.

Unlike standard MATLAB structures, you do not need to type the entire field name or use upper-case characters. You only need to type the minimum number of characters sufficient to identify the property name uniquely. Thus either of the commands

```
sys.InputName
sys.inputn
```

produces

```
ans =

    'u'
```

Any valid syntax for structures extends to LTI objects. For example, given the TF model $h(p) = 1/p$

```
h = tf(1,[1,0],'variable','p');
```

you can reset the numerator to $p + 2$ by typing

```
h.num{1} = [1 2];
```

or equivalently, with

```
h.num{1}(2) = 2;
```

Additional Insight into LTI Properties

By reading this section, you can learn more about using the `Ts`, `InputName`, `OutputName`, `InputGroup`, and `OutputGroup` LTI properties through a set of examples. For basic information on `Notes` and `Userdata`, see "Generic

Properties" on page 1-25. For detailed information on the use of `InputDelay`, `OutputDelay`, and `ioDelay`, see "Time Delays" on page 1-42.

Sample Time

The sample time property `Ts` is used to specify the sampling period (in seconds) for either discrete-time or discretized continuous-time LTI models. Suppose you want to specify

$$H(z) = \frac{z}{2z^2 + z + 1}$$

as a discrete-time transfer function model with a sampling period of 0.5 seconds. To do this, type

```
h = tf([1 0],[2 1 1],0.5);
```

This sets the `Ts` property to the value 0.5, as is confirmed by

```
h.Ts

ans =
    0.5000
```

For continuous-time models, the sample time property `Ts` is 0 by convention. For example, type

```
h = tf(1,[1 0]);
get(h,'Ts')

ans =
     0
```

To leave the sample time of a discrete-time LTI model unspecified, set `Ts` to –1. For example,

```
h = tf(1,[1  1], 1)
```

produces

```
Transfer function:
  1
-----
z   1
```

```
Sampling time: unspecified
```

The same result is obtained by using the `Variable` property.

```
h = tf(1,[1  1],'var','z')
```

In operations that combine several discrete-time models, all *specified* sample times must be identical, and the resulting discrete-time model inherits this common sample time. The sample time of the resultant model is unspecified if all operands have unspecified sample times. With this inheritance rule for `Ts`, the following two models are equivalent.

```
tf(0.1,[1  1],0.1) + tf(1,[1 0.5], 1)
```

and

```
tf(0.1,[1  1],0.1) + tf(1,[1 0.5],0.1)
```

Note that

```
tf(0.1,[1  1],0.1) + tf(1,[1 0.5],0.5)
```

returns an error message.

```
??? Error using ==> lti/plus
In SYS1+SYS2, both models must have the same sample time.
```

Caution. Resetting the sample time of a continuous-time LTI model `sys` from zero to a nonzero value does *not* discretize the original model `sys`. The command

```
set(sys,'Ts',0.1)
```

only affects the `Ts` property and does not alter the remaining model data. Use `c2d` and `d2c` to perform continuous-to-discrete and discrete-to-continuous conversions. For example, use

```
sysd = c2d(sys,0.1)
```

to discretize a continuous system `sys` at a 10Hz sampling rate.

Use `d2d` to change the sample time of a discrete-time system and resample it.

Input Names and Output Names

You can use the `InputName` and `OutputName` properties (in short, I/O names) to assign names to any or all of the input and output channels in your LTI model.

For example, you can create a SISO model with input `thrust`, output `velocity`, and transfer function $H(p) = 1/(p + 10)$ by typing

```
h = tf(1,[1 10]);
set(h,'inputname','thrust','outputname','velocity',...
                           'variable','p')
```

Equivalently, you can set these properties directly by typing

```
h = tf(1,[1 10],'inputname','thrust',...
                'outputname','velocity',...
                'variable','p')
```

This produces

```
Transfer function from input "thrust" to output "velocity":
  1
------
p + 10
```

Note how the display reflects the input and output names and the variable selection.

In the MIMO case, use cell vectors of strings to specify input or output channel names. For example, type

```
num = {3 , [1 2]};
den = {[1 10] , [1 0]};
H = tf(num,den);        % H(s) has one output and two inputs

set(H,'inputname',{'temperature' ; 'pressure'})
```

The specified input names appear in the display of `H`.

```
Transfer function from input "temperature" to output:
  3
------
s + 10

Transfer function from input "pressure" to output:
s + 2
-----
  s
```

To leave certain names undefined, use the empty string `''` as in

```
H = tf(num,den,'inputname',{ 'temperature' ; '' })
```

Input Groups and Output Groups

In many applications, you may want to create several (distinct or intersecting) groups of input or output channels and name these groups. For example, you may want to label one set of input channels as `noise` and another set as `controls`.

To see how input and output groups (I/O groups) work:

1. Create a random state-space model with one state, three inputs, and three outputs.
2. Assign the first two inputs to a group named `controls`, the first output to a group named `temperature`, and the last two outputs to a group named `measurements`.

To do this, type

```
h = rss(1,3,3);
set(h, 'InputGroup',{[1 2] 'controls'})
set(h, 'OutputGroup', {[1] 'temperature'; [2 3] 'measurements'})
h
```

and MATLAB returns a state-space model of the following form.

```
a =
                    x1
           x1      0.64884


b =
                    u1          u2          u3
           x1      0.12533          0          0


c =
                    x1
           y1       1.1909
           y2       1.1892
```

```
           y3            0


d =
                      u1           u2           u3
           y1      0.32729            0       0.1364
           y2            0            0            0
           y3            0       2.1832            0

I/O Groups:
   Group Name     I/O    Channel(s)
    controls       I        1,2
  temperature      0         1
  measurements     0        2,3

Continuous-time model.
```

Notice that the middle column of the I/O group listing indicates whether the group is an input group (`I`) or an output group (`O`).

In general, to specify M input groups (or output groups), you need an M-by-2 cell array organized as follows.

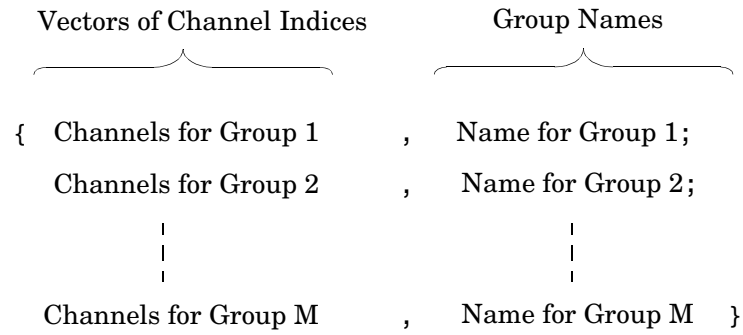


Figure 1-2: Two Column Cell Array

When you specify the cell array for input (or output) groups, keep in mind:

- Each row of this cell array designates a different input (output) group.

- You can add input (or output) groups by appending rows to the cell array.
- You can choose not to assign any of the group names when you assign the groups, and leave off the second column of this array. In that case,
  - Empty strings are assigned to the group names by default.
  - If you append rows to a cell array with no group names assigned, you have to assign empty strings (`''`) to the group names.

For example,

```
h.InputGroup = [h.InputGroup; {[3] 'disturbance'}];
```

adds another input group called `disturbance` to `h`.

You can use regular cell array syntax for accessing or modifying I/O group components. For example, to delete the first output group, `temperature`, type

```
h.OutputGroup(1,:) = []

ans =
    [1x2 double]    'measurements'
```

Similarly, you can add or delete channels from an existing input or output group. Recalling that input group channels are stored in the first column of the corresponding cell array, to add channel three to the input group `controls`, type

```
h.inputgroup{1,1} = [h.inputgroup{1,1} 3]
```

or, equivalently,

```
h.inputgroup{1,1} = [1 2 3]
```

Model Conversion

There are four LTI model types you can use with the Control System Toolbox: TF, ZPK, SS, and FRD. This section shows how to convert models from one type to the other.

Explicit Conversion

Model conversions are performed by `tf`, `ss`, `zpk`, and `frd`. Given any TF, SS, or ZPK model `sys`, the syntax for conversion to another model type is

```
sys = tf(sys)                    % Conversion to TF
sys = zpk(sys)                   % Conversion to ZPK
sys = ss(sys)                    % Conversion to SS
sys = frd(sys,frequency)         % Conversion to FRD
```

Notice that FRD models can't be converted to the other model types. In addition, you must also include a vector of frequencies (`frequency`) as an input argument when converting to an FRD model.

For example, you can convert the state-space model

```
sys = ss( 2,1,1,3)
```

to a zero-pole-gain model by typing

```
zpk(sys)
```

to which MATLAB responds

```
Zero/pole/gain:
3 (s+2.333)
-----------
   (s+2)
```

Note that the transfer function of a state-space model with data (A, B, C, D) is

$$H(s) = D + C(sI - A)^{-1}B$$

for continuous-time models, and

$$H(z) = D + C(zI - A)^{-1}B$$

for discrete-time models.

Automatic Conversion

Some algorithms operate only on one type of LTI model. For example, the algorithm for zero-order-hold discretization with `c2d` can only be performed on state-space models. Similarly, commands like `tfdata` expect one particular type of LTI models (TF). For convenience, such commands automatically convert LTI models to the appropriate or required model type. For example, in

```
sys = ss(0,1,1,0)
[num,den] = tfdata(sys)
```

`tfdata` first converts the state-space model `sys` to an equivalent transfer function in order to return numerator and denominator data.

Note that conversions to state-space models are not uniquely defined. For this reason, automatic conversions to state space are disabled when the result depends on the choice of state coordinates, for example, in commands like `initial` or `kalman`.

Caution About Model Conversions

When manipulating or converting LTI models, keep in mind that:

- The three LTI model types TF, ZPK, and SS, are not equally well-suited for numerical computations. In particular, the accuracy of computations using high-order transfer functions is often poor. Therefore, it is often preferable to work with the state-space representation. In addition, it is often beneficial to balance and scale state-space models using `ssbal`. You get this type of balancing automatically when you convert any TF or ZPK model to state space using `ss`.
- Conversions to the transfer function representation using `tf` may incur a loss of accuracy. As a result, the transfer function poles may noticeably differ from the poles of the original zero-pole-gain or state-space model.
- Conversions to state space are not uniquely defined in the SISO case, nor are they guaranteed to produce a minimal realization in the MIMO case. For a given state-space model `sys`,

```
ss(tf(sys))
```

may return a model with different state-space matrices, or even a different number of states in the MIMO case. Therefore, if possible, it is best to avoid converting back and forth between state-space and other model types.

Time Delays

Using the `ioDelay`, `InputDelay`, and `OutputDelay` properties of LTI objects, you can specify delays in both continuous- and discrete-time LTI models. With these properties, you can, for example, represent:

- LTI models with independent delays for each input/output pair. For example, the continuous-time model with transfer function

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

- State-space models with delayed inputs and/or delayed outputs. For example,

$$\dot{x}(t) = Ax(t) + Bu(t-\tau)$$
$$y(t) = Cx(t-\theta) + Du(t-(\theta+\tau))$$

  where τ is the time delay between the input $u(t)$ and the state vector $x(t)$, and θ is the time delay between $x(t)$ and the output $y(t)$.

You can assign the delay properties `ioDelay`, `InputDelay`, and `OutputDelay` either when first creating your model with the `tf`, `zpk`, `ss`, or `frd` constructors, or later with the `set` command (see "LTI Properties and Methods" on page 1-4 for details).

Supported Functionality

Most analysis commands support time delays, including:

- All time and frequency response commands
- Conversions between model types
- Continuous-to-discrete conversions (`c2d`)
- Horizontal and vertical concatenation
- Series, parallel, and feedback interconnections of discrete-time models with delays

- Interconnections of continuous-time delay systems as long as the resulting transfer function from input j to output i is of the form $\exp(-s\tau_{ij})\ h_{ij}(s)$ where $h_{ij}(s)$ is a rational function of s
- Padé approximation of time delays (`pade`)

Specifying Input/Output Delays

Using the `ioDelay` property, you can specify frequency-domain models with independent delays in each entry of the transfer function. In continuous time, such models have a transfer function of the form

$$H(s) = \begin{bmatrix} e^{-s\tau_{11}}h_{11}(s) & \dots & e^{-s\tau_{1m}}h_{1m}(s) \\ \vdots & & \vdots \\ e^{-s\tau_{p1}}h_{p1}(s) & \dots & e^{-s\tau_{pm}}h_{pm}(s) \end{bmatrix} = [\exp(-s\tau_{ij})\ h_{ij}(s)]$$

where the h_{ij}'s are rational functions of s, and τ_{ij} is the time delay between input j and output i. See "Specifying Delays in Discrete-Time Models" on page 1-49 for details on the discrete-time counterpart. We collectively refer to the scalars τ_{ij} as the *I/O delays*.

The syntax to create $H(s)$ above is

```
H = tf(num,den,'ioDelay',Tau)
```

or

```
H = zpk(z,p,k,'ioDelay',Tau)
```

where

- `num`, `den` (respectively, `z`, `p`, `k`) specify the rational part $[h_{ij}(s)]$ of the transfer function $H(s)$
- `Tau` is the matrix of time delays for each I/O pair. That is, `Tau(i,j)` specifies the I/O delay τ_{ij} in seconds. Note that `Tau` and $H(s)$ should have the same row and column dimensions.

You can also use the `ioDelay` property in conjunction with state-space models, as in

```
sys = ss(A,B,C,D,'ioDelay',Tau)
```

This creates the LTI model with the following transfer function.

$$H(s) = \left[\exp(-s\tau_{ij})\, r_{ij}(s)\right]$$

Here $r_{ij}(s)$ is the (i,j) entry of

$R(s) = D + C(sI - A)^{-1}B$

Note State-space models with I/O delays have only a frequency-domain interpretation. They cannot, in general, be described by state-space equations with delayed inputs and outputs.

Distillation Column Example

This example is adapted from [2] and illustrates the use of I/O delays in process modeling. The process of interest is the distillation column depicted by the figure below. This column is used to separate a mix of methanol and water (the *feed*) into *bottom products* (mostly water) and a methanol-saturated *distillate*.

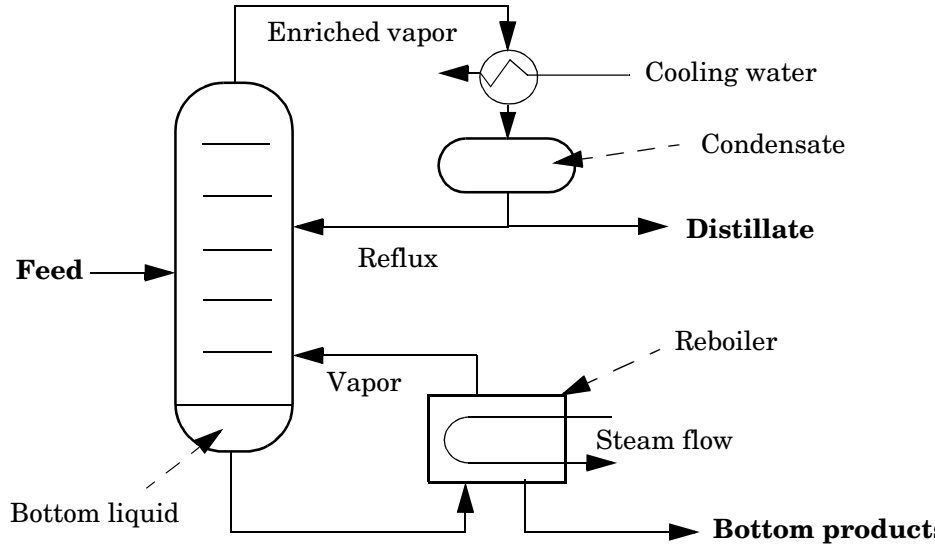


Figure 1-3: Distillation Column

Schematically, the distillation process functions as follows:

- Steam flows into the reboiler and vaporizes the bottom liquid. This vapor is reinjected into the column and mixes with the feed
- Methanol, being more volatile than water, tends to concentrate in the vapor moving upward. Meanwhile, water tends to flow downward and accumulate as the bottom liquid
- The vapor exiting at the top of the column is condensed by a flow of cooling water. Part of this condensed vapor is extracted as the distillate, and the rest of the condensate (the *reflux*) is sent back to the column.
- Part of the bottom liquid is collected from the reboiler as bottom products (waste).

The regulated output variables are:

- Percentage X_D of methanol in the distillate
- Percentage X_B of methanol in the bottom products.

The goal is to maximize X_D by adjusting the reflux flow rate R and the steam flow rate S in the reboiler.

To obtain a linearized model around the steady-state operating conditions, the transient responses to pulses in steam and reflux flow are fitted by first-order plus delay models. The resulting transfer function model is

$$\begin{bmatrix} X_D(s) \\ X_B(s) \end{bmatrix} = \begin{bmatrix} \frac{12.8e^{-1s}}{16.7e+1} & \frac{-18.9e^{-3s}}{21.0s+1} \\ \frac{6.6e^{-7s}}{10.9s+1} & \frac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix} \begin{bmatrix} R(s) \\ S(s) \end{bmatrix}$$

Note the different time delays for each input/output pair.

You can specify this MIMO transfer function by typing

```
H = tf({12.8  18.9;6.6  19.4},...
       {[16.7 1] [21 1];[10.9 1] [14.4 1]},...
       'iodelay',[1 3;7 3],...
       'inputname',{'R' , 'S'},...
       'outputname',{'Xd' , 'Xb'})
```

The resulting TF model is displayed as

```
Transfer function from input "R" to output...
                    12.8
 Xd:  exp( 1*s) * ----------
                  16.7 s + 1

                     6.6
 Xb:  exp( 7*s) * ----------
                  10.9 s + 1

Transfer function from input "S" to output...
                     18.9
 Xd:  exp( 3*s) * --------
                  21 s + 1

                     19.4
 Xb:  exp( 3*s) * ----------
                  14.4 s + 1
```

Specifying Delays on the Inputs or Outputs

While ideal for frequency-domain models with I/O delays, the `ioDelay` property is inadequate to capture delayed inputs or outputs in state-space models. For example, the two models

$$(M_1)\begin{cases}\dot{x}(t) = -x(t) + u(t-0.1)\\ y(t) = x(t)\end{cases} \qquad (M_2)\begin{cases}\dot{z}(t) = -z(t) + u(t)\\ y(t) = z(t-0.1)\end{cases}$$

share the same transfer function

$$h(s) = \frac{e^{-0.1s}}{s+1}$$

As a result, they cannot be distinguished using the `ioDelay` property (the I/O delay value is 0.1 seconds in both cases). Yet, these two models have different state trajectories since $x(t)$ and $z(t)$ are related by

$$z(t) = x(t-0.1)$$

Note that the 0.1 second delay is on the *input* in the first model, and on the *output* in the second model.

InputDelay and OutputDelay Properties

When the state trajectory is of interest, you should use the `InputDelay and OutputDelay` properties to distinguish between delays on the inputs and delays on the outputs in state-space models. For example, you can accurately specify the two models above by

```
M1 = ss( 1,1,1,0,'inputdelay',0.1)
M2 = ss( 1,1,1,0,'outputdelay',0.1)
```

In the MIMO case, you can specify a different delay for each input (or output) channel by assigning a vector value to `InputDelay` (or `OutputDelay`). For example,

```
sys = ss(A,[B1 B2],[C1;C2],[D11 D12;D21 D22])
sys.inputdelay = [0.1 0]
sys.outputdelay = [0.2 0.3]
```

creates the two-input, two-output model

$$\dot{x}(t) = Ax(t) + B_1u_1(t-0.1) \; + B_2u_2(t)$$
$$y_1(t+0.2) = C_1x(t) + D_{11}u_1(t-0.1) + D_{12}u_2(t)$$
$$y_2(t+0.3) = C_2x(t) + D_{21}u_1(t-0.1) + D_{22}u_2(t)$$

You can also use the `InputDelay` and `OutputDelay` properties to conveniently specify input or output delays in TF, ZPK, or FRD models. For example, you can create the transfer function

$$H(s) = \begin{bmatrix} \frac{1}{s} \\ \frac{2}{s+1} \end{bmatrix} e^{-0.1s}$$

by typing

```
s = tf('s');
H = [1/s ; 2/(s+1)];   % rational part
H.inputdelay = 0.1
```

The resulting model is displayed as

```
Transfer function from input to output...
                    1
 #1:  exp( 0.1*s) * -
                    s

                      2
 #2:  exp( 0.1*s) * -----
                    s + 1
```

By comparison, to produce an equivalent transfer function using the `ioDelay` property, you would need to type

```
H = [1/s ; 2/(s+1)];
H.iodelay = [0.1 ; 0.1];
```

Notice that the 0.1 second delay is repeated twice in the I/O delay matrix. More generally, for a TF, ZPK, or FRD model with input delays $[\alpha_1, ..., \alpha_m]$ and output delays $[\beta_1, ..., \beta_p]$, the equivalent I/O delay matrix is

$$\begin{bmatrix} \alpha_1+\beta_1 & \alpha_2+\beta_1 & \dots & \alpha_m+\beta_1 \\ \alpha_1+\beta_2 & \alpha_2+\beta_2 & & \alpha_m+\beta_2 \\ \vdots & \vdots & & \vdots \\ \alpha_1+\beta_p & \alpha_2+\beta_p & \dots & \alpha_m+\beta_p \end{bmatrix}$$

Specifying Delays in Discrete-Time Models

You can also use the `ioDelay`, `InputDelay`, and `OutputDelay` properties to specify delays in discrete-time LTI models. You specify time delays in discrete-time models with integer multiples of the sampling period. The integer `k` you supply for the time delay of a discrete-time model specifies a time delay of k sampling periods. Such a delay contributes a factor z^{-k} to the transfer function.

For example,

```
h = tf(1,[1 0.5 0.2],0.1,'inputdelay',3)
```

produces the discrete-time transfer function

```
Transfer function:
                1
z^( 3) * -----------------
         z^2 + 0.5 z + 0.2

Sampling time: 0.1
```

Notice the `z^( 3)` factor reflecting the three-sampling-period delay on the input.

Mapping Discrete-Time Delays to Poles at the Origin

Since discrete-time delays are equivalent to additional poles at $z = 0$, they can be easily absorbed into the transfer function denominator or the state-space equations. For example, the transfer function of the delayed integrator

$$y[k+1] = y[k] + u[k-2]$$

is

$$H(z) = \frac{z^{-2}}{z-1}$$

You can specify this model either as the first-order transfer function $1/(z-1)$ with a delay of two sampling periods on the input

```
Ts = 1;   % sampling period
H1 = tf(1,[1  1],Ts,'inputdelay',2)
```

or directly as a third-order transfer function:

```
H2 = tf(1,[1  1 0 0],Ts)   % 1/(z^3 z^2)
```

While these two models are mathematically equivalent, `H1` is a more efficient representation both in terms of storage and subsequent computations.

When necessary, you can map all discrete-time delays to poles at the origin using the command `delay2z`. For example,

```
H2 = delay2z(H1)
```

absorbs the input delay in `H1` into the transfer function denominator to produce the third-order transfer function

```
Transfer function:
    1
---------
z^3   z^2

Sampling time: 1
```

Note that

```
H2.inputdelay
```

now returns `0` (zero).

Retrieving Information About Delays

There are several ways to retrieve time delay information from a given LTI model `sys`:

- Use property display commands to inspect the values of the `ioDelay`, `InputDelay`, and `OutputDelay` properties. For example,

```
sys.iodelay
get(sys,'inputdelay')
```

- Use the helper function `hasdelay` to determine if `sys` has any delay at all. The syntax is

  ```
  hasdelay(sys)
  ```

  which returns `1` (true) if `sys` has any delay, and `0` (false) otherwise
- Use the function `totaldelay` to determine the total delay between each input and each output (cumulative contribution of the `ioDelay`, `InputDelay`, and `OutputDelay` properties). Type `help totaldelay` or see the Reference pages for details.

Padé Approximation of Time Delays

The function `pade` computes rational approximations of time delays in continuous-time LTI models. The syntax is

```
sysx = pade(sys,n)
```

where `sys` is a continuous-time model with delays, and the integer `n` specifies the Padé approximation order. The resulting LTI model `sysx` is of the same type as `sys`, but is delay free.

For models with multiple delays or a mix of input, output, and I/O delays, you can use the syntax

```
sysx = pade(sys,ni,no,nio)
```

where the vectors `ni` and `no`, and the matrix `nio` specify independent approximation orders for each input, output, and I/O delay, respectively. Set `ni=[]` if there are no input delays, and similarly for `no` and `nio`.

For example, consider the “Distillation Column Example” on page 1-44. The two-input, two-output transfer function in this example is

$$H(s) = \begin{bmatrix} \dfrac{12.8e^{-1s}}{16.7e+1} & \dfrac{-18.9e^{-3s}}{21.0s+1} \\ \dfrac{6.6e^{-7s}}{10.9s+1} & \dfrac{-19.4e^{-3s}}{14.4s+1} \end{bmatrix}$$

To compute a Padé approximation of $H(s)$ using:

- A first-order approximation for the 1 second and 3 second delays
- A second-order approximation for the 7 second delay,

type

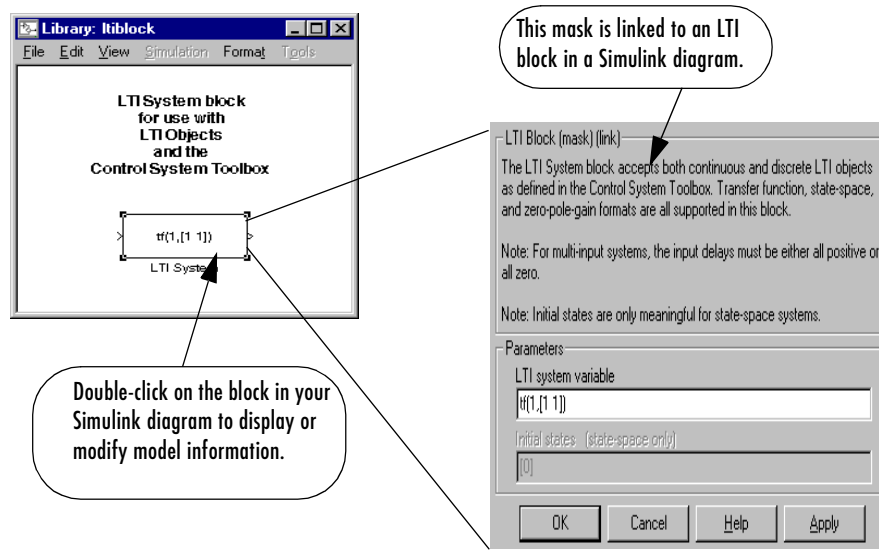
```
pade(H,[],[],[1 1;2 1])
```

where `H` is the TF representation of $H(s)$ defined in the distillation column example. This command produces a rational transfer function.

```
Transfer function from input "R" to output...
          12.8 s + 25.6
 Xd:  ---------------------
      16.7 s^2 + 34.4 s + 2

              6.6 s^2   5.657 s + 1.616
 Xb:  ---------------------------------------
      10.9 s^3 + 10.34 s^2 + 3.527 s + 0.2449

Transfer function from input "S" to output...
           18.9 s   12.6
 Xd:  ----------------------
      21 s^2 + 15 s + 0.6667

             19.4 s   12.93
 Xb:  -------------------------
      14.4 s^2 + 10.6 s + 0.6667
```

Simulink Block for LTI Systems

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



The LTI System block can be accessed either by typing

```
ltiblock
```

at the MATLAB prompt or by selecting **Control System Toolbox** from the **Blocksets and Toolboxes** 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. 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.

Note:

- For MIMO systems, the input delays stored in the LTI object must be either all positive or all zero.
- LTI blocks in a Simulink diagram cannot be used for FRD models or LTI arrays.

References

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

[2] Wood, R.K. and M.W. Berry, "Terminal Composition Control of a Binary Distillation Column," *Chemical Engineering Science*, 28 (1973), pp. 1707-1717.

2

Operations on LTI Models

You can perform basic matrix operations such as addition, multiplication, or concatenation on LTI models. Such operations are “overloaded,” which means that they use the same syntax as they do for matrices, but are adapted so as to apply to the LTI model context. These overloaded operations and their interpretation in this context are discussed in this chapter. You can read about discretization methods in this chapter as well. The following topics and operations on LTI models are covered in this chapter:

- Precedence and Property Inheritance
- Extracting and Modifying Subsystems
- Arithmetic Operations
- Model Interconnection Functions
- Continuous/Discrete-Time Conversions of LTI Models
- Resampling of Discrete-Time Models

These operations can be applied to LTI models of different types. As a result, before discussing operations on LTI models, we discuss model type precedence and how LTI model properties are inherited when models are combined using these operations. To read about how you can apply these operations to arrays of LTI models, see “Operations on LTI Arrays” on page 4-24. To read about the available functions with which you can analyze LTI models, see Chapter 5, “Model Analysis Tools,”

Precedence and Property Inheritance

You can apply operations to LTI models of different types. The resulting type is then determined by the rules discussed in "Precedence Rules" on page 2-5. For example, if `sys1` is a transfer function and `sys2` is a state-space model, then the result of their addition

```
sys = sys1 + sys2
```

is a state-space model, since state-space models have precedence over transfer function models.

To supersede the precedence rules and force the result of an operation to be a given type, for example, a transfer function (TF), you can either

- Convert all operands to TF *before* performing the operation
- Convert the result to TF *after* performing the operation

Suppose, in the above example, you want to compute the transfer function of `sys`. You can either use *a priori* conversion of the second operand

```
sys = sys1 + tf(sys2);
```

or *a posteriori* conversion of the result

```
sys = tf(sys1 + sys2)
```

Note These alternatives are not equivalent numerically; computations are carried out on transfer functions in the first case, and on state-space models in the second case.

Another issue is property inheritance, that is, how the operand property values are passed on to the result of the operation. While inheritance is partly operation-dependent, some general rules are summarized below:

- In operations combining discrete-time LTI models, all models must have identical or unspecified (`sys.Ts = 1`) sample times. Models resulting from such operations inherit the specified sample time, if there is one.
- Most operations ignore the `Notes` and `Userdata` properties.

- In general, when two LTI models `sys1` and `sys2` are combined using operations such as `+`, `*`, `[,]`, `[;]`, `append`, and `feedback`, the resulting model inherits its I/O names and I/O groups from `sys1` and `sys2`. 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.
- 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, `'z^ 1'` has precedence over `'q'` and `'z'`, while `'q'` has precedence over `'z'`.

Extracting and Modifying Subsystems

Subsystems relate subsets of the inputs and outputs of a system. The transfer matrix of a subsystem is a submatrix of the system transfer matrix. For example, if `sys` is a system with two inputs, three outputs, and I/O relation

$$y = Hu$$

then $H(3, 1)$ gives the relation between the first input and third output.

$$y_3 = H(3,1)\, u_1$$

Accordingly, use matrix-like subindexing to extract this subsystem.

```
SubSys = sys(3,1)
```

The resulting subsystem `SubSys` is an LTI model of the same type as `sys`, with its sample time, time delay, I/O name, and I/O group property values inherited from `sys`.

For example, if `sys` has an input group named `controls` consisting of channels one, two, and three, then `SubSys` also has an input group named `controls` with the first channel of `SubSys` assigned to it.

If `sys` is a state-space model with matrices `a`, `b`, `c`, `d`, the subsystem `sys(3,1)` is a state-space model with data `a`, `b(:,1)`, `c(3,:)`, `d(3,1)`. Note the following rules when extracting subystems:

- In the expression `sys(3,1)`, the first index selects the output channel while the second index selects the input channel.
- When extracting a subsystem from a given state-space model, the resulting state-space model may not be minimal. Use the command `sminreal` to eliminate unnecessary states in the subsystem.

You can use similar syntax to modify the LTI model `sys`. For example,

```
sys(3,1) = NewSubSys
```

redefines the I/O relation between the first input and third output, provided `NewSubSys` is a SISO LTI model.

The following rules apply when modifying LTI models:

- `sys`, the LTI model that has had a portion reassigned, retains its original model type (TF, ZPK, SS, or FRD) regardless of the model type of `NewSubSys`.
- Subsystem assignment does not reassign any I/O names or I/O group names of `NewSubSys` that are already assigned to `NewSubSys`.
- Reassigning parts of a MIMO state-space model generally increases its order.
- If `NewSubSys` is an FRD model, then `sys` must also be an FRD model. Furthermore, their frequencies must match.

Other standard matrix subindexing extends to LTI objects as well. For example,

```
sys(3,1:2)
```

extracts the subsystem mapping the first two inputs to the third output.

```
sys(:,1)
```

selects the first input and all outputs, and

```
sys([1 3],:)
```

extracts a subsystem with the same inputs, but only the first and third outputs.

For example, consider the two-input/two-output transfer function

$$.T(s) = \begin{bmatrix} \dfrac{1}{s+0.1} & 0 \\ \dfrac{s-1}{s^2+2s+2} & \dfrac{1}{s} \end{bmatrix}$$

To extract the transfer function $T_{11}(s)$ from the first input to the first output, type

```
T(1,1)

Transfer function:
   1
-------
s + 0.1
```

Next reassign $T_{11}(s)$ to $1/(s+0.5)$ and modify the second input channel of `T` by typing

```
T(1,1) = tf(1,[1 0.5]);
T(:,2) = [ 1 ; tf(0.4,[1 0]) ]
```

```
Transfer function from input 1 to output...
         1
 #1:  -------
      s + 0.5

          s   1
 #2:  -------------
      s^2 + 2 s + 2

Transfer function from input 2 to output...
 #1:  1

      0.4
 #2:  ---
       s
```

Referencing FRD Models Through Frequencies

You can extract subsystems from FRD models, as you do with other LTI model types, by indexing into input and output (I/O) dimensions. You can also extract subsystems by indexing into the frequencies of an FRD model.

To index into the frequencies of an FRD model, use the string `'Frequency'` (or any abbreviation, such as, `'freq'`, as long as it does not conflict with existing I/O channel or group names) as a keyword. There are two ways you can specify FRD models using frequencies:

- Using integers to index into the frequency vector of the FRD model
- Using a Boolean (logical) expression to specify desired frequency points in an FRD model

For example, if `sys` is an FRD model with five frequencies, (e.g., `sys.Frequency=[1 1.1 1.2 1.3 1.4]`), then you can create a new FRD model `sys2` by indexing into the frequencies of `sys` as follows.

```
sys2 = sys('frequency', 2:3);
```

```
sys2.Frequency

ans =
    1.1000
    1.2000
```

displays the second and third entries in the frequency vector.

Similarly, you can use logical indexing into the frequencies.

```
sys2 = sys('frequency',sys.Frequency >1.0 & sys.Frequency <1.15);
sys2.freq

ans =
    1.1000
```

You can also combine model extraction through frequencies with indexing into the I/O dimensions. For example, if `sys` is an FRD model with two inputs, two outputs, and frequency vector `[2.1 4.2 5.3]`, with `sys.Units` specified in rad/s, then

```
sys2 = sys(1,2,'freq',1)
```

specifies `sys2` as a SISO FRD model, with one frequency data point, 2.1 rad/s.

Referencing Channels by Name

You can also extract subsystems using I/O group or channel names. For example, if `sys` has an input group named `noise`, consisting of channels two, four, and five, then

```
sys(1,'noise')
```

is equivalent to

```
sys(1,[2 4 5])
```

Similarly, if `pressure` is the name assigned to an output channel of the LTI model `sys`, then

```
sys('pressure',1) = tf(1, [1 1])
```

reassigns the subsystem from the first input of `sys` to the output labeled `pressure`.

You can reference a set of channels by input or output name by using a cell array of strings for the names. For example, if `sys` has one output channel named `pressure` and one named `temperature`, then these two output channels can be referenced using

```
sys({'pressure','temperature'})
```

Resizing LTI Systems

Resizing a system consists of adding or deleting inputs and/or outputs. To delete the first two inputs, simply type

```
sys(:,1:2) = []
```

In deletions, at least one of the row/column indexes should be the colon (`:`) selector.

To perform input/output augmentation, you can proceed by concatenation or subassignment. Given a system `sys` with a single input, you can add a second input using

```
sys = [sys,h];
```

or, equivalently, using

```
sys(:,2) = h;
```

where `h` is any LTI model with one input, and the same number of outputs as `sys`. There is an important difference between these two options: while concatenation obeys the precedence rules (see page 2-5), subsystem assignment does not alter the model type. So, if `sys` and `h` are TF and SS objects, respectively, the first statement produces a state-space model, and the second statement produces a transfer function.

For state-space models, both concatenation and subsystem assignment increase the model order because they assume that `sys` and `h` have independent states. If you intend to keep the same state matrix and only update the input-to-state or state-to-output relations, use `set` instead and modify the corresponding state-space data directly. For example,

```
sys = ss(a,b1,c,d1)
set(sys,'b',[b1 b2],'d',[d1 d2])
```

adds a second input to the state-space model `sys` by appending the *B* and *D* matrices. You should *simultaneously* modify both matrices with a single `set` command. Indeed, the statements

```
sys.b = [b1 b2]
```

and

```
set(sys,'b',[b1 b2])
```

cause an error because they create invalid intermediate models in which the *B* and *D* matrices have inconsistent column dimensions.

Arithmetic Operations

You can apply almost all arithmetic operations to LTI models, including those shown below.

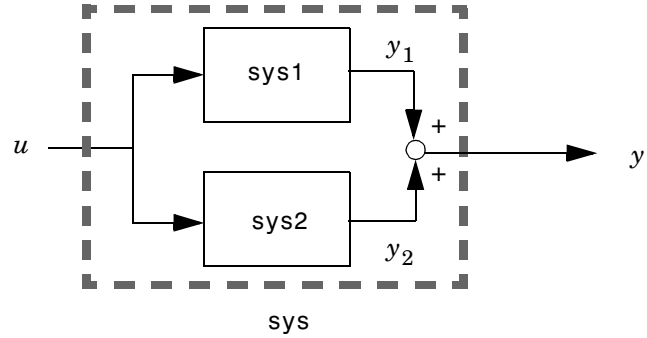
Operation	Description
`+`	Addition
–	Subtraction
`*`	Multiplication
`/`	Right matrix divide
`\`	Left matrix divide
`inv`	Matrix inversion
`'`	Pertransposition
`.'`	Transposition
`^`	Powers of an LTI model (as in `s^2`)

Addition and Subtraction

Adding LTI models is equivalent to connecting them in parallel. Specifically, the LTI model

```
sys = sys1 + sys2
```

represents the parallel interconnection shown below.



If `sys1` and `sys2` are two state-space models with data A_1, B_1, C_1, D_1 and A_2, B_2, C_2, D_2, the state-space data associated with `sys1 + sys2` is

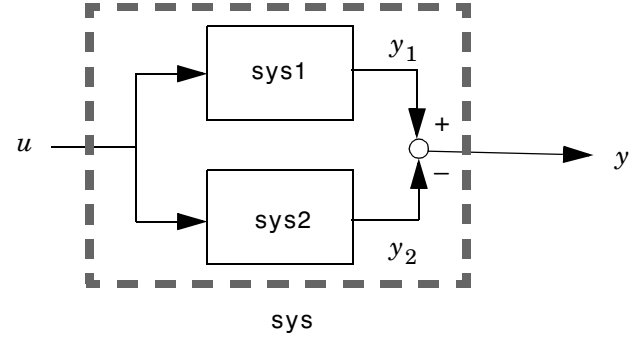
$$\begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix}, \qquad \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \qquad \begin{bmatrix} C_1 & C_2 \end{bmatrix}, \qquad D_1 + D_2$$

Scalar addition is also supported and behaves as follows: if `sys1` is MIMO and `sys2` is SISO, `sys1 + sys2` produces a system with the same dimensions as `sys1` whose *ij*th entry is `sys1(i,j) + sys2`.

Similarly, the subtraction of two LTI models

```
sys = sys1   sys2
```

is depicted by the following block diagram.

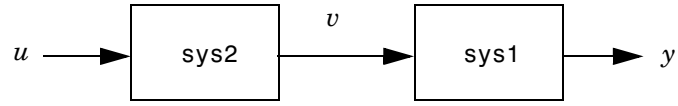


Multiplication

Multiplication of two LTI models connects them in series. Specifically,

```
sys = sys1 * sys2
```

returns an LTI model sys for the series interconnection shown below.



Notice the reverse orders of sys1 and sys2 in the multiplication and block diagram. This is consistent with the way transfer matrices are combined in a series connection: if sys1 and sys2 have transfer matrices H_1 and H_2, then

$$y = H_1 v = H_1(H_2 u) = (H_1 \times H_2)\, u$$

For state-space models sys1 and sys2 with data A_1, B_1, C_1, D_1 and A_2, B_2, C_2, D_2, the state-space data associated with sys1*sys2 is

$$\begin{bmatrix} A_1 & B_1C_2 \\ 0 & A_2 \end{bmatrix}, \qquad \begin{bmatrix} B_1D_2 \\ B_2 \end{bmatrix}, \qquad \begin{bmatrix} C_1 & D_1C_2 \end{bmatrix}, \qquad D_1D_2$$

Finally, if sys1 is MIMO and sys2 is SISO, then sys1*sys2 or sys2*sys1 is interpreted as an entry-by-entry scalar multiplication and produces a system with the same dimensions as sys1, whose ijth entry is sys1(i,j)*sys2.

Inversion and Related Operations

Inversion of LTI models amounts to inverting the following input/output relationship.

$$y = H\, u \qquad \rightarrow \qquad u = H^{-1} y$$

This operation is defined only for square systems (that is, systems with as many inputs as outputs) and is performed using

```
inv(sys)
```

The resulting inverse model is of the same type as sys. Related operations include:

- Left division `sys1\sys2`, which is equivalent to `inv(sys1)*sys2`
- Right division `sys1/sys2`, which is equivalent to `sys1*inv(sys2)`

For a state-space model `sys` with data A, B, C, D, `inv(sys)` is defined only when D is a square invertible matrix, in which case its state-space data is

$$A - BD^{-1}C, \qquad BD^{-1}, \qquad -D^{-1}C, \qquad D^{-1}$$

Transposition

You can transpose an LTI model `sys` using

```
sys.'
```

This is a literal operation with the following effect:

- For TF models (with input arguments, `num` and `den`), the cell arrays `num` and `den` are transposed.
- For ZPK models (with input arguments, `z`, `p`, and `k`), the cell arrays, `z` and `p`, and the matrix `k` are transposed.
- For SS models (with model data A, B, C, D), transposition produces the state-space model $A^{\mathrm{T}}, C^{\mathrm{T}}, B^{\mathrm{T}}, D^{\mathrm{T}}$.
- For FRD models (with complex frequency response matrix `Response`), the matrix of frequency response data at each frequency is transposed.

Pertransposition

For a continuous-time system with transfer function $H(s)$, the *pertransposed* system has the transfer function

$$G(s) = [H(-s)]^T$$

The discrete-time counterpart is

$$G(z) = [H(z^{-1})]^T$$

Pertransposition of an LTI model `sys` is performed using

```
sys'
```

You can use pertransposition to obtain the Hermitian (conjugate) transpose of the frequency response of a given system. The frequency response of the

pertranspose of $H(s)$, $G(s) = [H(-s)]^T$, is the Hermitian transpose of the frequency response of $H(s)$: $G(jw) = H(jw)^H$.

To obtain the Hermitian transpose of the frequency response of a system `sys` over a frequency range specified by the vector `w`, type

```
freqresp(sys', w);
```

Model Interconnection Functions

The Control System Toolbox provides a number of functions to help with the model building process. These include model interconnection functions to perform I/O concatenation (`[,]`, `[;]`, and `append`), general parallel and series connections (`parallel` and `series`), and feedback connections (`feedback` and `lft`). These functions are useful to model open- and closed-loop systems.

Interconnection Operator	Description
`[,]`	Concatenates horizontally
`[;]`	Concatenates vertically
`append`	Appends models in a block diagonal configuration
`augstate`	Augments the output by appending states
`connect`	Forms an SS model from a block diagonal LTI object for an arbitrary interconnection matrix
`feedback`	Forms the feedback interconnection of two models
`lft`	Produces the LFT interconnection (Redheffer Star product) of two models
`parallel`	Forms the generalized parallel connection of two models
`series`	Forms the generalized series connection of two models

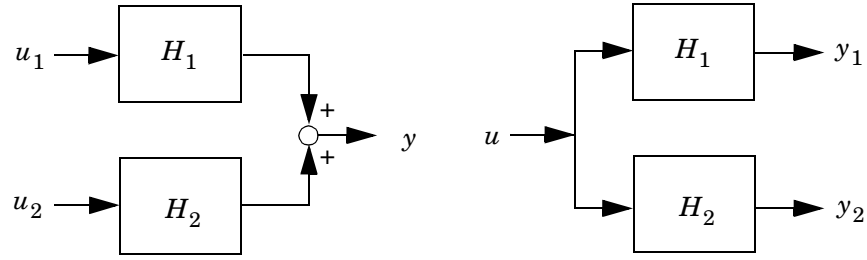
Concatenation of LTI Models

LTI model concatenation is done in a manner similar to the way you concatenate matrices in MATLAB, using

```
sys = [sys1 , sys2]% horizontal concatenation
sys = [sys1 ; sys2]% vertical concatenation
```

```
sys = append(sys1,sys2)% block diagonal appending
```

In I/O terms, horizontal and vertical concatenation have the following block-diagram interpretations (with H_1 and H_2 denoting the transfer matrices of `sys1` and `sys2`).



$$y = \begin{bmatrix} H_1 , H_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \qquad \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} H_1 \\ H_2 \end{bmatrix} u$$

Horizontal Concatenation **Vertical Concatenation**

You can use concatenation as an easy way to create MIMO transfer functions or zero-pole-gain models. For example,

```
H = [ tf(1,[1 0])  1 ; 0  tf([1  1],[1 1]) ]
```

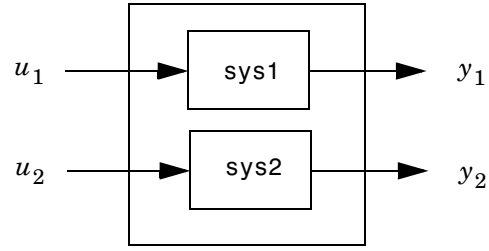
specifies

$$H(s) = \begin{bmatrix} \frac{1}{s} & 1 \\ 0 & \frac{s-1}{s+1} \end{bmatrix}$$

Use

```
append(sys1,sys2)
```

to specify the block-decoupled LTI model interconnection.



Appended Models

$$\begin{bmatrix} sys1 & 0 \\ 0 & sys2 \end{bmatrix}$$

Transfer Function

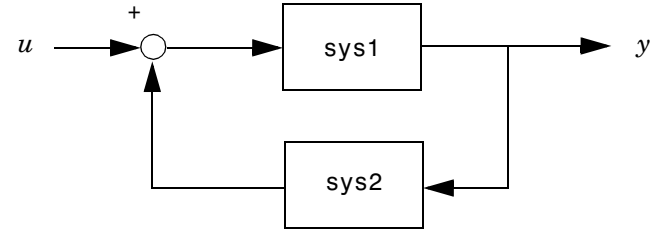
See `append` for more information on this function.

Feedback and Other Interconnection Functions

The following LTI model interconnection functions are useful for specifying closed- and open-loop model configurations:

- `feedback` puts two LTI models with compatible dimensions in a feedback configuration.
- `series` connects two LTI models in series.
- `parallel` connects two LTI models in parallel.
- `lft` performs the Redheffer star product on two LTI models.
- `connect` works with `append` to apply an arbitrary interconnection scheme to a set of LTI models.

For example, if `sys1` has m inputs and p outputs, while `sys2` has p inputs and m outputs, then the negative feedback configuration of these two LTI models



is realized with

```
feedback(sys1,sys2)
```

This specifies the LTI model with m inputs and p outputs whose I/O map is

$$(I + sys1 \cdot sys2)^{-1} sys1$$

See the reference pages online for more information on `feedback`, `series`, `parallel`, `lft`, and `connect`.

Continuous/Discrete Conversions of LTI Models

The function `c2d` discretizes continuous-time TF, SS, or ZPK models. Conversely, `d2c` converts discrete-time TF, SS, or ZPK models to continuous time. Several discretization/interpolation methods are supported, including zero-order hold (ZOH), first-order hold (FOH), Tustin approximation with or without frequency prewarping, and matched poles and zeros.

The syntax

```
sysd = c2d(sysc,Ts);     % Ts = sampling period in seconds
sysc = d2c(sysd);
```

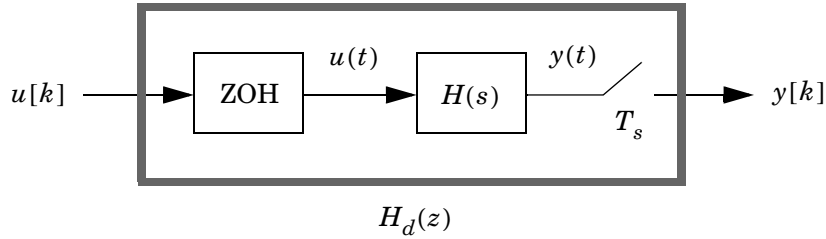
performs ZOH conversions by default. To use alternative conversion schemes, specify the desired method as an extra string input:

```
sysd = c2d(sysc,Ts,'foh');% use first-order hold
sysc = d2c(sysd,'tustin');% use Tustin approximation
```

The conversion methods and their limitations are discussed next.

Zero-Order Hold

Zero-order hold (ZOH) devices convert sampled signals to continuous-time signals for analyzing sampled continuous-time systems. The zero-order-hold discretization $H_d(z)$ of a continuous-time LTI model $H(s)$ is depicted in the following block diagram.



The ZOH device generates a continuous 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 then fed to the continuous system $H(s)$, and the resulting output $y(t)$ is sampled every T_s seconds to produce $y[k]$.

Conversely, given a discrete system $H_d(z)$, the `d2c` conversion produces a continuous system $H(s)$ whose ZOH discretization coincides with $H_d(z)$. This inverse operation has the following limitations:

- `d2c` cannot operate on LTI models with poles at $z = 0$ when the ZOH is used.
- Negative real poles in the z domain are mapped to *pairs* of complex poles in the s domain. As a result, the `d2c` conversion of a discrete system with negative real poles produces a continuous system with higher order.

The next example illustrates the behavior of `d2c` with real negative poles. Consider the following discrete-time ZPK model.

```
hd = zpk([], 0.5,1,0.1)

Zero/pole/gain:
   1
-------
(z+0.5)

Sampling time: 0.1
```

Use `d2c` to convert this model to continuous-time

```
hc = d2c(hd)
```

and you get a second-order model.

```
Zero/pole/gain:
   4.621 (s+149.3)
----------------------
(s^2 + 13.86s + 1035)
```

Discretize the model again

```
c2d(hc,0.1)
```

and you get back the original discrete-time system (up to canceling the pole/zero pair at z=–0.5):

```
Zero/pole/gain:
 (z+0.5)
```

```
---------
(z+0.5)^2

Sampling time: 0.1
```

First-Order Hold

First-order hold (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.

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

This method is generally more accurate than ZOH for systems driven by smooth inputs. Due to causality constraints, this option is only available for `c2d` conversions, and not `d2c` conversions.

Note This FOH method differs from standard causal FOH and is more appropriately called *triangle approximation* (see [2], p. 151). It is also known as *ramp-invariant* approximation because it is distortion-free for ramp inputs.

Tustin Approximation

The Tustin or bilinear approximation uses the approximation

$$z = e^{sT_s} \approx \frac{1 + sT_s/2}{1 - sT_s/2}$$

to relate s-domain and z-domain transfer functions. In `c2d` conversions, the discretization $H_d(z)$ of a continuous transfer function $H(s)$ is derived by

$$H_d(z) = H(s'), \text{ where } \qquad s' = \frac{2}{T_s}\frac{z-1}{z+1}$$

Similarly, the `d2c` conversion relies on the inverse correspondence

$$H(s) = H_d(z'), \text{ where } \qquad z' = \frac{1 + sT_s/2}{1 - sT_s/2}$$

Tustin with Frequency Prewarping

This variation of the Tustin approximation uses the correspondence

$$H_d(z) = H(s'), \qquad s' = \frac{\omega}{\tan(\omega T_s/2)} \frac{z-1}{z+1}$$

This change of variable ensures the matching of the continuous- and discrete-time frequency responses at the frequency ω.

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

Matched Poles and Zeros

The matched pole-zero method applies only to SISO systems. The continuous and discretized systems have matching DC gains and their poles and zeros correspond in the transformation

$$z = e^{sT_s}$$

See [2], p. 147 for more details.

Discretization of Systems with Delays

You can also use `c2d` to discretize SISO or MIMO continuous-time models with time delays. If `Ts` is the sampling period used for discretization:

- A delay of `tau` seconds in the continuous-time model is mapped to a delay of `k` sampling periods in the discretized model, where `k = fix(tau/Ts)`.
- The residual *fractional delay* `tau  k*Ts` is absorbed into the coefficients of the discretized model (for the zero-order-hold and first-order-hold methods only).

For example, to discretize the transfer function

$$H(s) = e^{-0.25s} \frac{10}{s^2 + 3s + 10}$$

using zero-order hold on the input, and a 10 Hz sampling rate, type

```
h = tf(10,[1 3 10],'inputdelay',0.25);
hd = c2d(h,0.1)
```

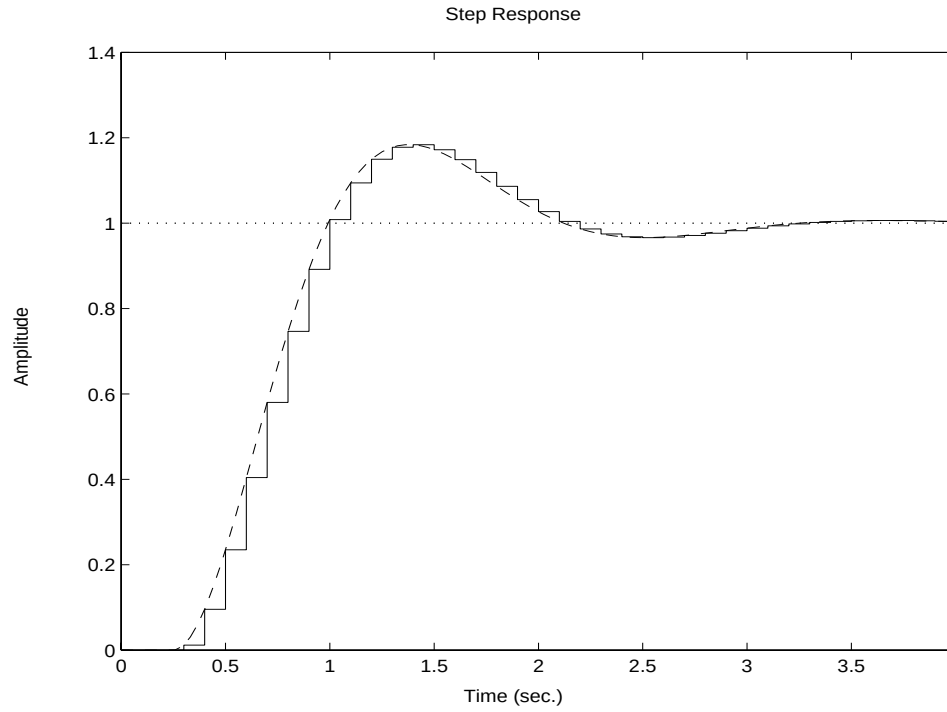
This produces the discrete-time transfer function

```
Transfer function:
         0.01187 z^2 + 0.06408 z + 0.009721
z^( 2) * ---------------------------------
             z^3   1.655 z^2 + 0.7408 z

Sampling time: 0.1
```

Here the input delay in $H(s)$ amounts to 2.5 times the sampling period of 0.1 seconds. Accordingly, the discretized model `hd` inherits an input delay of two sampling periods, as confirmed by the value of `hd.inputdelay`. The residual half-period delay is factored into the coefficients of `hd` by the discretization algorithm.

The step responses of the continuous and discretized models are compared in the figure below. This plot was produced by the command

```
step(h,'--',hd,'-')
```



Note The Tustin and matched pole/zero methods are accurate only for delays that are integer multiples of the sampling period. It is therefore preferable to use the `zoh` and `foh` discretization methods for models with delays.

Resampling of Discrete-Time Models

You can resample a discrete-time TF, SS, or ZPK model `sys1` by typing

```
sys2 = d2d(sys1,Ts)
```

The new sampling period `Ts` does not have to be an integer multiple of the original sampling period. For example, typing

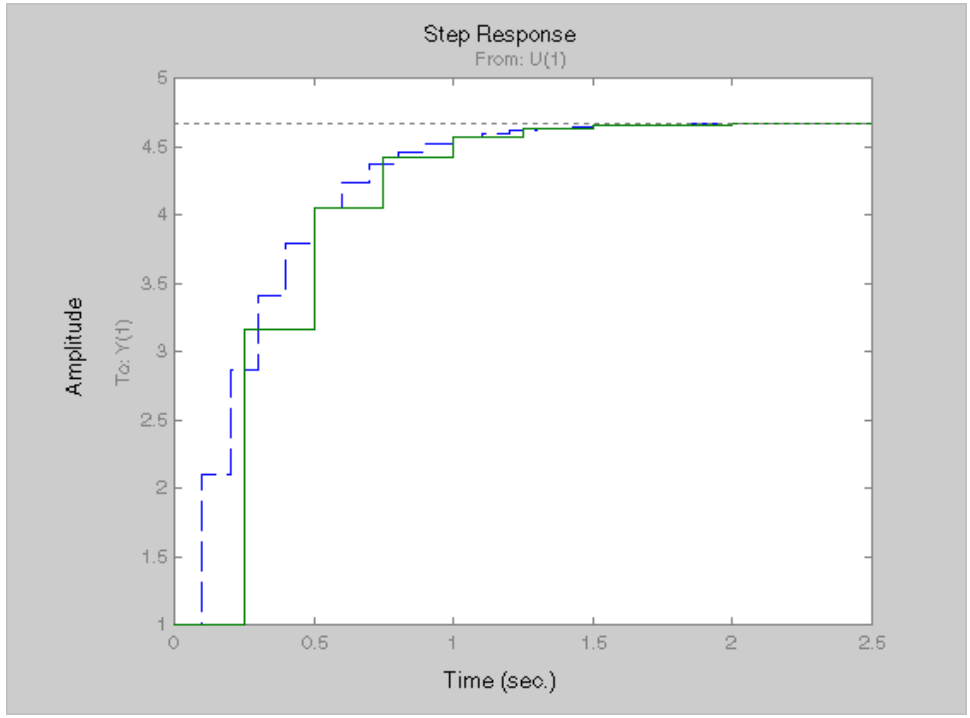
```
h1 = tf([1 0.4],[1  0.7],0.1);
h2 = d2d(h1,0.25);
```

resamples `h1` at the sampling period of 0.25 seconds, rather than 0.1 seconds.

You can compare the step responses of `h1` and `h2` by typing

```
step(h1,'--',h2,'-')
```

The resulting plot is shown on the figure below (`h1` is the dashed line).



References

[1] Åström, K.J. and B. Wittenmark, *Computer-Controlled Systems: Theory and Design*, Prentice-Hall, 1990, pp. 48–52.

[2] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems*, Second Edition, Addison-Wesley, 1990.

3

Model Analysis Tools

General Model Characteristics

General model characteristics include the model type, I/O dimensions, and continuous or discrete nature. Related commands are listed in the table below. These commands operate on continuous- or discrete-time LTI models or arrays of LTI models of any type.

General Model Characteristics Commands	
`class`	Display model type (`'tf'`, `'zpk'`, `'ss'`, or `'frd'`).
`hasdelay`	Test true if LTI model has any type of delay.
`isa`	Test true if LTI model is of specified class.
`isct`	Test true for continuous-time models.
`isdt`	Test true for discrete-time models.
`isempty`	Test true for empty LTI models.
`isproper`	Test true for proper LTI models.
`issiso`	Test true for SISO models.
`ndims`	Display the number of model/array dimensions.
`reshape`	Change the shape of an LTI array.
`size`	Output/input/array dimensions. Used with special syntax, `size` also returns the number of state dimensions for state-space models, and the number of frequencies in an FRD model.

This example illustrates the use of some of these commands. See the related reference pages for more details.

```
H = tf({1 [1  1]},{[1 0.1] [1 2 10]})

Transfer function from input 1 to output:
   1
-------
s + 0.1
```

```
Transfer function from input 2 to output:
    s   1
--------------
s^2 + 2 s + 10


class(H)

ans =
tf

size(H)
Transfer function with 2 input(s) and 1 output(s).

[ny,nu] = size(H)% Note: ny = number of outputs

ny =
     1

nu =
     2

isct(H)% Is this system continuous?

ans =
     1

isdt(H)% Is this system discrete?

ans =
     0
```

Model Dynamics

The Control System Toolbox offers commands to determine the system poles, zeros, DC gain, norms, etc. You can apply these commands to single LTI models or LTI arrays. The following table gives an overview of these commands.

Model Dynamics	
`covar`	Covariance of response to white noise.
`damp`	Natural frequency and damping of system poles.
`dcgain`	Low-frequency (DC) gain.
`dsort`	Sort discrete-time poles by magnitude.
`esort`	Sort continuous-time poles by real part.
`norm`	Norms of LTI systems (H_2 and L_∞).
`pole`, `eig`	System poles.
`pzmap`	Pole/zero map.
`zero`	System transmission zeros.

With the exception of L_∞ `norm`, these commands are not supported for FRD models.

Here is an example of model analysis using some of these commands.

```
h = tf([4 8.4 30.8 60],[1 4.12 17.4 30.8 60])

Transfer function:
     4 s^3 + 8.4 s^2 + 30.8 s + 60
-------------------------------------
s^4 + 4.12 s^3 + 17.4 s^2 + 30.8 s + 60

pole(h)

ans =
   1.7971 + 2.2137i
```

```
   1.7971   2.2137i
   0.2629 + 2.7039i
   0.2629   2.7039i

zero(h)
ans =
     0.0500 + 2.7382i
     0.0500   2.7382i
     2.0000

dcgain(h)

ans =
      1

[ninf,fpeak] = norm(h,inf)% peak gain of freq. response

ninf =
1.3402     % peak gain

fpeak =
1.8537     % frequency where gain peaks
```

These functions also operate on LTI arrays and return arrays. For example, the poles of a three dimensional LTI array `sysarray` are obtained as follows.

```
sysarray = tf(rss(2,1,1,3))

Model sysarray(:,:,1,1)
=======================
Transfer function:
    -0.6201 s - 1.905
  ---------------------
  s^2 + 5.672 s + 7.405

Model sysarray(:,:,2,1)
=======================
Transfer function:
  0.4282 s^2 + 0.3706 s + 0.04264
  -------------------------------
      s^2 + 1.056 s + 0.1719
```

```
Model sysarray(:,:,3,1)
=======================
Transfer function:
    0.621 s + 0.7567
  ---------------------
  s^2 + 2.942 s + 2.113

3x1 array of continuous-time transfer functions.

pole(sysarray)
ans(:,:,1) =
   -3.6337
   -2.0379
ans(:,:,2) =
   -0.8549
   -0.2011
ans(:,:,3) =
   -1.6968
   -1.2452
```

State-Space Realizations

The following functions are useful to analyze, perform state coordinate transformations on, and derive canonical state-space realizations for single state-space LTI models or LTI arrays of state-space models.

State-Space Realizations	
`canon`	Canonical state-space realizations.
`ctrb`	Controllability matrix.
`ctrbf`	Controllability staircase form.
`gram`	Controllability and observability gramians.
`obsv`	Observability matrix.
`obsvf`	Observability staircase form.
`ss2ss`	State coordinate transformation.
`ssbal`	Diagonal balancing of state-space realizations.

The function `ssbal` uses a simple *diagonal* similarity transformation

$$(A, B, C) \to (T^{-1}AT, T^{-1}B, CT)$$

to balance the state-space data (A, B, C) . This is accomplished by reducing the norm of the matrix.

$$\begin{bmatrix} T^{-1}AT & T^{-1}B \\ CT & 0 \end{bmatrix}$$

Such balancing usually improves the numerical conditioning of subsequent state-space computations. Note that conversions to state-space using `ss` produce balanced realizations of transfer functions and zero-pole-gain models.

By contrast, the canonical realizations produced by `canon`, `ctrbf`, or `obsvf` are often badly scaled, sensitive to perturbations of the data, and

poorly suited for state-space computations. Consequently, it is wise to use them only for analysis purposes and not in control design algorithms.

4

Arrays of LTI Models

Introduction

In many applications, it is useful to consider collections of linear, time invariant (LTI) models. For example, you may want to consider a model with a single parameter that varies, such as

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

and so on. A convenient way to store and analyze a collection like this is to use LTI arrays. Continuing this example, you can create this LTI array and store all three transfer functions in one variable.

```
sys_ltia = (sys1, sys2, sys3);
```

You can use the LTI array `sys_ltia` just like you would use, for example, sys1.

You can use LTI arrays to collect a set of LTI models into a single MATLAB variable. You then use this variable to manipulate or analyze the entire collection of models in a vectorized fashion. You access the individual models in the collection through indexing rather than by individual model names.

LTI arrays extend the concept of single LTI models in a similar way to how multidimensional arrays extend two-dimensional matrices in MATLAB (see Multidimensional Arrays in the MATLAB documentation).

When to Collect a Set of Models in an LTI Array

You can use LTI arrays to represent

- A set of LTI models arising from the linearization of a nonlinear system at several operating points
- A collection of transfer functions that depend on one or more parameters
- A set of LTI models arising from several system identification experiments applied to one plant
- A set of gain-scheduled LTI controllers
- A list of LTI models you want to collect together under the same name

Restrictions for LTI Models Collected in an Array

For each model in an LTI array, the following properties must be the same:

- The number of inputs and outputs
- The sample time, for discrete-time models
- The I/O names and I/O groups

Note You cannot specify Simulink LTI blocks with LTI arrays.

Where to Find Information on LTI Arrays

The next two sections give examples that illustrate the concept of an LTI array, its dimensions, and size. To read about how to build an LTI array, go to "Building LTI Arrays" on page 4-12. The remainder of the chapter is devoted to indexing and operations on LTI Arrays. You can also apply the analysis functions in the Control System Toolbox to LTI arrays. See Chapter 5, "Model Analysis Tools," for more information on these functions. You can also view response plots of LTI arrays with the LTI Viewer.

The Concept of an LTI Array

To visualize the concept of an LTI 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.

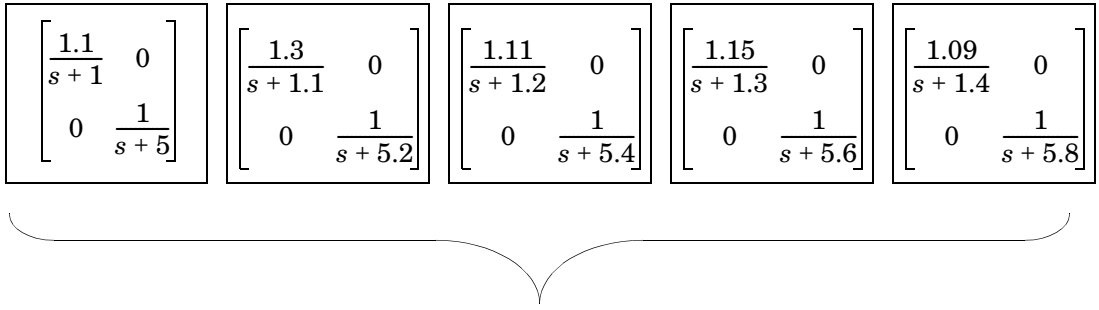


Figure 4-1: Five LTI Models to be Collected in an LTI Array

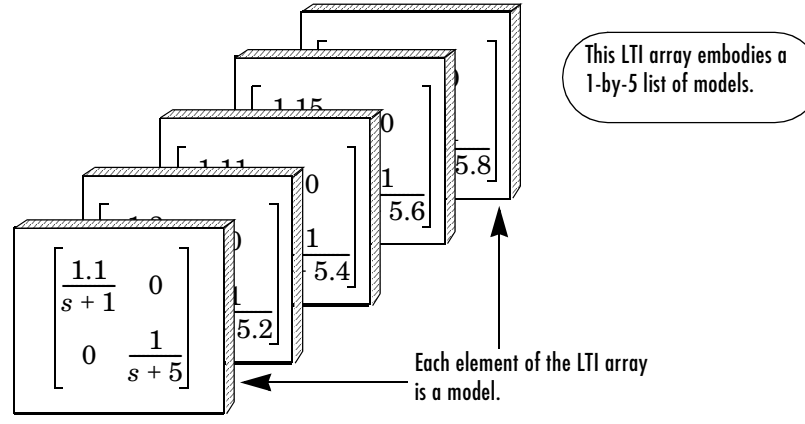
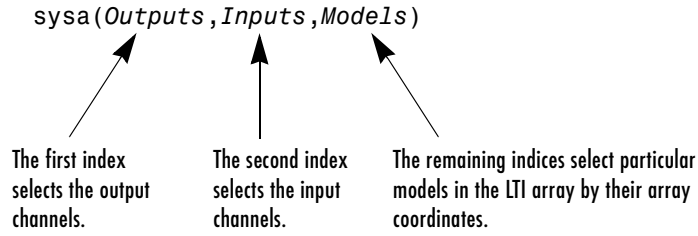


Figure 4-2: An LTI Array Containing These Five Models

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 an LTI array under one variable name, say, `sys`. Each element of the LTI array is an LTI model.

Individual models in the LTI array `sys` are accessed via indexing. The general form for the syntax you use to access data in an LTI array is



For example, you can access the third model in `sys` with `sys(:,:,3)`. The following illustrates how you can use indexing to select models or their components from `sys`.

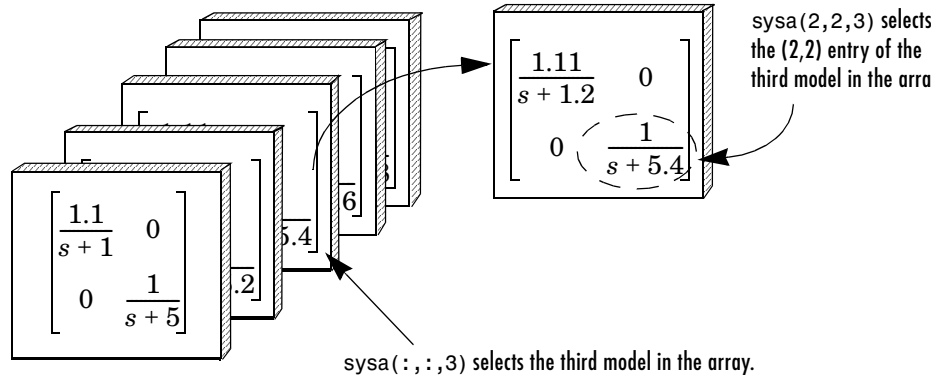


Figure 4-3: Using Indices to Select Models and Their Components

See "Indexing Into LTI Arrays" for more information on indexing.

Higher Dimensional Arrays of LTI Models

You can also collect a set of models in a two-dimensional array. The following diagram illustrates a 2-by-3 array of six, two-output, one-input models called `m2d`.

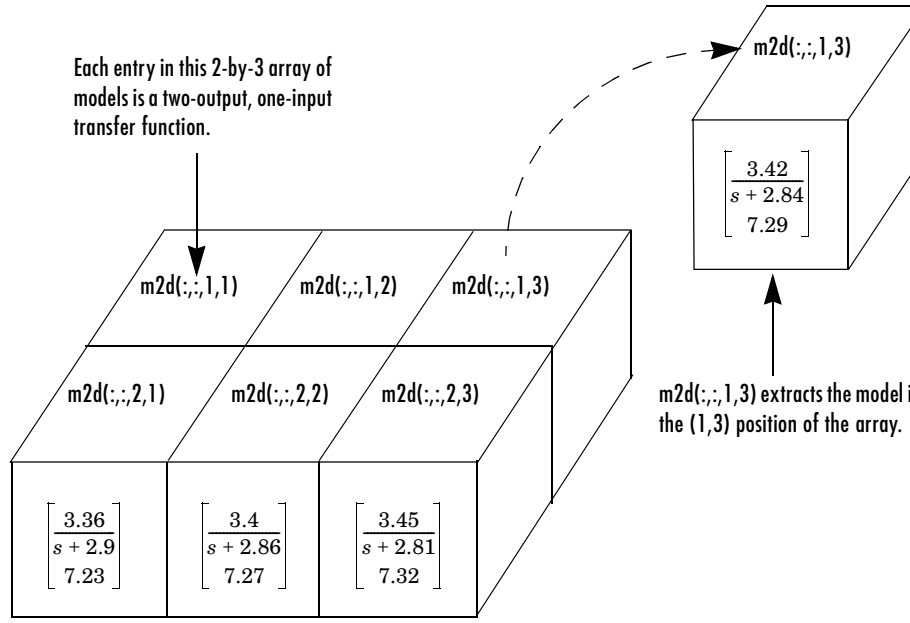


Figure 4-4: m2d: A 2-by-3 Array of Two-Output, One-Input Models

More generally, you can organize models into a 3-D or higher-dimensional array, in much the same way you arrange numerical data into multidimensional arrays (see Multidimensional Arrays in the MATLAB documentation).

Dimensions, Size, and Shape of an LTI Array

The dimensions and size of a single LTI model are determined by the output and input channels. An array of LTI models has additional quantities that determine its dimensions, size, and shape.

There are two sets of dimensions associated with LTI arrays:

- The *I/O dimensions*—the output dimension and input dimension common to all models in the LTI array
- The *array dimensions*—the dimensions of the array of models itself

The size of the LTI array is determined by:

- The lengths of the I/O dimensions—the number of outputs (or inputs) common to all models in the LTI array
- The length of each array dimension—the number of models along that array dimension

The next figure illustrates the concepts of dimension and size for the LTI array `m2d`, a 2-by-3 array of one-input, two-output transfer function models.

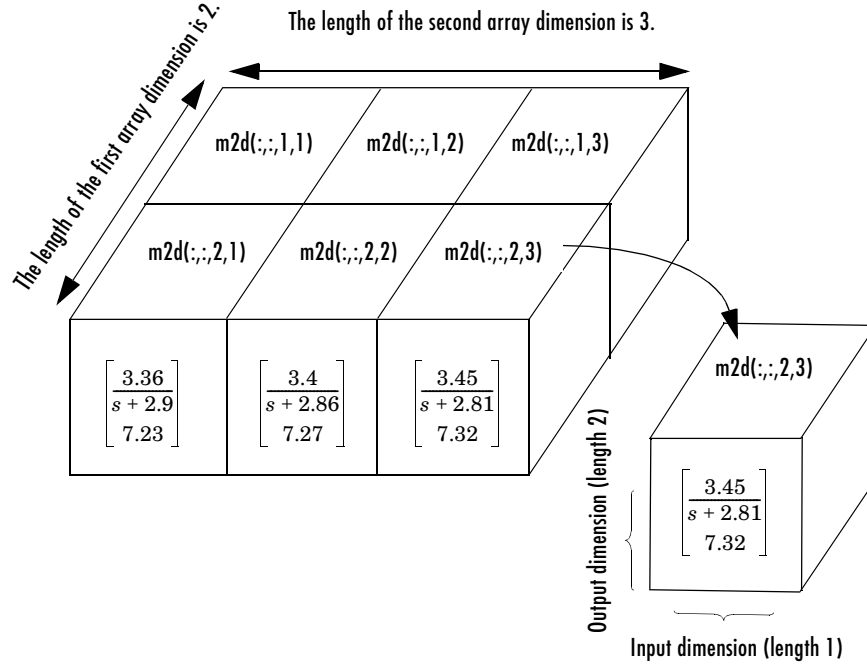


Figure 4-5: Dimensions and Size of m2d, an LTI Array

You can load this sample LTI array into your workspace by typing

```
load LTIexamples
size(m2d)

2x3 array of continuous-time transfer functions
Each transfer function has 2 outputs and 1 input.
```

The I/O dimensions correspond to the row and column dimensions of the transfer matrix. The two I/O dimensions are both of length 1 for SISO models. For MIMO models the lengths of these dimensions are given by the number of outputs and inputs of the model.

Five related quantities are pertinent to understanding the array dimensions:

- N, the number of models in the LTI array
- K, the number of array dimensions
- $S_1 S_2 \ldots S_K$, the list of lengths of the array dimensions
  - S_i is the number of models along the i^{th} dimension.
- S_1–by–S_2–by–…–by–S_K, the configuration of the models in the array
  - The configuration determines the shape of the array.
  - The product of these integers $S_1 \times S_2 \times \ldots \times S_K$ is N.

In the example model `m2d`,:

- The length of the output dimension, the first I/O dimension, is 2, since there are two output channels in each model.
- The length of the input dimension, the second I/O dimension, is 1, since there is only one input channel in each model.
- N, the number of models in the LTI array, is 6.
- K, the number of array dimensions, is 2.
- The array dimension lengths are [2 3].
- The array configuration is 2-by-3.

size and ndims

You can access the dimensions and shape of an LTI array using

- `size` to determine the lengths of each of the dimensions associated with an LTI array
- `ndims` to determine the total number of dimensions in an LTI array

When applied to an LTI array, `size` returns

```
[Ny Nu S1 S2 ... Sk]
```

where

- `Ny` is the number of outputs common to all models in the LTI array.
- `Nu` is the number of inputs common to all models in the LTI array.
- `S1 S2 ... Sk` are the lengths of the array dimensions of a k-dimensional array of models. `Si` is the number of models along the ith array dimension.

Note the following when using the `size` function:

- By convention, a single LTI model is treated as a 1-by-1 array of models. For single LTI models, size returns only the I/O dimensions `[Ny Nu]`.
- For LTI arrays, size always returns at least two array dimensions. For example, the size of a 2-by-1 LTI array in `[Ny Nu 2 1]`
- `size` ignores trailing singleton dimensions beyond the second array dimension. For example, `size` returns `[Ny Nu 2 3]` for a 2-by-3-by-1-by-1 LTI array of models with `Ny` outputs and `Nu` inputs.

The function `ndims` returns the total number of dimensions in an LTI array:

- 2, for single LTI models
- 2 + `p`, for LTI arrays, where `p` (greater than 2) is the number of array dimensions

Note that

```
ndims (sys) = length(size(sys))
```

To see how these work on the sample 2-by-3 LTI array `m2d` of two-output, one-input models, type

```
load LTIexamples
s = size(m2d)

s =

     2     1     2     3
```

Notice that `size` returns a vector whose entries correspond to the length of each of the four dimensions of `m2d`: two outputs and one input in a 2-by-3 array of models. Type

```
ndims(m2d)

ans =
     4
```

to see that there are indeed four dimensions attributed to this LTI array.

reshape

Use `reshape` to reorganize the arrangement (array configuration) of the models of an existing LTI array.

For example, to arrange the models in an LTI Array `sys` as a $w_1 \times \ldots \times w_p$ array, type

```
reshape(sys,w1,...,wp)
```

where `w1,...,wp` are any set of integers whose product is *N,* the number of models in `sys`.

You can reshape the LTI array `m2d` into a 3-by-2, a 6-by-1, or a 1-by-6 array using `reshape`. For example, type

```
load LTIexamples
sys = reshape(m2d,6,1);
size(sys)
```

```
6x1 array of continuous-time transfer functions
Each transfer function has 2 outputs and 1 inputs.
```

```
s = size(sys)
```

```
s =
     2     1     6     1
```

Building LTI Arrays

There are several ways to build LTI arrays:

- Using a `for` loop to assign each model in the array
- Using `stack` to concatenate LTI models into an LTI array
- Using `tf`, `zpk`, `ss`, or `frd`

In addition, you can use the command `rss` to generate LTI arrays of random state-space models.

Generating LTI Arrays Using rss

A convenient way to generate arrays of state-space models with the same number of states in each model is to use `rss`. The syntax is

```
rss(N,P,M,sdim1,...,sdimk)
```

where

- `N` is the number of states of each model in the LTI array.
- `P` is the number of outputs of each model in the LTI array.
- `M` is the number of inputs of each model in the LTI array.
- `sdim1,...,sdimk` are the lengths of the array dimensions.

For example, to create a 4-by-2 array of random state-space models with three states, one output, and one input, type

```
sys = rss(3,2,1,4,2);
size(sys)

4x2 array of continuous-time state-space models
Each model has 2 outputs, 1 input, and 3 states.
```

Building LTI Arrays Using for Loops

Consider the following second-order SISO transfer function that depends on two parameters, ζ and ω

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

Suppose, based on measured input and output data, you estimate confidence intervals $[\omega_1,\omega_2]$, and $[\zeta_1,\zeta_2]$ for each of the parameters, ω and ζ. All of the possible combinations of the confidence limits for these model parameter values give rise to a set of four SISO models.

	ω_1	ω_2
ζ_1	$H_{11}(s)=\dfrac{\omega_1^2}{s^2+2\zeta_1\omega_1 s+\omega_1^2}$	$H_{12}(s)=\dfrac{\omega_1^2}{s^2+2\zeta_2\omega_1 s+\omega_1^2}$
ζ_2	$H_{21}(s)=\dfrac{\omega_2^2}{s^2+2\zeta_1\omega_2 s+\omega_2^2}$	$H_{22}(s)=\dfrac{\omega_2^2}{s^2+2\zeta_2\omega_2 s+\omega_2^2}$

Figure 4-6: Four LTI Models Depending on Two Parameters

You can arrange these four models in a 2-by-2 array of SISO transfer functions called `H`.

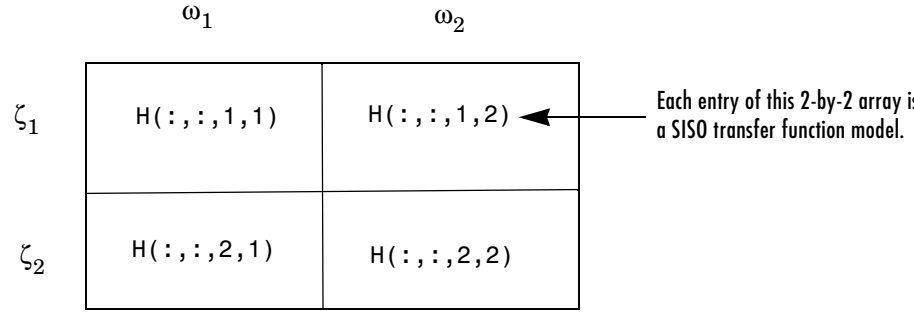


Figure 4-7: The LTI Array H

Here, for $i,j \in \{1,2\}$, `H(:,:,i,j)` represents the transfer function

$$\frac{\omega_j^2}{s^2+2\zeta_i\omega_j s+\omega_j^2}$$

corresponding to the parameter values $\zeta = \zeta_i$ and $\omega = \omega_j$.

The first two colon indices (`:`) select all I/O channels from the I/O dimensions of `H`. The third index of `H` refers to the first array dimension (ζ), while the fourth index is for the second array dimension (ω).

Suppose the limits of the ranges of values for ζ and ω are [0.66,0.76] and [1.2,1.5], respectively. Enter these at the command line.

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

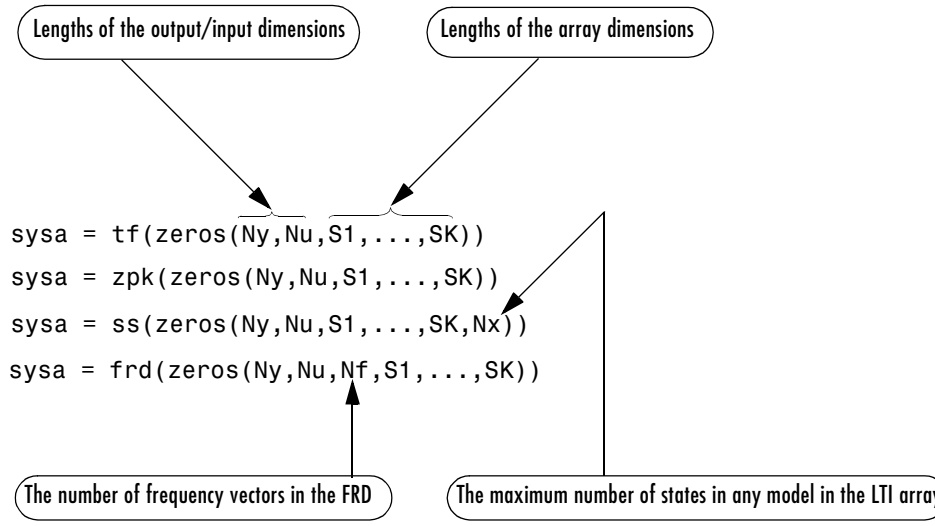
Since the four models have the same parametric structure, it's convenient to use two nested `for` loops to construct the LTI array.

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

`H` now contains the four models in a 2-by-2 array. For example, to display the transfer function in the (1,2) position of the array, type

```
H(:,:,1,2)

Transfer function:
       2.25
-------------------
s^2 + 1.98 s + 2.25
```

For the purposes of efficient computation, you can initialize an LTI array to zero, and then reassign the entire array to the values you want to specify. The general syntax for zero assignment of LTI arrays is



To initialize `H` in the above example to zero, type

```
H = tf(zeros(1,1,2,2));
```

before you implement the nested `for` loops.

Building LTI Arrays Using the stack Function

Another way to build LTI arrays is using the function `stack`. This function operates on single LTI models as well as LTI arrays. It concatenates a list of LTI arrays or single LTI models only along the array dimension. The general syntax for `stack` is

```
stack(Arraydim,sys1,sys2...)
```

where

- `Arraydim` is the array dimension along which to concatenate the LTI models or arrays.
- `sys1`, `sys2`, ... are the LTI models or LTI arrays to be concatenated.

When you concatenate several models or LTI arrays along the *j*th array dimension, such as in

```
stack(j,sys1,sys2,...,sysn)
```

- The lengths of the I/O dimensions of `sys1,...,sysn` must all match.
- The lengths of all but the *j*th array dimension of `sys1,...,sysn` must match.

For example, if two TF models `sys1` and `sys2` have the same number of inputs and outputs,

```
sys = stack(1,sys1,sys2)
```

concatenates them into a 2-by-1 array of models.

There are two principles that you should keep in mind:

- `stack` only concatenates along an array dimension, not an I/O dimension.
- To concatenate LTI models or LTI arrays along an input or output dimension, use the bracket notation (`[,] [;]`). See "Model Interconnection Functions" for more information on the use of bracket notation to concatenate models. See also "Special Cases for Operations on LTI Arrays" for some examples of this type of concatenation of LTI arrays.

Here's an example of how to build the LTI array `H` using the function `stack`.

```
% Set up the parameter vectors.

zeta = [0.66,0.75];
w = [1.2,1.5];

% Specify the four individual models with those parameters.
%
H11 = tf(w(1)^2,[1 2*zeta(1)*w(1) w(1)^2]);
H12 = tf(w(2)^2,[1 2*zeta(1)*w(2) w(2)^2]);
H21 = tf(w(1)^2,[1 2*zeta(2)*w(1) w(1)^2]);
H22 = tf(w(2)^2,[1 2*zeta(2)*w(2) w(2)^2]);

% Set up the LTI array using stack.

COL1 = stack(1,H11,H21); % The first column of the 2-by-2 array
COL2 = stack(1,H12,H22); % The second column of the 2-by-2 array
H = stack(2, COL1, COL2); % Concatenate the two columns of models.
```

Notice that this result is very different from the single MIMO LTI model returned by

```
H = [H11,H12;H21,H22];
```

Accessing LTI Arrays of Variable Order

For arrays of state-space models with variable order, you cannot use the dot operator (e.g., `sys.a`) to access arrays. Use the syntax

```
[a,b,c,d] = ssdata(sys,'cell')
```

to extract the state-space matrices of each model as separate cells in the cell arrays `a`, `b`, `c`, and `d`.

Building LTI Arrays Using tf, zpk, ss, and frd

You can also build LTI arrays using the `tf`, `zpk`, `ss`, and `frd` constructors. You do this by using multidimensional arrays in the input arguments for these functions.

Specifying Arrays of TF models tf

For TF models, use

```
sys = tf(num,den)
```

where

- Both `num` and `den` are multidimensional cell arrays the same size as `sys` (see "size and ndims" on page 4-9).
- `sys(i,j,n1,...,nK)` is the (i, j) entry of the transfer matrix for the model located in the $(n_1, \ldots, n_K)$ position of the array.
- `num(i,j,n1,...,nK)` is a row vector representing the numerator polynomial of `sys(i,j,n1,...,nK)`.
- `den(i,j,n1,...,nK)` is a row vector representing denominator polynomial of `sys(i,j,n1,...,nK)`.

See "MIMO Transfer Function Models" on page 2-10 for related information on the specification of single TF models.

Specifying Arrays of ZPK Models Using zpk

For ZPK models, use

```
sys = zpk(zeros,poles,gains)
```

where

- Both `zeros` and `poles` are multidimensional cell arrays whose cell entries contain the vectors of zeros and poles for each I/O pair of each model in the LTI array.
- `gains` is a multidimensional array containing the scalar gains for each I/O pair of each model in the array.
- The dimensions (and their lengths) of `zeros`, `poles`, and `gains`, determine those of the LTI array, `sys`.

Specifying Arrays of SS Models Using ss

To specify arrays of SS models, use

```
sys = ss(a,b,c,d)
```

where `a`, `b`, `c`, and `d` are real-valued multidimensional arrays of appropriate dimensions. All models in the resulting array of SS models have the same number of states, outputs, and inputs.

Note You cannot use the `ss` constructor to build an array of state-space models with different numbers of states. Use `stack` to build such LTI arrays.

The Size of LTI Array Data for SS Models

The size of the model data for arrays of state-space models is summarized in the following table.

Data	Size (Data)
a	$\left[N_s N_s S_1 S_2 \ldots S_K\right]$
b	$\left[N_s N_u S_1 S_2 \ldots S_K\right]$
c	$\left[N_y N_s S_1 S_2 \ldots S_K\right]$
d	$\left[N_y N_u S_1 S_2 \ldots S_K\right]$

where

- N_s is the maximum of the number of states in each model in the array.
- N_u is the number of inputs in each model.
- N_y is the number of outputs in each model.
- $S_1, S_2, \ldots, S_K$ are the lengths of the array dimensions.

Specifying Arrays of FRD Models Using frd

To specify a K-dimensional array of p-output, m-input FRD models for which $S_1, S_2, \ldots, S_K$ are the lengths of the array dimensions, use

```
sys = frd(response,frequency,units)
```

where

- `frequency` is a real vector of n frequency data points common to all FRD models in the LTI array.
- `response` is a p-by-m-by-n-by-S_1-by-…-by-S_K complex-valued multidimensional array.
- `units` is the optional string specifying `'rad/s'` or `'Hz'`.

Note that for specifying an LTI array of SISO FRD models, `response` can also be a multidimensional array of 1-by-n matrices whose remaining dimensions determine the array dimensions of the FRD.

Indexing Into LTI Arrays

You can index into LTI arrays in much the same way as you would for multidimensional arrays to

- Access models
- Extract subsystems
- Reassign parts of an LTI array
- Delete parts of an LTI array

When you index into an LTI array `sys`, the indices should be organized according to the following format

`sys(Outputs, Inputs,` $n_1, \ldots, n_K$ `)`

where

- `Outputs` are indices that select output channels.
- `Inputs` are indices that select input channels.
- $n_1, \ldots, n_K$ are indices into the array dimensions that select one model or a subset of models in the LTI array.

Note on Indexing into LTI Arrays of FRD models. For 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)
```

See "Referencing FRD Models Through Frequencies" for details on frequency point selection in FRD models.

Accessing Particular Models in an LTI Array

To access any given model in an LTI array:

- Use colon arguments `(:,:)` for the first two indices to select all I/O channels.
- The remaining indices specify the model coordinates within the array.

For example, if `sys` is a 5-by-2 array of state-space models defined by

```
sys = rss(4,3,2,5,2);
```

you can access (and display) the model located in the `(3,2)` position of the array `sys` by typing

```
sys(:,:,3,2)
```

If `sys` is a 5-by-2 array of 3-output, 2-input FRD models, with frequency vector `[1,2,3,4,5]`, then you can access the response data corresponding to the middle frequency (3 rad/s), of the model in the (3,1) position by typing

```
sys(:,:,3,1,'frequency',3.0)
```

To access all frequencies of this model in the array, you can simply type

```
sys(:,:,3,1)
```

Single Index Referencing of Array Dimensions

You can also access models using single index referencing of the array dimensions.

For example, in the 5-by-2 LTI array `sys` above, you can also access the model located in the `(3,2)` position by typing

```
sys(:,:,8)
```

since this model is in the eighth position if you were to list the 10 models in the array by successively scanning through its entries along each of its columns.

For more information on single index referencing, see “Advanced Indexing” under “M-File Programming” in the MATLAB online documentation.

Extracting LTI Arrays of Subsystems

To select a particular subset of I/O channels from all the models in an LTI array, use the syntax described in “Extracting and Modifying Subsystems” on page 3-5. For example,

```
sys = rss(4,3,2,5,2);
A = sys(1, [1 2])
```

or equivalently,

```
A = sys(1,[1 2],:,:)
```

selects the first two input channels, and the first output channel in each model of the LTI array `A`, and returns the resulting 5-by-2 array of one-output, two-input subsystems.

You can also combine model selection with I/O selection within an LTI array. For example, to access both:

- The state-space model in the `(3,2)` array position
- Only the portion of that model relating the second input to the first output

type

```
sys(1,2,3,2)
```

To access the subsystem from all inputs to the first two output channels of this same array entry, type

```
sys(1:2,:,3,2)
```

Reassigning Parts of an LTI Array

You can reassign entire models or portions of models in an LTI array. For example,

```
sys = rss(4,3,2,5,2); % 5X2 array of state-space models
H = rss(4,1,1,5,2);   % 5X2 array of SISO models
sys(1,2) = H
```

reassigns the subsystem from input two to output one, for all models in the LTI array `sys`. This SISO subsystem of each model in the LTI array is replaced with the LTI array `H` of SISO models. This one-line assignment command is equivalent to the following 10-step nested `for` loop.

```
for k = 1:5
  for j = 1:2
    sys(1,2,k,j) = H(:,:,k,j);
  end
end
```

Notice that you don't have to use the array dimensions with this assignment. This is because I/O selection applies to all models in the array when the array indices are omitted.

Similarly, the commands

```
sys(:,:,3,2) = sys(:,:,4,1);
sys(1,2,3,2) = 0;
```

reassign the entire model in the (3,2) position of the LTI array `sys` and the (1,2) subsystem of this model, respectively.

Deleting Parts of an LTI Array

You can use indexing to delete any part of an LTI array by reassigning it to be empty (`[]`). For instance,

```
sys = rss(4,3,2,5,2);
sys(1,:) = [];
size(sys)
```

```
5x2 array of continuous-time state-space models
Each model has 2 outputs, 2 inputs, and 4 states.
```

deletes the first output channel from every model of this LTI array.

Similarly,

```
sys(:,:,[3 4],:) = []
```

deletes the third and fourth rows of this two-dimensional array of models.

Operations on LTI Arrays

Using LTI arrays, you can apply almost all of the basic model operations that work on single LTI models to entire sets of models at once. These basic operations include

- The arithmetic operations: `+`, `-`, `*`, `/`, `\`, `'`, `.'`
- The functions: concatenation along I/O dimensions (`[,]`, `[;]`), `feedback`, `append`, `series`, `parallel`, and `lft`

When you apply any of these operations to two (or more) LTI arrays (for example, `sys1` and `sys2`), the operation is implemented on a model-by-model basis. Therefore, the *k*th model of the resulting LTI array is derived from the application of the given operation to the *k*th model of `sys1` and the *k*th model of `sys2`.

For example, if `sys1` and `sys2` are two LTI arrays and

```
sysa = op(sys1,sys2)
```

then the *k*th model in the resulting LTI array `sys` is obtained by adding the *k*th models in `sys1` to the *k*th model in `sys2`

```
sysa(:,:,k) = sys1(:,:,k) + sys2(:,:,k)
```

You can also apply any of the response plotting functions such as `step`, `bode`, and `nyquist` to LTI arrays. These plotting functions are also applied on a model by model basis.

Example: Addition of Two LTI Arrays

The following diagram illustrates the addition of two 3-by-1 LTI arrays `sys1+sys2`.

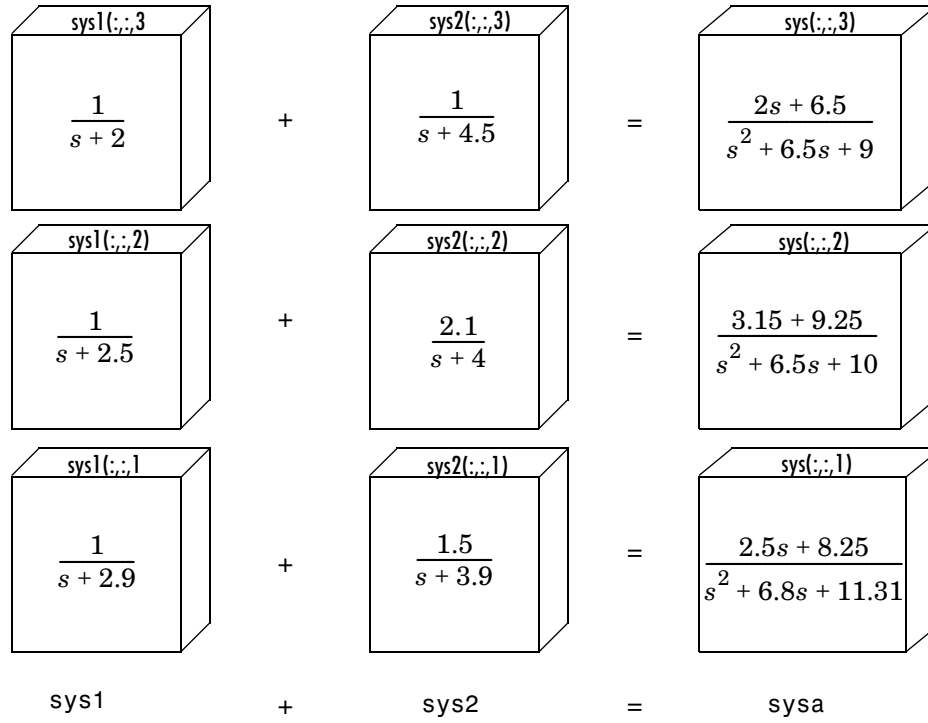


Figure 4-8: The Addition of Two LTI Arrays

The summation of these LTI arrays

```
sysa = sys1+sys2
```

is equivalent to the following model-by-model summation:

```
for k = 1:3
  sysa(:,:,k)=sys1(:,:,k) + sys2(:,:,k)
end
```

Note that:

- Each model in `sys1` and `sys2` must have the same number of inputs and outputs. This is required for the addition of two LTI arrays.
- The lengths of the array dimensions of `sys1` and `sys2` must match.

Dimension Requirements

In general, when you apply any of these basic operations to two or more LTI arrays:

- The I/O dimensions of each of the LTI arrays must be compatible with the requirements of the operation.
- The lengths of array dimensions must match.

The I/O dimensions of each model in the resulting LTI array are determined by the operation being performed. See Chapter 3, "Operations on LTI Models," for requirements on the I/O dimensions for the various operations.

For example, if `sys1` and `sys2` are both 1-by-3 arrays of LTI models with two inputs and two outputs, and `sys3` is a 1-by-3 array of LTI models with two outputs and 1 input, then

```
sys1 + sys2
```

is an LTI array with the same dimensions as `sys1` and `sys2`.

```
sys1 * sys3
```

is a 1-by-3 array of LTI models with two outputs and one input, and

```
[sys1,sys3]
```

is a 1-by-3 array of LTI models with two outputs and three inputs.

Special Cases for Operations on LTI Arrays

There are some special cases in coding operations on LTI arrays.

Consider

```
sysa = op(sys1,sys2)
```

where `op` is a symbol for the operation being applied. `sys1` is an LTI array, and `sysa` (the result of the operation) is an LTI array with the same array

dimensions as sys1. You can use shortcuts for coding sysa = op(sys1,sys2) in the following cases:

- For operations that apply to LTI arrays, sys2 does not have to be an array. It can be a single LTI model (or a gain matrix) whose I/O dimensions satisfy the compatibility requirements for op (with those of each of the models in sys1). In this case, op applies sys2 to each model in sys1, and the *k*th model in sys satisfies

  ```
  sysa(:,:,k) = op(sys1(:,:,k),sys2)
  ```

- For arithmetic operations, such as +, *, /, and \, sys2 can be either a single SISO model, or an LTI array of SISO models, even when sys1 is an LTI array of MIMO models. This special case relies on the MATLAB scalar expansion capabilities for arithmetic operations.
  - When sys2 is a single SISO LTI model (or a scalar gain), op applies sys2 to sys1 on an entry-by-entry basis. The *ij*th entry in the *k*th model in sysa satisfies

    ```
    sysa(i,j,k) = op(sys1(i,j,k),sys2)
    ```

  - When sys2 is an LTI array of SISO models (or a multidimensional array of scalar gains), op applies sys2 to sys1 on an entry-by-entry basis for each model in sysa.

    ```
    sysa(i,j,k) = op(sys1(i,j,k),sys2(:,:,k))
    ```

Examples of Operations on LTI Arrays with Single LTI Models

Suppose you want to create an LTI array containing three models, where, for τ in the set $\{1.1, 1.2, 1.3\}$, each model $H_\tau(s)$ has the form

$$H_\tau(s) = \begin{bmatrix} \dfrac{1}{s+\tau} & 0 \\ -1 & \dfrac{1}{s} \end{bmatrix}$$

You can do this efficiently by first setting up an LTI array h containing the SISO models $1/(s+\tau)$ and then using concatenation to form the LTI array H of MIMO LTI models $H_\tau(s)$, $\tau \in \{1.1, 1.2, 1.3\}$. To do this, type

```
tau = [1.1 1.2 1.3];
for i=1:3                    % Form LTI array h of SISO models.
```

```
  h(:,:,i)=tf(1,[1 tau]);
end
H = [h 0;  1 tf(1,[1 0])]; %Concatenation: array h & single models
size(H)

3x1 array of continuous-time transfer functions
Each transfer function has 2 output(s) and 2 input(s).
```

Similarly, you can use `append` to perform the diagonal appending of each model in the SISO LTI array `h` with a fixed single (SISO or MIMO) LTI model.

```
S = append(h,tf(1,[1 3])); % Append a single model to h.
```

specifies an LTI array `S` in which each model has the form

$$S_\tau(s) = \begin{bmatrix} \frac{1}{s+\tau} & 0 \\ 0 & \frac{1}{s+3} \end{bmatrix}$$

You can also combine an LTI array of MIMO models and a single MIMO LTI model using arithmetic operations. For example, if `h` is the LTI array of three SISO models defined above,

```
[h,h] + [tf(1,[1 0]);tf(1,[1 5])]
```

adds the single one-output, two-input LTI model `[1/s 1/(s + 5)]` to every model in the 3-by-1 LTI array of one-output, two-input models `[h,h]`. The result is a new 3-by-2 array of models.

Examples: Arithmetic Operations on LTI Arrays and SISO Models

Using the LTI array of one-output, two-input state-space models `[h,h]`, defined in the previous example,

```
tf(1,[1 3]) + [h,h]
```

adds a single SISO transfer function model to each entry in each model of the LTI array of MIMO models `[h,h]`.

Finally,

```
G = rand(1,1,3,1);
sysa = G + [h,h]
```

adds the array of scalars to each entry of each MIMO model in the LTI array `[h,h]` on a model-by-model basis. This last command is equivalent to the following `for` loop.

```
hh = [h,h];
for k = 1:3
  sysa(:,:,k) = G(1,1,k) + hh(:,:,k);
end
```

Other Operations on LTI Arrays

You can also apply the analysis functions, such as `bode`, `nyquist`, and `step`, to LTI arrays.

5

Customization

Introduction

The Control System Toolbox provides editors that allow you to set properties and preferences in the SISO Design Tool, the LTI Viewer, 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.

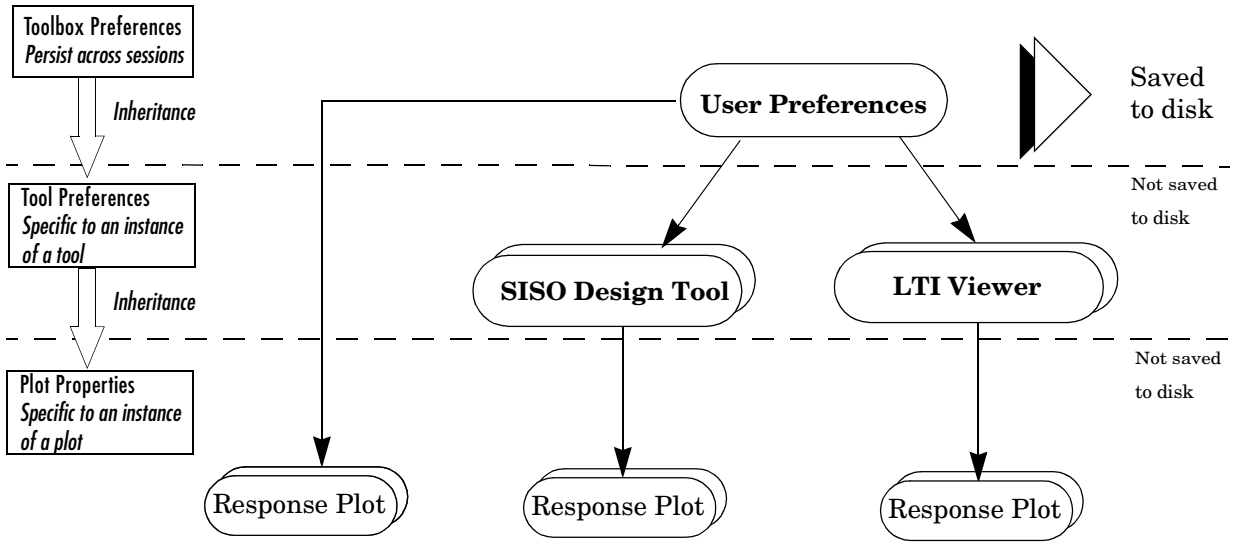
Preferences refers to properties that persist either:

- Within a single session for a specific instance of an LTI Viewer or a SISO Design Tool
- Across Control System Toolbox sessions

The former are called *tool preferences*, the latter *toolbox preferences*.

The Property and Preferences Hierarchy

This diagram explains the hierarchy from properties, which are local, to toolbox preferences, which are global and persist from session to session.



6

Setting Toolbox Preferences

The Toolbox Preferences editor allows you to set plot preferences that will persist from session to session. This is the highest level shown in “The Property and Preferences Hierarchy” on page 5-3.

Toolbox Preferences Editor

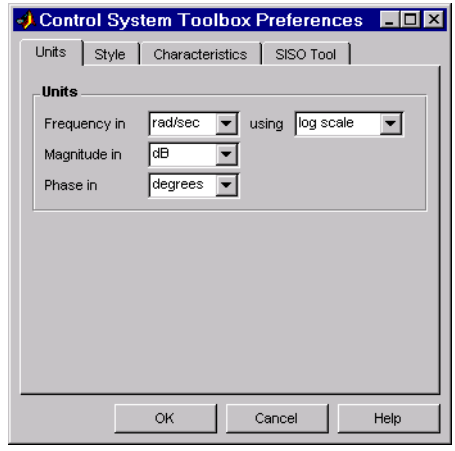
To open the Toolbox Preferences Editor, select **Toolbox Preferences** under the **File** menu of the LTI Viewer or the SISO Design Tool. Alternatively, you can type

```
ctrlpref
```

at the MATLAB prompt.

Note To get help on panes in the Control System Toolbox Preferences editor, you can click on the tabs.

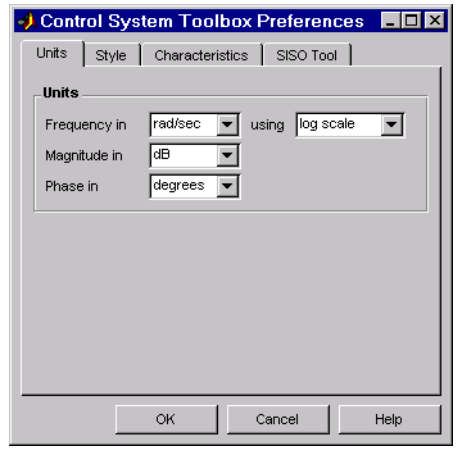
.



The Control System Toolbox Preferences Editor

Units Pane

Note To get help on panes in the Control System Toolbox Preferences editor, click on the tabs.



Use the Units pane to set preferences for the following:

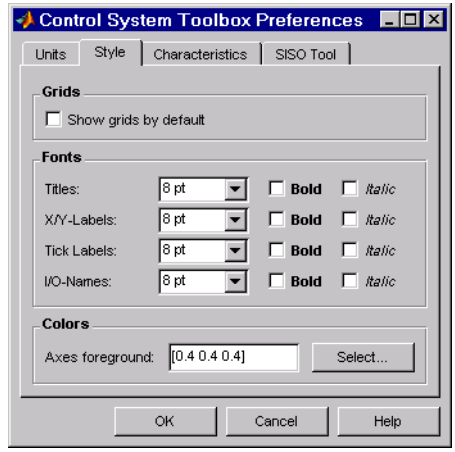
- Frequency — Radians per second (rad/s) or Hertz (Hz)
- Magnitude — Decibels (dB) or absolute value (abs)
- Phase — Degrees or radians

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

Style Pane

Note Click on the tabs to get help on panes in the Control System Toolbox Preferences editor.

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for all plots you create using the Control System Toolbox. This figure shows the Style pane.



You have the following choices:

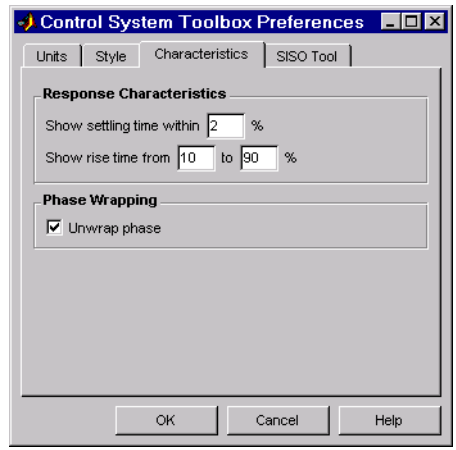
- Grid — Activate grids by default in new plots
- Font preferences — Set the font size, weight (bold), and angle (italic)
- Colors — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and 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** window. See “Select colors” on page 7-9 for more information.

Characteristics Pane

Note Click on the tabs to get help on panes in the Control System Toolbox Preferences editor.

The Characteristics pane has selections for response characteristics and phase wrapping. This figure shows the Characteristics pane with default settings.



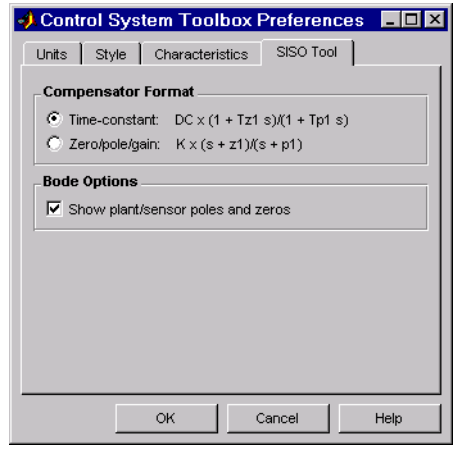
The following are the available options for the Characteristics pane:

- **Response Characteristics**:
  - Specify settling time tolerance — You can set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.
  - Specify rise time boundaries — 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.
- **Phase Wrapping** — By default, the phase is not wrapped. 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 [-180°, 180°).

SISO Tool Pane

Note Click on the tabs below to get help on panes in the Control System Toolbox Preferences editor.

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



You can make the following selections:

- **Compensator Format** — You can select either the time-constant format or the zero/pole/gain format. The time-constant format is

  $$dcgain \times \frac{(1 + Tz_1 s)}{(1 + Tp_1 s)} \ldots$$

  where Tz_1, Tz2, ..., are the zero time constants, and Tp_1, Tp_2, ..., are the pole time constants.

  The zero/pole/gain format is a variation on the time-constant format.

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

  In this case, the gain is compensator gain; 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.

7

Setting Tool Preferences

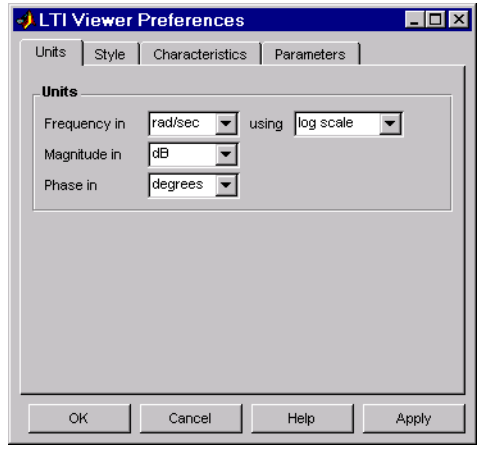
Both the LTI Viewer and the SISO Design Tool have Tool Preferences Editors. These editors comprise the middle layer of “The Property and Preferences Hierarchy” on page 5-3.

Both editors allow you to set default characteristics for specific instances of LTI Viewers and SISO Design Tools. If you open a new instance of either, each defaults to the characteristics specified in the Toolbox Preferences editor.

LTI Viewer Preferences Editor

Select **LTI Viewer Preferences** under the **Edit** menu of the LTI Viewer to open the **LTI Viewer Preferences** editor, which is a tool for customizing various LTI Viewer properties, including units, fonts, and various other viewer characteristics. This figure shows the editor open to its first pane.

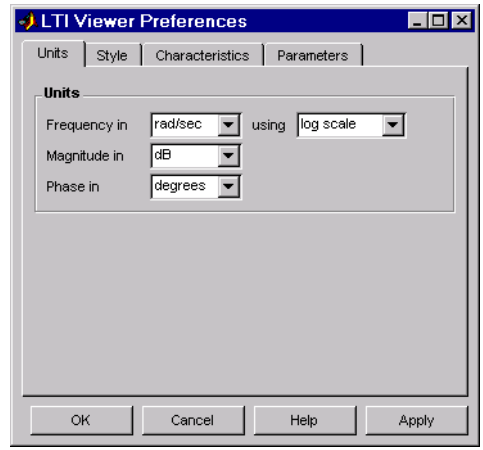
Note Click on the tabs to get help on LTI Viewer Preference editor.



The LTI Viewer Preferences Editor

Units Pane

Note Click on the pane tabs below to get help on LTI Viewer Preference editor panes.



You can select the following on the Units pane:

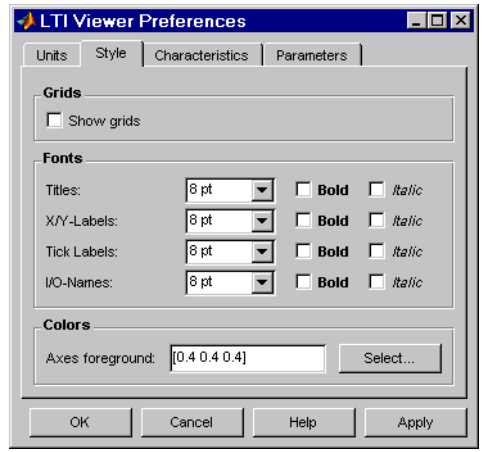
- Frequency — Radians per second (rad/sec) or Hertz (Hz)
- Magnitude — Decibels (dB) or absolute value (abs)
- Phase — Degrees or radians

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

Style Pane

Note Click on the tabs to get help on LTI Viewer Preference editor.

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



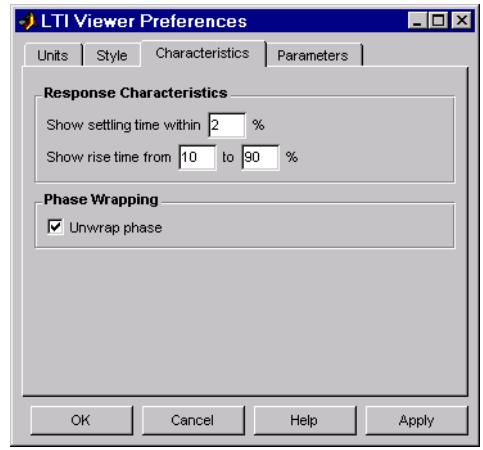
You have the following choices:

- **Grid** — Activate grids for all plots in the LTI Viewer
- **Fonts** — Set the font size, weight (bold), and angle (italic)
- **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 7-9 for more information.

Characteristics Pane

Note Click on the tabs to get help on LTI Viewer Preference editor.

The Characteristics pane, shown below, has selections for response characteristics and phase wrapping.



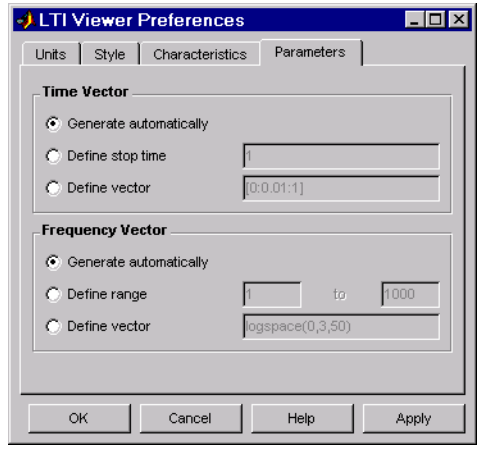
The following choices are available:

- **Response Characteristics**:
  - Specify settling time tolerance — You can set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.
  - Specify rise time boundaries — 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.
- **Phase Wrapping** — By default, the phase is not wrapped. 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 [-180°, 180°).

Parameters Panpanee

Note Click on the tabs to get help on LTI Viewer Preference editor.

Use the Parameters pane, shown below, to specify input vectors for time and frequency simulation.



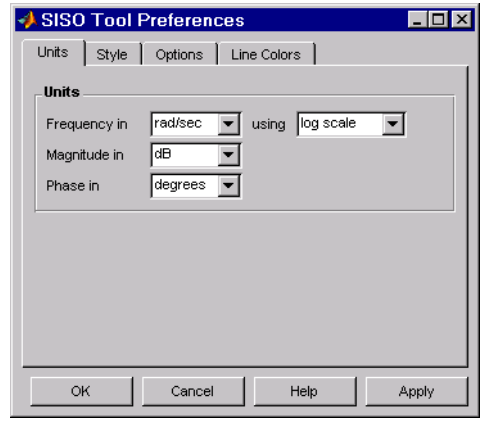
The defaults are to generate time and frequency vectors for your plots automatically. You can, however, override the defaults as follows:

- **Time Vector**:
  - Define stop time — Specify the final time value for your simulation
  - Define vector — Specify the time vector manually using equal-sized time steps
- **Frequency Vector**:
  - Define range — Specify the bandwidth of your response. Whether it's in rad/sec or Hz depends on the selection you made in the Units pane.
  - Define vector — Specify the vector for your frequency values. Any real, positive, strictly monotonically increasing vector is valid.

SISO Tool Preferences Editor

Note Click on the tabs to get help on SISO Tool Preference editor.

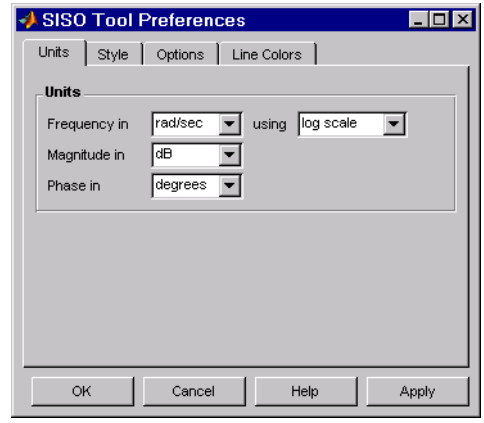
To open the **SISO Tool Preferences** editor, select **SISO Tool Preferences** from the **Edit** menu of the SISO Design Tool. This window opens.



The SISO Tool Preferences Editor

Units Pane

Note Click on the pane tabs below to get help on SISO Tool Preference editor panes.



The Units pane has settings for the following units:

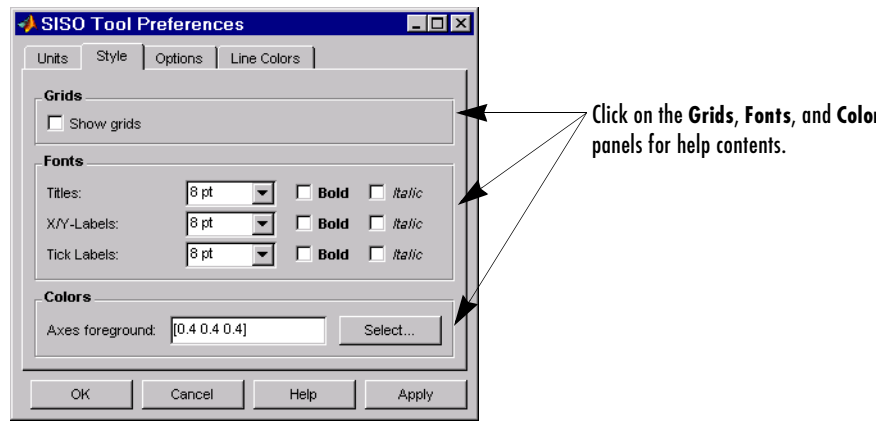
- Frequency — Radians per second (rad/sec) or Hertz (Hz)
- Magnitude — Decibels (dB) or absolute value (abs)
- Phase — Degrees or radians

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

Style Pane

Note Click on the tabs to get help on SISO Tool Preference editor.

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for all plots in the SISO Design Tool. This figure shows the Style pane.



Grids Panel

Select the box to activate grids for all plots in the SISO Design Tool

Fonts Panel

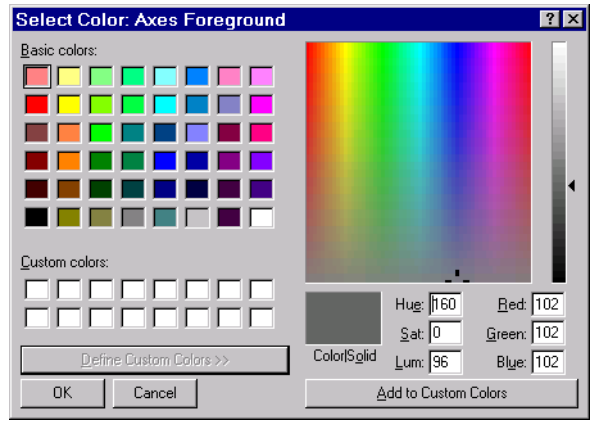
Set the font size, weight (bold), and angle (italic) by using the menus and check boxes.

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 **Define Custom Colors** button. This extends the Select Color window, as shown in this figure.



You can pick colors from the color spectrum located in the upper right corner of the window. To select a custom color, follow these steps:

1 Place your cursor at a point in the color spectrum that has a color you want to define.

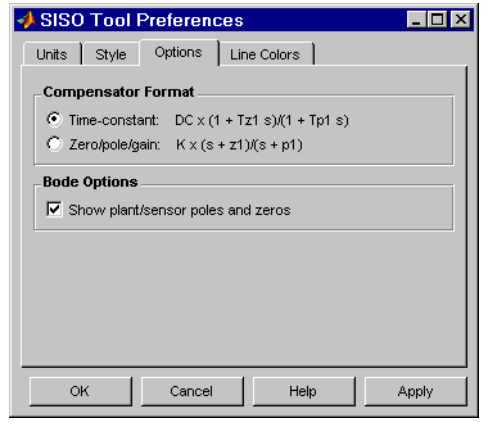
2 Left-click. Notice that the hue, saturation, luminescence (lum.), red, green, and blue fields specify the numerical values for the selected color.

3 Press **Add to Custom Colors**. This adds the selected color to the row of boxes labeled **Custom Color.** You can now use this color just like the basic colors.

Options Pane

Note Click on the tabs to get help on SISO Tool Preference editor.

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



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 compenator transfer function of the form

  $$dcgain \times \frac{(1 + Tz_1 s)}{(1 + Tp_1 s)} \ldots$$

  where Tz_1, Tz_2, ..., are the zero time constants, and Tp_1, Tp_2, ..., are the pole time constants.

  The natural frequency format is

  $$dcgain \times \frac{(1 + s/\omega_{z_1})}{(1 + s/(\omega_{p_1}))} \ldots$$

  where ω_{z1},ω_{z2}, ... and ω_{p1}, ω_{p2}, ..., are the natural frequencies of the zeros and poles, respectively.

  The zero/pole/gain format is

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

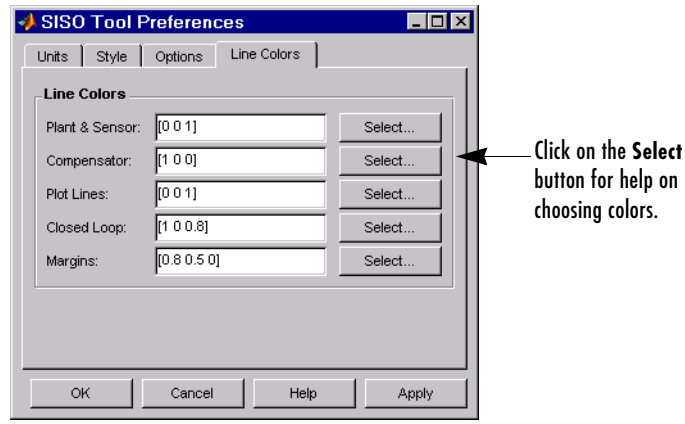
  where z_1, z_2, ... and p_1, p_2, ..., are the zero and pole locations, respectively.

- **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 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

Note Click on the pane tabs below to get help on SISO Tool Preference editor.

The Line Colors pane, shown below, has selections for specify the colors of the lines in the response plots of the SISO Design Tool.



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 7-9 for more information.

8

Customizing Response Plot Properties

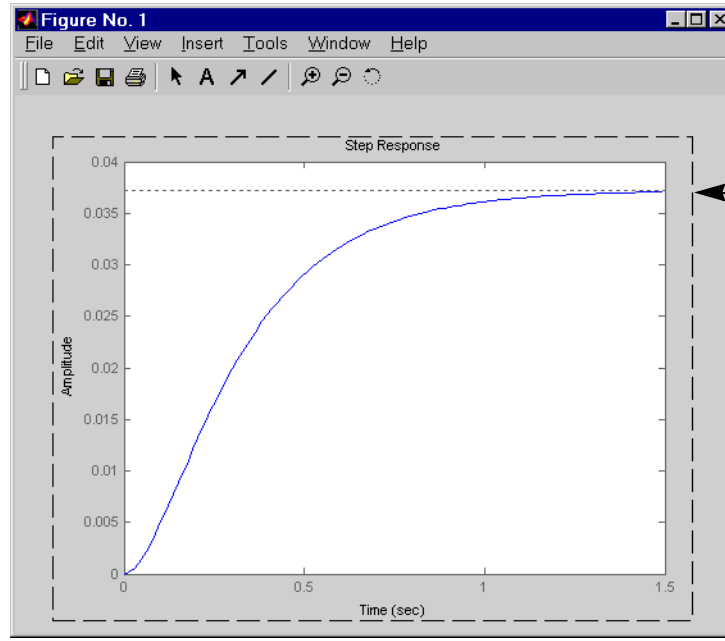
The lowest level of "The Property and Preferences Hierarchy" on page 5-3 is setting response plot properties. If you have created a response plot, there are two ways to open the Property Editor:

- 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 this commands.

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

This creates a step plot. Select **Properties** from the right-click window. Note that when you open the **Property Editor**, a black dashed box appears around the step response, as this figure shows.

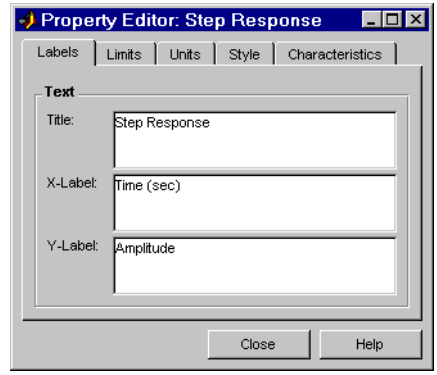


A SISO System Step Response

Response Plots Property Editor

Note Click on the tabs to get help on panes in the Property Editor.

This figure shows the **Property Editor** dialog box for this step response.



The Property Editor for the Step Response

In general, you can change the following properties of response plots:

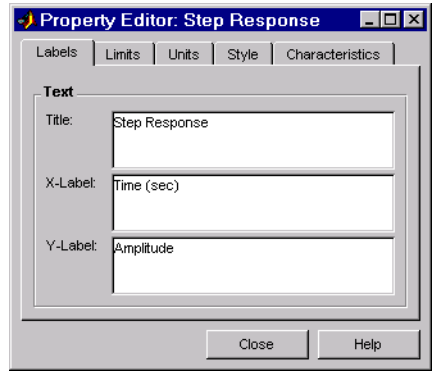
- **Labels** — Titles and X- and Y-labels
- **Limits** — Numerical ranges of the X and Y axes
- **Units** — Where applicable (e.g., rad/s to Hertz). If you cannot customize units, as is the case with step responses, the Property Editor will display that no units ar available for the selected plot.
- **Style** — Show a grid and adjust font properties, such as font size, bold and italics
- **Characteristics** — Where applicable, these include peak response, settling time, phase and gain margins, etc. Plot characteristics change with each plot response type. The Property Editor displays only the characteristics 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

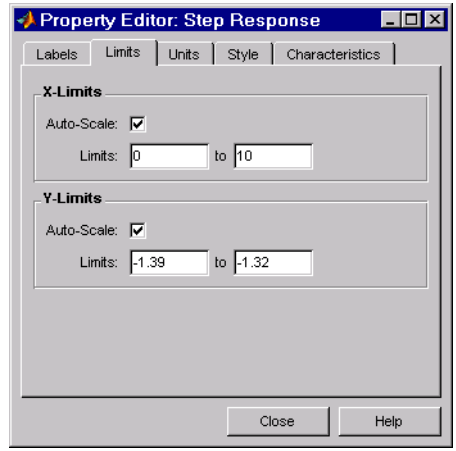
Note Click on the tabs below to get help on the Property Editor.



To specify new text for plot titles and axis labels, type the new string in the field next to the label you want to change. Note that the label changes immediately as you type, so you can see how the new text looks as you are typing.

Limits Pane

Note Click on the tabs to get help on the Property Editor.

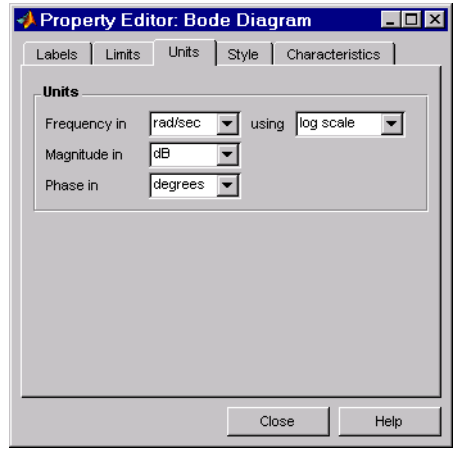


The Control System Toolbox selects default values for the axes limits to make sure that the maximum and minimum x and y values are displayed. If you want to override the default settings, change the values in the Limits fields. The **Auto-Scale** box automatically clears if you click on a different field. The new limits appear immediately in the response plot.

To reestablish the default values, select the **Auto-Scale** box again.

Units Pane

Note Click on the tabs to get help on the Property Editor.



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.

Note that for step and impulse responses, there are no alternate units available (only time and amplitude are possible in the toolbox). This table lists the options available for the other response objects. Use the menus to toggle between units.

Table 8-1: Optional Unit Conversions for Response Plots

Response Plot	Unit Conversions
Bode and Bode Magnitude	Frequency in rad/s or Hertz (Hz) using logarithmic or linear scale Magnitude in decibels (dB) or the absolute value Phase in degrees or radians
Impulse	None
Nichols Chart and Nyquist Diagram	Frequency in rad/s or Hertz Magnitude in decibels or the absolute value Phase in degrees or radians
Pole/Zero Map	Frequency in rad/s or Hertz

Table 8-1: Optional Unit Conversions for Response Plots

Response Plot	Unit Conversions
Singular Values	Frequency in rad/s or Hertz using logarithmic or linear scale Magnitude in decibels or the absolute value
Step	None

Style Pane

Note Click on the tabs to get help on the Property Editor.

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for response plots.

You have the following choices:

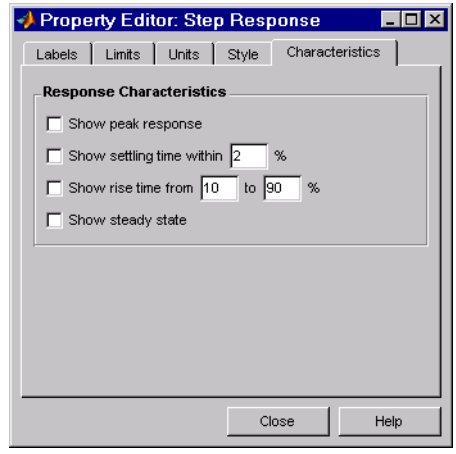
- **Grid** — Activate grids by default in new plots
- **Fonts** — Set the font size, weight (bold), and angle (italic)
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector

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** window. See “Select colors” on page 7-9 for more information.

Characteristics Pane

Note Click on the tabs to get help on the Property Editor.



The **Characteristics** pane allows you to customize response characteristics for plots. Each response plot has its own set of characteristics; the table below lists

them. Use the check boxes to activate the feature and the fields to specify rise or settling time percentages.

Table 8-2: Response Characteristic Options for Response Plots

Plot	Customizable Feature
Bode Diagram	Show peak response Show minimum stability margins Show all stability margins Unwrap phase (default is wrapped)
Bode Magnitude	Show peak response
Impulse	Show peak response Show settling time within *xx*% (specify the percentage)
Nichols Chart	Show peak response Show minimum stability margins Show all stability margins Unwrap phase (default is wrapped)
Nyquist Diagram	Show peak response Show minimum stability margins Show all stability margins
Pole/Zero Map	None
Sigma	Show peak response
Step	Show peak response Show settling time within *xx*% (specify the percentage) Show rise time from *xx* to *yy*% (specify the percentages) Show steady state

Property Editing for Subplots

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 dashed line appear around the step response, indicating that it is the active plot for the editor. To switch to the lower (impulse response) plot, just click once in the impulse response plot region. The dashed box switches to the impulse response, and the **Property Editor** updates as well.

Customizing Plots Inside the SISO Design Tool

Customizing plots inside the SISO Design Tool is similar to how you customize any response plot. The Control System Toolbox provides the following property editors 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

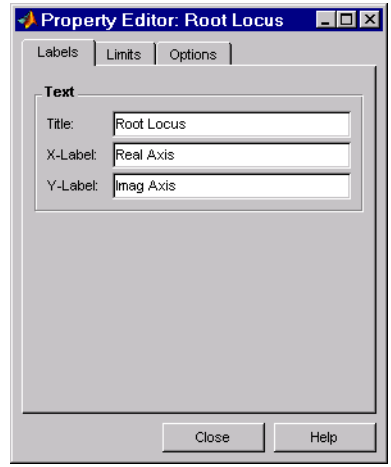
There are three ways to open the Property Editor for root locus plots:

- 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

---

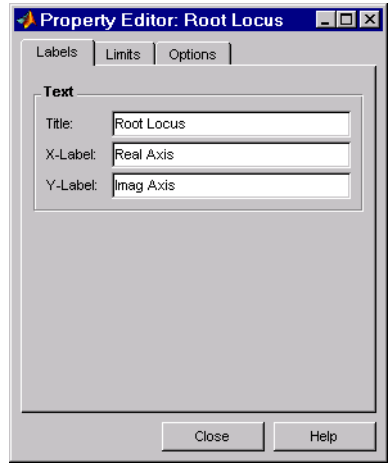
Note Click on the tabs to get help on the Root Locus Property Editor.

This figure shows the **Property Editor: Root Locus** window.



Labels Pane

Note Click on the tabs to get help on the Root Locus Property Editor.

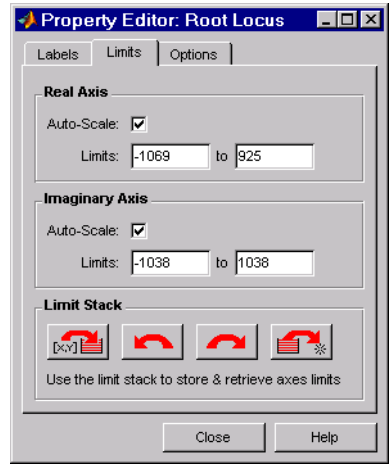


You can use the **Label** pane to specify plot titles and axis labels. To specify a new label, type the string in the appropriate field. The root locus plot automatically updates.

Limits Pane

Note Click on the pane tabs below to get help on panes in the Root Locus Property Editor.

The SISO Design Tool specifies default values for the real and imaginary axes ranges to make sure that all the poles and zeros in your model appear in the root locus plot. Use the Limits pane, shown below, to override the default settings.



To change the limits, specify the new limits in the real and imaginary axes **Limits** fields. The **Auto-Scale** check box automatically clears once you click in a different field. Your root locus diagram updates immediately. If you want to reapply the default limits, select the **Auto-Scale** check boxes again.

The Limit Stack panel 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

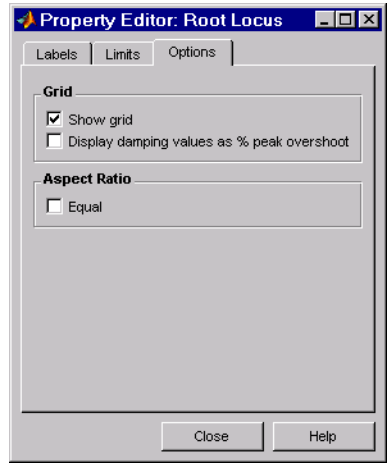
 — Remove the current limits from the stack

Using these buttons, you can store and retrieve any number of saved custom axes limits.

Options Pane

Note Click on the tabs to get help on the Root Locus Property Editor.

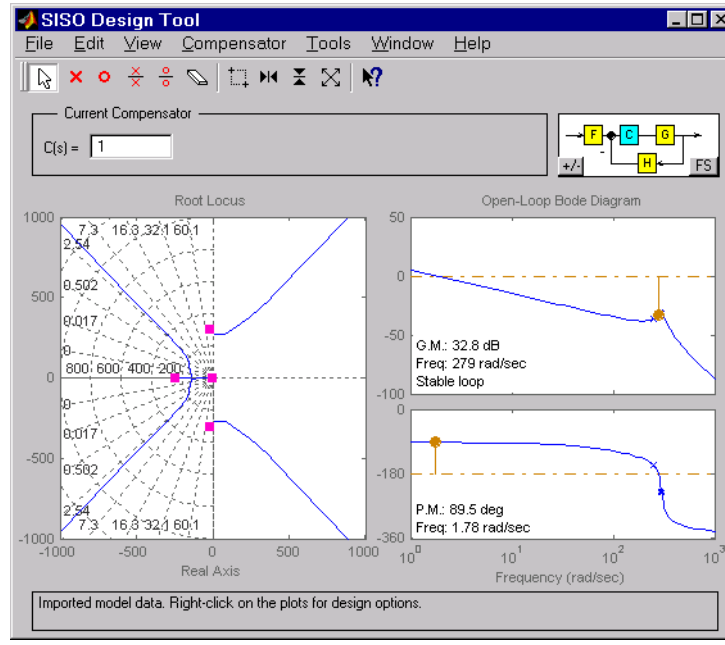
The Options pane contains settings for adding a grid and changing the plot's aspect ratio. This figure shows the Options pane.



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.



Displaying Damping Ratio Values

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** panel, the x and y-axes are set to equal limit values.

Open-Loop Bode Property Editor

The Property Editor for open-loop Bode diagrams is identical to the one for root locus, with one exception, the Options pane. Also, note the prefilter and open-loop Bode diagram property editors are identical.

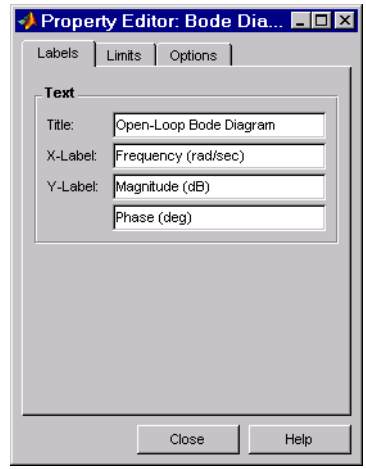
As is the case with the root locus Property Editor, there are three ways to open the Bode diagram property editor:

- 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

---

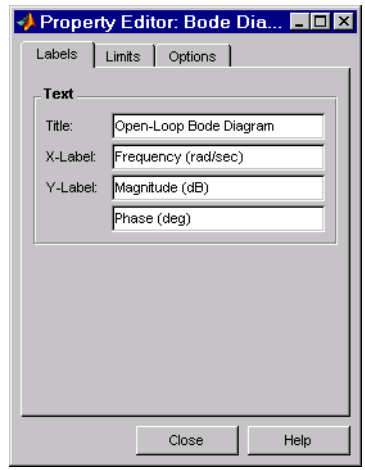
Note Click on the tabs to get help on the Open-Loop Bode Property editor.

This figure shows the **Property Editor: Open-Loop Bode** editor.



Labels Pane

Note Click on the tabs to get help on the Open-Loop Bode Property editor.

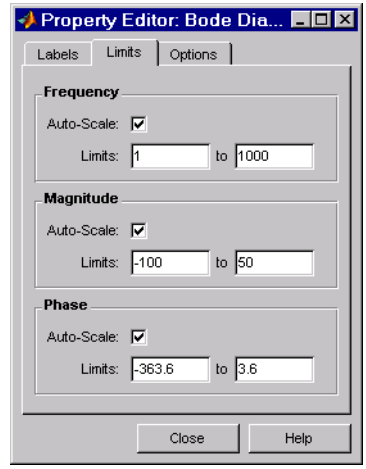


You can use the **Label** pane to specify plot titles and axis labels. To specify a new label, type the string in the appropriate field. The Bode diagram automatically updates.

Limits Pane

Note Click on the tabs to get help on the Open-Loop Bode Property editor.

The Control System Toolbox sets default limits for the frequency, magnitude, and phase scales for your plots. Use the Limits pane to override the default values.



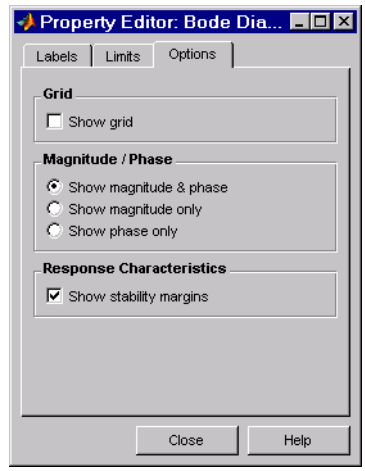
To change the limits, specify the new values in the **Limits** fields for frequency, magnitude, and phase. The **Auto-Scale** check box automatically deactivates once you click in a different field. The Bode diagram updates immediately.

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

Options Pane

Note Click on the tabs to get help on the Open-Loop Bode Property editor.

This figure shows the Options pane for Bode diagrams.



The following options are available from this pane:

- **Grid** — Select **Show grid** to display grid lines.
- **Magnitude/Phase** — There are three radio buttons; you can toggle between the following displays:
  - **Show magnitude & phase**
  - **Show magnitude only**
  - **Show phase only**
- **Response Characteristics** — Select **Show stability margins** to display the phase and gain margins on your Bode diagram. The margins appear as brown stems, and the Bode diagram displays the numerical values of the margins in one of the bottom corners of the gain and phase plots.

  The Bode diagram in “Displaying Damping Ratio Values” on page 8-15, has the stability margins displayed.

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:

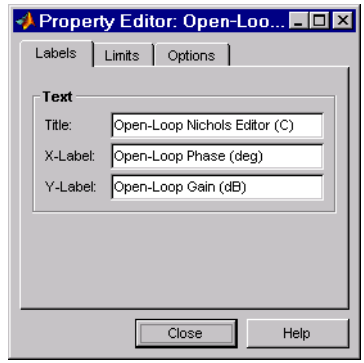
- Double-click in the Nichols plot away from the curve
- Select **Properties** from the right-click menu

- Select **Open-Loop Nichols** and then **Properties** from **Edit** in the menu bar

---

Note Click on the tabs to get help on the Open-Loop Nichols Property editor.

This figure shows the **Property Editor: Open-Loop Nichols** editor.



Labels Pane

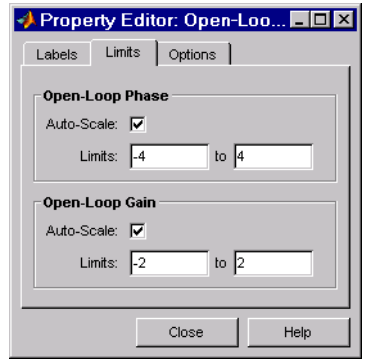
Note Click on the tabs to get help on the Open-Loop Nichols Property editor.

You can use the Label pane to specify plot titles and axis labels. To specify a new label, type the string in the appropriate field. The Nichols plot automatically updates.

Limits Pane

Note Click on the tabs to get help on the Open-Loop Nichols Property editor.

The Control System Toolbox sets default limits for the frequency, magnitude, and phase scales for your plots. Use the Limits pane to override the default values.



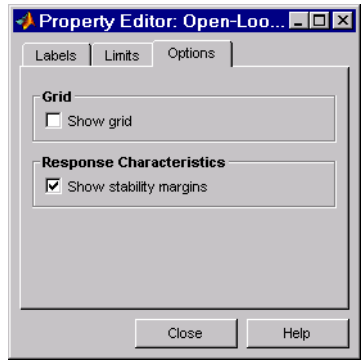
To change the limits, specify the new values in the **Limits** fields for open-loop phase and/or gain. The **Auto-Scale** check box automatically deactivates once you click in a different field. The Nichols plot updates immediately.

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

Options Pane

Note Click on the tabs to get help on the Open-Loop Nichols Property editor.

This figure shows the Options pane for Bode diagrams.



The following options are available from this pane:

- **Grid** — Select **Show grid** to display grid lines.
- **Response Characteristics** — Select **Show stability margins** to display the phase and gain margins on your Nichols plot.

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 8-16 for a description of the features of this editor.

9

Design Case Studies

This chapter contains four detailed case studies of control system design and analysis using the Control System Toolbox:

- "Yaw Damper for a 747 Jet Transport" — Illustrating the classical design process
- "Hard-Disk Read/Write Head Controller" — Illustrating classical digital controller design
- "LQG Regulation: Rolling Mill Example" — Using linear quadratic Gaussian techniques to regulate the beam thickness in a steel rolling mill
- "Kalman Filtering"— Kalman filtering that illustrates both steady-state and time-varying Kalman filter design and simulation

Demonstration files for these case studies are available as `jetdemo.m`, `diskdemo.m`, `milldemo.m`, and `kalmdemo.m`. To run any of these demonstrations, type the corresponding name at the command line, for example,

```
jetdemo
```

Yaw Damper for a 747 Jet Transport

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.

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 = [ 0.0729    0.0000
     -4.7500    0.00775
      .15300    0.1430
           0         0];

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

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

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`. MATLAB responds with

```
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   0.0729            0
          yaw    -4.75      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.
```

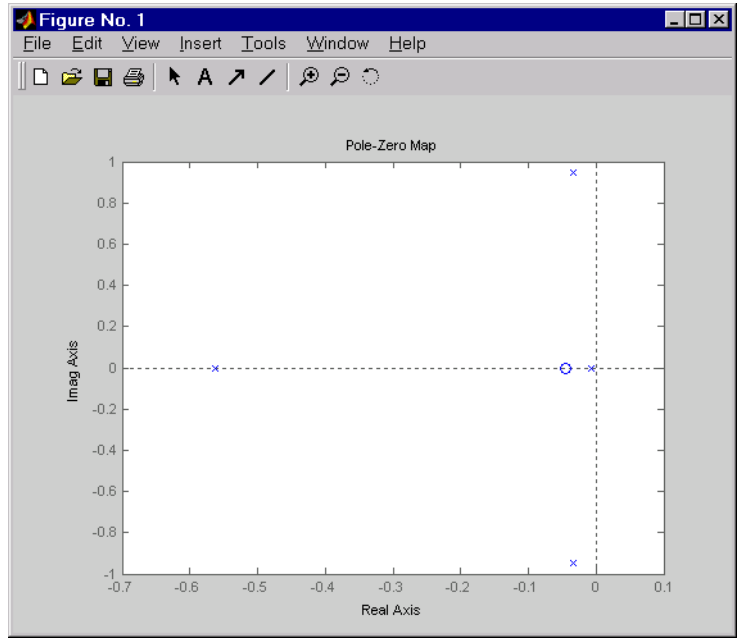
The model has two inputs and two outputs. The units are radians for `beta` (sideslip angle) and `phi` (bank angle) and radians/sec for `yaw` (yaw rate) and `roll` (roll rate). The rudder and aileron deflections are in radians as well.

Computing Open-Loop Eigenvalues

Compute the open-loop eigenvalues and plot them in the *s* -plane.

```
damp(sys)

Eigenvalue                  Damping        Freq. (rad/s)

 -7.28e-003                 1.00e+000      7.28e-003
 -5.63e-001                 1.00e+000      5.63e-001
 -3.29e-002 + 9.47e-001i    3.48e-002      9.47e-001
 -3.29e-002 - 9.47e-001i    3.48e-002      9.47e-001
```

```
pzmap(sys)
```



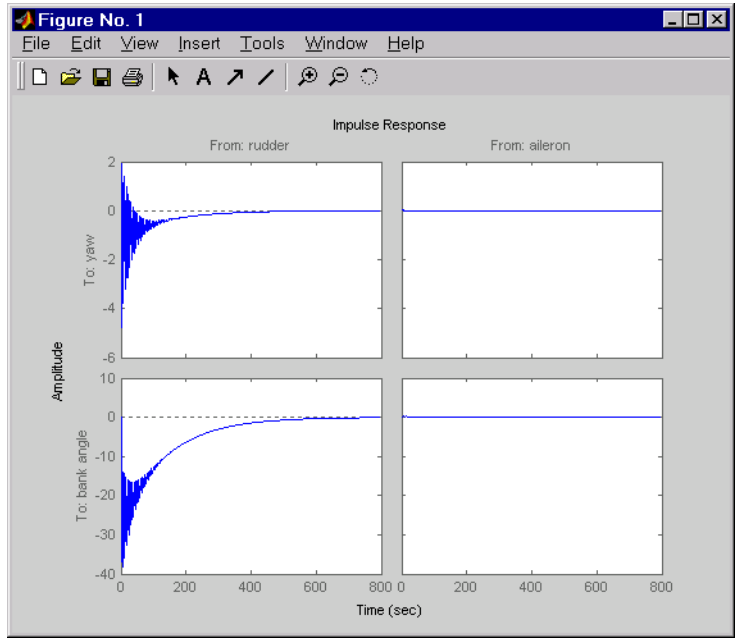
This model has one pair of lightly damped poles. They correspond to what is called the “Dutch roll mode.”

Suppose you want to design a compensator that increases the damping of these poles, so that the resulting complex poles have a damping ratio $\zeta > 0.35$ with natural frequency $\omega_n < 1$ rad/sec. You can do this using the Control System toolbox analysis tools.

Open-Loop Analysis

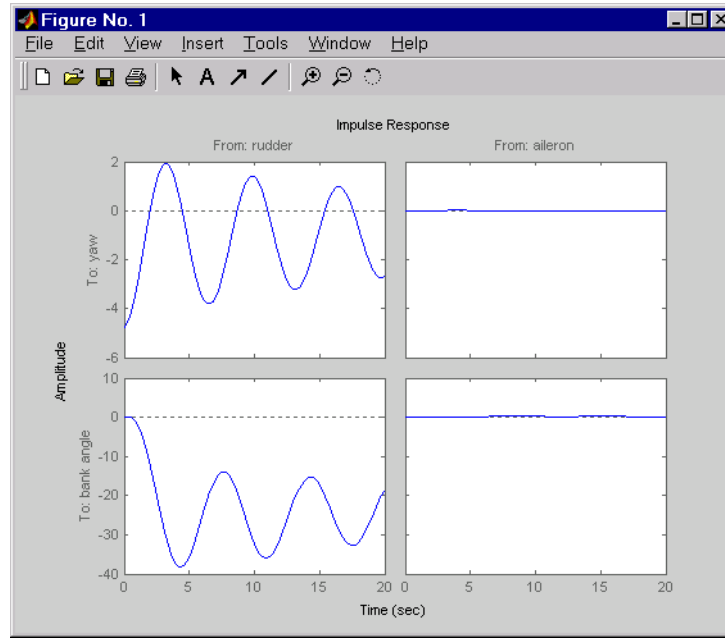
First, perform some open-loop analysis to determine possible control strategies. Start with the time response (you could use `step` or `impulse` here).

```
impulse(sys)
```

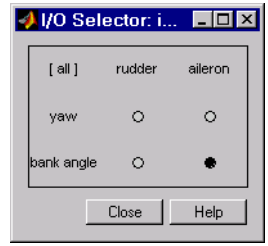


The impulse response confirms that the system is lightly damped. But the time frame is much too long because the passengers and the pilot are more concerned about the behavior during the first few seconds rather than the first few minutes. Next look at the response over a smaller time frame of 20 seconds.

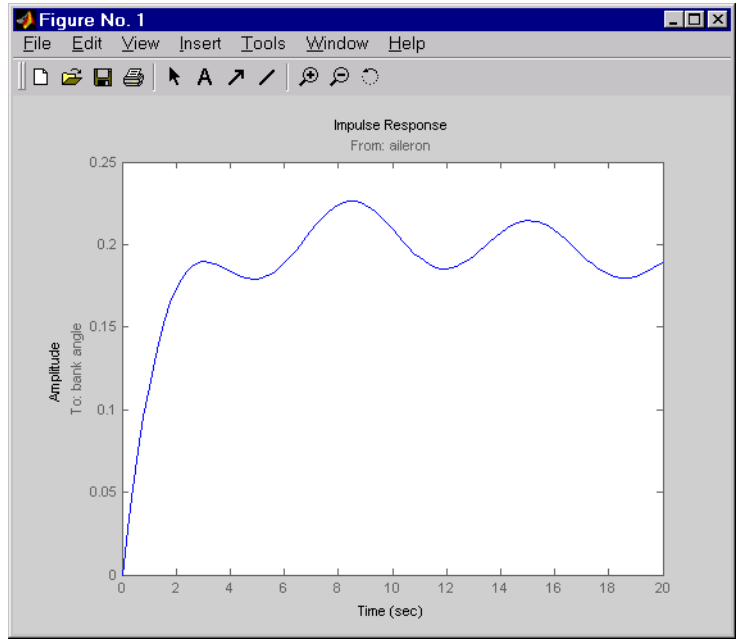
```
impulse(sys,20)
```



Look at the plot from aileron (input 2) to bank angle (output 2). To show only this plot, right-click and choose **I/O Selector**, then click on the (2,2) entry. The I/O Selector should look like this.



The new figure is shown below.

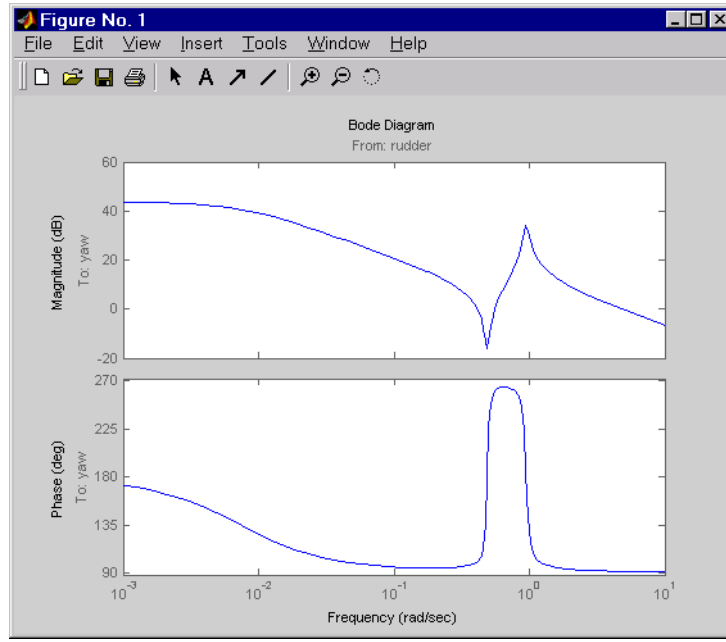


The aircraft is oscillating around a nonzero bank angle. Thus, the aircraft is turning in response to an aileron impulse. This behavior will prove important later in this case study.

Typically, yaw dampers are designed using the yaw rate as sensed output and the rudder as control input. Look at the corresponding frequency response.

```
sys11=sys('yaw','rudder') % Select I/O pair.
```

```
bode(sys11)
```



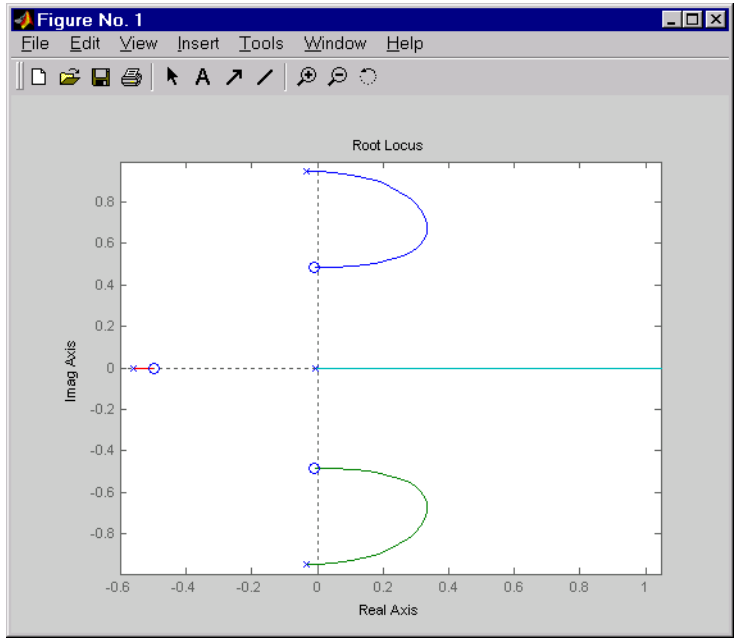
From this Bode diagram, you can see that the rudder has significant effect around the lightly damped Dutch roll mode (that is, near $\omega = 1$ rad/sec).

Root Locus Design

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

```
% Plot the root locus for the rudder to yaw channel
```

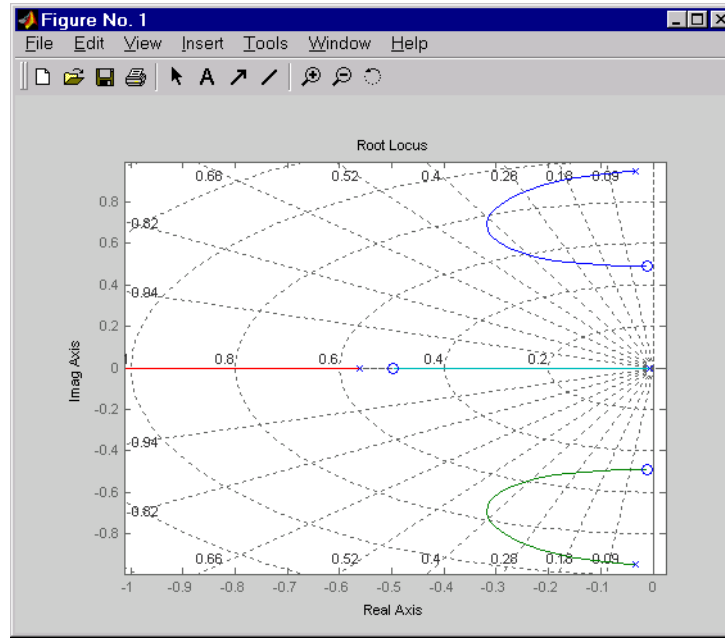
```
rlocus(sys11)
```



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

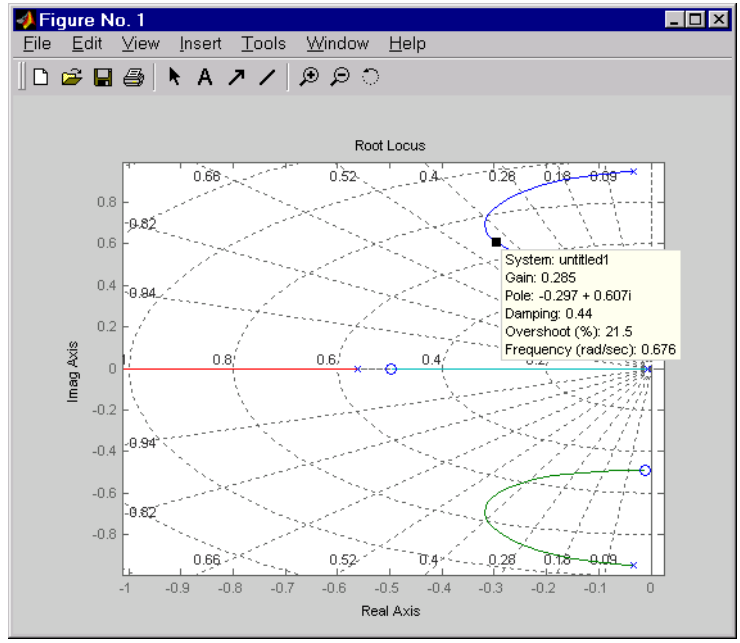
```
rlocus(-sys11)
```

```
sgrid
```



This looks better. By using simple feedback, you can achieve a damping ratio of $\zeta = 0.45$. Click on the blue curve and move the data marker to track the

gain and damping values. To achieve a 0.45 damping ratio, the gain should be about 2.85. This figure shows the data marker with similar values.

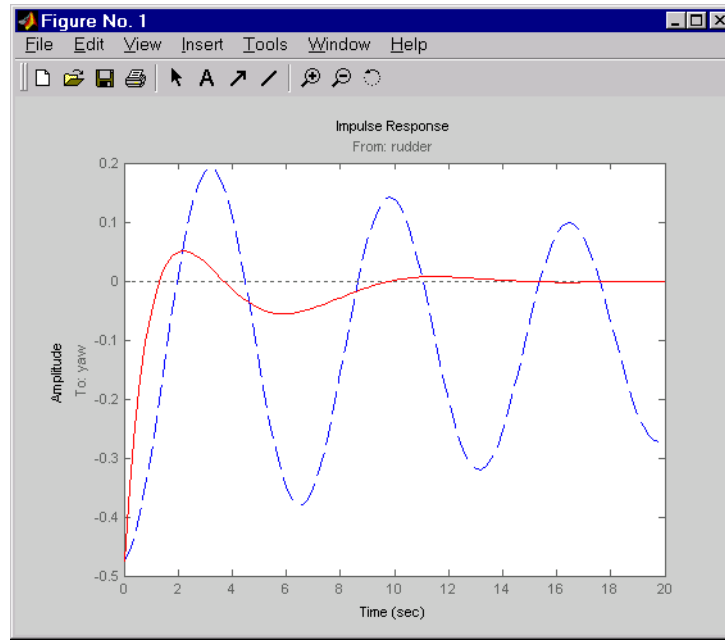


Next, close the SISO feedback loop.

```
K = 2.85;
cl11 = feedback(sys11,-K);   % Note: feedback assumes negative
                             % feedback by default
```

Plot the closed-loop impulse response for a duration of 20 seconds, and compare it to the open-loop impulse response.

```
impulse(sys11,'b--',cl11,'r',20)
```



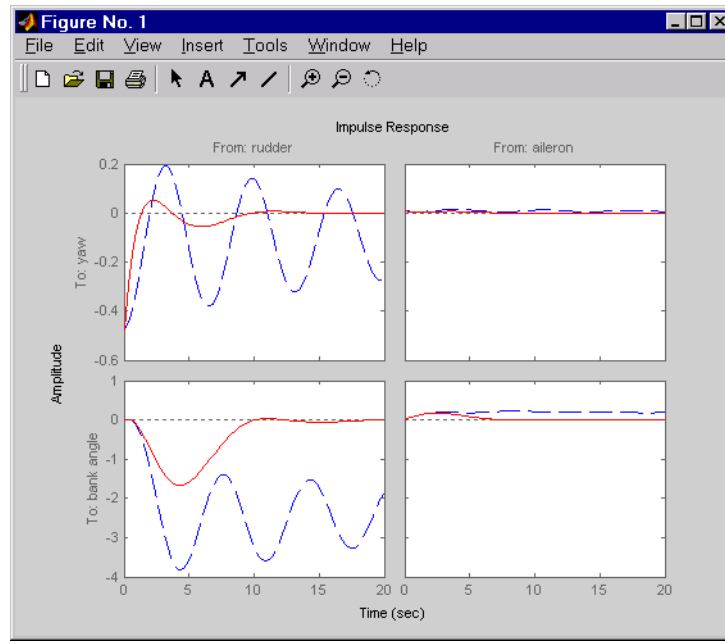
The closed-loop response settles quickly and does not oscillate much, particularly when compared to the open-loop response.

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

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

Eigenvalue            Damping     Freq. (rad/s)

-3.42e-001                 1.00e+000      3.42e-001
-2.97e-001 + 6.06e-001i    4.40e-001      6.75e-001
-2.97e-001 - 6.06e-001i    4.40e-001      6.75e-001
-1.05e+000                 1.00e+000      1.05e+000
```

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

$$H(s) = \frac{s}{s+a}$$

The washout filter places a zero at the origin, which constrains the spiral mode pole to remain near the origin. We choose $a = 0.2$ for a time constant of five seconds and use the root locus technique to select the filter gain `H`. First specify the fixed part $s/(s+a)$ of the washout by

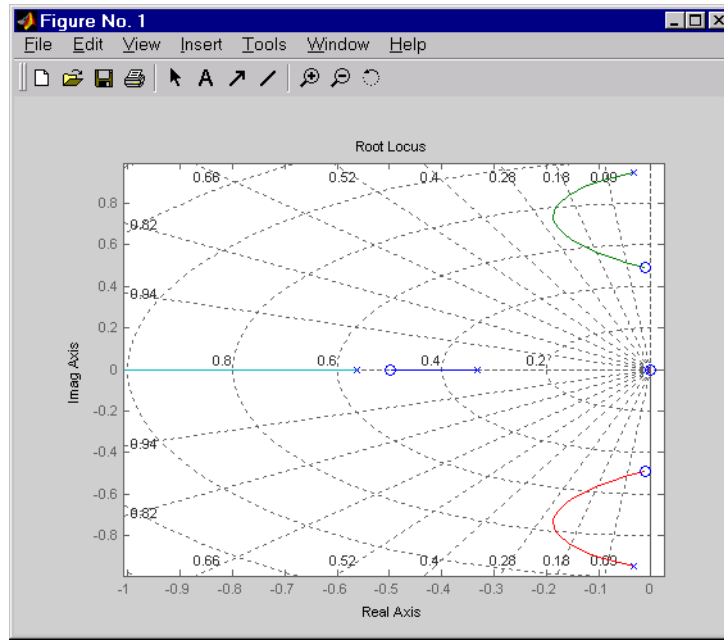
```
H = zpk(0,-0.2,1);
```

Connect the washout in series with the design model `sys11` (relation between input 1 and output 1) to obtain the open-loop model

```
oloop = H * sys11;
```

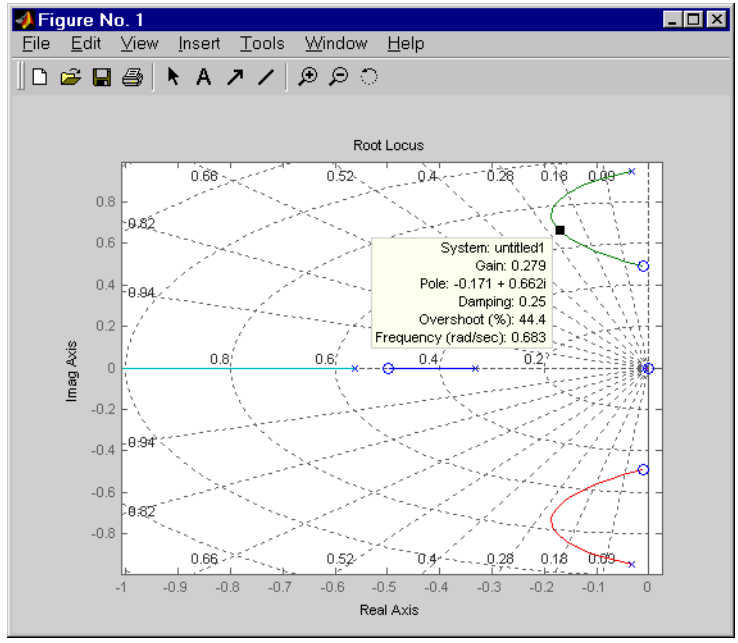
and draw another root locus for this open-loop model.

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



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

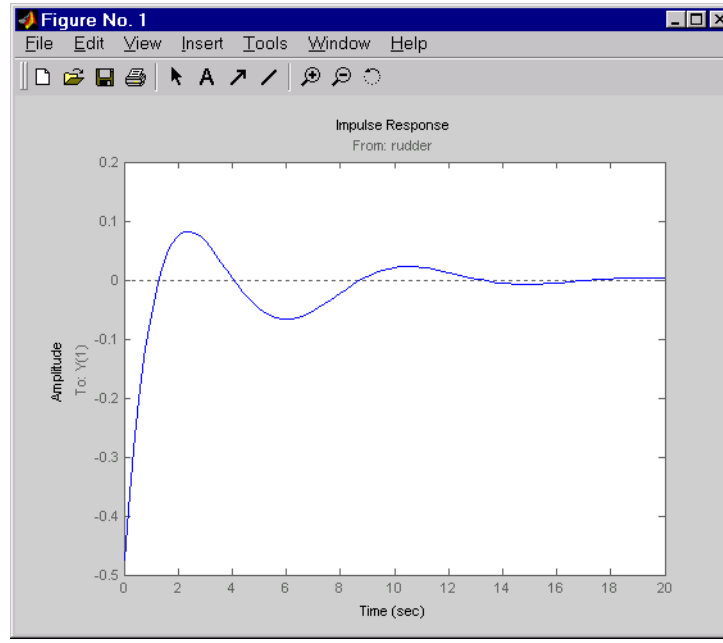
This figure shows a data marker at the maximum damping ratio; the gain is approximately 2.07.



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

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

```
impulse(cl11,20)
```



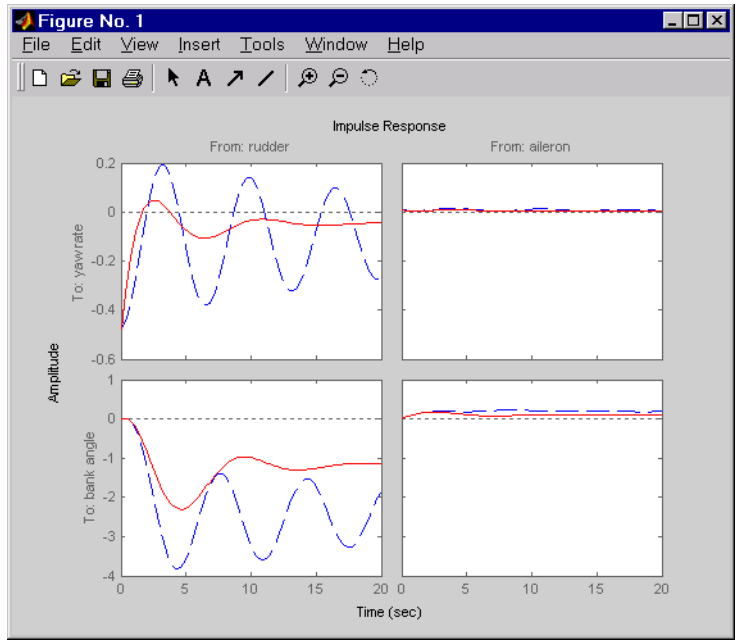
The response settles nicely but has less damping than your previous design. Finally, you can verify that the washout filter has fixed the spiral mode problem. First form the complete washout filter $kH(s)$ (washout + gain).

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

Then close the loop around the first I/O pair of the MIMO model `sys` and simulate the impulse response.

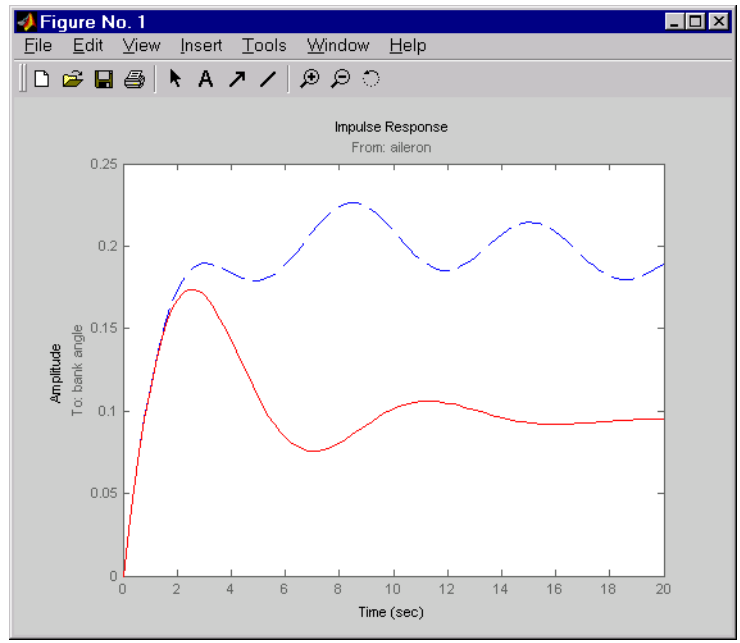
```
cloop = feedback(sys,WOF,1,1);

% Final closed-loop impulse response
```

```
impulse(sys,'b--',cloop,'r',20)
```



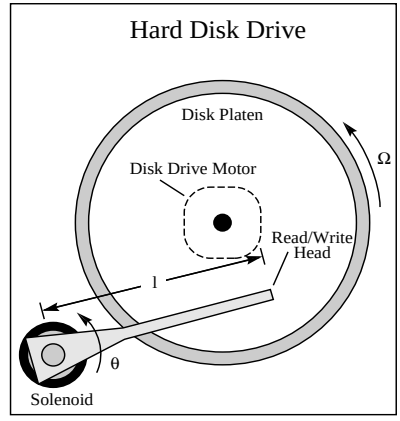
The bank angle response (output 2) due to an aileron impulse (input 2) now has the desired nearly constant behavior over this short time frame. To inspect the

response more closely, use the I/O Selector in the right-click menu to select the (2,2) I/O pair.



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

Hard-Disk Read/Write Head Controller



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.

Deriving the 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 $J = 0.01\,\text{kg m}^2$, $C = 0.004\,\text{Nm/(rad/sec)}$, $K = 10\,\text{Nm/rad}$, and $K_i = 0.05\,\text{Nm/rad}$, 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)
```

MATLAB responds with

```
Transfer function:
         0.05
-----------------------
0.01 s^2 + 0.004 s + 10
```

Model Discretization

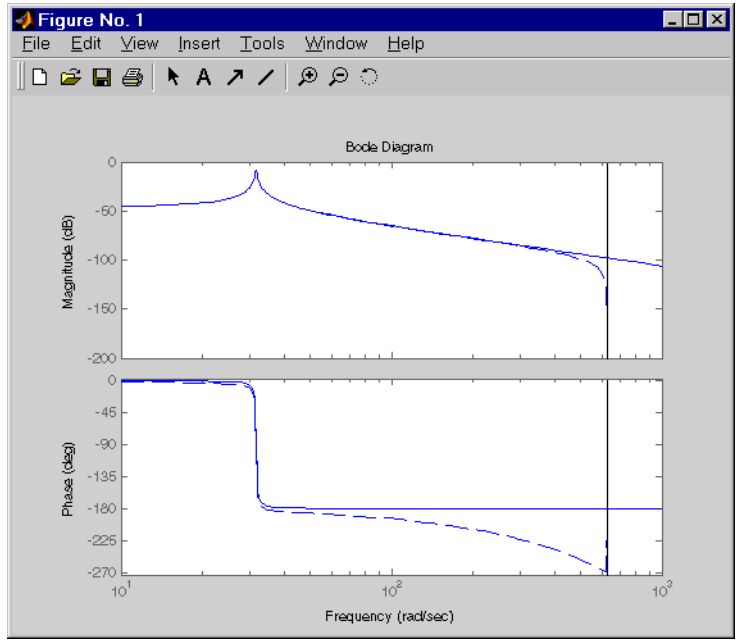
The task here is to design a digital controller that provides accurate positioning of the read/write head. The design is performed in the digital domain. First, discretize the continuous plant. Because our plant will be equipped with a digital-to-analog converter (with a zero-order hold) connected to its input, use `c2d` with the `'zoh'` discretization method. Type

```
Ts = 0.005;     % sampling period = 0.005 second
Hd = c2d(H,Ts,'zoh')
```

```
Transfer function:
6.233e-05 z + 6.229e-05
-----------------------
 z^2 - 1.973 z + 0.998

Sampling time: 0.005
```

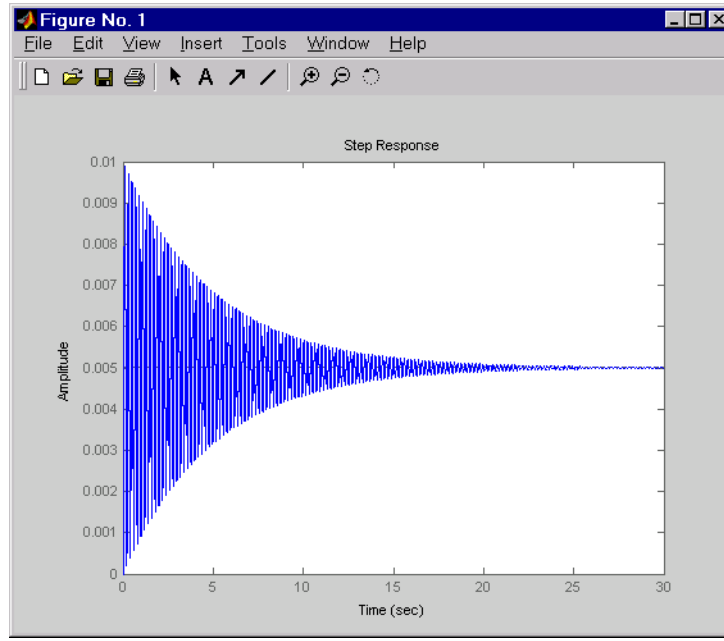
You can compare the Bode plots of the continuous and discretized models with

```
bode(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)
```

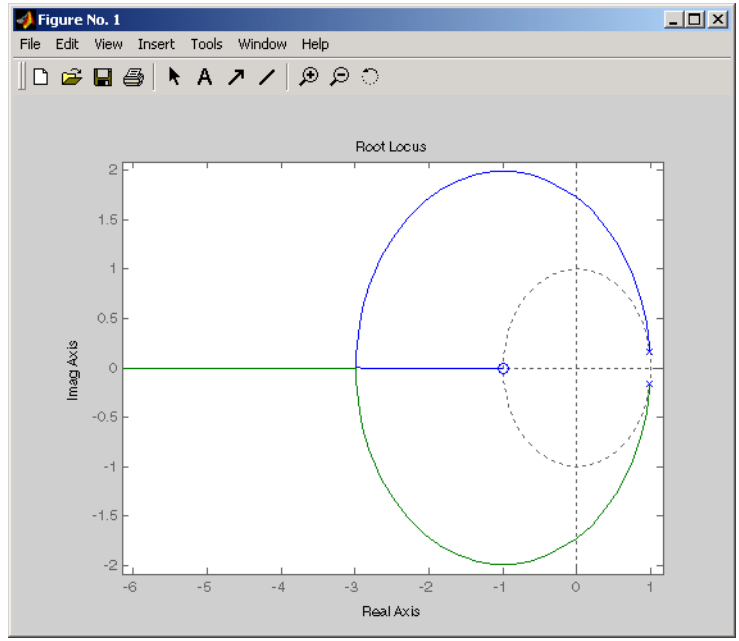
```
    Eigenvalue          Magnitude    Equiv. Damping    Equiv. Freq.

9.87e-01 + 1.57e-01i    9.99e-01       6.32e-03         3.16e+01
9.87e-01 - 1.57e-01i    9.99e-01       6.32e-03         3.16e+01
```

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.

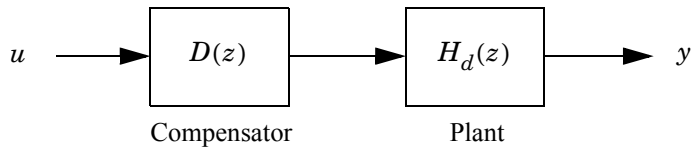
Adding a Lead Network

Try the compensator

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

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

The corresponding open-loop model



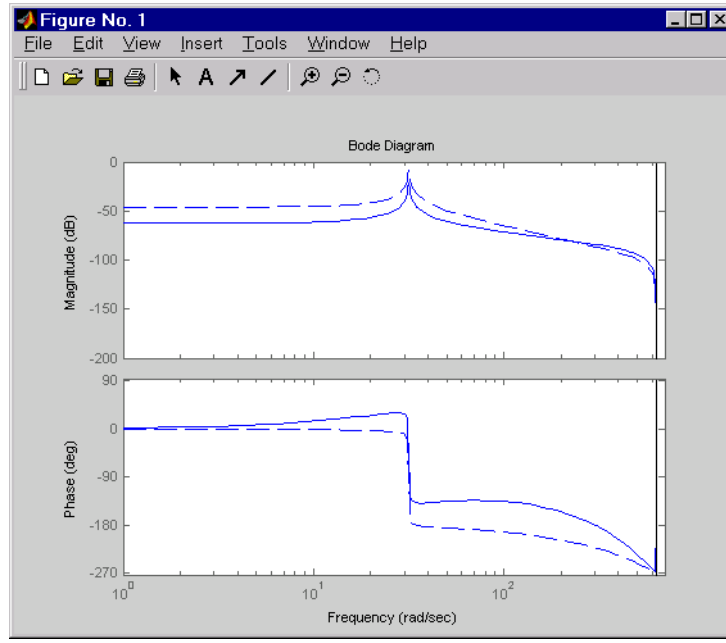
is obtained by the series connection

```
D = zpk(0.85,0,1,Ts)
oloop = Hd * D
```

Now see how this compensator modifies the open-loop frequency response.

```
bode(Hd,'--',oloop,'-')
```

The plant response is the dashed line and the open-loop response with the compensator is the solid line.

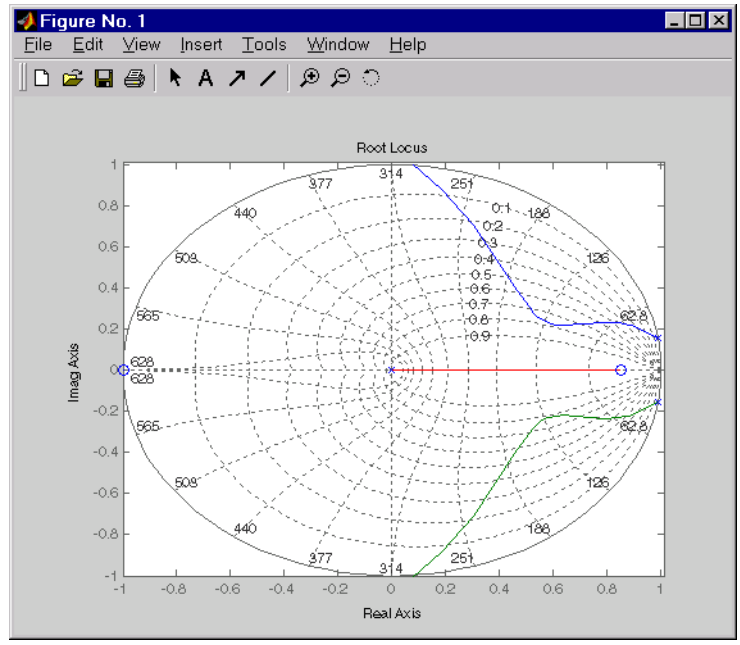


The plot above shows that the compensator has shifted up the phase plot (added lead) in the frequency range $\omega > 10$ rad/sec.

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

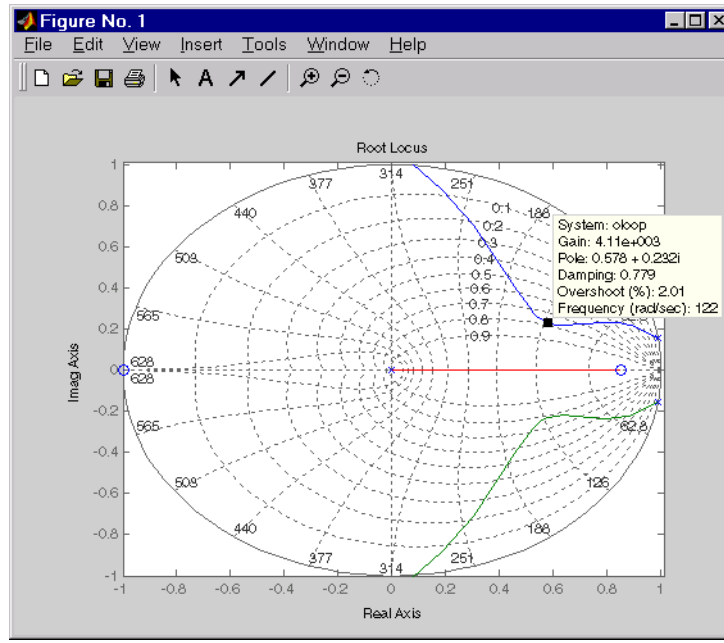
```
rlocus(oloop)
zgrid
```

Open the **Property Editor** by right-clicking in the plot away from the curve. On the **Limits** page, set the *x*- and *y*-axis limits from -1 to 1.01. This figure shows the result.



This time, the poles stay within the unit circle for some time (the lines drawn by `zgrid` show the damping ratios from $\zeta = 0$ to 1 in steps of 0.1). Use a data

marker to find the point on the curve where the gain equals 4.111e+03. This figure shows the data marker at the correct location.

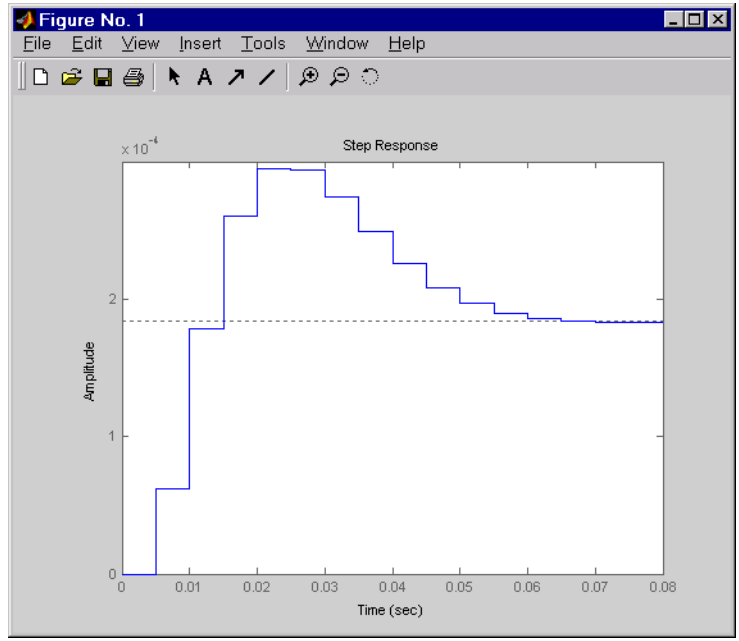


Design Analysis

To analyze this design, form the closed-loop system and plot the closed-loop step response.

```
K = 4.11e+03;
cloop = feedback(oloop,K);
```

```
step(cloop)
```

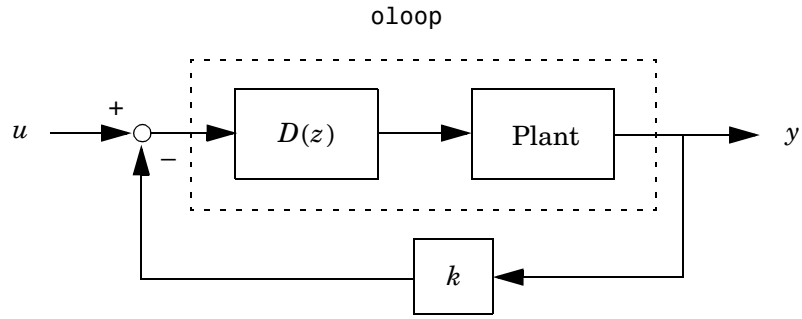


This response depends on your closed loop set point. The one shown here is relatively fast and settles in about 0.07 seconds. Therefore, this closed loop disk drive system has a seek time of about 0.07 seconds. This is slow by today's standards, but you also started with a very lightly damped system.

Now look at the robustness of your design. The most common classical robustness criteria are the gain and phase margins. Use the function `margin` to determine these margins. With output arguments, `margin` returns the gain and phase margins as well as the corresponding crossover frequencies. Without output argument, `margin` plots the Bode response and displays the margins graphically.

To compute the margins, first form the unity-feedback open loop by connecting the compensator $D(z)$, plant model, and feedback gain k in series.

```
olk = K * oloop;
```



Next apply `margin` to this open-loop model. Type

```
[Gm,Pm,Wcg,Wcp] = margin(olk);
Margins = [Gm Wcg Pm Wcp]
```

```
Margins =

    3.7987  296.7978   43.2031  106.2462
```

To obtain the gain margin in dB, type

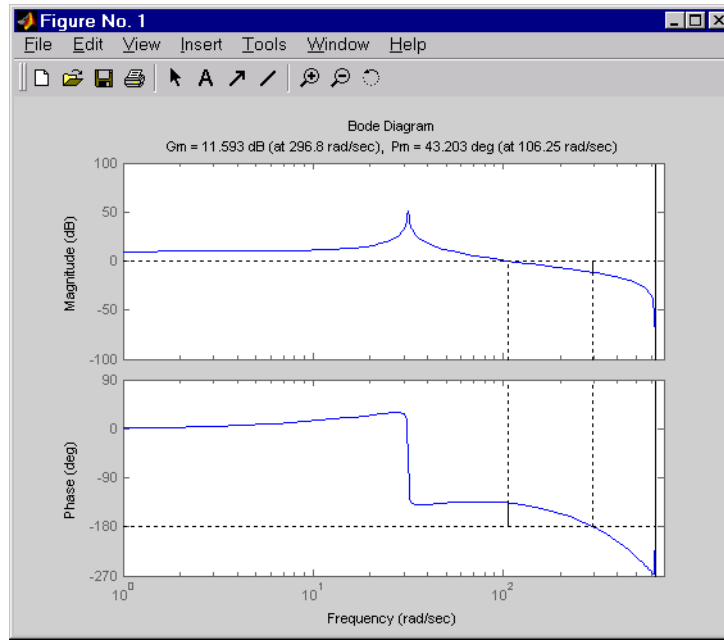
```
20*log10(Gm)
```

```
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 Example

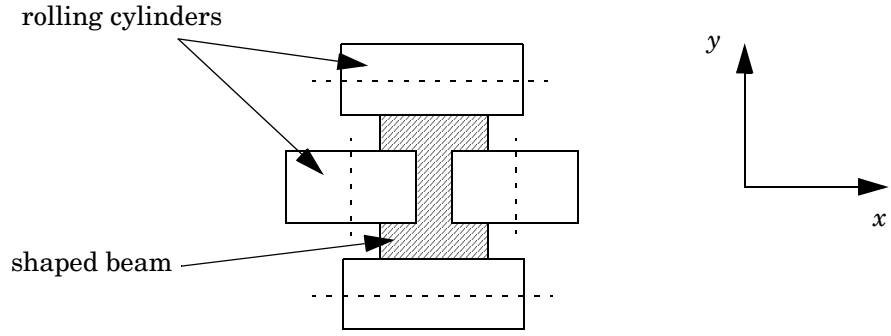
This case study demonstrates the use of the LQG design tools in a process control application. The goal is to regulate the horizontal and vertical thickness of the beam produced by a hot steel rolling mill. This example is adapted from [1]. The full plant model is MIMO and the example shows the advantage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

```
milldemo
```

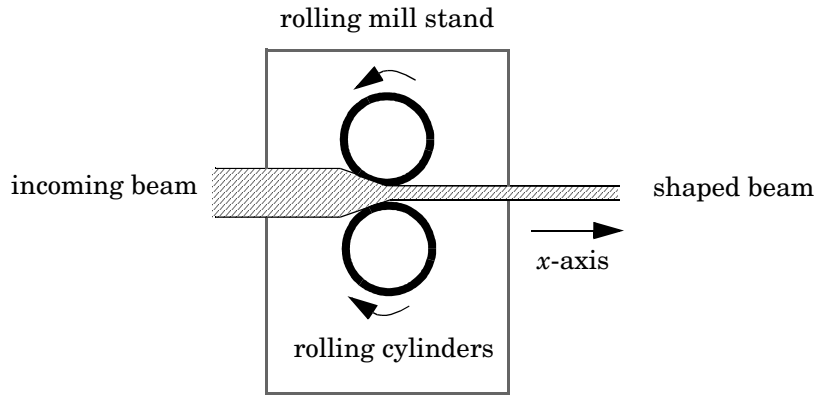
at the command line to run this demonstration interactively.

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:

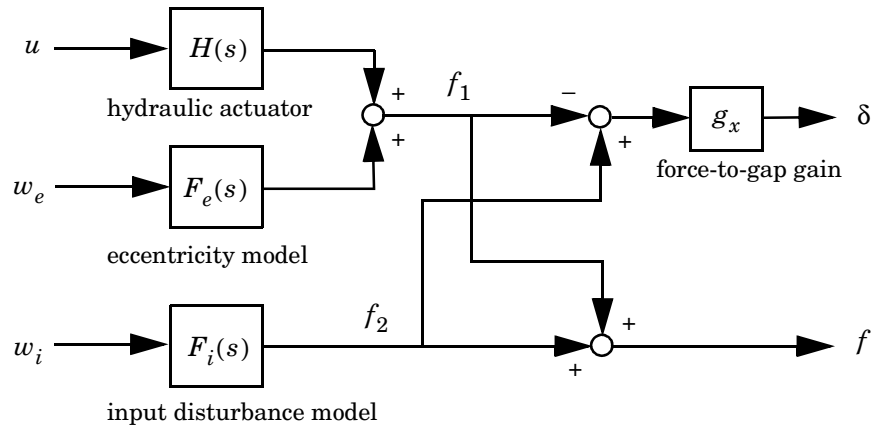
- Variations in the thickness/hardness of the incoming beam
- Eccentricity in the rolling cylinders

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 "Open-loop Model for x- or y-axis" above. 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)
```

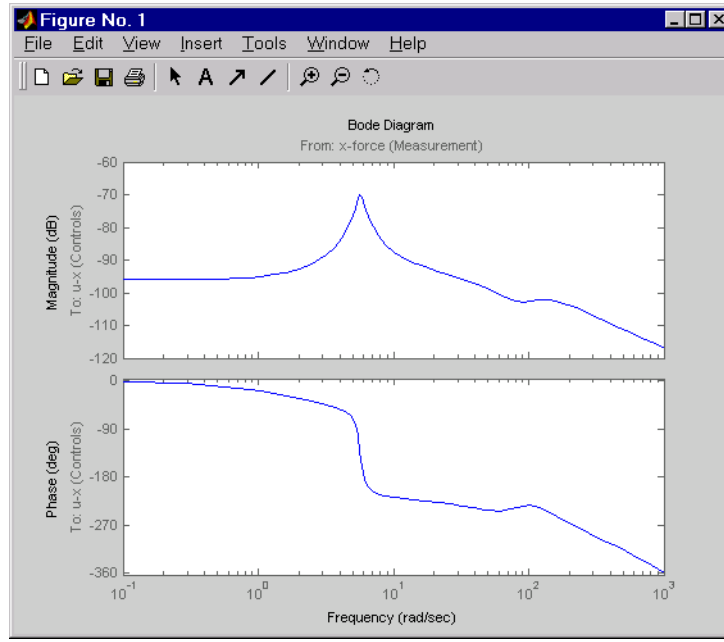
Finally, connect the state-feedback gain `kx` and state estimator `estx` to form the LQG regulator.

```
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.

```
bode(Regx,{0.1 1000})
```



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^o 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 =
```

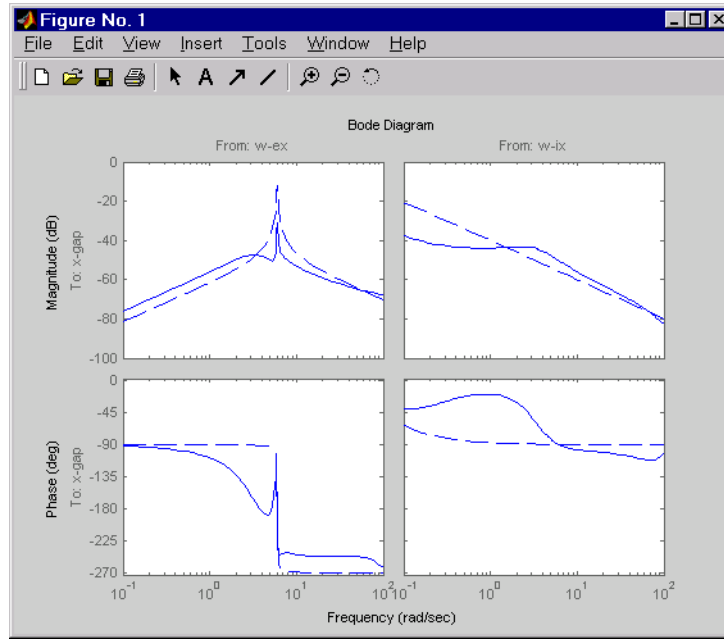
```
    'u-x'
    'w-ex'
    'w-ix'

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.

```
bode(Px(1,2:3),'--',clx(1,2:3),'-',{0.1 100})
```

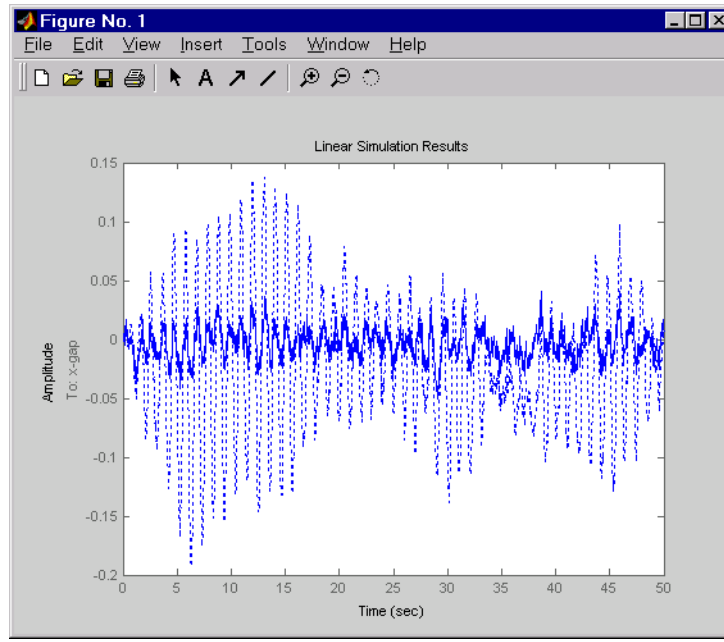


The dashed lines show the open-loop response. Note that the peak gain of the eccentricity-to-gap response and the low-frequency gain of the input-thickness-to-gap response have been reduced by about 20 dB.

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 sampling period 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)
```



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.

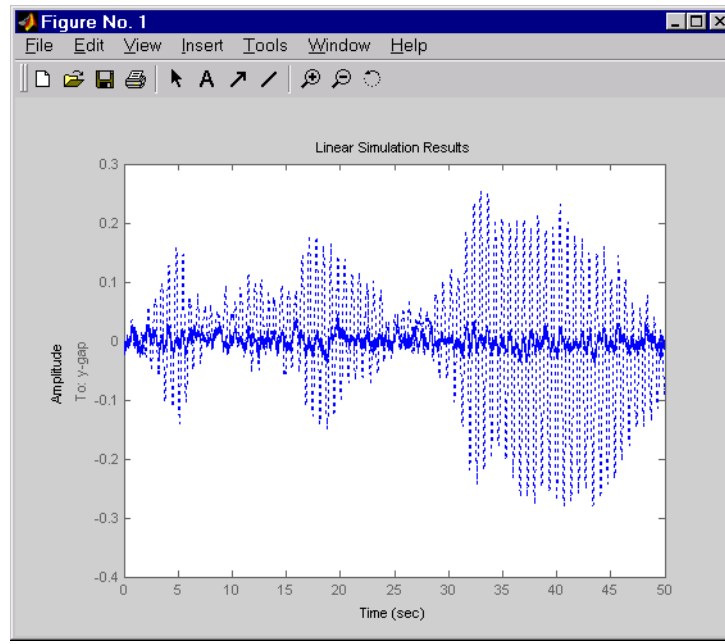
```
% Specify model components
Hy = tf(7.8e8,[1 71 88^2],'inputn','u-y')
Fiy = tf(2e4,[1 0.05],'inputn','w-iy')
Fey = tf([1e5 0],[1 0.19 9.4^2],'inputn','w-ey')
gy = 0.5e-6    % force-to-gap gain

% 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)
```



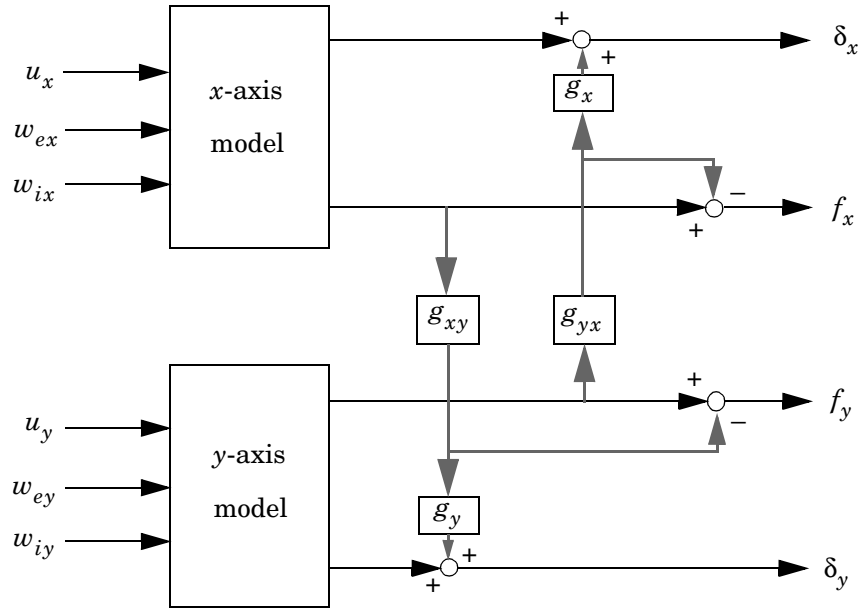
The dotted lines correspond to the open-loop response. The simulation results are comparable to those for the x -axis.

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 $\delta_x, f_x, \ldots$ of the x- and y-axis models by

$$\begin{bmatrix} \delta_x \\ \delta_y \\ f_x \\ f_y \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & g_{yx}g_x \\ 0 & 1 & g_{xy}g_y & 0 \\ 0 & 0 & 1 & -g_{yx} \\ 0 & 0 & -g_{xy} & 1 \end{bmatrix}}_{\text{cross-coupling matrix}} \begin{bmatrix} \bar{\delta}_x \\ \bar{\delta}_y \\ \bar{f}_x \\ \bar{f}_y \end{bmatrix}$$

Let's see how the previous "decoupled" LQG design fares when cross-coupling is taken into account. To build the two-axes model, shown in "Coupling Between the x- and y-axes" 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.

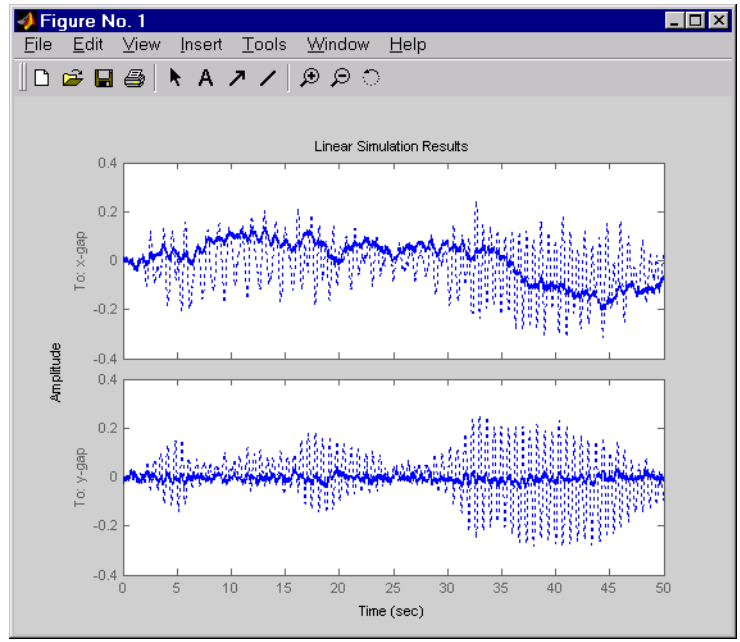
```
gxy = 0.1; gyx = 0.4;
CCmat = [eye(2) [0 gyx*gx;gxy*gy 0] ; zeros(2) [1 -gyx;-gxy 1]]
Pc = CCmat * P
Pc.outputname = P.outputname
```

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)
```



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-space model `Pc` derived above. 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).

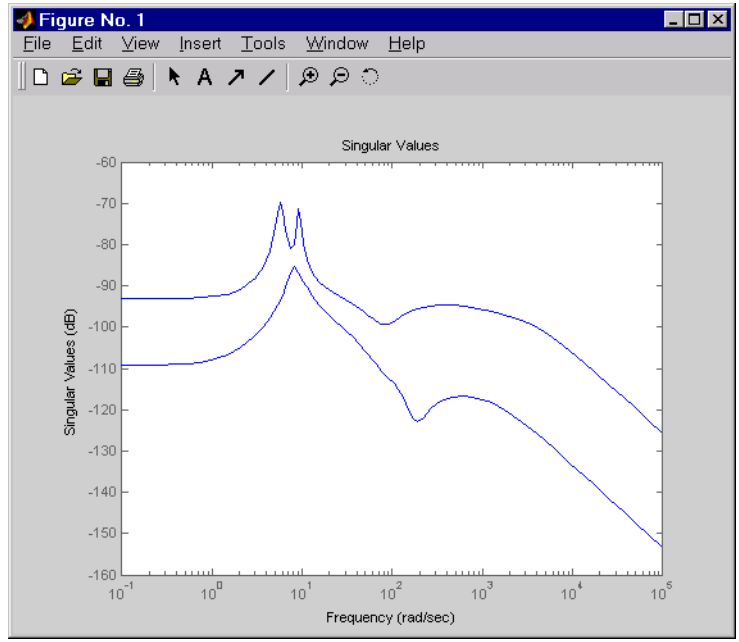
```
k = lqry(Pdes(1:2,1:2),eye(2),1e-4*eye(2))    % LQ gain
est = kalman(Pdes(3:4,:),eye(4),1e3*eye(2))   % Kalman estimator

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).

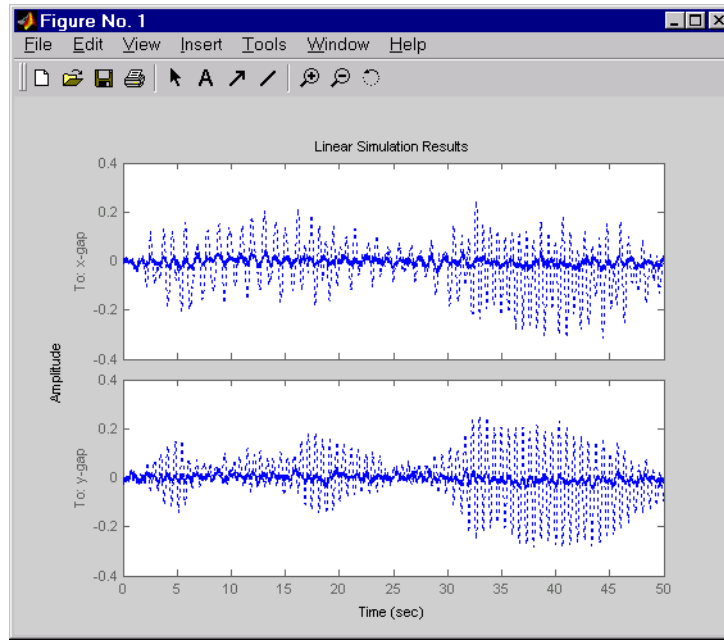
```
sigma(RegMIMO)
```



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)
```



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.

Kalman Filtering

This final case study illustrates the use of the Control System Toolbox for Kalman filter design and simulation. Both steady-state and time-varying Kalman filters are considered.

Consider the discrete plant

$$x[n+1] = Ax[n] + B(u[n] + w[n])$$
$$y[n] = Cx[n]$$

with additive Gaussian noise $w[n]$ on the input $u[n]$ and data

```
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];
```

Our goal is to design a Kalman filter that estimates the output $y[n]$ given the inputs $u[n]$ and the noisy output measurements

$$y_v[n] = Cx[n] + v[n]$$

where $v[n]$ is some Gaussian white noise.

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.

$$y_v[n+1] - C\hat{x}[n+1|n] = C(x[n+1] - \hat{x}[n+1|n])$$

between the measured and predicted values of $y[n+1]$. The innovation gain M is chosen to minimize the steady-state covariance of the estimation error given the noise covariances

$$E(w[n]w[n]^T) = Q, \qquad E(v[n]v[n]^T) = R$$

You can combine the time and measurement update equations into one state-space model (the Kalman filter).

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

$$\hat{y}[n|n] = C(I-MC)\,\hat{x}[n|n-1] + CM\,y_v[n]$$

This filter generates an optimal estimate $\hat{y}[n|n]$ of $y[n]$. Note that the filter state is $\hat{x}[n|n-1]$.

Steady-State Design

You can design the steady-state Kalman filter described above with the function `kalman`. First specify the plant model with the process noise.

$$x[n+1] = Ax[n] + Bu[n] + Bw[n] \qquad \text{(state equation)}$$

$$y[n] = Cx[n] \qquad \text{(measurement equation)}$$

This is done by

```
% Note: set sample time to -1 to mark model as discrete
Plant = ss(A,[B B],C,0,-1,'inputname',{'u' 'w'},...
```

```
                                    'outputname','y');
```

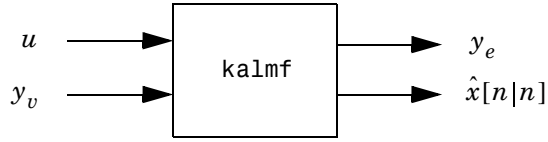
Assuming that $Q = R = 1$, you can now design the discrete Kalman filter by

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

This returns a state-space model `kalmf` of the filter as well as the innovation gain

```
M

M =
    3.7980e-01
    8.1732e-02
   -2.5704e-01
```

The inputs of `kalmf` are u and y_v, and its outputs are the plant output and state estimates $y_e = \hat{y}[n|n]$ and $\hat{x}[n|n]$.



Kalman filter

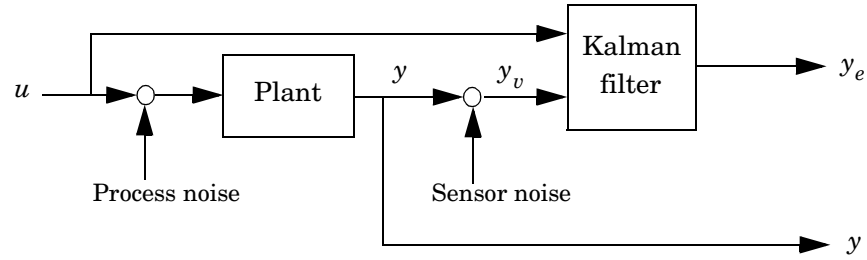
Because you are interested in the output estimate y_e, keep only the first output of `kalmf`. Type

```
kalmf = kalmf(1,:);
kalmf
a =
                      x1_e          x2_e          x3_e
         x1_e       0.7683        -0.494        0.1129
         x2_e       0.6202             0             0
         x3_e    -0.081732             1             0


b =
                         u             y
         x1_e       -0.3832        0.3586
```

```
          x2_e        0.5919        0.3798
          x3_e        0.5191      0.081732


c =
                        x1_e          x2_e          x3_e
           y_e        0.6202             0             0


d =
                           u             y
           y_e             0        0.3798

I/O groups:
       Group name       I/O    Channel(s)
       KnownInput        I         1
      Measurement        I         2
     OutputEstimate      O         1

Sampling time: unspecified
Discrete-time model.
```

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.

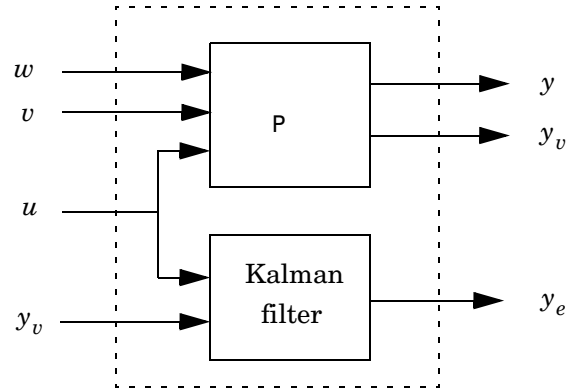
The block diagram below shows how to generate both true and filtered outputs.



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 following parallel connection.




```
sys = parallel(P,kalmf,1,1,[],[])
```

Finally, close the sensor loop by connecting the plant output y_v to the filter input y_v with positive feedback.

```
% Close loop around input #4 and output #2
SimModel = feedback(sys,1,4,2,1)
% Delete yv from I/O list
SimModel = SimModel([1 3],[1 2 3])
```

The resulting simulation model has w, v, u as inputs and y, y_e as outputs.

```
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)
randn('seed',0)
w = sqrt(Q)*randn(n,1);
v = sqrt(R)*randn(n,1);
```

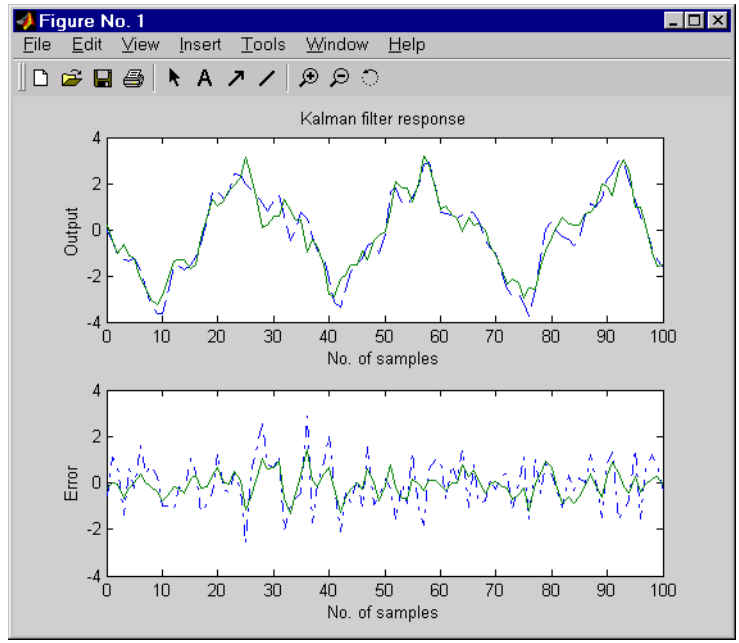
Now simulate with `lsim`.

```
[out,x] = lsim(SimModel,[w,v,u]);

y = out(:,1);   % true response
ye = out(:,2);  % filtered response
yv = y + v;     % measured response
```

and 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 the following error covariance computations.

```
MeasErr = y-yv;
MeasErrCov = sum(MeasErr.*MeasErr)/length(MeasErr);
EstErr = y-ye;
EstErrCov = sum(EstErr.*EstErr)/length(EstErr);
```

The error covariance before filtering (measurement error) is

```
MeasErrCov

MeasErrCov =
    1.1138
```

while the error covariance after filtering (estimation error) is only

```
EstErrCov

EstErrCov =
    0.2722
```

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. Given the plant state and measurement equations

$$x[n+1] = Ax[n] + Bu[n] + Gw[n]$$
$$y_v[n] = Cx[n] + v[n]$$

the time-varying Kalman filter is given by the recursions

Measurement update

$$\hat{x}[n|n] = \hat{x}[n|n-1] + M[n](y_v[n] - C\hat{x}[n|n-1])$$
$$M[n] = P[n|n-1]C^T(R[n] + CP[n|n-1]C^T)^{-1}$$
$$P[n|n] = (I - M[n]C)\ P[n|n-1]$$

Time update

$$\hat{x}[n+1|n] = A\hat{x}[n|n] + Bu[n]$$
$$P[n+1|n] = AP[n|n]A^T + GQ[n]G^T$$

with $\hat{x}[n|n-1]$ and $\hat{x}[n|n]$ as defined in “Discrete Kalman Filter”, and in the following.

$$Q[n] = E(w[n]w[n]^T)$$
$$R[n] = E(v[n]v[n]^T)$$
$$P[n|n] = E(\{x[n] - x[n|n]\}\{x[n] - x[n|n]\}^T)$$
$$P[n|n-1] = E(\{x[n] - x[n|n-1]\}\{x[n] - x[n|n-1]\}^T)$$

For simplicity, we have dropped the subscripts indicating the time dependence of the state-space matrices.

Given initial conditions $x[1|0]$ and $P[1|0]$, you can iterate these equations to perform the filtering. Note that you must update both the state estimates $x[n|.]$ and error covariance matrices $P[n|.]$ at each time sample.

Time-Varying Design

Although the Control System Toolbox does not offer specific commands to perform time-varying Kalman filtering, it is easy to implement the filter recursions in MATLAB. This section shows how to do this for the stationary plant considered above.

First generate noisy output measurements

```
% Use process noise w and measurement noise v generated above
sys = ss(A,B,C,0,-1);
y = lsim(sys,u+w);     % w = process noise
yv = y + v;            % v = measurement noise
```

Given the initial conditions

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

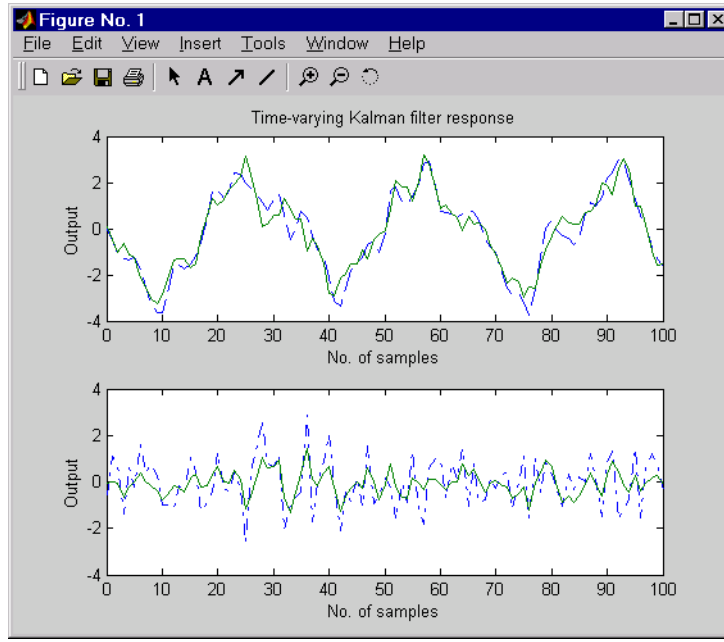
you can implement the time-varying filter with the following `for` loop.

```
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
```

You can now 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')
```

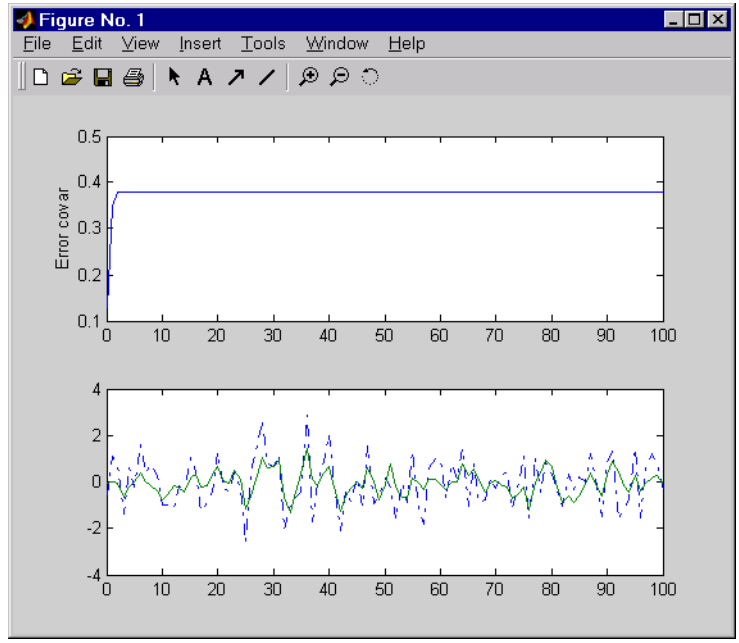


The first plot shows the true response y (dashed line) and the filtered response y_e (solid line). The second plot compares the measurement error (dash-dot) with the estimation error (solid).

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. Type

```
EstErr = y-ye;
EstErrCov = sum(EstErr.*EstErr)/length(EstErr)

EstErrCov =
    0.2718
```

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, M
```

```
Mn =
    0.3798
    0.0817
   -0.2570

M =
    0.3798
    0.0817
   -0.2570
```

References

[1] Grimble, M.J., *Robust Industrial Control: Optimal Design Approach for Polynomial Systems*, Prentice Hall, 1994, p. 261 and pp. 443-456.

10

Reliable Computations

Introduction

When working with low-order SISO models (less than five states), computers are usually quite forgiving and insensitive to numerical problems. You generally won't encounter any numerical difficulties and MATLAB will give you accurate answers regardless of the model or conversion method you choose. For high order SISO models and MIMO models, however, the finite-precision arithmetic of a computer is not so forgiving and you must exercise caution.

In general, to get a numerically accurate answer from a computer, you need

- A well-conditioned problem
- An algorithm that is numerically stable in finite-precision arithmetic
- A good software implementation of the algorithm

A problem is said to be well-conditioned if small changes in the data cause only small corresponding changes in the solution. If small changes in the data have the potential to induce large changes in the solution, the problem is said to be ill-conditioned. An algorithm is numerically stable if it does not introduce any more sensitivity to perturbation than is already inherent in the problem. Many numerical linear algebra algorithms can be shown to be backward stable; i.e., the computed solution can be shown to be (near) the exact solution of a slightly perturbed original problem. The solution of a slightly perturbed original problem will be close to the true solution if the problem is well-conditioned.

Thus, a stable algorithm cannot be expected to solve an ill-conditioned problem any more accurately than the data warrant, but an unstable algorithm can produce poor solutions even to well-conditioned problems. For further details and references to the literature see [5].

While most of the tools in the Control System Toolbox use reliable algorithms, some of the tools do not use stable algorithms and some solve ill-conditioned problems. These unreliable tools work quite well on some problems (low-order systems) but can encounter numerical difficulties, often severe, when pushed on higher-order problems. These tools are provided because

- They are quite useful for low-order systems, which form the bulk of real-world engineering problems.
- Many control engineers think in terms of these tools.
- A more reliable alternative tool is usually available in this toolbox.

- They are convenient for pedagogical purposes.

At the same time, it is important to appreciate the limitations of computer analyses. By following a few guidelines, you can avoid certain tools and models when they are likely to get you into trouble. The following sections try to illustrate, through examples, some of the numerical pitfalls to be avoided. We also encourage you to get the most out of the good algorithms by ensuring, if possible, that your models give rise to problems that are well-conditioned.

Conditioning and Numerical Stability

Two of the key concepts in numerical analysis are the conditioning of problems and the stability of algorithms.

Conditioning

Consider the linear system $Ax = b$ given by

```
A =
    0.7800    0.5630
    0.9130    0.6590
b =
    0.2170
    0.2540
```

The true solution is `x = [1, -1]'` and you can calculate it approximately using MATLAB.

```
x = A\b
x =
     1.0000
    -1.0000

format long, x
x =
     0.99999999991008
    -0.99999999987542
```

Of course, in real problems you almost never have the luxury of knowing the true solution. This problem is very ill-conditioned. To see this, add a small perturbation to `A`

```
E =
     0.0010     0.0010
    -0.0020    -0.0010
```

and solve the perturbed system $(A + E)x = b$

```
xe = (A+E)\b
xe =
    -5.0000
     7.3085
```

Notice how much the small change in the data is magnified in the solution.

One way to measure the magnification factor is by means of the quantity

$$\|A\| \, \|A^{-1}\|$$

called the condition number of A with respect to inversion. The condition number determines the loss in precision due to roundoff errors in Gaussian elimination and can be used to estimate the accuracy of results obtained from matrix inversion and linear equation solution. It arises naturally in perturbation theories that compare the perturbed solution $(A+E)^{-1}b$ with the true solution $A^{-1}b$.

In MATLAB, the function `cond` calculates the condition number in 2-norm. `cond(A)` is the ratio of the largest singular value of `A` to the smallest. Try it for the example above. The usual rule is that the exponent `log10(cond(A))` on the condition number indicates the number of decimal places that the computer can lose to roundoff errors.

IEEE standard double precision numbers have about 16 decimal digits of accuracy, so if a matrix has a condition number of 10^{10}, you can expect only six digits to be accurate in the answer. If the condition number is much greater than `1/sqrt(eps)`, caution is advised for subsequent computations. For IEEE arithmetic, the machine precision, `eps`, is about 2.2×10^{-16}, and `1/sqrt(eps)` = 6.7×10^{8}.

Another important aspect of conditioning is that, in general, residuals are reliable indicators of accuracy only if the problem is well-conditioned. To illustrate, try computing the residual vector $r = Ax - b$ for the two candidate solutions `x = [0.999 -1.001]'` and `x = [0.341 -0.087]'`. Notice that the second, while clearly a much less accurate solution, gives a far smaller residual. The conclusion is that residuals are unreliable indicators of relative solution accuracy for ill-conditioned problems. This is a good reason to be concerned with computing or estimating accurately the condition of your problem.

Another simple example of an ill-conditioned problem is the n-by-n matrix with ones on the first upper-diagonal.

```
A = diag(ones(1,n-1),1);
```

This matrix has n eigenvalues at 0. Now consider a small perturbation of the data consisting of adding the number 2^{-n} to the first element in the last (n th)

row of A. This perturbed matrix has n distinct eigenvalues $\lambda_1, ..., \lambda_n$ with $\lambda_k = 1/2 \exp(j2\pi k/n)$. Thus, you can see that this small perturbation in the data has been magnified by a factor on the order of 2^n to result in a rather large perturbation in the solution (the eigenvalues of A). Further details and related examples are to be found in [7].

It is important to realize that a matrix can be ill-conditioned with respect to inversion but have a well-conditioned eigenproblem, and vice versa. For example, consider an upper triangular matrix of ones (zeros below the diagonal) given by

```
A = triu(ones(n));
```

This matrix is ill-conditioned with respect to its eigenproblem (try small perturbations in A(n,1) for, say, n=20), but is well-conditioned with respect to inversion (check its condition number). On the other hand, the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1+\delta \end{bmatrix}$$

has a well-conditioned eigenproblem, but is ill-conditioned with respect to inversion for small δ.

Numerical Stability

Numerical stability is somewhat more difficult to illustrate meaningfully. Consult the references in [5], [6], and [7] for further details. Here is one small example to illustrate the difference between stability and conditioning.

Gaussian elimination with no pivoting for solving the linear system $Ax = b$ is known to be numerically unstable. Consider

$$A = \begin{bmatrix} 0.001 & 1.000 \\ 1.000 & -1.000 \end{bmatrix} \qquad b = \begin{bmatrix} 1.000 \\ 0.000 \end{bmatrix}$$

All computations are carried out in three-significant-figure decimal arithmetic. The true answer $x = A^{-1}b$ is approximately

$$x = \begin{bmatrix} 0.999 \\ 0.999 \end{bmatrix}$$

Using row 1 as the pivot row (i.e., subtracting 1000 times row 1 from row 2) you arrive at the equivalent triangular system.

$$\begin{bmatrix} 0.001 & 1.000 \\ 0 & -1000 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.000 \\ -1000 \end{bmatrix}$$

Note that the coefficient multiplying x_2 in the second equation should be -1001, but because of roundoff, becomes -1000. As a result, the second equation yields $x_2 = 1.000$, a good approximation, but now back-substitution in the first equation

$$0.001x_1 = 1.000 - (1.000)(1.000)$$

yields $x_1 = 0.000$. This extremely bad approximation of x_1 is the result of numerical instability. The problem itself can be shown to be quite well-conditioned. Of course, MATLAB implements Gaussian elimination with pivoting.

Choice of LTI Model

Now turn to the implications of the results in the last section on the linear modeling techniques used for control engineering. The Control System Toolbox includes the following types of LTI models that are applicable to discussions of computational reliability:

- State space
- Transfer function, polynomial form
- Transfer function, factored zero-pole-gain form

The following subsections show that state space is most preferable for numerical computations.

State Space

The state-space representation is the most reliable LTI model to use for computer analysis. This is one of the reasons for the popularity of "modern" state-space control theory. Stable computer algorithms for eigenvalues, frequency response, time response, and other properties of the (A, B, C, D) quadruple are known [5] and implemented in this toolbox. The state-space model is also the most natural model in the MATLAB matrix environment.

Even with state-space models, however, accurate results are not guaranteed, because of the problems of finite-word-length computer arithmetic discussed in the last section. A well-conditioned problem is usually a prerequisite for obtaining accurate results and makes it important to have reasonable scaling of the data. Scaling is discussed further in the "Scaling" section later in this chapter.

Transfer Function

Transfer function models, when expressed in terms of expanded polynomials, tend to be inherently ill-conditioned representations of LTI systems. For systems of order greater than 10, or with very large/small polynomial coefficients, difficulties can be encountered with functions like `roots`, `conv`, `bode`, `step`, or conversion functions like `ss` or `zpk`.

A major difficulty is the extreme sensitivity of the roots of a polynomial to its coefficients. This example is adapted from Wilkinson, [6] as an illustration. Consider the transfer function

$$H(s) = \frac{1}{(s+1)(s+2)...(s+20)} = \frac{1}{s^{20} + 210s^{19} + ... + 20!}$$

The A matrix of the companion realization of $H(s)$ is

$$A = \begin{bmatrix} 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 1 & ... & 0 \\ : & : & . & . & : \\ 0 & 0 & ... & . & 1 \\ -20! & . & ... & . & -210 \end{bmatrix}$$

Despite the benign looking poles of the system (at -1,-2,..., -20) you are faced with a rather large range in the elements of A, from 1 to $20! \approx 2.4 \times 10^{18}$. But the difficulties don't stop here. Suppose the coefficient of s^{19} in the transfer function (or $A(n,n)$) is perturbed from 210 to $210 + 2^{-23}$ ($2^{-23} \approx 1.2 \times 10^{-7}$). Then, computed on a VAX (IEEE arithmetic has enough mantissa for only $n = 17$), the poles of the perturbed transfer function (equivalently, the eigenvalues of A) are

```
eig(A)'

ans =

Columns 1 through 7
-19.9998 -19.0019 -17.9916 -17.0217 -15.9594 -15.0516 -13.9504

Columns 8 through 14
-13.0369 -11.9805 -11.0081  -9.9976  -9.0005  -7.9999  -7.0000

Columns 15 through 20
 -6.0000  -5.0000  -4.0000  -3.0000  -2.0000  -1.0000
```

The problem here is not roundoff. Rather, high-order polynomials are simply intrinsically very sensitive, even when the zeros are well separated. In this case, a relative perturbation of the order of 10^{-9} induced relative perturbations of the order of 10^{-2} in some roots. But some of the roots changed

very little. This is true in general. Different roots have different sensitivities to different perturbations. Computed roots may then be quite meaningless for a polynomial, particularly high-order, with imprecisely known coefficients.

Finding all the roots of a polynomial (equivalently, the poles of a transfer function or the eigenvalues of a matrix in controllable or observable canonical form) is often an intrinsically sensitive problem. For a clear and detailed treatment of the subject, including the tricky numerical problem of deflation, consult [6].

It is therefore preferable to work with the factored form of polynomials when available. To compute a state-space model of the transfer function $H(s)$ defined above, for example, you could expand the denominator of H, convert the transfer function model to state space, and extract the state-space data by

```
H1 = tf(1,poly(1:20))
H1ss = ss(H1)
[a1,b1,c1] = ssdata(H1)
```

However, you should rather keep the denominator in factored form and work with the zero-pole-gain representation of $H(s)$.

```
H2 = zpk([],1:20,1)
H2ss = ss(H2)
[a2,b2,c2] = ssdata(H2)
```

Indeed, the resulting state matrix `a2` is better conditioned.

```
[cond(a1)  cond(a2)]

ans =
   2.7681e+03   8.8753e+01
```

and the conversion from zero-pole-gain to state space incurs no loss of accuracy in the poles.

```
format long e
[sort(eig(a1))  sort(eig(a2))]

ans =
     9.999999999998792e-01     1.000000000000000e+00
     2.000000000001984e+00     2.000000000000000e+00
     3.000000000475623e+00     3.000000000000000e+00
     3.999999981263996e+00     4.000000000000000e+00
```

```
   5.000000270433721e+00     5.000000000000000e+00
   5.999998194359617e+00     6.000000000000000e+00
   7.000004542844700e+00     7.000000000000000e+00
   8.000013753274901e+00     8.000000000000000e+00
   8.999848908317270e+00     9.000000000000000e+00
   1.000059459550623e+01     1.000000000000000e+01
   1.099854678336595e+01     1.100000000000000e+01
   1.200255822210095e+01     1.200000000000000e+01
   1.299647702454549e+01     1.300000000000000e+01
   1.400406940833612e+01     1.400000000000000e+01
   1.499604787386921e+01     1.500000000000000e+01
   1.600304396718421e+01     1.600000000000000e+01
   1.699828695210055e+01     1.700000000000000e+01
   1.800062935148728e+01     1.800000000000000e+01
   1.899986934359322e+01     1.900000000000000e+01
   2.000001082693916e+01     2.000000000000000e+01
```

There is another difficulty with transfer function models when realized in state-space form with ss. They may give rise to badly conditioned eigenvector matrices, even if the eigenvalues are well separated. For example, consider the normal matrix

```
A = [5 4 1 1
     4 5 1 1
     1 1 4 2
     1 1 2 4]
```

Its eigenvectors and eigenvalues are given as follows.

```
[v,d] = eig(A)

v =
    0.7071   -0.0000   -0.3162    0.6325
   -0.7071    0.0000   -0.3162    0.6325
    0.0000    0.7071    0.6325    0.3162
   -0.0000   -0.7071    0.6325    0.3162

d =
    1.0000         0         0         0
         0    2.0000         0         0
         0         0    5.0000         0
```

```
         0         0         0   10.0000
```

The condition number (with respect to inversion) of the eigenvector matrix is

```
cond(v)

ans =
    1.000
```

Now convert a state-space model with the above `A` matrix to transfer function form, and back again to state-space form.

```
b = [1 ; 1 ; 0 ; -1];
c = [0 0 2 1];
H = tf(ss(A,b,c,0));     % Transfer function
[Ac,bc,cc] = ssdata(H)    % Convert back to state space
```

The new `A` matrix is

```
Ac =
   18.0000   -6.0625    2.8125   -1.5625
   16.0000         0         0         0
         0    4.0000         0         0
         0         0    1.0000         0
```

Note that `Ac` is not a standard companion matrix and has already been balanced as part of the `ss` conversion (see `ssbal` for details).

Note also that the eigenvectors have changed.

```
[vc,dc] = eig(Ac)

vc =
   -0.5017    0.2353    0.0510    0.0109
   -0.8026    0.7531    0.4077    0.1741
   -0.3211    0.6025    0.8154    0.6963
   -0.0321    0.1205    0.4077    0.6963

dc =
   10.0000         0         0         0
         0    5.0000         0         0
         0         0    2.0000         0
         0         0         0    1.0000
```

The condition number of the new eigenvector matrix

```
cond(vc)

ans =
    34.5825
```

is thirty times larger.

The phenomenon illustrated above is not unusual. Matrices in companion form or controllable/observable canonical form (like `Ac`) typically have worse-conditioned eigensystems than matrices in general state-space form (like `A`). This means that their eigenvalues and eigenvectors are more sensitive to perturbation. The problem generally gets far worse for higher-order systems. Working with high-order transfer function models and converting them back and forth to state space is numerically risky.

In summary, the main numerical problems to be aware of in dealing with transfer function models (and hence, calculations involving polynomials) are

- The potentially large range of numbers leads to ill-conditioned problems, especially when such models are linked together giving high-order polynomials.
- The pole locations are very sensitive to the coefficients of the denominator polynomial.
- The balanced companion form produced by `ss`, while better than the standard companion form, often results in ill-conditioned eigenproblems, especially with higher-order systems.

The above statements hold even for systems with distinct poles, but are particularly relevant when poles are multiple.

Zero-Pole-Gain Models

The third major representation used for LTI models in MATLAB is the factored, or zero-pole-gain (ZPK) representation. It is sometimes very convenient to describe a model in this way although most major design methodologies tend to be oriented towards either transfer functions or state-space.

In contrast to polynomials, the ZPK representation of systems can be more reliable. At the very least, the ZPK representation tends to avoid the

extraordinary arithmetic range difficulties of polynomial coefficients, as illustrated in the "Transfer Function" section. The transformation from state space to zero-pole-gain is stable, although the handling of infinite zeros can sometimes be tricky, and repeated roots can cause problems.

If possible, avoid repeated switching between different model representations. As discussed in the previous sections, when transformations between models are not numerically stable, roundoff errors are amplified.

Scaling

State space is the preferred model for LTI systems, especially with higher order models. Even with state-space models, however, accurate results are not guaranteed, because of the finite-word-length arithmetic of the computer. A well-conditioned problem is usually a prerequisite for obtaining accurate results.

You should generally normalize or scale the (A, B, C, D) matrices of a system to improve their conditioning. An example of a poorly scaled problem might be a dynamic system where two states in the state vector have units of light years and millimeters. You would expect the A matrix to contain both very large and very small numbers. Matrices containing numbers widely spread in value are often poorly conditioned both with respect to inversion and with respect to their eigenproblems, and inaccurate results can ensue.

Normalization also allows meaningful statements to be made about the degree of controllability and observability of the various inputs and outputs.

A set of (A, B, C, D) matrices can be normalized using diagonal scaling matrices N_u, N_x, and N_y to scale u, x, and y.

$$u = N_u u_n \qquad x = N_x x_n \qquad y = N_y y_n$$

so the normalized system is

$$\dot{x}_n = A_n x_n + B_n u_n$$
$$y_n = C_n x_n + D_n u_n$$

where

$$A_n = N_x^{-1} A N_x \qquad B_n = N_x^{-1} B N_u$$
$$C_n = N_y^{-1} C N_x \qquad D_n = N_y^{-1} D N_u$$

Choose the diagonal scaling matrices according to some appropriate normalization procedure. One criterion is to choose the maximum range of each of the input, state, and output variables. This method originated in the days of analog simulation computers when u_n, x_n, and y_n were forced to be between ± 10 Volts. A second method is to form scaling matrices where the diagonal entries are the smallest deviations that are significant to each variable. An

excellent discussion of scaling is given in the introduction to the *LINPACK Users' Guide,* [1].

Choose scaling based upon physical insight to the problem at hand. If you choose not to scale, and for many small problems scaling is not necessary, be aware that this choice affects the accuracy of your answers.

Finally, note that the function `ssbal` performs automatic scaling of the state vector. Specifically, it seeks to minimize the norm of

$$\begin{bmatrix} N_x^{-1} A N_x & N_x^{-1} B \\ C N_x & 0 \end{bmatrix}$$

by using diagonal scaling matrices N_x. Such diagonal scaling is an economical way to compress the numerical range and improve the conditioning of subsequent state-space computations.

Summary

This chapter has described numerous things that can go wrong when performing numerical computations. You won't encounter most of these difficulties when you solve practical lower-order problems. The problems described here pertain to all computer analysis packages. MATLAB has some of the best algorithms available, and, where possible, notifies you when there are difficulties. The important points to remember are

- State-space models are, in general, the most reliable models for subsequent computations.
- Scaling model data can improve the accuracy of your results.
- Numerical computing is a tricky business, and virtually all computer tools can fail under certain conditions.

References

[1] Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart, *LINPACK User's Guide*, SIAM Publications, Philadelphia, PA, 1978.

[2] Franklin, G.F. and J.D. Powell, *Digital Control of Dynamic Systems*, Addison-Wesley, 1980.

[3] Kailath, T., *Linear Systems*, Prentice-Hall, 1980.

[4] Laub, A.J., "Numerical Linear Algebra Aspects of Control Design Computations," *IEEE Transactions on Automatic Control*, Vol. AC-30, No. 2, February 1985, pp. 97-108.

[5] Wilkinson, J.H., *Rounding Errors in Algebraic Processes*, Prentice-Hall, 1963.

[6] Wilkinson, J.H., *The Algebraic Eigenvalue Problem*, Oxford University Press, 1965.

11

Tool and Viewer Quick Start

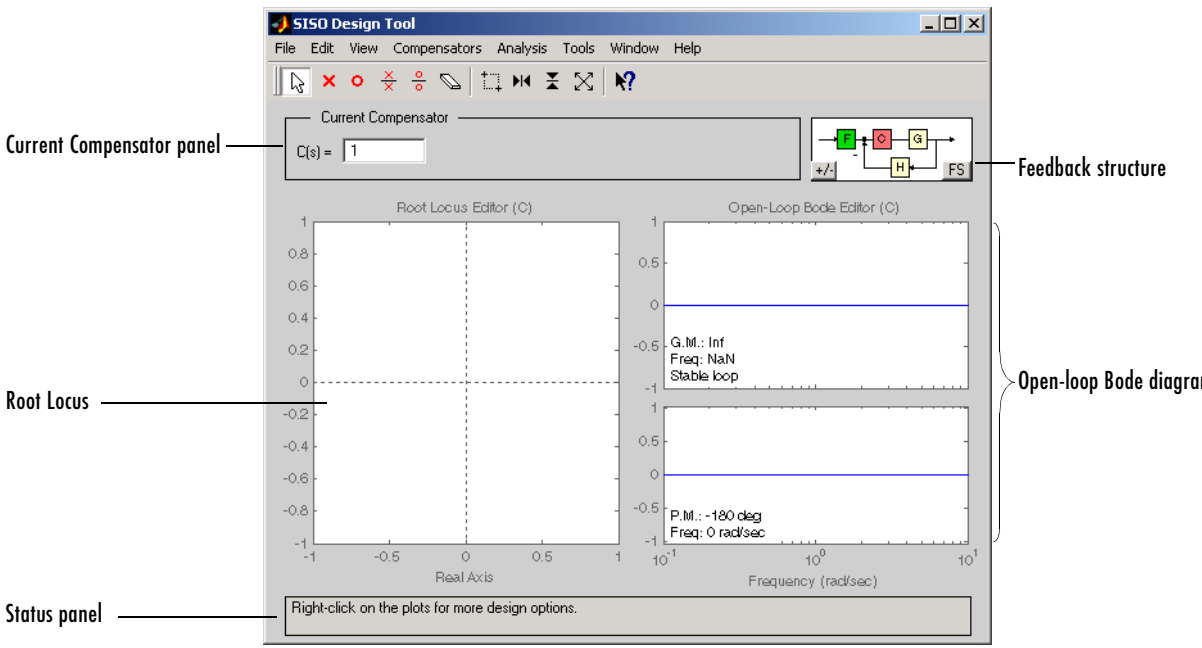
Introduction

The following two sections are a brief introduction to the basics of the tools provided with the Control System Toolbox. The graphical user interfaces (GUI's) covered are as follows:

- "SISO Design Tool"— A GUI for designing prefilters and compensators for SISO systems
- "LTI Viewer" — A tool for viewing and analyzing LTI system data

SISO Design Tool

The SISO Design Tool is a graphical user interface (GUI) that allows you to analyze and tune SISO feedback control systems. Using the SISO Design Tool, you can graphically tune the gains and dynamics of the compensator (**C**) and prefilter (**F**) using a mix of root locus and loop shaping techniques. For example, you can use the root locus view to stabilize the feedback loop and enforce some minimum damping, and use the Bode diagrams to adjust the bandwidth, check the gain and phase margins, or add a notch filter for disturbance rejection. You can also bring up an open-loop Nichols view or Bode diagram of the prefilter (**F**) by selecting these items from the View menu. All views 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 Tool is designed to work closely with the LTI Viewer, allowing you to rapidly iterate on your design and immediately see the results in the LTI Viewer. When you make a change in your compensator, the LTI Viewer associated with your SISO Design Tool automatically updates the response plots that you have chosen. By default, the SISO Design Tool displays

the root locus and open-loop Bode diagrams for your imported systems. You can also bring up an open-loop Nichols view or prefilter Bode diagram by selecting these items in the View menu.

Imported systems can include any of elements of the feedback structure diagram located to the right of the Current Compensator panel. You cannot change imported plant (**G**) or sensor (**H**) models, but you can use the SISO Design Tool for designing a new (or modifying an existing) prefilter (**F**) or compensator (**C**) for your imported plant and sensor configuration.

For a quick discussion of basic tasks you can do with the SISO Design Tool, see the following:

- "Importing and Exporting Models"
- "Configuring the Feedback Structure"
- "Tuning Compensators"
  - "Root-Locus"
  - "Open-Loop Bode Diagram"
  - "Open-Loop Nichols Plot"
- "Viewing Loop Responses"
- "Viewing Loop Responses"
- "Storing and Retrieving Designs"
- "Customizing the SISO Design Tool"

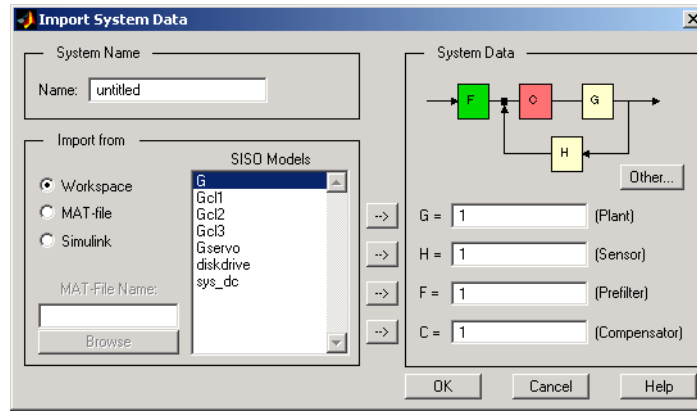
In addition, there is an extensive discussion of how to use the SISO Design Tool in Designing Compensators, chapter 4 in *Getting Started with the Control System Toolbox*. See SISO Design Tool for a description of all the features available.

Importing and Exporting Models

The SISO Design Tool provides graphical user interfaces to facilitate importing and exporting of linear models.

Importing Models

To import models into the SISO Design Tool, select **Import** under the **File** menu. This opens the **Import System Data** window.



To import a model:

- Specify whether you want to import a SISO model from the MATLAB workspace, a MAT-file, or from a Simulink model. The window lists the available models for each format under SISO Models.
- Click on the desired model.
- Click a right arrow to specify whether you want to import the model as the plant (**G**), Sensor (**H**), Prefilter (**F**), or Compensator (**C**).

  Click the **OK** Button

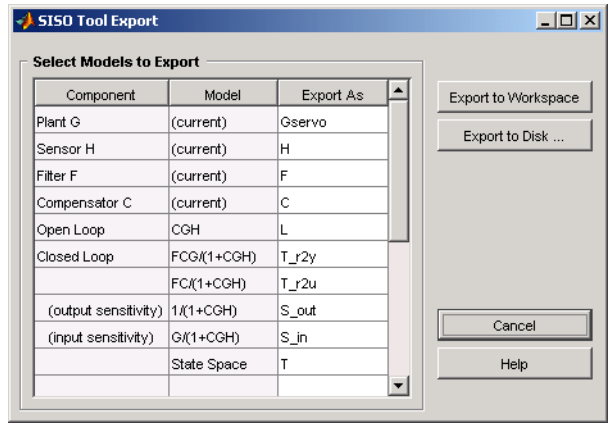
Alternatively, you can directly import a model into the SISO Design Tool using the `sisotool` function, as in

```
sisotool(modelname)
```

In this case, *`modelname`* is imported as the plant (**G**). See the `sisotool` function for more information.

Exporting Models

Use **Export** in the **File** menu to open the **SISO Tool Export** window.



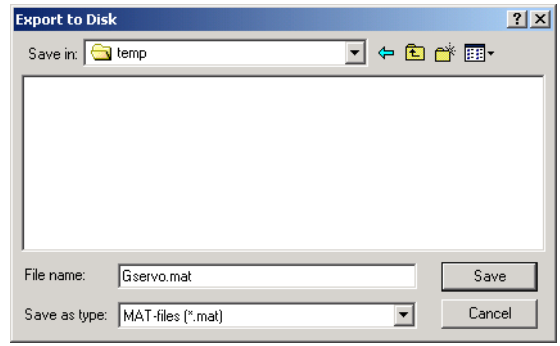
Selecting Models to Export. The SISO Tool Export window not only lists all the models displayed in your SISO Design Tool, but also contains various transfer functions associated with standard control analysis techniques. These include open and closed loop transfer functions, input and output sensitivity functions, and the state-space model of the overall feedback loop.

To select a model for export, left-click on the row containing the component name. To specify a different export name, double-click on the Export As cell for the component. This makes the name in the cell editable.

You can export models back to the MATLAB workspace or to disk. In the latter case, the models are saved in a MAT-file.

Exporting to Workspace. To export models to the MATLAB workspace, simply click **Export to Workspace**.

Exporting to Disk. If you choose **Export to Disk**, this window opens.



The Export to Disk window provides a default file name. If you want to change the name, specify the new name for your model(s) and click **Save**. Your models are stored in a MAT-file.

Exporting Multiple Models . There are two ways to export multiple models:

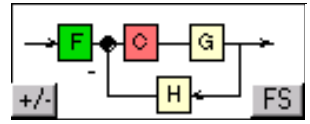
- If the models are adjacent in the model selection list, hold down the **Shift** key while selecting the models with your mouse.
- If the models are nonadjacent, hold down the **Ctrl** key and select the models by left-clicking

Configuring the Feedback Structure

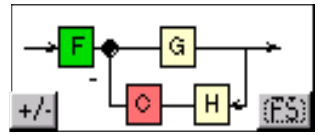
The **Feedback Structure** panel displays the current configuration of these components:

- Compensator (**C**)
- Prefilter (**F**)
- Plant (**G**)
- Sensor (**H**)

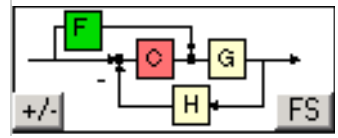
The default configuration is shown below.



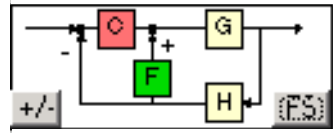
To cycle through the feedback structures, click the **FS** button. This figure shows the alternate feedback structures.



Compensator in the feedback path



Feedforward controller



Cascade configuration with filter F in the minor loop

Clicking the **+/-** button toggles between positive and negative feedback signs. Negative feedback is the default.

Tuning Compensators

The SISO Design Tool simplifies the task of designing compensators. Regardless of which views you have open--root-locus, Bode diagrams, or Nichols plots--there are three ways to alter compensator designs:

- Interactive graphics let you tune compensator gains and adjust dynamics (poles and zeros).
- Right-click menus allow you to add/remove dynamics and customize the view (for example, add a grid, zoom in/out, add design constraints, or customize plot properties).
- The Edit Compensator window is a GUI with fields for keyboard entry of gain values and pole/zero locations.

You can perform any of these tasks in the root locus, open-loop and prefilter Bode diagrams, or Nichols plots in the SISO Design Tool. Once you've added dynamics to your compensator, you can dynamically update pole and zero

locations by dragging them. The SISO Design Tool is designed so that a change in any one view is automatically reflected in other views in the GUI. In particular, the **Current Compensator** panel always reflects the current compensator design. The next sections discuss some of the ways you can tune compensators in different SISO Design Tool views.

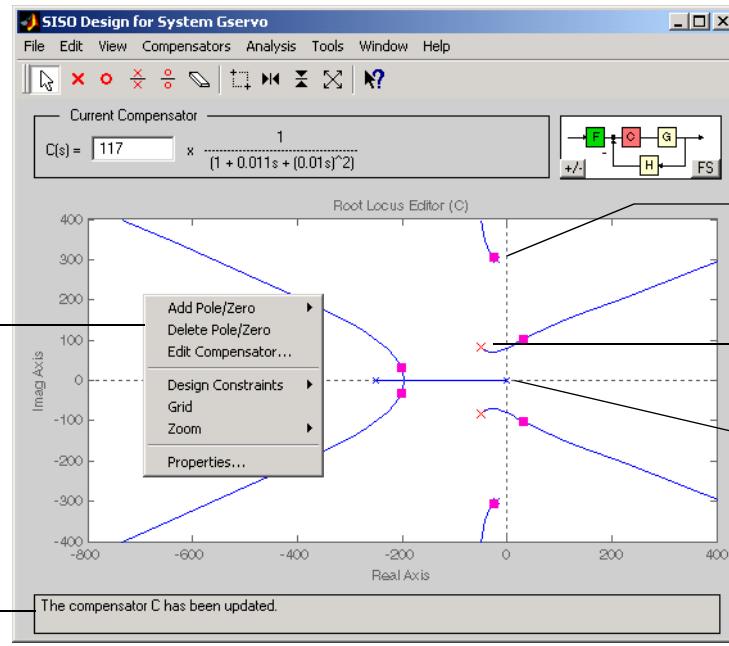
You can tune compensators in any of the views in the SISO Design Tool. These views include the following:

- "Root-Locus"
- "Open-Loop Bode Diagram"
- "Open-Loop Nichols Plot"
- "Prefilter Bode Diagram"

Root-Locus

You can tune your compensator using root-locus techniques. This figure shows an imported compensator and plant model; use the right-click menus and

interactive graphics features to add, adjust, and remove compensator dynamics.



Use the right-click menu to add poles and zeros to your compensator design.

Move any red square to change the compensator gain.

You can move the compensator poles and zeros (red x's and o's, respectively) with your mouse.

The blue x's and o's represent plant poles and zeros. You cannot move them.

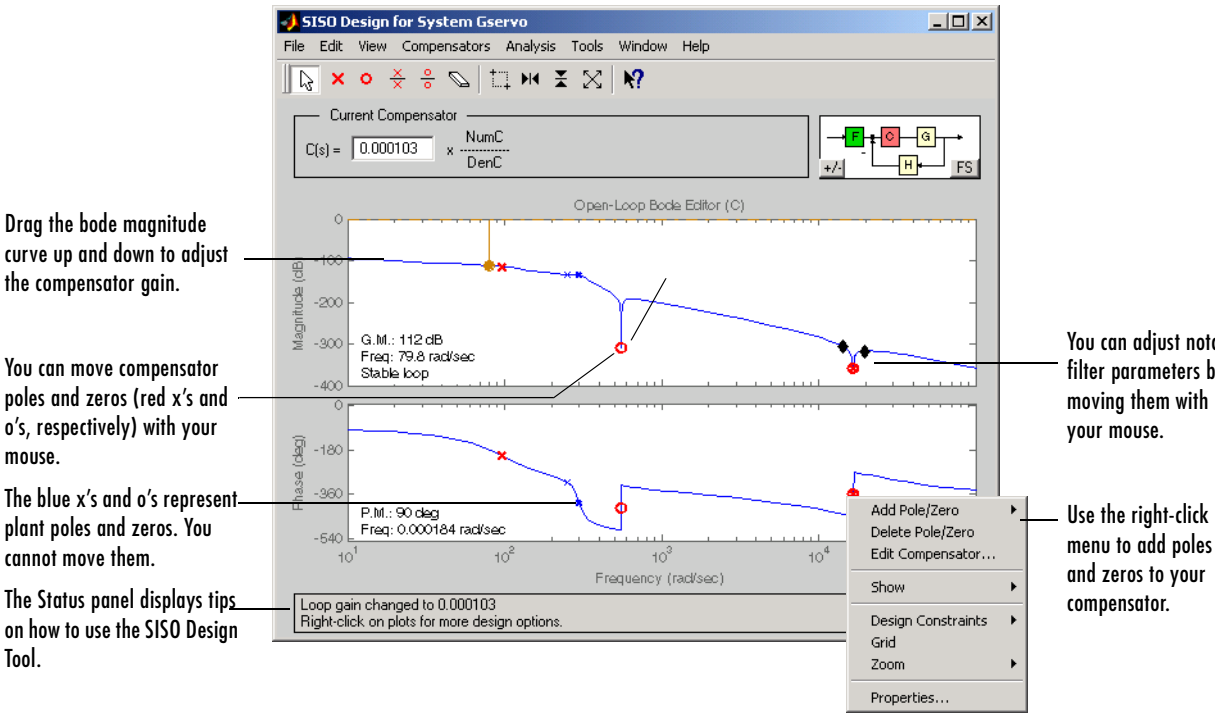
The Status panel displays tips on how to use the SISO Design Tool.

See Root Locus Design in *Getting Started with the Control System Toolbox* for an example of how to use root locus design techniques.

Open-Loop Bode Diagram

The SISO Design Tool supports the open-loop Bode diagram view of your system. You can use the right-click menu and interactive graphics features to

add, adjust, and remove compensator dynamics. This figure shows some of the features of the open-loop Bode diagram.

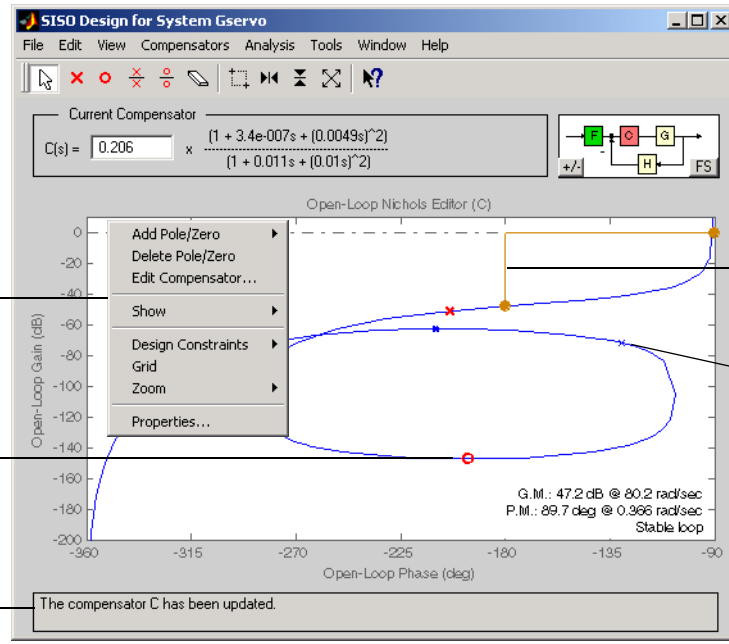


See Bode Diagram Design in *Getting Started with the Control System Toolbox* for an example of how to use Bode diagram design techniques.

Open-Loop Nichols Plot

An alternative method for compensator design is the open-loop Nichols plot. You can use the right-click menu and interactive graphics features to add,

adjust, and remove compensator dynamics. This figure shows some of the features of the open-loop Nichols plot.



Use the right-click menu to add poles and zeros to your compensator.

You can move compensator poles and zeros (red x's and o's, respectively) with your mouse.

The Status panel displays tips on how to use the SISO Design Tool.

The brown stems display the gain and phase margins.

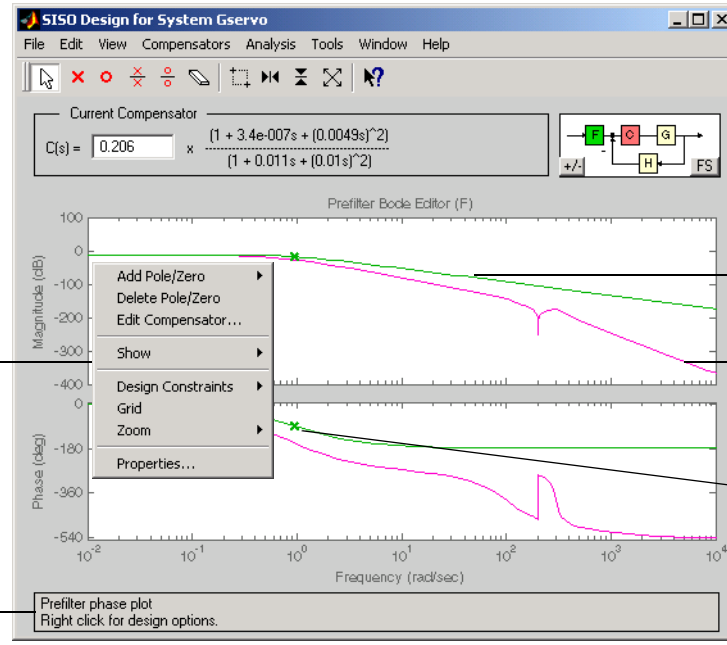
The blue x's and o's represent plant poles and zeros. You cannot move them.

See Nichols Plot Design in *Getting Started with the Control System Toolbox* for an example of how to use Nichols plot design techniques.

Prefilter Bode Diagram

The SISO Design Tool supports the prefilter Bode diagram view. You can add dynamics to your prefilter design using the right-click menu, and you can

adjust dynamics by dragging poles and zeros with your mouse. This figure shows some of the features.



Use the right-click menu to add poles and zeros to your prefilter design.

The Status panel displays tips on how to use the SISO Design Tool.

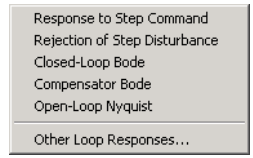
The green curve is the prefilter Bode magnitude. Move it up and down to adjust the prefilter gain.

The magenta curve is the closed-loop response from the prefilter input to the plant output.

The green x is a prefilter pole. Use your mouse to move this pole to a new location.

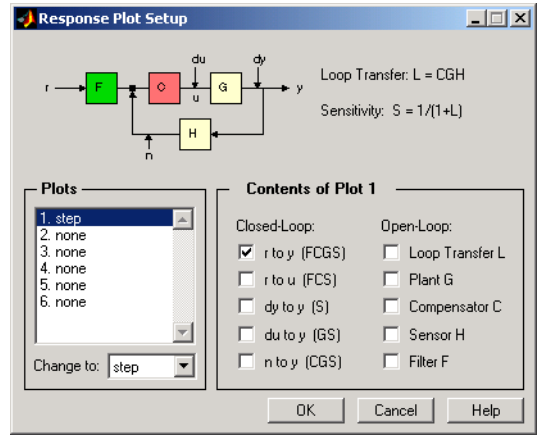
Viewing Loop Responses

The SISO Design Tool provides support for viewing loop responses for your system. To see available frequency and time domain responses, pull down the **Analysis** menu.



Choose the response you want to see; an LTI Viewer opens with the view plotted. For example, if you select **Response to Step Command**, you will see a closed-loop step response of your system.

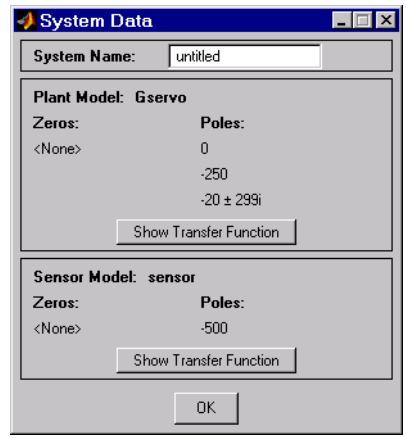
If you select **Other**, the **Response Plot Setup** window opens.



This window allows for more specialized loop responses. Click here for more information.

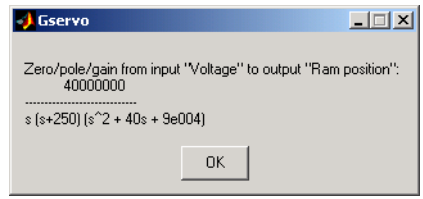
Viewing System Data

You can view data about your model by selecting **System Data** under the **View** menu.



The **System Data** window displays the poles and zeros of your imported plant and sensor models. Click **Show Transfer Function** to see the associated

transfer function. For example, this picture shows the `Gservo` plant's transfer function.



Type

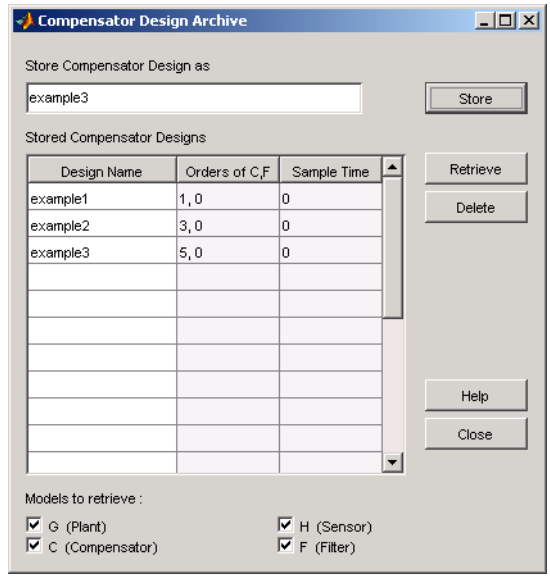
```
load ltiexamples
```

at the MATLAB prompt to load the `Gservo` plant model.

Storing and Retrieving Designs

The SISO Design Tool provides a graphical user interface (GUI's) for storing and retrieving compensator designs. Each design consists of a pair (**C**, **F**) of compensator and prefilter models

To open the **Compensator Design Archive** window, select **Store/Retrieve** from the **Compensators** menu.



You can use this window both to store and retrieve compensator designs.

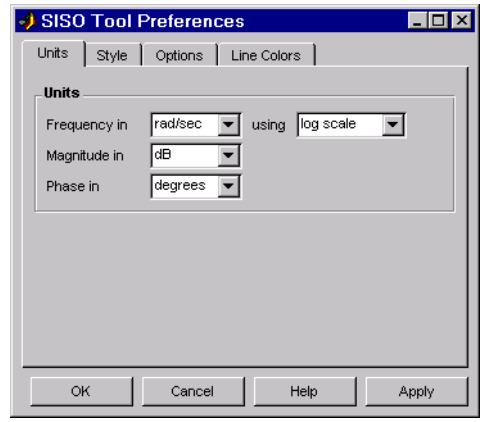
To store a design, specify the name you want to save it under and click **Store**. To retrieve any of the prefilter and/or compensator designs that you have created during a SISO Design Tool session, right-clicking on the Design Name you want to retrieve. Click **Retrieve** and the design is sent back to the SISO Design Tool.

Customizing the SISO Design Tool

The SISO Design Tool provides a graphical user interface (GUI), called the **SISO Tool Preferences** window, for customizing units, linestyles, axes foreground and linestyle colors, fonts, compensator format, and Bode plot options. Any options you set in this window apply to all the plots of the current instance of the SISO Design Tool. If you open another instance of the SISO Design Tool, it inherits its options from the Toolbox Preferences Editor.

Opening the SISO Tool Preferences Window

To open the SISO Tool Preferences window, select **SISO Tool Preferences** from the **Edit** menu.



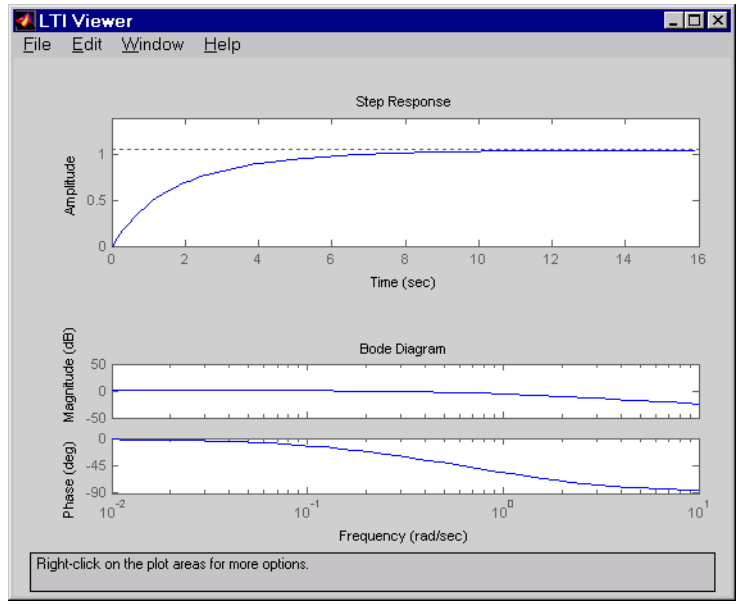
There are four panes in the **SISO Tool Preferences** window:

- Units
- Style
- Options
- Line Colors

For complete descriptions of options available on the four panes, click on the links.

LTI Viewer

LTI Viewer is a graphical user interface (GUI) that simplifies the analysis of linear, time-invariant systems. You use the LTI Viewer 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.



The LTI Viewer 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.

Right-Click Menu

Using right-click menu options, you can access several LTI Viewer controls and options, including:

- **Plot Type** — Changes the plot type

- **Systems** — Selects or deselects any of the models loaded in the LTI Viewer
- **Characteristics** — Displays key response characteristics and parameters
- **Grid** — Adds grids to your plot
- **Properties** — Opens the **Property Editor**, where you can customize plot attributes

In addition to right-click menus, all response plots include data markers. These allow you to scan the plot data, identify key data, and determine the source system for a given plot.

LTI Viewer Toolbar

The LTI Viewer has a tool bar that you can use to do the following:

- Open a new LTI Viewer
- Print
- Zoom in and out

Basic LTI Viewer Tasks

For descriptions of basic tasks you can perform with the LTI Viewer, see the other **Help** menu items:

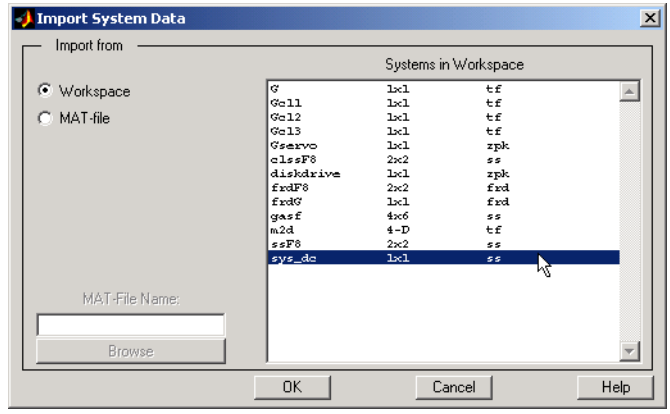
- “Importing and Exporting Models”
- “Selecting Response Types”
- “Analyzing MIMO Models”
- “Customizing the LTI Viewer”

For examples of how to use the LTI Viewer, see Analyzing Models in *Getting Started with the Control System Toolbox*. See LTI Viewer for descriptions of all the features available in the LTI Viewer.

See `ltiview` for help on the function that opens an LTI Viewer.

Importing and Exporting Models

To import models into the LTI Viewer, select **Import** under the **Edit** menu. This opens the **LTI Browser**, shown below.



Use the **LTI Browser** to import LTI models into or from the LTI Viewer 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.

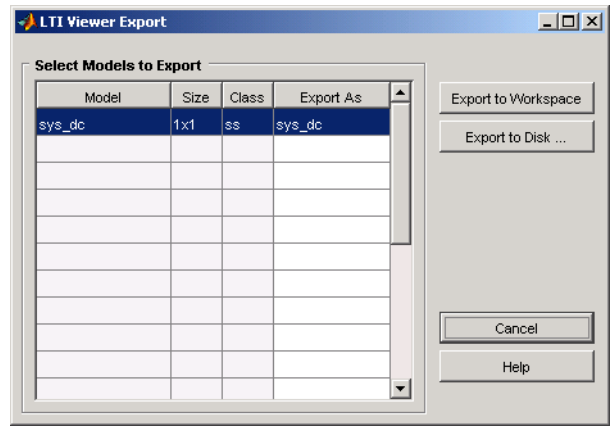
Alternatively, you can directly import a model into the LTI Viewer using the ltiview function, as in

```
ltiview({'step', 'bode'}, modelname)
```

See the `ltiview` function for more information.

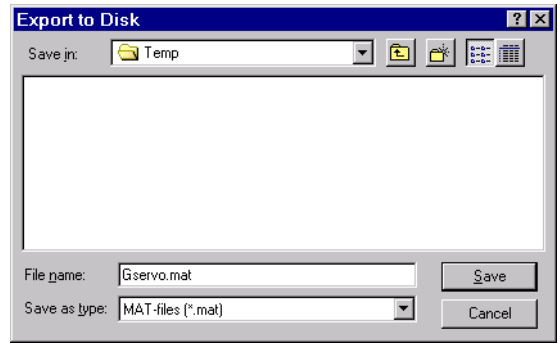
Exporting Models

Use **Export** in the **File** menu to open the **LTI Viewer Export** window, shown below.



The **LTI Viewer Export** window lists all the models with responses currently displayed in your LTI Viewer. You can export models back to the MATLAB workspace or to disk. In the latter case, the Control System Toolbox saves the files as MAT-files.

To export single or multiple models, follow the steps described in the importing models section above. If you choose **Export to Disk**, this window opens.



Choose a name for your model(s) and click **Save**. Your models are stored in a MAT-file.

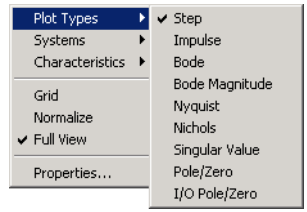
Selecting Response Types

There are two methods for selecting response plots in the LTI Viewer:

- Selecting **Plot Type** from the right-click menus
- Opening the **Plot Configurations** window

Right Click Menu: Plot Type

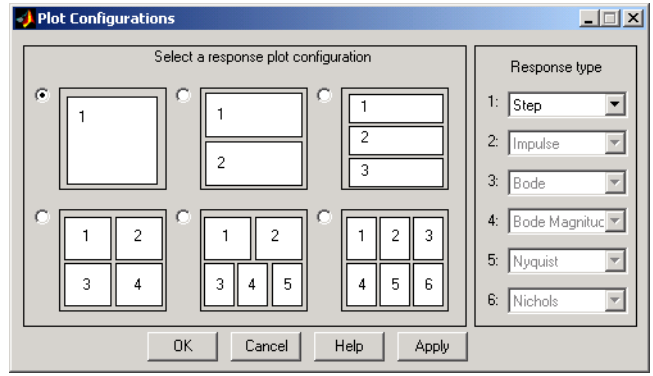
If you have a plot open in the LTI Viewer, you can switch to any other response plot available by selecting **Plot Type** from the right click menu.



To change the response plot, select the new plot type from the **Plot Type** submenu. The LTI Viewer automatically displays the new response plot.

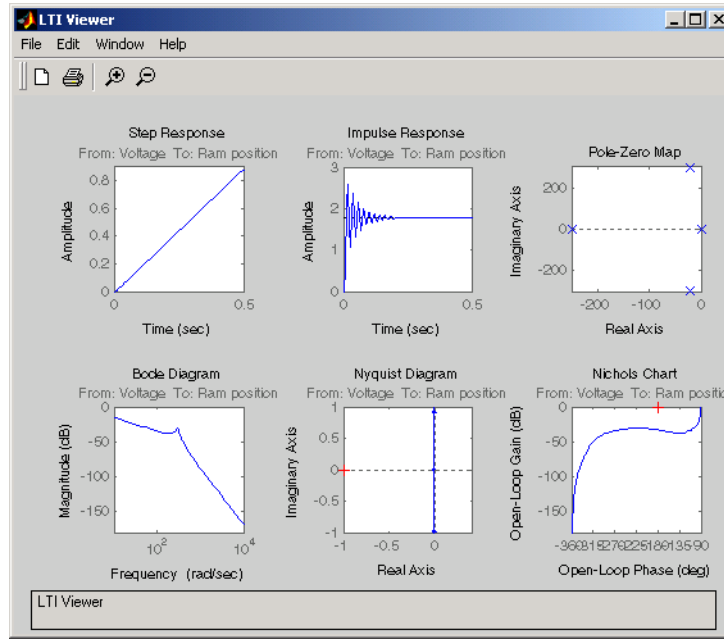
Plot Configurations Window

The Plot Type feature of the right-click menu works on existing plots, but you can also add plots to an LTI Viewer by using the **Plot Configurations** window. By default, the LTI Viewer opens with a closed-loop step response. To reconfigure an open viewer, select **Plot Configuration** in the **Edit** menu.



Use the radio buttons to select the number of plots you want displayed in your LTI Viewer. 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 LTI Viewer to contain up to six response plots.



Available response plots include: step, impulse, Bode (magnitude and phase, or magnitude only), Nyquist, Nichols, sigma, pole/zero maps, and I/O pole/zero maps.

Analyzing MIMO Models

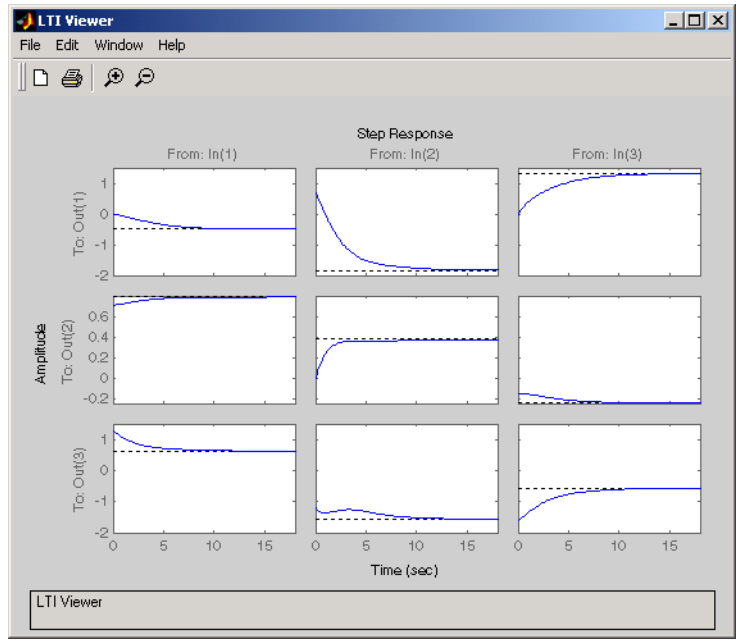
If you import a MIMO system, or an LTI array containing multiple linear models, you can use special features of the right-click menu to group the response plots by input/output (I/O) pairs or select individual plots for display. For example, if you randomly generate a 3-input, 3-output MIMO system,

```
sys_mimo=rss(3,3,3);
```

and open an LTI Viewer,

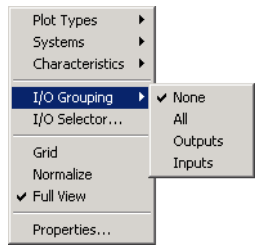
```
ltiview(sys_mimo);
```

the default is an unwrapped set of 9 plots, one from each input to each output.

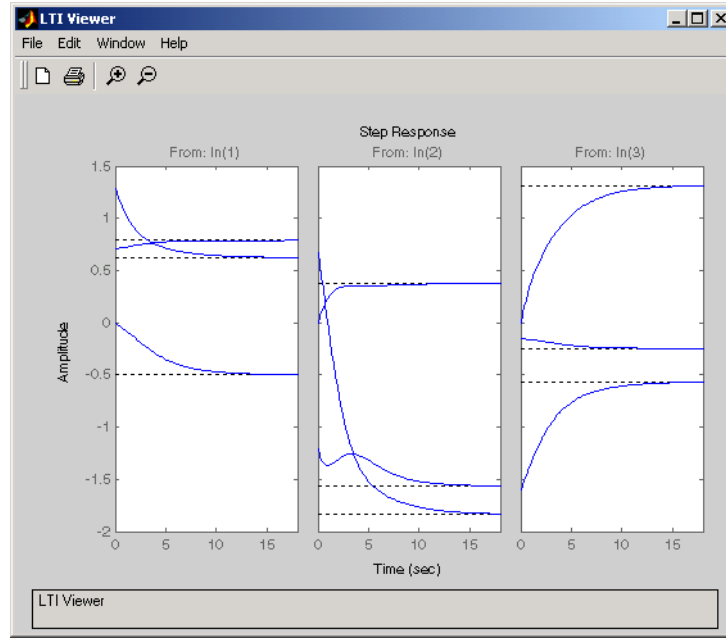


I/O Grouping

You can group this by inputs, by outputs, or both by selecting **I/O Grouping** and then **Inputs**, **Outputs**, or **All**, respectively, from the right-click menu.



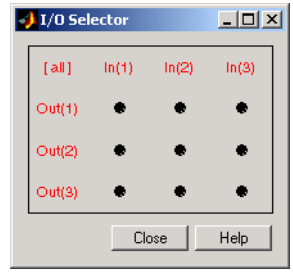
For example, if you select **Outputs**, the LTI Viewer reconfigures into 3 plots, one for each input.



Selecting **None** returns to the default configuration, where all I/O pairs are displayed individually.

I/O Selector

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]

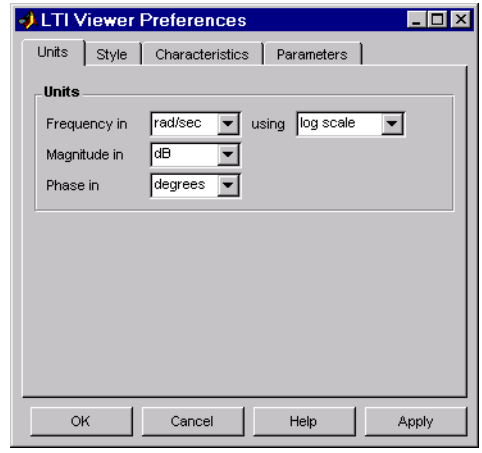
Using these options, you can inspect individual I/O pairs, or look at particular I/O channels in detail.

Customizing the LTI Viewer

The LTI Viewer has a tool preferences editor, which allows you to set default characteristics for specific instances of LTI Viewers. If you open a new instance of either, each defaults to the characteristics specified in the **Toolbox Preferences** editor.

LTI Viewer Preferences Editor

Select **Viewer Preferences** in the **Edit** menu of the LTI Viewer to open the **LTI Viewer Preferences** editor. This figure shows the editor open to its first pane.



The **LTI Viewer 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

If you want to customize the settings for all instances of LTI Viewers, see the Toolbox Preferences editor.

Index

M

N

O

P

R

S

T

U

Z