

Control System Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

Version 4.1

How to Contact The MathWorks:



508-647-7000

Phone



508-647-7001

Fax



The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500

Mail



<http://www.mathworks.com>
<ftp.mathworks.com>
<comp.soft-sys.matlab>

Web
Anonymous FTP server
Newsgroup



support@mathworks.com
suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
subscribe@mathworks.com
service@mathworks.com
info@mathworks.com

Technical support
Product enhancement suggestions
Bug reports
Documentation error reports
Subscribing user registration
Order status, license renewals, passcodes
Sales, pricing, and general information

Control System Toolbox User's Guide

© COPYRIGHT 1992 - 1998 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

U.S. GOVERNMENT: If Licensee is acquiring the Programs on behalf of any unit or agency of the U.S. Government, the following shall apply: (a) For units of the Department of Defense: the Government shall have only the rights specified in the license under which the commercial computer software or commercial software documentation was obtained, as set forth in subparagraph (a) of the Rights in Commercial Computer Software or Commercial Software Documentation Clause at DFARS 227.7202-3, therefore the rights set forth herein shall apply; and (b) For any other unit or agency: NOTICE: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction, and disclosure are as set forth in Clause 52.227-19 (c)(2) of the FAR.

MATLAB, Simulink, Handle Graphics, and Real-Time Workshop are registered trademarks and Stateflow and Target Language Compiler are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: July 1992 First printing
December 1996 Second printing Revised for MATLAB 5
January 1998 Third printing Revised for MATLAB 5.2

Preface

Installation	3
Getting Started	4
What's New in Version 4?	5
Typographic Conventions	6

Quick Start

1

LTI Models	1-3
MIMO Systems	1-5
Model Conversion	1-5
LTI Properties	1-7
Model Characteristics	1-9
Operations on LTI Models	1-10
Continuous/Discrete Conversions	1-12
Time and Frequency Response	1-13
The LTI Viewer	1-16
System Interconnections	1-18

Control Design Tools	1-19
The Root Locus Design GUI	1-22

LTI Models

2

Introduction	2-2
LTI Objects	2-2
Precedence Rules	2-3
Viewing LTI Systems As Matrices	2-4
Command Summary	2-5
Creating LTI Models	2-6
Transfer Function	2-6
Zero-Pole-Gain	2-8
State Space	2-10
Descriptor State-Space Models	2-12
Discrete-Time Models	2-13
Discrete Transfer Functions in DSP Format	2-14
Data Retrieval	2-15
LTI Properties	2-18
Generic Properties	2-18
Model-Specific Properties	2-19
Setting LTI Properties	2-21
Accessing Property Values	2-23
Direct Property Referencing	2-24
Model Conversion	2-26
Explicit Conversion	2-26
Automatic Conversion	2-27
Caution	2-27
Operations on LTI Models	2-28
Precedence and Property Inheritance	2-28

Arithmetic Operations	2-29
Addition and Subtraction	2-29
Multiplication	2-30
Inversion and Related Operations	2-31
Concatenation	2-31
Transposition and Pertransposition	2-33
Extracting and Modifying Subsystems	2-33
Resizing LTI Systems	2-35
Input Delays	2-37
Specifying Input Delays	2-37
Supported Functionalities	2-38
Pade Approximation of Input Delays	2-39
Continuous/Discrete Conversions	2-40
Zero-Order Hold	2-40
First-Order Hold	2-42
Tustin Approximation	2-42
Tustin with Frequency Prewarping	2-43
Matched Poles and Zeros	2-43
Discretization of Delay Systems	2-43
Resampling Discrete-Time Models	2-45
Simulink Block for LTI Systems	2-47
References	2-49

Model Analysis Tools

3

General Characteristics	3-2
Model Dynamics	3-4
State-Space Realizations	3-6

Time and Frequency Response	3-7
Time Responses	3-7
Frequency Response	3-9
Plotting and Comparing Multiple Systems	3-11
Customized Plots	3-13
 Model Order Reduction	 3-16

The LTI Viewer

4

Introduction	4-2
Getting Started	4-2
Getting Help	4-5
Static Help	4-5
Interactive Help	4-6
 The LTI Viewer Workspace	 4-7
Using the Workspace Browser	4-7
Refreshing the Workspace Browser	4-7
Selecting LTI Objects in the Workspace Browser	4-7
Interpreting the Selected Browser	4-9
Displaying/Hiding a Particular System's Response	4-9
Deleting an LTI Object from the Selected Field	4-10
Selection or Deselection of Several Models in a Browser ..	4-10
 SISO LTI Viewer Example	 4-11
Calculating Response Characteristics	4-11
Changing the Displayed Response Type	4-12
Reading Information off the Response Plots	4-13
Quick Exercise	4-14
Zooming into Regions of the Response Plots	4-16
Generating Hard Copies of the Response Plots	4-17
Showing/Hiding the Viewer Controls	4-18
Opening a New LTI Viewer	4-18
 MIMO LTI Viewer Example	 4-19

Setting up the MIMO Example	4-21
Rearranging the Display	4-22
Specifying Which I/O Channels to Display	4-23
Changing the Displayed MIMO Response Type	4-25
Accessing Additional LTI Viewer Controls	4-26
Response Preferences	4-28
Setting Response Durations	4-28
Specifying Settling Time Target Percentage	4-31
Changing Bode Diagram Units	4-31
Linestyle Preferences	4-33
Altering the Response Plot Line Style Properties	4-34
Determining the Line Property Order	4-35
Changing the Line Property Order	4-35
Simulink LTI Viewer	4-37
Using the Simulink LTI Viewer	4-37
A Sample Analysis Task	4-38
Opening the Simulink LTI Viewer	4-39
Specifying the Simulink Model Portion for Analysis	4-41
Adding Input Point or Output Point Blocks to the Diagram	4-41
Removing Input Points and Output Points	4-44
Specifying Open- Versus Closed-Loop Analysis Models ...	4-44
Setting the Operating Conditions	4-45
Modifying the Block Parameters	4-46
Performing Linear Analysis	4-46
Importing a Linearized Analysis Model to the LTI Viewer .	4-47
Analyzing the Bode Plot of the Linearized Analysis Model	4-48
Specifying Another Analysis Model	4-48
Comparing the Bode Plots of the	
Two Linearized Analysis Models	4-48
Saving Analysis Models	4-50

System Modeling	5-3
Root Locus Design	5-4
Pole Placement	5-6
State-Feedback Gain Selection	5-6
State Estimator Design	5-6
Pole Placement Tools	5-7
LQG Design	5-9
Optimal State-Feedback Gain	5-10
Kalman State Estimator	5-10
LQG Regulator	5-11
LQG Design Tools	5-11

The Root Locus Design GUI

Introduction	6-2
A Servomechanism Example	6-4
Controller Design Using the Root Locus Design GUI	6-6
Opening the Root Locus Design GUI	6-6
Importing Models into the Root Locus Design GUI	6-7
Opening the Import LTI Design Model Window	6-9
Choosing a Feedback Structure	6-10
Specifying the Design Model	6-11
Changing the Gain Set Point and Zooming	6-13
Dragging Closed-loop Poles to Change the Gain Set Point	6-14
Zooming	6-15
Storing and Retrieving Axes Limits	6-19
Displaying System Responses	6-20

Designing the Compensator to Meet Specifications	6-23
Specifying Design Region Boundaries on the Root Locus . .	6-24
Placing Compensator Poles and Zeros	6-26
Placing Compensator Poles and Zeros	
Using the Root Locus Toolbar	6-27
Editing Compensator Pole and Zero Locations	6-30
Activating the LTI Viewer	6-35
Saving the Compensator and Models	6-36
Additional Root Locus Design GUI Features	6-38
Specifying Design Models: General Concepts	6-38
Creating Models Manually Within the GUI	6-38
Designating the Model Source	6-39
Getting Help with the Root Locus Design GUI	6-39
Using the Help Menu	6-40
Using the Status Bar for Help	6-40
Tooltips	6-40
Erasing Compensator Poles and Zeros	6-41
Listing Poles and Zeros	6-41
Printing the Root Locus	6-42
Drawing a Simulink Diagram	6-43
Converting Between Continuous and Discrete Models	6-44
Clearing Data	6-45
References	6-46

Design Case Studies

7

Yaw Damper for a Jet Transport	7-3
Open-Loop Analysis	7-6
Root Locus Design	7-9
Washout Filter Design	7-14
Hard-Disk Read/Write Head Controller	7-19
LQG Regulation	7-30

Process and Disturbance Models	7-30
Model Data for the x-Axis	7-33
Model Data for the y-Axis	7-33
LQG Design for the x-Axis	7-33
LQG Design for the y-Axis	7-41
Cross-Coupling Between Axes	7-42
MIMO LQG Design	7-46
Kalman Filtering	7-49
Discrete Kalman Filter	7-49
Steady-State Design	7-50
Time-Varying Kalman Filter	7-56
Time-Varying Design	7-57
References	7-62

Reliable Computations

8

Conditioning and Numerical Stability	8-4
Conditioning	8-4
Numerical Stability	8-6
Choice of LTI Model	8-8
State Space	8-8
Transfer Function	8-8
Zero-Pole-Gain Models	8-14
Scaling	8-15
Summary	8-17
References	8-18

Category Tables	9-3
Modal Form	9-26
Companion Form	9-26
Example 1	9-29
Example 2	9-30
Continuous Time	9-56
Discrete Time	9-56
H_2 Norm	9-142
Infinity Norm	9-142
Creation of State-Space Models	9-195
Conversion to State Space	9-196
Example 1	9-196
Example 2	9-197
Creation of Transfer Functions	9-209
Conversion to Transfer Function	9-210
Example 1	9-211
Example 2	9-211
Example 3	9-212
Creation of Zero-Pole-Gain Models	9-221
Conversion to Zero-Pole-Gain Form	9-223
Example 1	9-223
Example 2	9-223

Preface

Installation	3
Getting Started	4
What’s New in Version 4?	5
Typographic Conventions	6

MATLAB® has a rich collection of functions immediately useful to the control engineer or system theorist. Complex arithmetic, eigenvalues, root-finding, matrix inversion, and FFTs are just a few examples of MATLAB's important numerical tools. More generally, MATLAB's linear algebra, matrix computation, and numerical analysis capabilities provide a reliable foundation for control system engineering as well as many other disciplines.

The Control System Toolbox uses MATLAB matrix structures and builds upon the foundations of MATLAB to provide functions specialized to control engineering. The Control System Toolbox is a collection of algorithms, expressed mostly in M-files, which implements common control system design, analysis, and modeling techniques.

Control systems can be modeled as transfer functions or in zero-pole-gain or state-space form, allowing you to use both classical and modern techniques. You can manipulate both continuous-time and discrete-time systems. Conversions between various model representations are provided. Time responses, frequency responses, and root loci can be computed and graphed. Other functions allow pole placement, optimal control, and estimation. Finally, and most importantly, tools that are not found in the toolbox can be created by writing new M-files.

Installation

Instructions for installing the Control System Toolbox can be found in the *MATLAB Installation Guide* for your platform. We recommend that you store the files from this toolbox in a directory named `control` off the main `matlab` directory. To determine if the Control System Toolbox is already installed on your system, check for a subdirectory named `control` within the main toolbox directory or folder.

Five demonstration files are available. The demonstration M-file called `ctrl demo.m` runs through some basic control design and analysis functions and the demonstration files `jet demo.m`, `diskdemo.m`, `mlldemo.m`, and `kalmdemo.m` go through the design case studies described in Chapter 7. To start a demo, type `ctrl demo`, for example, at the MATLAB prompt.

Getting Started

If you are a new user, begin with Chapters 2 and 3 for an overview of

- How to specify and manipulate linear time-invariant models
- How to analyze such models and plot their time and frequency response

If you are an experienced toolbox user, see

- The next section for details on the Version 4 release
- Chapter 1 for an overview of the new features
- Chapter 4 for an introduction to the LTI Viewer GUI
- Chapter 6 for an introduction to the Root Locus Design GUI

All toolbox users should use Chapter 9 for information on specific tools. For functions, reference descriptions include a synopsis of the function's syntax, as well as a complete explanation of options and operation. Many reference descriptions also include helpful examples, a description of the function's algorithm, and references to additional reading material. For GUI-based tools, the descriptions include options for invoking the tool.

What's New in Version 4?

Version 4 of the Control System Toolbox contains several major new features including:

- The *LTI objects*, customized data structures that enable you to manipulate linear-time invariant models as single MATLAB variables
- A comprehensive set of tools for multi-input/multi-output system analysis
- New graphical user interfaces for system response analysis (LTI Viewer) and compensator design (Root Locus Design GUI)
- Support for continuous-time input delays
- Streamlined LQG design tools

Thanks to the LTI objects, the overall number of commands has been reduced and the syntax of most commands simplified. Thus, Version 4 is easier to use while offering more powerful analysis and design tools.

Version 4 is backward compatible, which means that the Version 3.x syntax is still fully supported. While the online help emphasizes the new LTI-oriented syntax, all M-files still contain the help for previous versions. For example, to find help on the Version 3.x syntax of the function `damp`, display the content of the M-file `damp.m` with the command

```
type damp
```

The old help appears immediately after the Version 4 help in the resulting display.

Typographic Conventions

To Indicate	This Guide Uses	Example
Example code	Monospace type	To assign the value 5 to A, enter A = 5
Function names	Monospace type	The cos function finds the cosine of each array element.
Function syntax	Monospace type for text that must appear as shown. <i>Monospace italics</i> for components you can replace with any variable.	The magi c function uses the syntax M= magi c(n)
Keys	Boldface	Press the Return key.
Mathematical expressions	Variables in <i>italics</i> . Functions, operators, and constants in standard type.	This vector represents the polynomial $p = x^2 + 2x + 3$
MATLAB output	Monospace type	MATLAB responds with A = 5
Menu names, menu items, and controls	Boldface	Choose the File menu.

To Indicate	This Guide Uses	Example
New terms	<i>italics</i>	An <i>array</i> is an ordered collection of information.

Quick Start

LTI Models	1-3
MIMO Systems	1-5
Model Conversion	1-5
LTI Properties	1-7
Model Characteristics	1-9
Operations on LTI Models	1-10
Continuous/Discrete Conversions	1-12
Time and Frequency Response	1-13
The LTI Viewer	1-16
System Interconnections	1-18
Control Design Tools	1-19
The Root Locus Design GUI	1-22

This chapter is intended for users already familiar with Version 3.1 of the Control System Toolbox. It gives a quick overview of the new features and the simplified syntax of the Version 4 release of the Control Systems Toolbox.

LTI Models

You can specify linear time-invariant (LTI) systems as transfer functions, zero/pole/gain models, or state-space models. The corresponding commands are:

```
sys = tf(num, den)      % transfer function
sys = zpk(z, p, k)      % zero/pole/gain
sys = ss(a, b, c, d)    % state space
```

For more information, see page 2-6.

The output `sys` is a model-specific data structure called TF object, ZPK object, and SS object, respectively. These objects store the model data and enable you to manipulate the LTI system as a single entity; see page 2-2 for more information. For example, type:

```
h = tf(1, [1 1]);      % creates transfer function 1/(s+1)
h
```

MATLAB responds with:

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

Type:

```
1+h
```

MATLAB responds with:

```
Transfer function:
s + 2
-----
s + 1
```

To create discrete-time systems, simply append the sample time `Ts` to the previous calling sequences:

```
sys = tf(num, den, Ts)
sys = zpk(z, p, k, Ts)
sys = ss(a, b, c, d, Ts)
```

For more information, see page 2-12.

Note the customized display for discrete systems. Type:

```
sys = zpk(0.5, [-0.1 0.3], 1, 0.05)
```

MATLAB responds with:

Zero/pole/gain:

(z-0.5)

(z+0.1) (z-0.3)

Sampling time: 0.05

Finally, you can retrieve the system data stored in the LTI object `sys` with the following commands (see page 2-15 for more information):

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

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

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

or by direct structure-like referencing (see page 2-23 for more information), as in this example:

```
sys.num
```

```
sys.a
```

```
sys.Ts
```


MIMO Systems

You can also create multi-input/multi-output (MIMO) models, including arbitrary MIMO transfer functions and zero/pole/gain models. MIMO transfer functions are arrays of single-input/single-output (SISO) transfer functions where each SISO entry is characterized by its numerator and denominator. Cell arrays provide an ideal means to specify the resulting arrays of numerators and denominators; see page 2-7 for more information. For example,

```
num = {0.5, [1 1]}           % 1-by-2 cell array of numerators
den = {[1 0], [1 2]}         % 1-by-2 cell array of denominators
H = tf(num, den)
```

creates the one-output/two-input transfer function

$$H(s) = \begin{bmatrix} \frac{0.5}{s} & \frac{s+1}{s+2} \end{bmatrix}$$

Alternatively, you can create the same transfer function by matrix-like concatenation of its SISO entries:

```
h11 = tf(0.5, [1 0])          % 0.5/s
h12 = tf([1 1], [1 2])        % (s+1)/(s+2)
H = [h11, h12]
```

MIMO zero/pole/gain systems are defined in a similar fashion. For example, the following commands specify $H(s)$ above as a zero/pole/gain model:

```
Zeros = {[ ], -1}             % Note: use [ ] when no zero
Poles = {0, -2}
Gains = [0.5, 1]              % Note: use regular matrix for gains
H = zpk(Zeros, Poles, Gains)
```

Model Conversion

The functions `tf`, `zpk`, and `ss` also perform model conversion; see page 2-25 for more information. For example,

```
sys_ss = ss(sys)
```

converts some arbitrary LTI model `sys` to state space. Similarly, if you type

```
h = tf(1, [1 2 1])      % transfer function 1/(s^2+2s+1)
zpk(h)
```

MATLAB derives the zero/pole/gain representation of the transfer function `h`:

Zero/pole/gain:

$$\frac{1}{(s+1)^2}$$

LTI Properties

In addition to the model data, the TF, ZPK, and SS objects can store extra information, such as the system sample time, input delays, and input or output names. The various pieces of information that can be attached to an LTI object are called the *LTI properties*; see page 2-17 for more information. Use `set` to list all LTI properties and their assignable values, and `get` to display the current properties of the system. For example, type:

```
sys = ss(-1, 1, 1, 0, 0.5)    % 1/(z+1), sample time = 0.5 sec.
set(sys)
```

MATLAB returns:

```
a:  Nx-by-Nx matrix (Nx = no. of states)
b:  Nx-by-Nu matrix (Nu = no. of inputs)
c:  Ny-by-Nx matrix (Ny = no. of outputs)
d:  Ny-by-Nu matrix
e:  Nx-by-Nx matrix (or empty)
StateName:  Nx-by-1 cell array of strings
Ts:  scalar
Td:  vector of length Nu (for continuous systems only)
InputName:  Nu-by-1 cell array of strings
OutputName:  Ny-by-1 cell array of strings
Notes:  array or cell array of strings
UserData:  arbitrary
```

Type:

```
get(sys)
```

MATLAB returns:

```
a = -1
b = 1
c = 1
d = 0
e = []
StateName = {' '}
Ts = 0.5
Td = []
InputName = {' '}
OutputName = {' '}
Notes = {}
UserData = []
```

You can also use `set` and `get` to access/modify LTI properties in a Handle Graphics[®] fashion; see page 2-20 for more information. For example, give names to the input and output of the SISO state-space model `sys`. Type:

```
set(sys, 'inputname', 'thrust', 'outputname', 'velocity')
get(sys, 'inputn')
```

MATLAB responds with:

```
ans =

'thrust'
```

Finally, an alternative for accessing or modifying a single property is the structure-like syntax. For example, type:

```
sys.Ts = 0.3;           % Set sample time to 0.3 sec.
sys.Ts                 % Get sample time value
```

MATLAB returns:

```
ans =

3.0000e-01
```

Model Characteristics

The Control System Toolbox contains commands to query such model characteristics as the I/O dimensions, poles, zeros, and DC gain. See page 3-2 for more information. These commands apply to both continuous- and discrete-time systems. Their LTI-based syntax is summarized below (with `sys` denoting an arbitrary LTI model):

<code>size(sys)</code>	% number of inputs and outputs
<code>isct(sys)</code>	% returns 1 for continuous systems
<code>isdt(sys)</code>	% returns 1 for discrete systems
<code>pole(sys)</code>	% system poles
<code>tzzero(sys)</code>	% system (transmission) zeros
<code>dcgain(sys)</code>	% DC gain
<code>norm(sys)</code>	% system norms (H2 and Linfinity)
<code>covar(sys, W)</code>	% covariance of response to white noise
<code>pade(sys)</code>	% Pade approximation of input delays

Operations on LTI Models

You can perform simple matrix operations on LTI systems, such as addition, multiplication, or concatenation; see page 2-28 for more information. Thanks to MATLAB object-oriented programming capabilities, these operations assume appropriate functionalities when applied to LTI systems. For example, addition performs a parallel interconnection. Type:

```
tf(1, [1 0]) + tf([1 1], [1 2])    % 1/s + (s+1)/(s+2)
```

MATLAB responds:

Transfer function:

$$s^2 + 2s + 2$$

$$s^2 + 2s$$

Multiplication performs a series interconnection. Type:

```
2 * tf(1, [1 0]) * tf([1 1], [1 2])    % 2*1/s*(s+1)/(s+2)
```

MATLAB responds:

Transfer function:

$$2s + 2$$

$$s^2 + 2s$$

If the operands are models of different types, the resulting model type is determined by precedence rules; see page 2-3 for more information. State-space models have highest precedence while transfer functions have lowest precedence. Hence the sum of a transfer function and a state-space model is always a state-space model.

Other available operations include system inversion, transposition, and pertransposition; see page 2-32 for more information. Matrix-like subindexing and subassignment are also supported; see page 2-32 for more information. For instance, if `sys` is a MIMO system with two inputs and three outputs,

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

extracts the subsystem describing the relation between the first input and third output. Note that row indices select the outputs while column indices select the inputs. Similarly,

```
sys(3, 1) = tf(1, [1 0])
```

redefines the relation between the first input and third output as an integrator.

Continuous/Discrete Conversions

The commands `c2d`, `d2c`, and `d2d` perform continuous to discrete, discrete to continuous, and discrete to discrete (resampling) conversions, respectively (see page 2-39 for more information):

```
sysd = c2d(sysc, Ts)    % discretization w/ sample period Ts
sysc = d2c(sysd)        % equivalent continuous-time model
sysd1= d2d(sysd, Ts)    % resampling at the period Ts
```

Various discretization/interpolation methods are available, including zero-order hold (default), first-order hold, Tustin approximation with or without prewarping, and matched zero-pole. For example,

```
sysd = c2d(sysc, Ts, 'foh')    % uses first-order hold
sysc = d2c(sysd, 'tustin')     % uses Tustin approx.
```


Time and Frequency Response

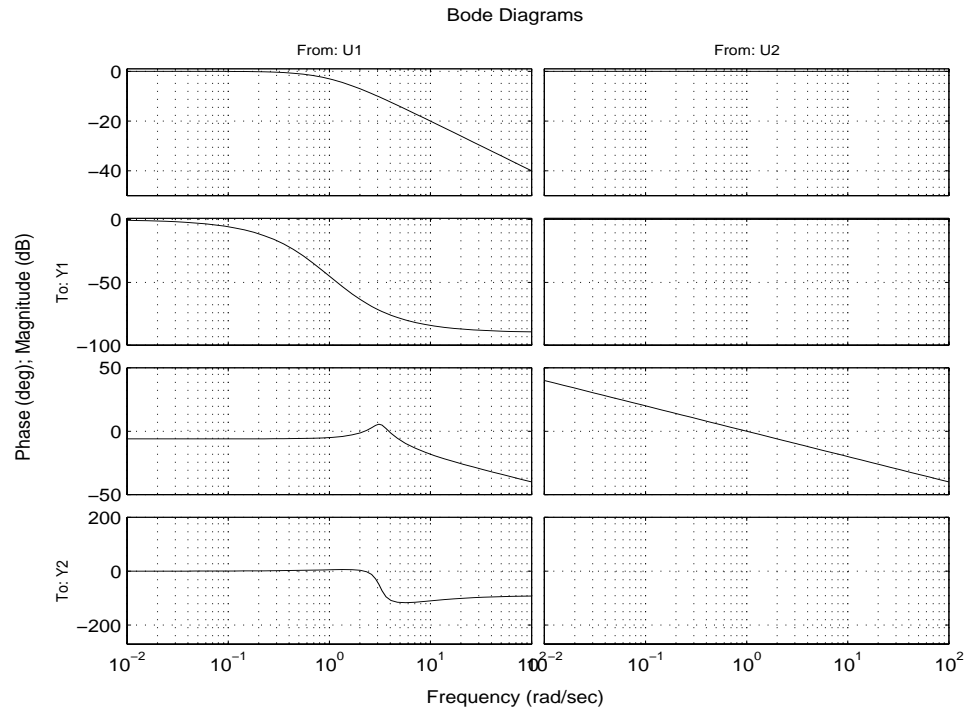
The following commands produce various time and frequency response plots for LTI systems (see page 3-7 for more information):

<code>step(sys)</code>	% step response
<code>impulse(sys)</code>	% impulse response
<code>initial(sys, x0)</code>	% undriven response to initial condition
<code>lsim(sys, u, t, x0)</code>	% response to input u
<code>bode(sys)</code>	% Bode plot
<code>nyquist(sys)</code>	% Nyquist plot
<code>nichols(sys)</code>	% Nichols plot
<code>sigma(sys)</code>	% singular value plot
<code>freqresp(sys, w)</code>	% complex frequency response

These commands work for both continuous- and discrete-time LTI models `sys` without restriction on the number of inputs or outputs. For MIMO systems, they produce an array of plots with one plot per I/O channel. For example,

```
sys = [tf(1, [1 1]) 1 ; tf([1 5], [1 1 10]) tf(-1, [1 0])]
bode(sys)
```

produces the Bode plot shown below.



To superimpose and compare the responses of several LTI systems, use the syntax

```
bode(sys1, sys2, sys3, ...)
```

You can also control the plot style by specifying a color/linestyle/marker for each system, much as with the `plot` command; see page 3-11 for more information. For example,

```
bode(sys1, 'r', sys2, 'b--')
```

draws the response of `sys1` with a red solid line and the response of `sys2` with a dashed blue line.

These commands automatically determine an appropriate simulation horizon or frequency range based on the system dynamics. To override the default range, type:

```
step(sys, tfinal)           % final time = tfinal  
bode(sys, {wmin, wmax})     % freq. range = [wmin, wmax]
```

The LTI Viewer

You can also analyze time and frequency domain responses using the LTI Viewer; see page 4-1 for more information. The LTI Viewer is an interactive user interface that assists you with the analysis of LTI model responses by facilitating such functions as:

- Toggling between types of response plots
- Plotting responses of several LTI models
- Zooming into regions of the response plots
- Calculating response characteristics, such as settling time
- Displaying different I/O channels
- Changing the plot styles of the response plots

To initialize an LTI Viewer, simply type:

```
lti view
```

`lti view` can also be called with additional input arguments that allow you to specify the type of LTI model response displayed when the window is first opened. The generic syntax is:

```
lti view(plottype, sys, extras)
```

where *sys* is the name of an LTI object in the MATLAB workspace and *plottype* is one of the following strings:

```
'step'  
'impulse'  
'initial'  
'lsim'  
  
'bode'  
'nyquist'  
'nichols'  
'sigma'
```

For example, you can initialize an LTI Viewer showing the step response of the LTI model *sys* by:

```
lti view('step', sys)
```

`extras` is a compilation of the additional input arguments supported by the function named in `plottype`. In the example above, `plottype` is 'step' and an additional input argument specifying the final time of the response may be entered, as in:

```
ltiview('step', sys, Tfinal)
```

However, if `plottype` is 'initial', the `extras` arguments must include the initial conditions `x0`. This can be followed by other arguments (See the Reference entry for `initial` for more information on this command):

```
ltiview('initial', sys, x0, Tfinal)
```

In general, the input arguments you supply after `plottype` are identical to those used when calling the `plottype` command at the MATLAB prompt. Therefore, as with any of the LTI model response functions, you can plot the responses of multiple systems and specify distinctive plot styles for each system. See “Time and Frequency Response” on page 1-13. Also see the Reference entry for `ltiview` for more detail and examples on initializing the LTI Viewer for multiple systems.

The LTI Viewer can now also be used to perform linear analysis directly on a Simulink diagram. By selecting the **Linear Analysis** item from the **Tools** menu of a Simulink diagram, you link an LTI Viewer to your Simulink diagram and open a set of Input/Output Point blocks that can be used to indicate the input and output signals for the (linearized) model you will perform linear analysis on.

For more detail on the use of the LTI Viewer and how it can be integrated into a Simulink diagram, see Chapter 4.

System Interconnections

You can derive LTI models for various system interconnections ranging from simple series connections to more complex block diagrams; see page 5-3 for more information. Related commands include:

```
append(sys1, sys2, . . .) % appends systems inputs and outputs
parallel(sys1, sys2)      % general parallel connection
series(sys1, sys2)        % general series connection
feedback(sys1, sys2)      % feedback loop
star(sys1, sys2)          % star product (LFT interconnection)
connect(sys, q)           % state-space model of block diagram
```

Note that simple parallel and series interconnections can be performed by direct addition and multiplication, respectively.

When combining LTI models of different types (for example, state-space `sys1` and transfer function `sys2`), the type of the resultant model is determined by the same precedence rules as for arithmetic operations. See page 2-3 for more information. Specifically, the ranking of the different types of LTI models from highest to lowest precedence is SS, ZPK, and TF.

Control Design Tools

The Control System Toolbox supports three mainstream control design methodologies: gain selection from root locus, pole placement, and linear-quadratic-Gaussian (LQG) regulation. The first two techniques are covered by the `rl locus` and `pl ace` commands. The LQG design tools include commands to compute the LQ-optimal state-feedback gain (`l qr`, `dl qr`, and `l qry`), to design the Kalman filter (`kal man`), and to form the resulting LQG regulator (`l qgreg`). See page 5-9 for more information.

As an example of LQG design, consider the regulation problem illustrated by Figure 1-1. The goal is to regulate the plant output y around zero. The system is driven by the white noise disturbance d , there is some measurement noise n , and the noise intensities are given by

$$E(d^2) = 1, \quad E(n^2) = 0.01$$

The cost function

$$J(u) = \int_0^{\infty} (10y^2 + u^2) dt$$

is used to specify the trade-off between regulation performance and cost of control. Note that an open-loop state-space model is

$$\begin{aligned} \dot{x} &= Ax + Bu + Bd && \text{(state equations)} \\ y_n &= Cx + n && \text{(measurements)} \end{aligned}$$

where (A, B, C) is a state-space realization of $100/(s^2 + s + 100)$.

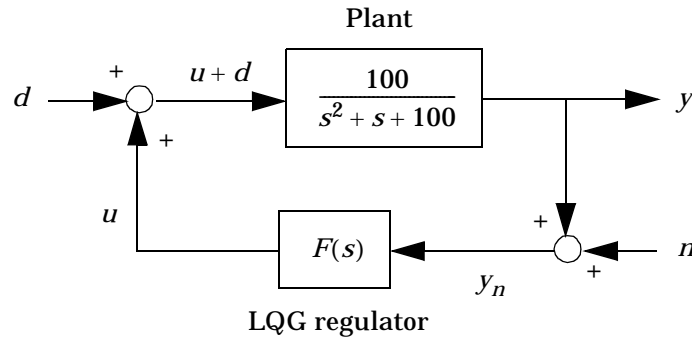


Figure 1-1: Simple Regulation Loop

The following commands design the optimal LQG regulator $F(s)$ for this problem:

```
sys = ss(tf(100, [1 1 100])) % state-space plant model

% Design LQ-optimal gain K
K = lqry(sys, 10, 1) % u = -Kx minimizes J(u)

% Separate control input u and disturbance input d
[A, B, C, D] = ssdata(sys)
P = ss(A, [B B], C, [D D]) % input [u; d], output y

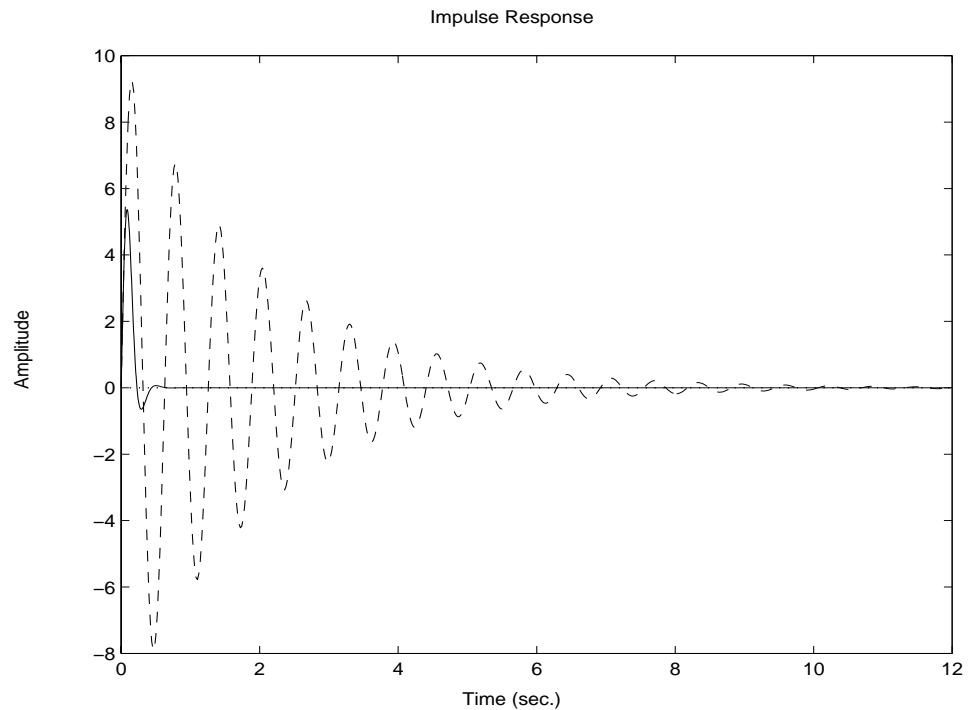
% Design Kalman state estimator KEST
Kest = kalman(P, 1, 0.01)

% Form LQG regulator = LQ gain + Kalman filter
F = lqgreg(Kest, K)
```

The last command returns a state-space model F of the LQG regulator $F(s)$. Note that `lqry`, `kalman`, and `lqgreg` perform discrete-time LQG design when applied to discrete plants.

To validate the design, close the loop with feedback and compare the open- and closed-loop impulse responses with `impz`:

```
% Close loop  
cl sys = feedback(sys, F, +1)    % Note positive feedback  
  
% Open- vs. closed-loop impulse responses  
impz(sys, 'r--', cl sys, 'b-')
```



The Root Locus Design GUI

You can also design a compensator using the Root Locus Design Graphical User Interface (GUI); see Chapter 6 for more information. The Root Locus Design GUI is an interactive graphical user interface that assists you in designing a compensator by providing the following features:

- Adding/removing compensator poles and zeros directly on the root locus plot
- Dragging compensator poles and zeros around in the root locus plane
- Changing the value of the root locus gain
- Examining changes in the closed-loop response whenever the compensator is changed
- Drawing boundaries on the root locus plane for parameters such as minimum damping ratio, etc.
- Zooming in to regions of the root locus

To initialize the Root Locus Design GUI, simply type:

```
rl gui
```

`rl gui` can also be called with additional input arguments that allow you to initialize the design model and compensator used in the Root Locus Design GUI. For example:

```
rl gui (sys)
```

initializes a Root Locus Design GUI with the linear time invariant (LTI) object `sys` as the design model. Adding a second input argument, as in:

```
rl gui (sys, comp)
```

also initializes the LTI object `comp` as the root locus compensator.

When one or two input arguments are provided, the root locus of the closed-loop poles and their locations for the current compensator gain are drawn. The closed-loop model is generated by placing the compensator and design model in the forward loop of a negative unity feedback system. You can provide additional input arguments to `rl gui` in order to override these defaults, as shown below:

```
rl gui (sys, comp, Locati onFl ag, FeedbackSi gn)
```

where `LocationFlag` indicates the location of the compensator, and `FeedbackSign` indicates the sign of the feedback, as follows:

```
LocationFlag = 1;    % Compensator in the forward loop  
LocationFlag = 2;    % Compensator in the feedback loop
```

```
FeedbackSign = -1;   % Negative feedback  
FeedbackSign = 1;    % Positive feedback
```

For more detail on the Root Locus Design GUI, see Chapter 6.

LTI Models

Introduction	2-2
Creating LTI Models	2-6
LTI Properties	2-17
Model Conversion	2-25
Operations on LTI Models	2-27
Input Delays	2-36
Continuous/Discrete Conversions	2-39
Resampling Discrete-Time Models	2-44
Simulink Block for LTI Systems	2-46
References	2-48

Introduction

The Control System Toolbox offers extensive tools to manipulate and analyze linear time-invariant (LTI) systems. It supports both continuous- and discrete-time systems. In addition, systems can be single-input/single-output (SISO) or multiple-input/multiple-output (MIMO). You can specify LTI systems as:

- Transfer functions (TF), for example,

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

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

$$H(z) = \left[\begin{array}{cc} \frac{2(z - 0.5)}{z(z + 0.1)} & \frac{(z^2 + z + 1)}{(z + 0.2)(z + 0.1)} \end{array} \right]$$

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

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

where x is the state vector and u and y are the input and output vectors.

This section introduces key concepts about the MATLAB representation of LTI models, including LTI objects, LTI properties, associated methods, precedence, and the analogy between LTI models and matrices.

LTI Objects

Depending on the choice of model, the system data ranges from a simple numerator/denominator pair for SISO transfer functions (`tf`) to four matrices for state-space models (`ss`), or multiple sets of zeros and poles for MIMO zero-pole-gain (`zpk`) models. For convenience, the Control System Toolbox provides customized data structures for each model called the TF, ZPK, and SS *objects*. These three *LTI objects* encapsulate the model data and enable you to

manipulate LTI systems as single entities rather than collections of data vectors or matrices. 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

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

You can then manipulate this transfer function by referring to the single MATLAB variable P, for example, plot its Bode response by

```
bode(P)
```

The LTI object implementation relies on MATLAB object-oriented programming capabilities. Objects are MATLAB structures with an additional flag indicating their class (TF, ZPK, or SS for LTI objects). Like MATLAB structures, they have pre-defined fields called *object properties*. For LTI objects, these properties include the model data, sample time, and input/output names (see page 2-17 for details). The functions operating on a particular object are called the object *methods*. These may include customized versions of simple operations such as addition or multiplication. For example,

```
Q = 2 + P
```

performs the transfer function addition

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

We refer to the object-specific versions of such standard operations as *overloaded* operations. For more details on objects, methods, and object-oriented programming, refer to Chapter 14 of *Using MATLAB*.

Precedence Rules

Operations like addition and commands like `feedback` operate on several LTI models. If these LTI models have different types (for example, the first operand is TF and the second operand is SS), it is unclear what resulting type to expect (for example, TF or SS). Such type conflicts are resolved by *precedence rules*.

Specifically, the TF, ZPK, and SS models are ranked according to the precedence hierarchy:

$$\text{SS} > \text{ZPK} > \text{TF}$$

Thus ZPK takes precedence over TF and SS takes precedence over both TF and ZPK. In other words, any operation involving two or more LTI models will produce

- An SS object if at least one operand is SS
- A ZPK object if no operand is SS and at least one is ZPK
- A TF object if *all* operands are TF models.

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 a linear input/output map

$$y = Hu$$

This map is characterized by its transfer matrix H , a function of the Laplace or Z-transform variable. The “matrix” H has as many columns as inputs and as many rows as outputs.

If you think of LTI systems as (transfer) matrices, certain basic operations on LTI systems are naturally expressed with a matrix-like syntax. For example,

$$\text{sys} = \text{sys1} + \text{sys2}$$

connects the LTI systems `sys1` and `sys2` in parallel because parallel connection amounts to adding the transfer matrices. Similarly, matrix-like subindexing is used to extract subsystems of a given LTI model `sys`. For instance,

$$\text{sys}(3, 1:2)$$

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

$$y = Hu \quad \Rightarrow \quad y_3 = \begin{bmatrix} H(3, 1) & H(3, 2) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

Command Summary

The next two tables give an overview of the main commands discussed in this chapter.

LTI Models	
dss	Create descriptor state-space model.
filt	Create discrete filter with DSP convention.
get	Query LTI model properties.
set	Set LTI model properties.
ss	Create state-space model.
ssdata, dssdata	Retrieve state-space data.
tf	Create transfer function.
tfdata	Retrieve transfer function data.
zpk	Create zero-pole-gain model.
zpkdata	Retrieve zero-pole-gain data.
Model Conversions	
c2d	Continuous- to discrete-time conversion.
d2c	Discrete- to continuous-time conversion.
d2d	Resampling of discrete-time models.
pade	Pade 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`, and `ss` create transfer function, zero-pole-gain, and state-space models, respectively. These functions take the model data as input and produce TF, ZPK, or SS 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`, and `ss`.

Transfer Function

A 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 variable s . MATLAB represents polynomials by the row vector of their coefficients ordered in *descending* powers of s . For example, `[1 3 5]` represents $s^2 + 3s + 5$. If `num` and `den` are the vector representations of $n(s)$ and $d(s)$, the command

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

creates the SISO transfer function $h(s) = n(s)/d(s)$. The variable `h` is a TF object containing the numerator and denominator data. For example,

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

creates the transfer function $h(s) = s/(s^2 + 2s + 10)$

Transfer function:

```
      s
-----
s^2 + 2 s + 10
```

Note the customized display used for TF objects.

MIMO transfer functions are arrays of elementary SISO transfer functions. As such, they are characterized by the numerators and denominators of their

entries, or equivalently by the array of numerators and the array of denominators. For example, the numerator and denominator arrays

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

specify

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

Using cell arrays of row vectors to represent the polynomial arrays $N(s)$ and $D(s)$, you can specify the MIMO transfer function $H(s)$ by typing:

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

and MATLAB responds with

```
Transfer function from input to output...

      s - 1
#1:  -----
      s + 1

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

For general MIMO transfer functions $H(s)$, the cell array entries $N\{i,j\}$ and $D\{i,j\}$ should be row-vector representations of the numerator and denominator of $H_{ij}(s)$.

Alternatively, you can also specify $H(s)$ by concatenation of its SISO entries:

```
h11 = tf([1 -1], [1 1]) % (s-1)/(s+1)
h21 = tf([1 2], [1 4 5]) % (s+2)/(s^2+4s+5)
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 page 2-30 for more details on concatenation operations for LTI systems.

Finally, you can use `tf` to tag 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
```

returns an empty transfer function.

Zero-Pole-Gain

SISO zero-pole-gain models are of the form

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

where k is a scalar (the *gain*), and z_1, \dots, z_m and p_1, \dots, p_n are the 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.

The syntax to create such models 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 , k data. For example, typing:

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

produces

Zero/pole/gain:

$$\frac{-2s}{(s-2)(s^2 - 2s + 2)}$$

The MIMO case is handled as for transfer functions. The syntax to create a p -by- m MIMO zero-pole-gain model is

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

where

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

For example, typing

```
Z = {[ ], -5; [1-i 1+i] [ ]} % Note: use [ ] when no zero
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

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

As for transfer functions, you can also specify this MIMO model by concatenation of its SISO entries (see page 2-30).

State Space

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

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

where x is the state vector and u, 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:

$$\text{sys} = \text{ss}(A, B, C, D)$$

This produces an SS object `sys` storing the state-space matrices (A, B, C, D) . These matrices should have compatible row/column dimensions as illustrated below:

$$\begin{array}{cc} N_x \{ & A \quad B \\ N_y \{ & \underbrace{C} \quad \underbrace{D} \\ & N_x \quad N_u \end{array} \left\{ \begin{array}{l} N_x: \text{ no. of states} \\ N_u: \text{ no. of inputs} \\ N_y: \text{ no. of outputs} \end{array} \right.$$

For models with a zero D matrix, use `D = 0` (zero) as a shorthand for a zero matrix of 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

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

where

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

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      1.0000
x2    -5.0000    -2.0000
```

```
b =
      u1
x1      0
x2      3.0000
```

```
c =
      x1      x2
y1      0      1.0000
```

```
d =
      u1
y1      0
```

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
```

Discrete-Time Models

To create discrete-time LTI models, simply append the sample time T_s (in seconds) to the model data in the calling syntax of `tf`, `zpk`, and `ss`:

```
sys = tf(num, den, Ts)
sys = zpk(z, p, k, Ts)
sys = ss(a, b, c, d, 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 second, and

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


specifies the discrete-time state-space model

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

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 n th sample.

By convention, the sample time of continuous systems is $T_s = 0$. Use the value $T_s = -1$ to leave the sample time of a discrete system unspecified, for example,

$$h = \text{tf}([1 \ -0.2], [1 \ 0.3], -1)$$

produces

Transfer function:

$$\begin{array}{r}z - 0.2 \\ \hline z + 0.3\end{array}$$

Sampling time: unspecified

Do not simply omit T_s in this case; otherwise the resulting system will be marked as continuous.

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 page 2-6 and see the `tf` entry in the Reference chapter). Indeed,

$$h = \text{tf}([1 \ 0.5], [1 \ 2 \ 3])$$

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 T_s . 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} . As for `tf`, the input arguments `num` and `den` are then cell arrays of row vectors (see page 2-6 for details). Note that each row vector should comply with the “ascending powers of z^{-1} ” convention.

Data Retrieval

The functions `tf`, `zpk`, and `ss` pack the model data and sample time in a single LTI object. Conversely, how can you extract this data from an existing LTI object? The following commands perform a convenient one-shot data retrieval:

```
[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)
```

Note that the output arguments `num`, `den` of `tfdata` and `z`, `p` of `zpkdata` are always cell arrays, even in the SISO case. These cell arrays have as many rows as outputs and as many columns as inputs, and their (i, j) entry specifies the transfer function from the j th input to the i th output. For example,

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

creates the one-output/two-input transfer function

$$H(s) = \begin{bmatrix} \frac{s-1}{s^2+2s+10} & \frac{1}{s} \end{bmatrix}$$

Typing

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

produces

```
ans =
```

```
0      1     -1
```

and typing

```
den{1, 1}
```

produces

```
ans =
```

```
1      2     10
```

which retrieves the numerator and denominator of the first input channel.

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] = zpndata(sys, 'v')
```

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

Finally, note that these functions are applicable to any LTI model regardless of its type. For example, `tfdata` can be applied to state-space models, in which case the model is first converted to a transfer function before extracting the numerator and denominator data (see “Automatic Conversion” on page 2-26 for details).

LTI Properties

The previous section shows how to create LTI objects that encapsulate the model data and sample time. Optionally, LTI objects can also store additional information such as the input names or notes on the model history. 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, and SS objects. Type `help lti props` for online help on available LTI properties.

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

Generic Properties

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

Table 2-1: Generic LTI Properties

Property Name	Description	Property Value
InputName	Input names	Cell vector of strings
Notes	Notes on the model history	Text
OutputName	Output names	Cell vector of strings
Ts	Sample time	Scalar
Td	Input delay(s)	Vector
Userdata	Additional data	Arbitrary

The sample time property `Ts` keeps track of the sample time (in seconds) of discrete systems. By convention, the values 0 (zero) and -1 are used for continuous systems and discrete systems with unspecified sample time, respectively.

The input delay property `Td` is available only for continuous-time systems. Its value is the vector of time delays (in seconds) for each input channel. The default value is zero (no delay). See page 2-36 for details on delay systems.

The `InputName` and `OutputName` properties enable you to give names to the system inputs and outputs. Their value 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.

Finally, `Notes` and `Userdata` are available to store additional information on the model. The `Notes` property is dedicated to text input (model history) while the `Userdata` property can accommodate arbitrary user-supplied data. They are both empty by default.

Model-Specific Properties

The remaining LTI properties are specific to one of the three model types (TF, ZPK, or SS). They are summarized in the three tables below.

Table 2-2: TF-Specific Properties

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

Table 2-3: ZPK-Specific Properties

Property Name	Description	Property Value
k	Gains	Two-dimensional matrix
p	Poles	Cell array of column vectors

Table 2-3: ZPK-Specific Properties (Continued)

Property Name	Description	Property Value
Variabl e	Transfer function variable	String ' s' , ' p' , ' z' , ' q' , or ' z ⁻¹ '
z	Zeros	Cell array of column vectors

Table 2-4: SS-Specific Properties

Property Name	Description	Property Value
a	State matrix A	Two-dimensional matrix
b	Input-to-state matrix B	Two-dimensional matrix
c	State-to-output matrix C	Two-dimensional matrix
d	Feedthrough matrix D	Two-dimensional matrix
e	Descriptor E matrix	Two-dimensional matrix
StateName	State names	Cell vector of strings

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

The Variabl e property of TF and ZPK objects 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` entry in Reference pages for details).

Note that `tf` produces the same result as `filt` when the Variabl e 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`, or `ss`.
- You can set or modify the properties of an existing LTI model with `set`.
- You can also set property values by structure-like assignments.

This section discusses the first two options. See “Direct Property Referencing” on page 2-23 for details on the third option.

The function `set` 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. It can be the property name itself, or any case-insensitive abbreviation with enough characters to uniquely identify the property. For example, 'user' refers to the `Userdata` property.
- *PropertyValue* is the value to assign to the property (see `set` entry in Reference pages for details on admissible property values).

This syntax is also supported by the functions `tf`, `zpk`, and `ss`.

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

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

or equivalently by

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

Note how the display reflects the I/O names and variable selection. Typing:

```
h
```


produces

Transfer function from input "thrust" to output "velocity":

$$\frac{1}{p + 10}$$

In the MIMO case, use cell vectors of strings to attach names to each input or output. 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'})
H
```

MATLAB responds with:

Transfer function from input "temperature" to output:

$$\frac{3}{s + 10}$$

Transfer function from input "pressure" to output:

$$\frac{s + 2}{s}$$

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

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

The syntax `set(sys)` displays all valid property values for the LTI model `sys`. For the transfer function `H` created above, this produces:

```
set(H)
    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
    Td: vector of length Nu (for continuous systems only)
    InputName: Nu-by-1 cell array of strings
    OutputName: Ny-by-1 cell array of strings
    Notes: array or cell array of strings
    UserData: arbitrary
```

For more details on valid property values, refer to the `set` entry in the Reference pages.

Caution: Resetting the sample time from zero (continuous system) to a nonzero value (discrete system) does *not* discretize the system. The command

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

is a literal operation that affects *only* the sample time property. 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.

Accessing Property Values

Property values are accessed 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 properties and their values at once, use the syntax `get(sys)`:

```
get(h)

    num = {[0 0 100]}
    den = {[1 5 100]}
    Variable = 's'
    Ts = 0
    Td = 0
    InputName = {'voltage'}
    OutputName = {'current'}
    Notes = {'A simple circuit'}
    UserData = []
```

Finally, you can also access property values by direct structure-like referencing; see “Direct Property Referencing” below for details.

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, or SS objects (see page 2-2). Their field names are simply 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

```
PropertyVal ue = get(sys, 'PropertyName')
set(sys, 'PropertyName', PropertyVal ue)
```

Here are two examples. Type:

```
sys = ss(1, 2, 3, 4)
sys.a = -1           % resets A matrix to -1

h = tf(1, [1 0], 'inputname', 'u', 'variable', 'p');    % h(p)=1/p
h.Variable
```

and MATLAB responds

```
ans =
p
```

Unlike standard MATLAB structures, you need not type the entire field name nor be concerned with upper-case letters. Thus typing

```
h.inp
```

produces

```
ans =

    'u'
```

which is the same answer as if you typed h.InputName, that is, the string cell {'u'}. Any valid syntax for structures extends to LTI objects, for example,

```
h.num{1} = [1 2]      % resets numerator to p+2
h.num{1}(1) = 3       % resets numerator to 3p+2
```

Model Conversion

There are three types of LTI models: transfer function, zero-pole-gain, and state-space. This section shows how to convert models from one type to the other.

Explicit Conversion

Model conversions are performed by `tf`, `ss`, and `zpk`. Given any LTI model `sys`, the syntax is simply

```
sys = tf(sys)           % Conversion to TF
sys = zpk(sys)          % Conversion to ZPK
sys = ss(sys)           % Conversion to SS
```

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 general state-space model (A, B, C, D) is

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

in continuous time and

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

in discrete time.

Automatic Conversion

Most functions and operations perform automatic conversion to the appropriate model type. For example, in

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

`tfdata` first converts the state-space model `sys` to a transfer function, then returns the numerator and denominator of this transfer function.

Note that conversions to state space (also called *realizations*) 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 functions like `initial` or `kalman`.

Caution

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

- The three LTI models are not equally suited for numerical computations. In particular, the accuracy of computations with high-order transfer functions is often poor. You should work with (balanced) state-space models as much as possible and use transfer functions only for display or interpretation.
- Conversions to transfer function format 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 (type `help roots` for an illustration).
- Conversions to state space are not uniquely defined in the SISO case, nor guaranteed to produce a minimal realization in the MIMO case. So given a 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. Hence, converting models back and forth should be avoided whenever possible.

Operations on LTI Models

You can perform basic operations such as addition, multiplication, or concatenation on LTI systems. Such operations are “overloaded,” which means that they retain the same syntax but assume a different and adequate functionality when applied to LTI models. Overloaded operations and their interpretation in the LTI context are discussed next.

Precedence and Property Inheritance

Operations may combine models of different types. The resulting type is then determined by the precedence rules discussed on page 2-3. To force the result to a given type, for example, the TF object, you can either convert all operands to TF *before* performing the operation, or convert the result to TF *after* performing the operation. Suppose, for instance, that `sys1` is a transfer function and `sys2` is a state-space model. To compute the transfer function of their sum, you can use *a priori* conversion of the second operand

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

or *a posteriori* conversion of the result

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

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

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

- You can only combine systems with identical sample times. Systems with an unspecified sample time (`sys.Ts = -1`) are neutral provided that all operands are discrete-time systems.
- Most operations ignore the `Notes` and `Userdata` properties.
- The input and output names are passed on to the result when appropriate. Conflicting sets of names are deleted.

- For TF and ZPK models, the result inherits its Variable property from the operands. Conflicts are resolved by the following rules: 'p' supersedes 's', and the discrete-time variables are ranked in order 'z⁻¹', 'q', and 'z', with 'z' having the lowest precedence.

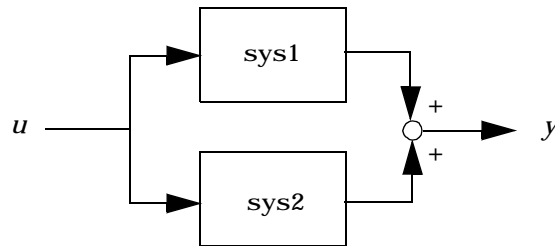
Arithmetic Operations

Addition and Subtraction

System addition connects systems in parallel. Specifically,

$$\text{sys} = \text{sys1} + \text{sys2}$$

returns an LTI model for the parallel interconnection shown below:



Why use addition for parallel connection? This is best understood in terms of transfer matrix. If H_1 and H_2 are the transfer matrices of sys1 and sys2, then u and y are related by:

$$y = (H_1 + H_2) u$$

Thus parallel connection amounts to adding the transfer matrices.

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 , a state-space realization of sys1+sys2 is

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

Scalar addition is also supported and behaves as follows: if sys1 is MIMO and sys2 is SISO, $\text{sys1} + \text{sys2}$ produces a system with the same dimensions as sys1 and with generic entry $\text{sys1}(i, j) + \text{sys2}$.

Finally, the subtraction

$$\text{sys} = \text{sys1} - \text{sys2}$$

produces the system

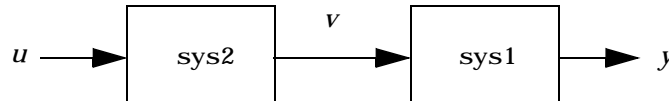
$$y = (H_1 - H_2) u$$

Multiplication

System multiplication connects systems in series. Specifically,

$$\text{sys} = \text{sys1} * \text{sys2}$$

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 , a state-space realization of $\text{sys1} * \text{sys2}$ is

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

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

Inversion and Related Operations

System inversion amounts to inverting the input/output relationship:

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

This operation is defined only for square systems (that is, with as many inputs as outputs) and performed by `inv`:

$$i\text{ sys} = \text{inv}(\text{sys})$$

The inverse model `i sys` 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 state-space models with data A, B, C, D , the inverse is defined only when D is a square invertible matrix, in which case a state-space realization of the inverse is

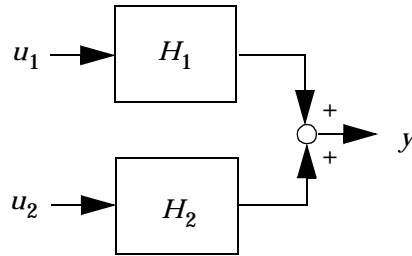
$$A - BD^{-1}C, \quad BD^{-1}, \quad -D^{-1}C, \quad D^{-1}$$

Concatenation

System concatenation is done in a matrix-like fashion by

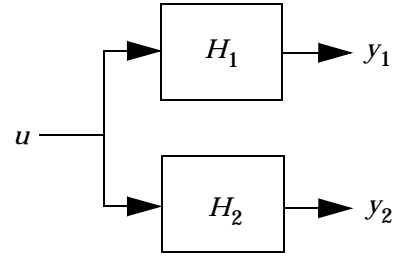
```
sys = [sys1 , sys2] % horizontal concatenation
sys = [sys1 ; sys2] % vertical concatenation
```

In I/O terms, these two operations have the following block-diagram interpretation (with H_1 and H_2 denoting the transfer matrices of sys1 and sys2):



$$y = \begin{bmatrix} H_1 & H_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

Horizontal concatenation



$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} H_1 \\ H_2 \end{bmatrix} u$$

Vertical concatenation

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

$$H = [\text{tf}(1, [1 \ 0]) \quad 1 ; \ 0 \quad \text{tf}([1 \ -1], [1 \ 1])]$$

specifies

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

Transposition and Pertransposition

You can transpose LTI models by

```
sys.'
```

This is a literal operation with the following effect:

- For TF models, the cell arrays num and den are transposed.
- For ZPK models, the cell arrays z and p and the matrix k are transposed.
- For SS models with matrices A, B, C, D , transposition produces the state-space model A^T, C^T, B^T, D^T

Pertransposition is performed by

```
sys'
```

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

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

where M^H denotes the Hermitian transpose of a matrix M . The discrete-time counterpart is

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

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, we use matrix-like subindexing to extract this subsystem:

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

The resulting subsystem `SubSys` is a model of the same type as `sys`. When `sys` is a state-space model with matrices `a`, `b`, `c`, `d`, the subsystem `SubSys` has matrices `a`, `b(:, 1)`, `c(3, :)`, `d(3, 1)`.

Note: In the expression `sys(3, 1)`, the first index selects the output channel while the second index selects the input channel.

Similarly,

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

redefines the I/O relation between the first input and third output. The new relation is specified by the LTI model `NewSubSys`. Note that `sys` retains its original type regardless of whether `NewSubSys` is a TF, ZPK, or SS model. Also, reassigning parts of a MIMO state-space model generally increases its order.

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.

As an illustration, consider the two-input/two-output transfer function

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

You can extract the transfer function $T_{11}(s)$ from first input to first output by typing `T(1, 1)`, to which MATLAB responds:

Transfer function:

$$\frac{1}{s + 0.1}$$

You can also reset $T_{11}(s)$ to $1/(s+0.5)$ by typing

```
T(1, 1) = tf(1, [1 0.5])
```

or modify the second input channel by typing

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

and MATLAB responds

Transfer function from input 1 to output...

$$\begin{array}{l} \#1: \frac{1}{s + 0.1} \\ \\ \#2: \frac{s - 1}{s^2 + 2s + 2} \end{array}$$

Transfer function from input 2 to output...

$$\begin{array}{l} \#1: 1 \\ \\ \#2: \frac{0.4}{s} \end{array}$$

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 by

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

or

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

where `h` is any LTI model for the new input channel. There is an important difference between these two options: while concatenation obeys the precedence rules (see page 2-3), subassignment does not alter the model type. So, if `sys` and `h` are TF and SS objects, respectively, the first statement will produce a state-space model and the second statement a transfer function.

For state-space models, both concatenation and subassignment increase the model order because they assume that `sys` and `h` have independent states. If you intend to keep the same state matrix and merely update the input-to-state or state-to-output relations, use `set` instead and directly update the corresponding state-space data. 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 updating the B and D matrices. You should *simultaneously* update 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 where the B and D matrices have inconsistent column dimensions.

Input Delays

Input delays are supported for *continuous-time* systems within the limits of tractability and cross-model compatibility. Discrete delay systems are not supported *per se* since they admit a standard LTI representation as long as the input delays are integer multiples of the sampling period. You can specify such systems with d2d (see d2d entry in Reference pages for details).

In state space, continuous systems with uniform input delay τ (in seconds) are described by

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t - \tau) \\ y(t) &= Cx(t) + Du(t - \tau)\end{aligned}$$

The corresponding transfer function is

$$y = H(s) e^{-s\tau} u$$

where $H(s) = D + C(sI - A)^{-1}B$ and $\exp(-s\tau)$ is the Laplace transform of the input delay. Multi-input systems can also have a different delay for each input channel:

$$\begin{aligned}\dot{x}(t) &= Ax(t) + B_1 u_1(t - \tau_1) + \dots + B_m u_m(t - \tau_m) \\ y(t) &= Cx(t) + D_1 u_1(t - \tau_1) + \dots + D_m u_m(t - \tau_m)\end{aligned}$$

or in transfer function form:

$$y = H_1(s) e^{-s\tau_1} u_1 + \dots + H_m(s) e^{-s\tau_m} u_m$$

where $H_j(s) = D_j + C(sI - A)^{-1}B_j$.

Specifying Input Delays

To specify delay systems as LTI objects, first create a regular LTI model with state-space matrices

$$A, \quad B = \begin{bmatrix} B_1 & \dots & B_m \end{bmatrix}, \quad C, \quad D = \begin{bmatrix} D_1 & \dots & D_m \end{bmatrix}$$

or transfer function

$$H(s) = \begin{bmatrix} H_1(s) & \dots & H_m(s) \end{bmatrix}$$

Then set the input delay property Td of this model to the vector $[\tau_1 \dots \tau_m]$. For example, typing:

```
H = tf({1 , [1 2]}, {[1 0] [1 2 5]}, 'td', [0 0.2])
```

produces

Transfer function from input 1 to output:

$$\frac{1}{s}$$

Transfer function from input 2 to output:

$$\frac{s + 2}{s^2 + 2s + 5}$$

Input delays (listed by channel): 0 0.2

Or type

```
sys = ss(-1, [1 2], [2; 1], 0)    % two inputs, two outputs
set(sys, 'td', [0.1 0.05])
```

Setting the Td property to a scalar τ specifies a uniform time delay for all input channels. Thus

```
sys = ss(-1, [1 2], [2; 1], 0)
set(sys, 'td', 0.05)
```

is equivalent to

```
sys = ss(-1, [1 2], [2; 1], 0)
set(sys, 'td', [0.05 0.05])
```

Supported Functionalities

In general, interconnections of delay systems mix rational transfer functions and delay operators $\exp(-s\tau_i)$ in an intractable way for the LTI object. As a

result, only a limited number of operations and functions can handle LTI models with input delays. These include:

- Horizontal concatenation
- Addition (parallel connection) of systems with identical input delays
- Multiplication $\text{sys1} * \text{sys2}$ when sys1 is delay free
- Pade approximation (`pade`)
- Continuous-to-discrete conversion (`c2d`)
- All time and frequency response functions

Pade Approximation of Input Delays

The function `pade` computes rational approximations of time delays. The syntax is

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

where `sys` is a continuous-time LTI model with input delays, and the integer `n` specifies the Pade approximation order. The resulting LTI model `rsys` is delay free.

For multi-input systems with nonuniform input delays, for example,

```
sys = ss(-1, [1 2], 1, [2 0], 'td', [0.1 0.3])
```

you can specify a different approximation order for each input delay by

```
rsys = pade(sys, [n1 n2])
```

In this case, `pade` performs a Pade approximation of order `n1` for the first delay (0.1 second), and a Pade approximation of order `n2` for the second delay (0.3 second). The resulting state-space model `rsys` has order $n1+n2+1$.

Continuous/Discrete Conversions

The function `c2d` discretizes continuous-time systems. Conversely, `d2c` converts discrete-time systems 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)
```

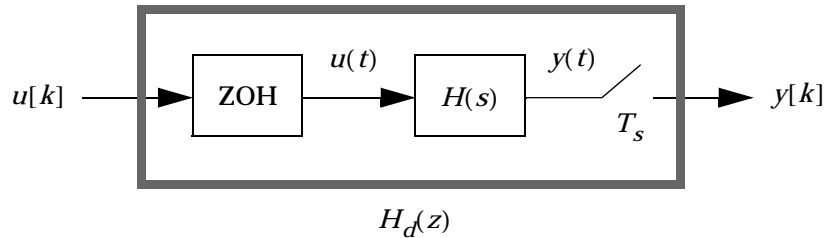
performs ZOH conversions by default. To use alternative conversion schemes, specify the desired method as an extra string input:

```
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

The zero-order-hold discretization $H_d(z)$ of a continuous system $H(s)$ is constructed as follows:



The ZOH device generates a continuous input signal $u(t)$ by holding each sample value $u[k]$ over one sample period:

$$u(t) = u[k], \quad kT_s \leq t \leq (k+1)T_s$$

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 with ZOH cannot handle systems with poles at $z = 0$.
- 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 with real negative poles. Type:

```
hd = zpke([], -0.5, 1, 0.1)
```

and MATLAB responds with

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

```
Sampling time: 0.1
```

Type

```
hc = d2c(hd) % Note: result hc is a second-order system
```

and MATLAB responds with

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

Type

```
c2d(hc, 0.1) % C2D conversion should give back hd
```

and MATLAB responds with

```
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:

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

This scheme is generally more accurate for systems driven by smooth inputs. For causality reasons, this option is available only for c2d conversions.

Note: This FOH scheme 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

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

Similarly, the d2c conversion relies on the inverse correspondence

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

Tustin with Frequency Prewarping

This variation of the Tustin approximation uses the correspondence

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

to ensure matching continuous and discrete 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 Delay Systems

The function `c2d` can also discretize continuous-time systems with input delays [1]. Only the zero- and first-order hold methods are supported for delay systems. For example, you can discretize the delay system by:

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

Type

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

and MATLAB responds with

Transfer function:

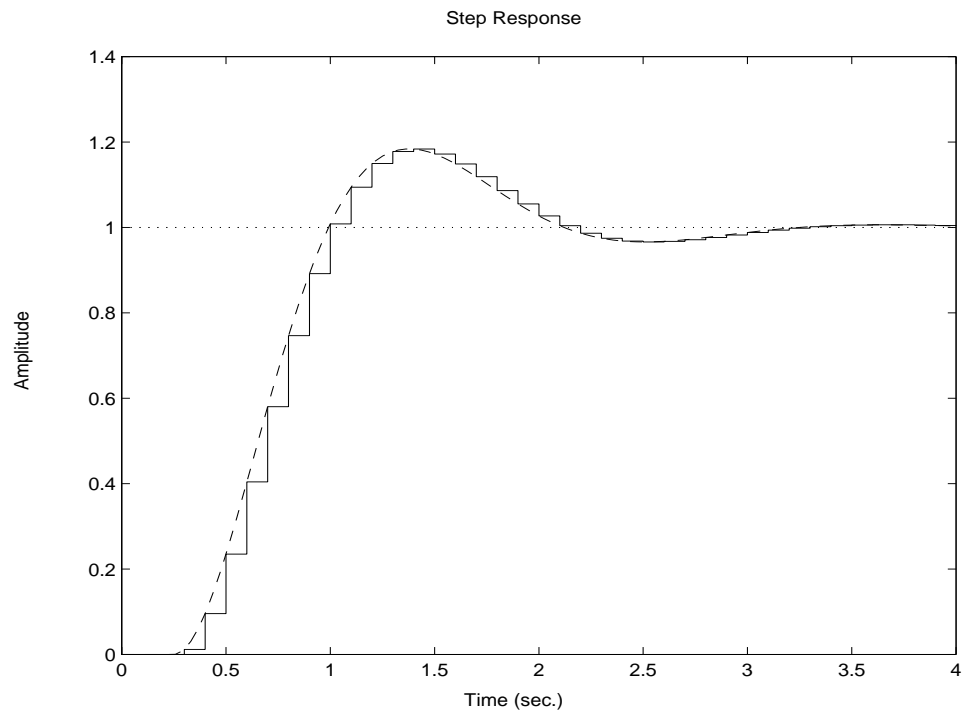
$$0.01187 z^2 + 0.06408 z + 0.009721$$

$$z^5 - 1.655 z^4 + 0.7408 z^3$$

Sampling time: 0.1

Note that the resulting discrete system `hd` is delay free. The step responses of the continuous and discretized systems are compared in the figure below. This plot was produced by the command

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



Resampling Discrete-Time Models

You can resample a discrete-time LTI model `sys1` by

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

The new sampling period `Ts` need not 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)
```

produces

Transfer function:

$z + 1.754$

$z - 0.41$

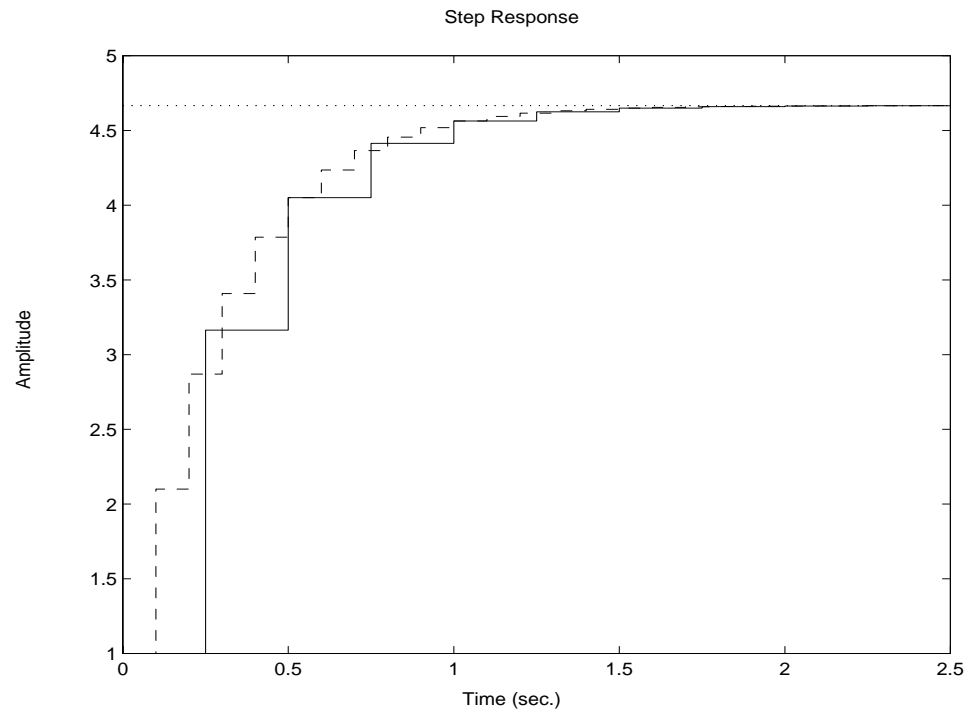
Sampling time: 0.25

Typing

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

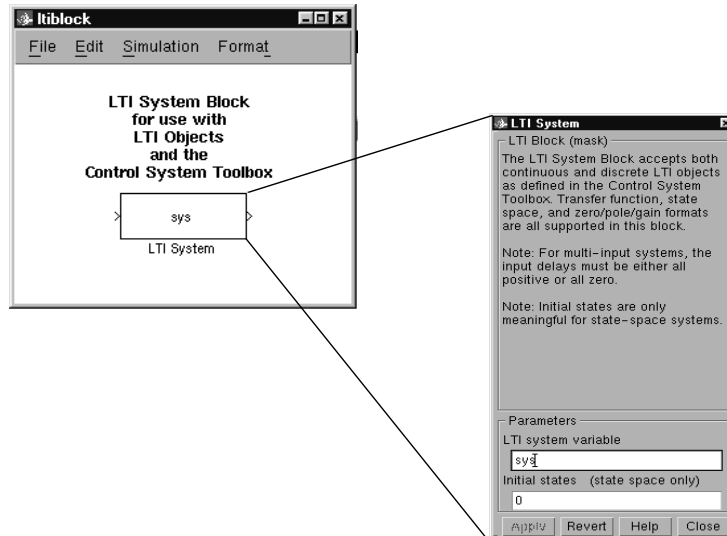
resamples at two-and-a-half times the initial sampling period of 0.1 second.

The step responses of the two discrete systems are compared on the figure below (h1 is the dashed line).



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:

`l t i b l o c k`

at the MATLAB prompt or by opening **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 notion of “initial state” is not well-defined for

transfer functions or zero-pole-gain models (it depends on the choice of state coordinates used by the realization algorithm). Accordingly, the simulation will not run when nonzero initial states are specified for transfer function or zero-pole-gain LTI objects.

Note: For MIMO systems, the input delays stored in the LTI object must be either all positive or all zero. Otherwise, the simulation will not run.

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.

Model Analysis Tools

General Characteristics	3-2
Model Dynamics	3-4
State-Space Realizations	3-6
Time and Frequency Response	3-7
Time Responses	3-7
Frequency Response	3-9
Plotting and Comparing Multiple Systems	3-11
Customized Plots	3-13
Model Order Reduction	3-16

General 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 models of any type.

General Characteristics	
class	Model type ('tf', 'zpk', or 'ss').
isa	True if LTI model is of specified type.
isct	True for continuous-time models.
isdt	True for discrete-time models.
isempty	True for empty LTI models.
isproper	True for proper LTI models.
issiso	True for SISO models.
size	Output/input/state dimensions.

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:

$$\frac{1}{s + 0.1}$$

Transfer function from input 2 to output:

$$\frac{s - 1}{s^2 + 2s + 10}$$

```
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. The next 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.
tzero	System transmission zeros.

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:

$$4 s^3 + 8.4 s^2 + 30.8 s + 60$$

$$s^4 + 4.12 s^3 + 17.4 s^2 + 30.8 s + 60$$

pol e(h)

```
ans =
-1.7971e+00 + 2.2137e+00i
-1.7971e+00 - 2.2137e+00i
-2.6291e-01 + 2.7039e+00i
-2.6291e-01 - 2.7039e+00i
```

tzero(h)

```
ans =
-5.0000e-02 + 2.7382e+00i
-5.0000e-02 - 2.7382e+00i
-2.0000e+00
```

dcgai n(h)

```
ans =
1
```

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

```
ninf =
1.3447e+00 % peak gain
```

```
fpeak =
1.8921e+00 % frequency where gain peaks
```

State-Space Realizations

The following functions are useful to analyze state-space models, perform state coordinate transformations, and derive canonical state-space realizations.

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 simple *diagonal* similarity transformations

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

to balance the state-space data (A, B, C, D) , that is, reduce 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 `ss` produces balanced state-space realizations of transfer functions and zero-pole-gain models.

By contrast, the canonical realizations produced by `canon`, `ctrbf`, or `obsvf` are often badly scaled, very 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.

Time and Frequency Response

The Control System Toolbox contains a set of commands that provide the basic time- and frequency-domain analysis tools required for control system engineering. These commands apply to any kind of LTI model (TF, ZPK, or SS, continuous or discrete, SISO or MIMO). In addition, the LTI Viewer provides an integrated graphical user interface (GUI) to analyze and compare LTI models (see Chapter 4 for details).

Time Responses

Time responses investigate the time-domain transient behavior of LTI systems for particular classes of inputs and disturbances. You can determine such system characteristics as rise time, settling time, overshoot, and steady-state error from the time response. The Control System Toolbox provides functions for step response, impulse response, initial condition response, and general linear simulations. Note that you can simulate the response to white noise inputs using `l sim` and the function `rand` (see *Using MATLAB* to generate random input vectors).

Time Response	
<code>i mpul se</code>	Impulse response.
<code>i ni ti al</code>	Initial condition response.
<code>gensig</code>	Input signal generator.
<code>l sim</code>	Simulation of response to arbitrary inputs.
<code>step</code>	Step response.

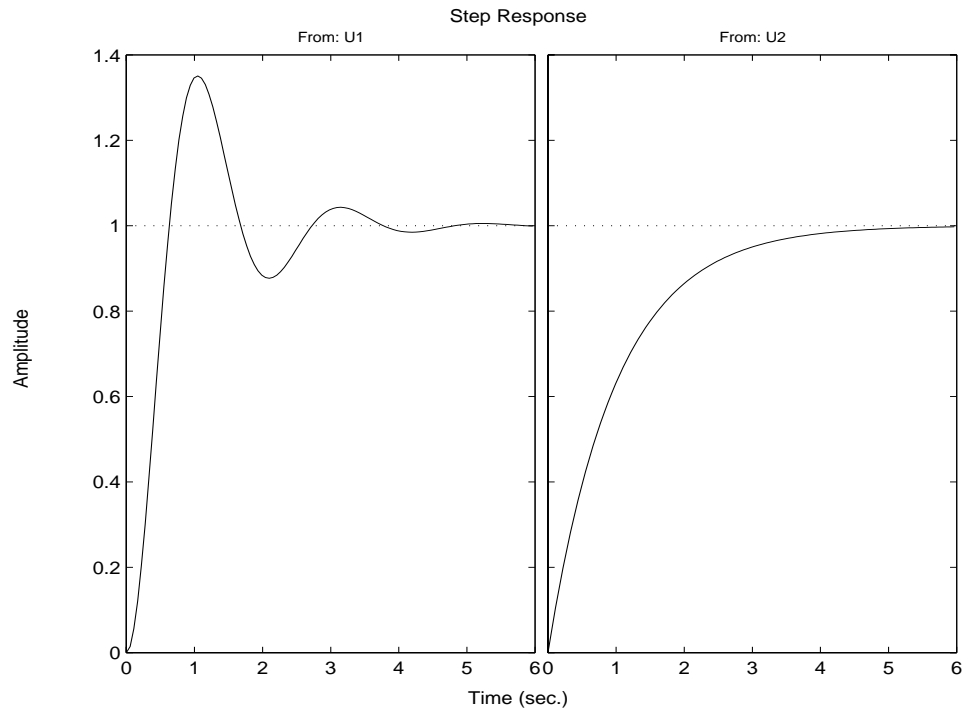
The functions `step`, `i mpul se`, and `i ni ti al` automatically generate an appropriate simulation horizon for the time response plots. Their syntax is

```
step(sys)
i mpul se(sys)
i ni ti al (sys, x0)    % x0 = i n i t i a l   s t a t e   v e c t o r
```

where `sys` is any continuous or discrete LTI model. For MIMO systems, these commands produce an array of plots with one plot per I/O channel (or per output for `initial` and `lsim`). For example,

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

produces the following plot.



The simulation horizon is automatically determined based on the model dynamics. You can override this automatic mode by specifying a final time:

```
step(sys, 10)           % simulates from 0 to 10 seconds
```

or a vector of evenly spaced time samples:

```
t = 0:0.01:10           % time samples spaced every 0.01 second
step(sys, t)
```

Note: When specifying a time vector `t = 0:dt:tf`, remember the following constraints on the spacing `dt` between time samples:

- For discrete systems, `dt` should match the system sample time.
- Continuous systems are first discretized using zero-order hold and `dt` as sampling period, and `step` simulates the resulting discrete system. Hence, you should pick `dt` small enough to capture the main features of the continuous transient response.

The syntax `step(sys)` automatically takes care of these issues.

Finally, the function `lsim` simulates the response to more general classes of inputs. For example,

```
t = 0:0.01:10
u = sin(t)
lsim(sys, u, t)
```

simulates the response of the LTI system `sys` to a sine wave for a duration of 10 seconds.

Frequency Response

Frequency response investigates the frequency-domain behavior of LTI systems. System characteristics such as bandwidth, resonance, DC gain, gain and phase margins, and closed-loop stability can be determined from the frequency response. The Control System Toolbox provides functions for Bode, Nichols, Nyquist, and singular value responses. In addition, the function `margin` determines the gain and phase margins for a given SISO open-loop model.

Frequency Response	
bode	Bode plot.
eval fr	Response at single complex frequency.
freqresp	Response over frequency grid.
margi n	Gain and phase margins.
ngri d	Grid lines for Nichols plot.
ni chol s	Nichols plot.
nyqui st	Nyquist plot.
si gma	Singular value plot.

As for time response functions, the commands

```
bode(sys)
ni chol s(sys)
nyqui st (sys)
si gma (sys)
```

handle both continuous and discrete models and produce a frequency response plot or array of plots in the MIMO case. The frequency grid used to evaluate the response is automatically selected based on the system poles and zeros.

To set the frequency range explicitly to some interval [wmi n, wmax], use the syntax

```
bode(sys, {wmi n , wmax})    % Note the curly braces
```

For example,

```
bode(sys, {0.1 , 100})
```

draws the Bode plot between 0.1 and 100 radians/second. You can also specify a particular vector of frequency points as in

```
w = logspace(-1, 2, 100)
bode(sys, w)
```

The `logspace` command generates a vector `w` of logarithmically spaced frequencies starting at $10^{-1} = 0.1$ rad/s and ending at $10^2 = 100$ rad/s. See the reference page for `linspace` for linearly spaced frequency vectors.

Note: In discrete time, the frequency response is evaluated on the unit circle and the notion of “frequency” should be understood as follows. The upper half of the unit circle is parametrized by

$$z = e^{j\omega T_s}, \quad 0 \leq \omega \leq \omega_N = \frac{\pi}{T_s}$$

where T_s is the system sample time and ω_N is called the *Nyquist frequency*. The variable ω plays the role of continuous-time frequency. We use this “equivalent frequency” as an x -axis variable in all discrete-time frequency response plots. In addition, the frequency response is plotted only up to the Nyquist frequency ω_N because it is periodic with period $2\omega_N$ (a phenomenon known as *aliasing*).

Plotting and Comparing Multiple Systems

To draw the response of several LTI systems on a single plot, invoke the corresponding function with the list `sys1,..., sysN` of systems as input:

```
step(sys1, sys2, ..., sysN)
impz(sys1, sys2, ..., sysN)
...
bode(sys1, sys2, ..., sysN)
nichols(sys1, sys2, ..., sysN)
...
```

All systems must have the same number of inputs and outputs. To differentiate the plots easily, you can also specify a distinctive color/linestyle/marker for each system just as you would with the `plot` command. For example,

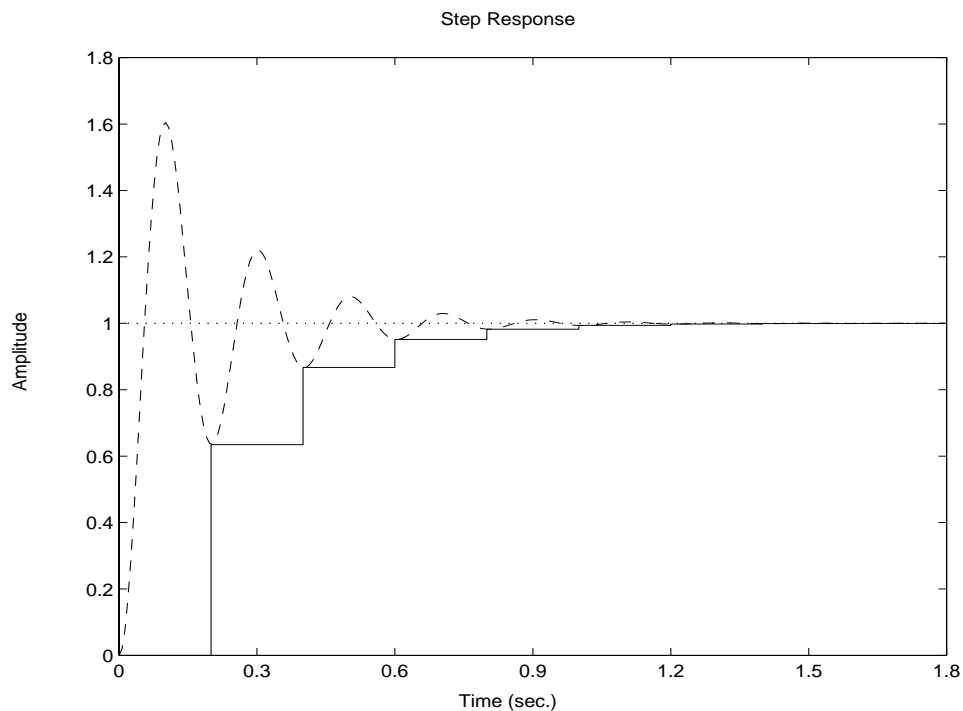
```
bode(sys1, 'r', sys2, 'y--', sys3, 'gx')
```

plots `sys1` with solid red lines, `sys2` with yellow dashed lines, and `sys3` with green `x` markers. Note that multisystem plots can mix continuous and discrete systems.

The following example compares a continuous system with its zero-order-hold discretization.

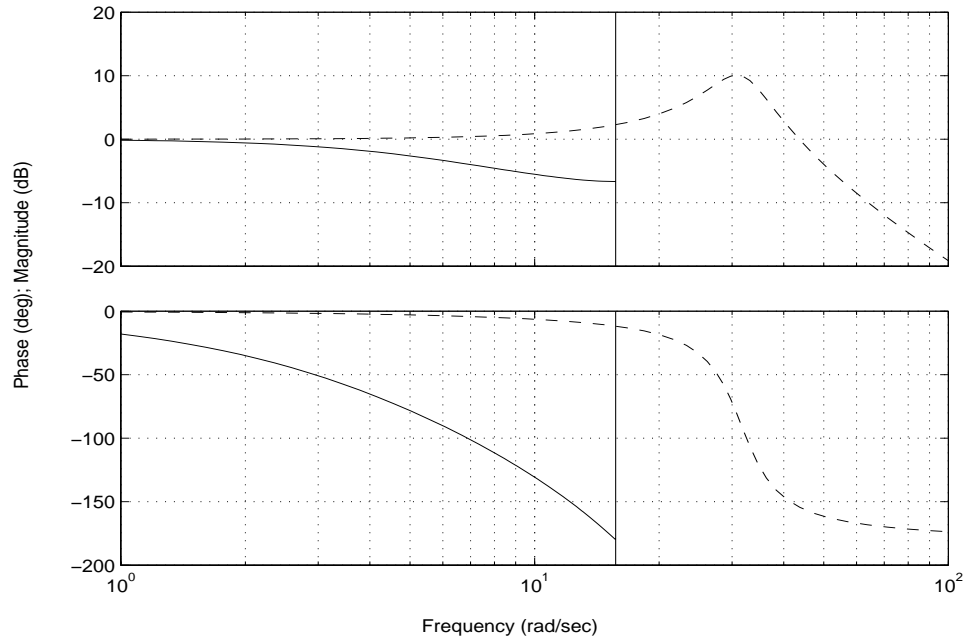
```
sysc = tf(1000, [1 10 1000])
sysd = c2d(sysc, 0.2)           % ZOH sampled at 0.2 second

step(sysc, '--', sysd, '-')    % compare step responses
```




```
bode(syssc, '-', sysd, '-') % compare Bode responses
```

Bode Diagrams



A comparison of the continuous and discretized responses reveals a drastic undersampling of the continuous system. Specifically, there are hidden oscillations in the discretized time response and aliasing obliterates the continuous-time resonance near 300 rad/sec.

Customized Plots

You can also store the time or frequency response data in MATLAB arrays by invoking `step`, `bode`, ... with output arguments, such as,

```
[y, t] = step(sys)
[mag, phase, w] = bode(sys)
[re, im, w] = nyquist(sys)
```

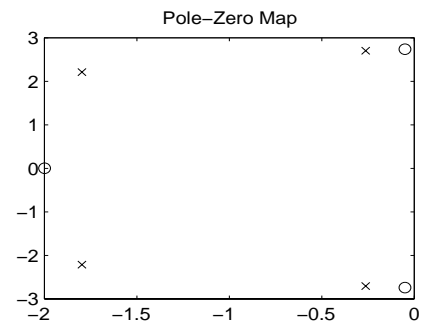
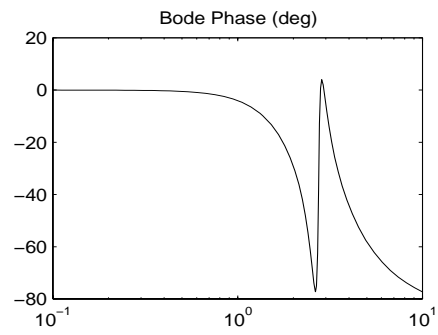
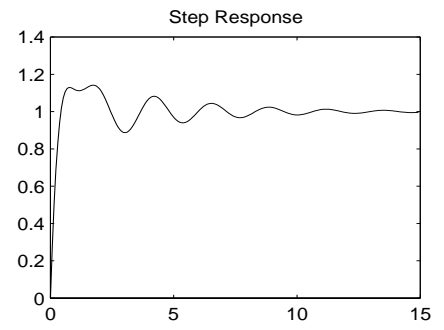
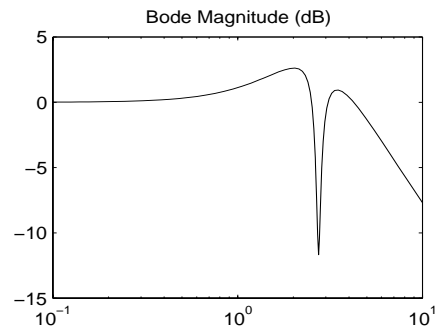
(see related entries in Reference pages for details on the syntax). It is then possible to generate customized plots with MATLAB standard plotting tools. For example, the following sequence of commands displays the Bode plot, step response, and pole/zero map on a single figure window:

```
h = tf([4 8.4 30.8 60], [1 4.12 17.4 30.8 60])
[mag, phase, w] = bode(h)
[y, t] = step(h, 15)
[p, z] = pzmap(h)

% Bode Plot
subplot(221)
semilogx(w, 20*log10(mag(:))), grid on
title('Bode Magnitude (dB)')
subplot(223)
semilogx(w, phase(:)), grid on
title('Bode Phase (deg)')

% Step response
subplot(222)
plot(t, y)
title('Step Response')

% Pole/zero map
subplot(224)
plot(z, 'go'), hold, plot(p, 'bx')
title('Pole-Zero Map')
```



Model Order Reduction

You can derive reduced-order models with the following commands:

Model Order Reduction	
bal real	Input/output balancing.
mi nreal	Minimal realization or pole/zero cancellation.
modred	State deletion in I/O balanced realization.

Use `mi nreal` to delete uncontrollable or unobservable states in state-space models, or cancel pole/zero pairs in transfer functions or zero-pole-gain models. For already minimal models, you can further reduce the model order using a combination of `bal real` and `modred`. See corresponding Reference pages for details.

The LTI Viewer

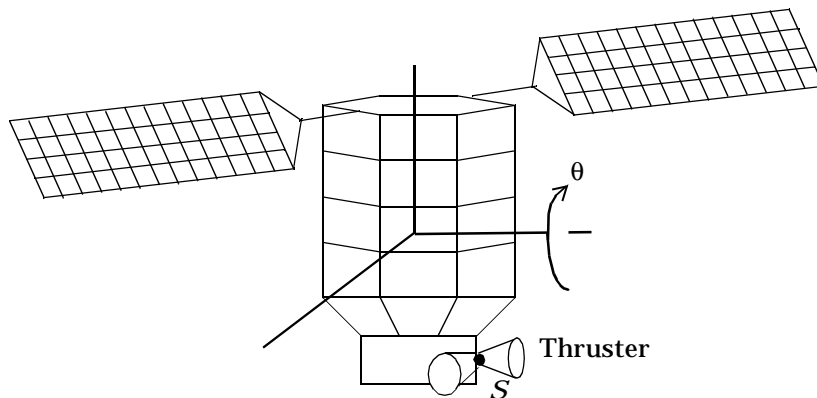
Introduction	4-2
The LTI Viewer Workspace	4-7
SISO LTI Viewer Example	4-11
MIMO LTI Viewer Example	4-19
Response Preferences	4-28
Linestyle Preferences	4-33
Simulink LTI Viewer	4-37

Introduction

This chapter shows you how to use the LTI Viewer controls and point-and-click functions through two examples. The first example steps through the analysis of three compensator designs for a simple single-input/single-output (SISO) satellite model. The second example demonstrates the more advanced features of the LTI Viewer through an open- and closed-loop multi-input/multi-output (MIMO) F-8 aircraft model example.

Getting Started

Suppose you are given the task of designing a compensator to control the orientation of one axis of the three-axis spacecraft shown below.



The attitude sensor, located at S , measures the deviation of the angular position θ from the desired value, and a rate gyro measures the rate of change of the deviation angle $\dot{\theta}$. The thrusters provide a control torque \hat{T} to adjust the attitude of the satellite and are colocated with the sensors, as shown in the figure. After working out the satellite dynamics and obtaining all the necessary physical constants, such as moments of inertia, you obtain the following simplified open-loop satellite model. Here, G is the transfer function from the control torque \hat{T} to the position angle θ . Type:

$$G = \text{tf}([1 \ 0.1 \ 7.5], [1 \ 0.12 \ 9 \ 0 \ 0])$$

and MATLAB returns:

Transfer function:

$$\frac{s^2 + 0.1s + 7.5}{s^4 + 0.12s^3 + 9s^2}$$

Notice that the satellite has a highly resonant pair of poles. Type:

`pole(G)`

and MATLAB returns:

```
ans =
      0
      0
-0.0600+ 2.9994i
-0.0600- 2.9994i
```

With the poles at these locations, the satellite will exhibit an undesirable response to a step input. You can attenuate the effects of these resonant poles by adding a proportional-derivative (PD) compensator in the feedback path. Assuming that three different PD compensators have already been designed, the remainder of your task is to decide which closed-loop system has the most satisfactory response.

Each of the PD compensators yields one of the following three closed-loop systems. Type:

Gcl 1

and MATLAB returns:

Transfer function:

$$\frac{4s^3 + 8.4s^2 + 30.8s + 60}{s^4 + 4.12s^3 + 17.4s^2 + 30.8s + 60}$$

Type:

Gcl 2

and MATLAB returns:

Transfer function:

$$2 s^3 + 1.2 s^2 + 15.1 s + 7.5$$

$$\text{-----}$$

$$s^4 + 2.12 s^3 + 10.2 s^2 + 15.1 s + 7.5$$

Type:

Gcl3

and MATLAB returns:

Transfer function:

$$1.2 s^3 + 1.12 s^2 + 9.1 s + 7.5$$

$$\text{-----}$$

$$s^4 + 1.32 s^3 + 10.12 s^2 + 9.1 s + 7.5$$

You can analyze and compare the various system responses of these three closed-loop models using the LTI Viewer. Typing

lti view

at the command line brings up the LTI Viewer shown in the next figure.

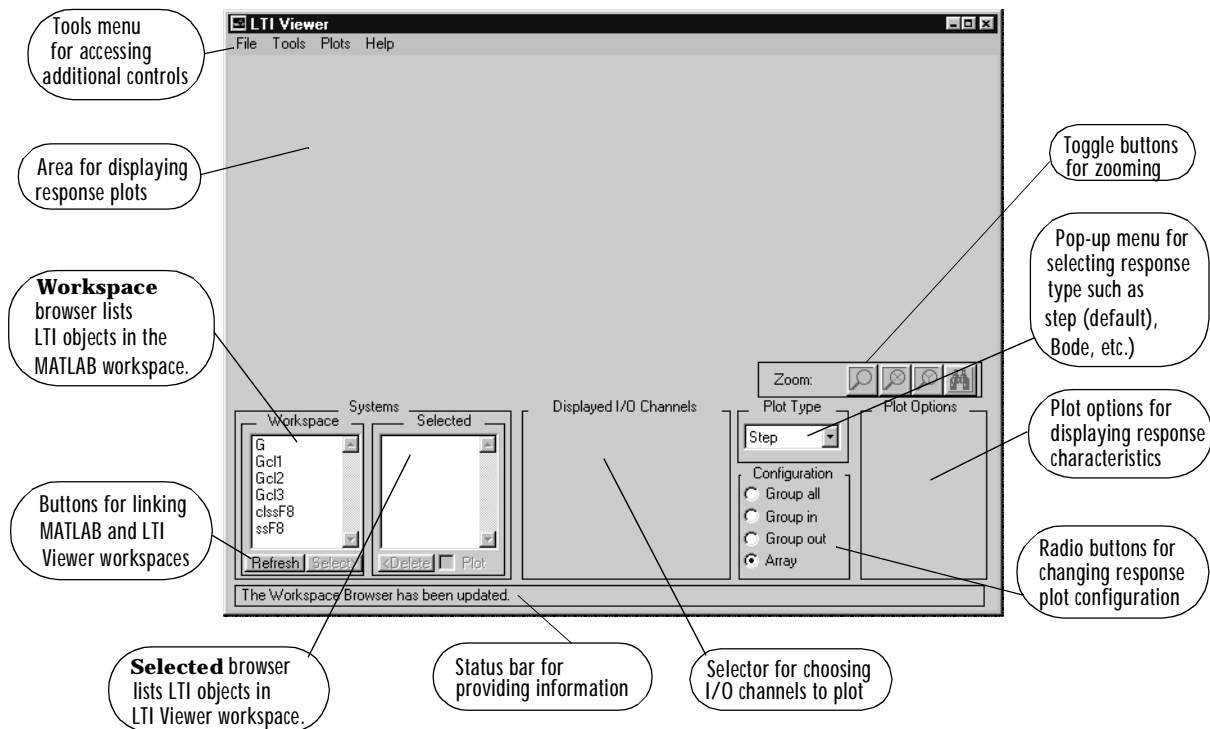


Figure 4-1: LTI Viewer

Getting Help

You can obtain instructions on how to use the previously mentioned fields directly from the LTI Viewer.

Static Help

Click on the **Help** menu; it contains three submenus.

- Viewer Controls
- Response Preferences
- Linestyle Preferences

The first submenu, **Viewer Controls**, opens the help text describing how to use the controls located on the main LTI Viewer. The remaining submenus pertain

to controls located on lower level LTI Viewer windows, the **Response** and **Linestyle Preferences** windows. These windows provide additional tools for manipulating the system responses. See the “Response Preferences” and the “Linestyle Preferences” sections for more information on the **Response** and **Linestyle Preferences** windows.

Interactive Help

In addition, the status bar at the bottom of the LTI Viewer provides you with instructions, hints, and error messages as you proceed through your analysis. In general, you can consult the status bar to learn:

- If you have tried to perform an unsupported function
- If a function has successfully been completed
- If additional information on the use of an LTI Viewer control is available

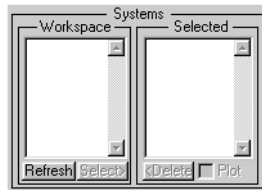
You can obtain simple reminders (tooltips) on how to use the LTI Viewer by moving your mouse and putting the cursor over one of these features. For example, if you put the cursor over the **Refresh** button, its tooltip, **Update Workspace browser**, appears just below the button. When a tooltip is available, a small bubble containing information about the feature you selected appears. This bubble disappears when you move the mouse again.

The LTI Viewer Workspace

The LTI Viewer operates out of its own workspace, which is directly linked to but completely independent of the main MATLAB workspace. You can use the **Workspace** browser to transfer information between the LTI Viewer and the MATLAB workspaces.

Using the Workspace Browser

The **Workspace** browser shown on the left in the figure below contains all the LTI models located in the MATLAB workspace at the time the LTI Viewer is opened. In effect, it is a snapshot of the LTI models that would display if you typed `whos` at the MATLAB prompt just before typing the `lti view` command.



Refreshing the Workspace Browser

If you have not already loaded the open- and closed-loop satellite models into the MATLAB workspace, do so by typing:

```
load LTIView
```

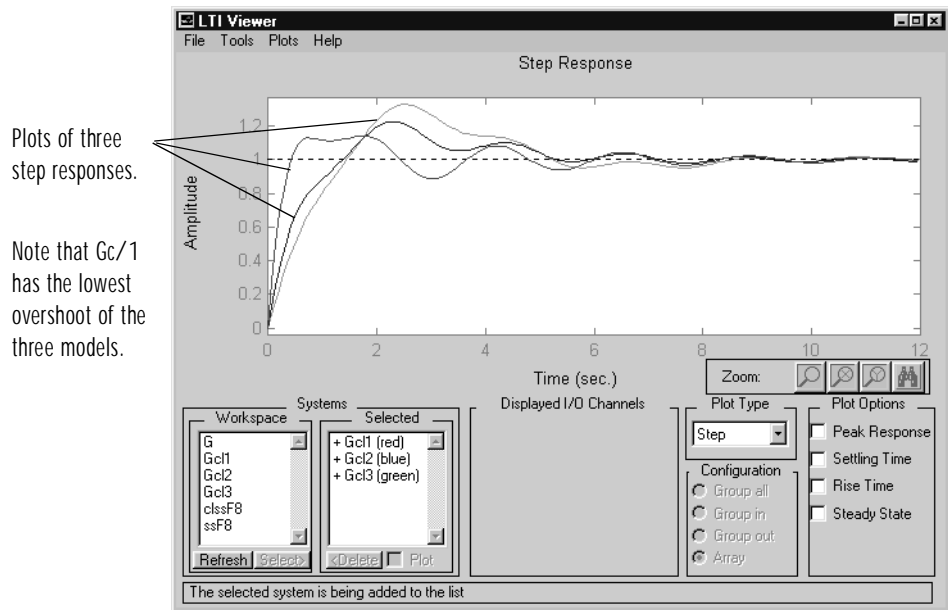
at the command line. Notice that when you add the LTI objects to the MATLAB workspace, the **Workspace** browser does not automatically reflect these additions. Now, update the LTI Viewer **Workspace** browser so it reflects the current MATLAB workspace, and press the **Refresh** button.

Selecting LTI Objects in the Workspace Browser

The systems located in the **Workspace** browser exist in the main MATLAB workspace. However, the LTI Viewer can only manipulate systems in its own workspace. The LTI Viewer workspace is reflected in the **Selected** browser shown on the right in the previous figure. To begin your analysis, move the closed-loop satellite models from the **Workspace** browser to the **Selected** browser. To move LTI objects into the **Selected** browser:

- 1 Highlight the name of the LTI object in the **Workspace** browser you want to place in the **Selected** browser. (You can also drag the cursor over several models or control-click to select multiple systems.)
- 2 Press the **Select** button. (A single system can be instantly placed in the **Selected** browser by double-clicking on its name in the **Workspace** browser.)

For this example, click on Gc1 1 and drag the cursor over the other two closed-loop models, Gc1 2 and Gc1 3. Note that the **Select** button enables as soon as you release the mouse button after selecting the objects in the **Workspace** browser. Press the **Select** button, transferring the models into the **Selected** browser. Your LTI Viewer should now look like this:



Interpreting the Selected Browser

Adding LTI objects into the **Selected** browser significantly changes the appearance of the LTI Viewer. In particular:

- The **Selected** browser contains the three closed-loop models, preceded by plus signs and followed by color codes in parenthesis.
- Display axes appear, showing plots of the three step responses.
- The **Plot Options** field contains four checkbox entries.

Generally, whenever you enter an LTI object into the **Selected** browser, the LTI Viewer automatically plots the response specified by the **Plot Type** menu described later, and adds the appropriate checkboxes in the **Plot Options** field. When the plot of an LTI object is displayed, its *plot state* is said to be on, and the name of the LTI object in the **Selected** browser is preceded by a plus sign. In this example, you can see that the display axes contain all three closed-loop step responses, as indicated by the plus sign before each system's name.

You can use the color codes following the LTI object names in the **Selected** browser to determine which response plot is associated with each model. The color code is a legend, depicting the plot styles used when plotting the responses of each model. In this example, Gcl 1 is plotted in red while Gcl 2 is in blue and Gcl 3 is in green.

Displaying/Hiding a Particular System's Response

The original plot state of any LTI object placed in the **Selected** browser is on. If, at any time, you do not want to view the plots associated with an LTI object, you can toggle its plot state off using the **Plot** checkbox. To toggle the plot state of an LTI object:

- 1 Highlight the name(s) of the model(s) in the **Selected** browser whose plot state you want to toggle.
- 2 Check/uncheck the **Plot** checkbox.

Unchecking the **Plot** checkbox turns the plot state off and hides the particular system's response whereas checking the box turns the plot state back on and displays the response. If you highlight multiple systems in the **Selected** browser, the **Plot** checkbox reflects the plot state of the majority of these

systems, or is unchecked when there is no majority among the highlighted models.

Deleting an LTI Object from the Selected Field

If there are objects in the LTI Viewer workspace that you no longer want to analyze, you can delete them using the **Delete** button below the **Selected** browser. To delete an LTI object from the LTI Viewer workspace:

- 1 In the **Selected** browser, highlight the name(s) of the model(s) that you want to remove from the LTI Viewer workspace.
- 2 Press the **Delete** button.

When you delete a system from the **Selected** browser, the LTI Viewer can no longer perform any analyses on that system. To perform further analyses on a deleted system, you must place the system back in the **Selected** browser.

Selection or Deselection of Several Models in a Browser

To select (or deselect) several models listed in either browser, hold down the control key (**Ctrl**) as you click on the individual items you want to select (or deselect).

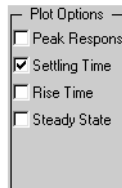
SISO LTI Viewer Example

Now that the closed-loop satellite models are loaded in the LTI Viewer workspace, you can use the other enabled Viewer Controls to perform an extensive analysis of the closed-loop satellite models.

Calculating Response Characteristics

The three closed-loop step responses indicate that Gcl 1, in red, yields the lowest overshoot (peak response). If you were concerned most about having a low overshoot, you could choose the compensator used in Gcl 1 and declare your analysis over. However, when designing compensators for systems where obtaining an accurate pointing is crucial, you may want to find the compensator that yields the lowest settling time.

You can determine which closed-loop system has the minimum settling time by using the **Plot Options** field shown below.

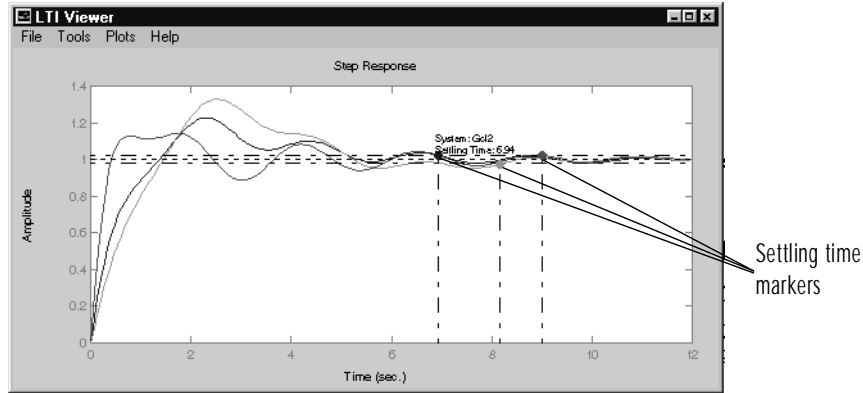


To use the **Plot Options** field to find various response characteristics:

- 1 Check the checkbox associated with the characteristic you want to display.
- 2 Hold the left mouse button down on the markers that represent the calculated quantity on the response plot, displaying the actual value.
- 3 Uncheck the checkbox to remove the markers.

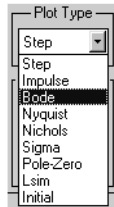
For this example, check the **Settling Time** checkbox. By default, the LTI Viewer calculates the settling time using a 2% envelope around the steady state value. Now, successively hold the left mouse button down on the three settling time markers, as shown in the plot region below. You find that the

closed-loop system Gcl 2 has the minimum settling time with a value of 6.94 seconds.



Changing the Displayed Response Type

Before deciding to use the compensator in Gcl 2, look at a frequency domain response and make sure the system has an adequate phase margin. You can use the **Plot Type** pop-up menu shown below to toggle between the different LTI model response functions.



To use the **Plot Type** menu:

- 1 Hold down the mouse button on the **Plot Type** pop-up menu.
- 2 Scroll down and select the desired type of response.
- 3 Release the mouse button.

Any time you select a new type of response, the response plots and the associated **Plot Options** field are updated.

Try selecting **Bode** from the **Plot Type** menu. The plots are updated and the names of the checkbox items under **Plot Options** change. Check the **Gain/Phase Margin** checkbox. By holding the left mouse button down on the blue circle marking the location of the phase margin of Gc1 2, you find that the closed-loop system has a phase margin of 136.9 degrees.

Note: Gain and Phase margins can only be calculated for SISO systems. In the upcoming MIMO example, the **Gain/Phase Margin** checkbox under **Plot Options** is disabled.

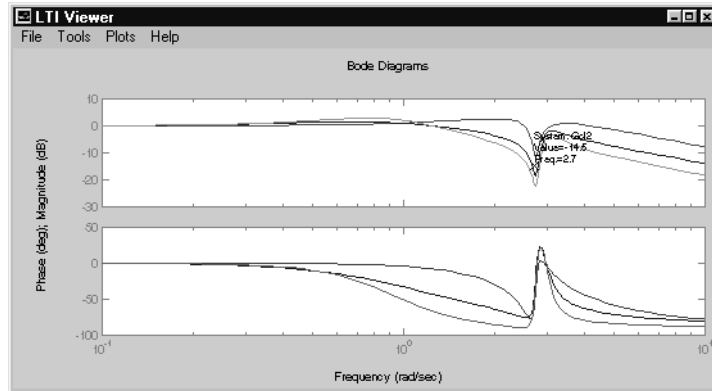
Reading Information off the Response Plots

You can obtain additional information directly from the response plots using point-and-click functions. There are two different point-and-click functions.

- Hold the right mouse button down on a response plot to display the name of the I/O channel, as well as the LTI model name.
- Hold the left mouse button down on a point on the response plot to display the exact value of that point.

Try reading additional information off the Bode diagrams using the left mouse button point-and-click feature. The Bode diagrams reveal that the models contain a pair of resonant poles and zeros, evident by the sharp spike and dip in the responses. By holding the left mouse button down on the dip in the Gc1 2

magnitude response, you find the zeros are at approximately 2.7 radians per second, as shown in the plot below.

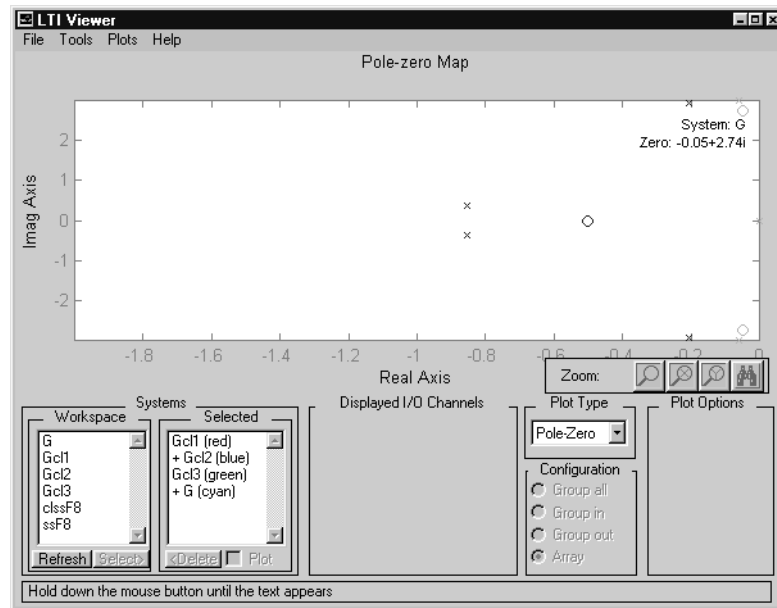


Quick Exercise

To find the exact frequency of the complex poles and zeros and compare the locations of the closed-loop poles and zeros with their open-loop counterparts, look at a pole-zero diagram. To do this, follow this quick exercise.

- 1 From the **Plot Type** menu, select **Pole-Zero**.
- 2 Toggle the plot states of Gcl 1 and Gcl 3 off by holding the **Control** key down while clicking on Gcl 1 and Gcl 3 in the **Selected** browser and unchecking the **Plot** checkbox.
- 3 Place G in the **Selected** browser by highlighting G in the **Workspace** browser and pressing the **Select** button.
- 4 Hold the left mouse button down on one of the complex zeros.

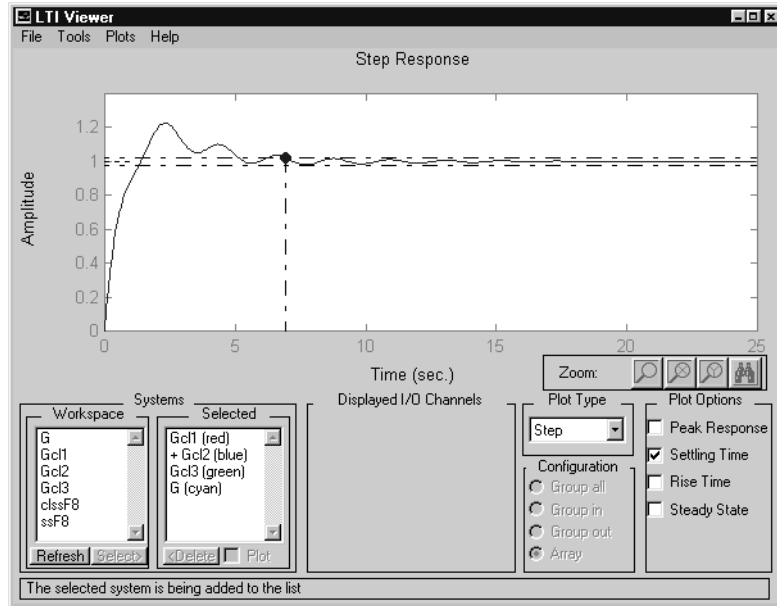
Your LTI Viewer should now look like this:



The plots show only one pair of complex zeros. By left-clicking on these zeros, you find that they are the open-loop zeros and have a frequency of 2.74 radians per second. The closed-loop zeros are at the same location, which you can confirm by toggling off the plot state of the open-loop model. Remember, to toggle off the plot of G:

- 1 Highlight G in the **Selected** browser.
- 2 Uncheck the **Plot** checkbox.

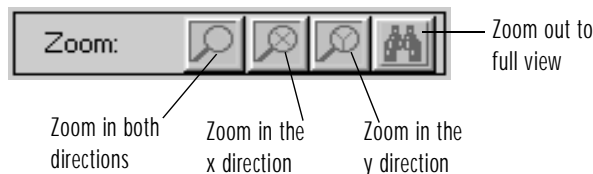
Now, with the plot states of G, Gc1 1, and Gc1 2 turned off, switch the **Plot Type** menu back to **Step**. Your LTI Viewer should look like this:



Note that when you reverted to the step response, the plot was automatically drawn with the settling time marked. The LTI Viewer remembers whenever you check a particular plot option and continues to display the checked characteristic whenever applicable, until you uncheck the box.

Zooming into Regions of the Response Plots

Now, zoom into the step response between zero seconds and the settling time. The **Zoom** buttons shown below allow you to zoom in three different directions; the x-direction only, the y-direction only, or both the x- and y-directions, simultaneously.



To perform a zoom:

- 1 Press the **Zoom** button associated with the type of zoom you want to perform.
- 2 Click the pointer in the plot region where you want to perform the zoom.
- 3 Drag the pointer over the region to zoom into.

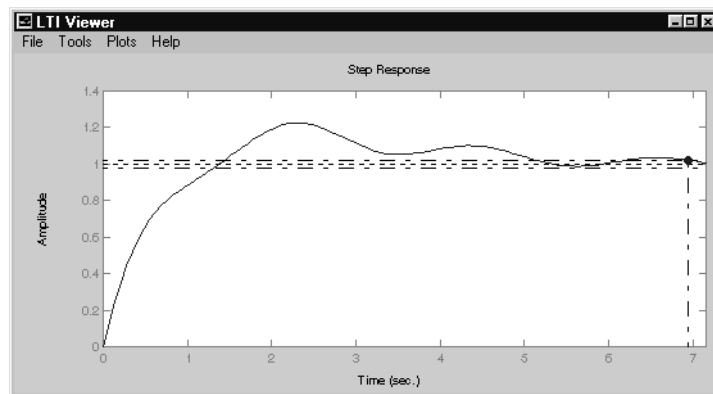
For this example, zoom into the desired time range by pressing the zoom in-x button, clicking near the settling time marker, and dragging the pointer to the left-hand axis. When you release the mouse button, the axis limits update appropriately. You can zoom back out to the original axis limits by pressing the full view button.

Generating Hard Copies of the Response Plots

To generate a hardcopy of the zoomed step response, go to the **File** menu. The **File** menu gives you several options, including:

- **New Viewer** – open a new LTI Viewer
- **Print Response** – generate a hardcopy of the LTI model response
- **Close Viewer** – close the current LTI Viewer

Choose the **Print Response** option to obtain a hardcopy, which should look something like this, depending on the time range you zoomed into:



Note that the LTI Viewer controls do not appear in any hardcopy generated from the **Print Response** option. To obtain a hardcopy of the figure with the controls, return to the MATLAB command window and type `print`.

Showing/Hiding the Viewer Controls

Whenever you print the figure from the **File** menu, you should notice that the Viewer controls are hidden and a printing dialog box opens. With the Viewer controls off, you have a preview of your hardcopy before you actually generate it. The Viewer controls are brought back as soon as you exit the printing dialog box.

You can turn the Viewer controls on and off at other times using the **Tools** menu. To toggle the Viewer controls on and off:

- 1 Open the **Tools** menu.
- 2 Select **Viewer Controls**.

When **Viewer Controls** is checked, the controls are shown, otherwise the controls are hidden.

Opening a New LTI Viewer

The **New Viewer** option in the **File** menu enables you to initialize a new LTI Viewer. This is the same as typing `lti view` at the MATLAB prompt. Select the **New Viewer** option and use this Viewer for the MIMO example shown in the next section.

MIMO LTI Viewer Example

This section analyzes the MIMO F-8 aircraft models and demonstrates the use of the **Displayed I/O Channels** and **Configuration** fields. Suppose your task is to design a state feedback control law for the longitudinal axis of an F-8 aircraft with the following LTI state-space model. Type:

ssF8

and MATLAB returns:

a =					
	Pi tchRate	Vel oci ty	AOA	Pi tchAngle	
Pi tchRate	-0. 70000	-0. 04580	-12. 20000	0	
Vel oci ty	0	-0. 01400	-0. 29040	-0. 56200	
AOA	1. 00000	-0. 00570	-1. 40000	0	
Pi tchAngle	1. 00000	0	0	0	

b =					
	El evator	Fl aperon			
Pi tchRate	-19. 10000	-3. 10000			
Vel oci ty	-0. 01190	-0. 00960			
AOA	-0. 14000	-0. 72000			
Pi tchAngle	0	0			

c =					
	Pi tchRate	Vel oci ty	AOA	Pi tchAngle	
Fl ightPath	0	0	-1. 00000	1. 00000	
Accel eration	0	0	0. 73300	0	

d =					
	El evator	Fl aperon			
Fl ightPath	0	0			
Accel eration	0. 07680	0. 11340			

Continuous-time system.

The state variable AOA is the angle of attack, and the Velocity state and Acceleration output are both measured in the horizontal direction. The Elevator and Flaperon inputs are measures of the deviation of their deflection angles about a trim angle. Again assume you already designed a control law and have the following closed-loop model. Type:

```
classF8
```

and MATLAB returns:

a =				
	PitchRate	Velocity	AOA	PitchAngle
PitchRate	-4.69515	0.01448	12.25661	-0.12750
Velocity	-0.03002	-0.01386	-0.27637	-0.56250
AOA	-1.53251	0.00443	-1.33007	-0.03949
PitchAngle	1.00000	0	0	0

b =		
	Elevator	Flaperon
PitchRate	-19.10000	-3.10000
Velocity	-0.01190	-0.00960
AOA	-0.14000	-0.72000
PitchAngle	0	0

c =				
	PitchRate	Velocity	AOA	PitchAngle
FlightPath	0	0	-1.00000	1.00000
Acceleration	0	0	0.73300	0

d =		
	Elevator	Flaperon
FlightPath	0	0
Acceleration	0.07680	0.11340

Continuous-time system.

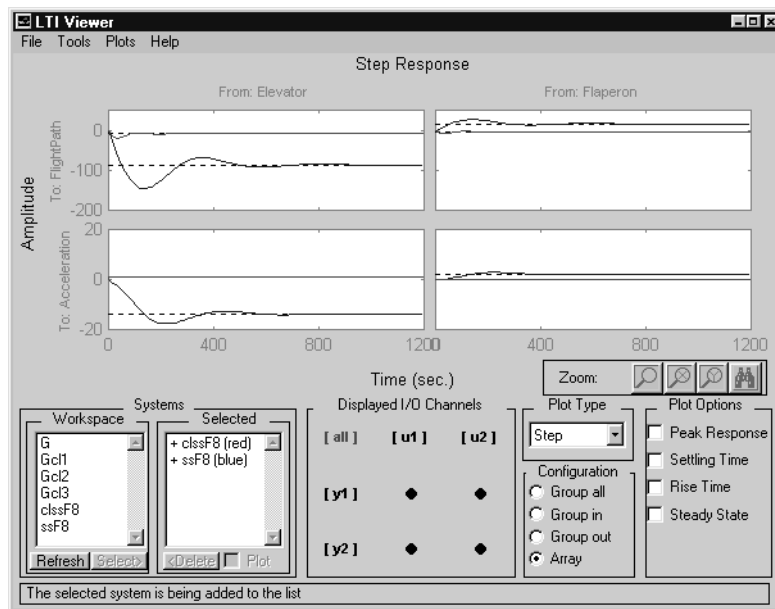
Use the new LTI Viewer to analyze the open- and closed-loop F-8 aircraft model responses.

Setting up the MIMO Example

First, place the open- and closed-loop F-8 aircraft models in the **Selected** browser. To do this, follow these steps:

- 1 Highlight the names of the LTI models ssF8 and cl ssF8 in the **Workspace** browser.
- 2 Press the **Select** button.

Your LTI Viewer should now look like this:



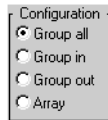
Note a few features specific to the MIMO analysis.

- Responses are plotted in an array of display axes, one for each I/O channel.
- The **Configuration** field is enabled.
- The **Displayed I/O Channels** field is activated.

Rearranging the Display

When an LTI Viewer is initialized for a MIMO model, the responses are always displayed in an array of plots, exactly as if you called the response function directly in the MATLAB command window.

The **Configuration** field shown below gives you three options for collapsing the plot configuration and an option for returning to the default array of axes.



- **Group all** places all the displayed I/O channels on one axis.
- **Group in** groups all the inputs and displays all of the plots corresponding to each input on a single axis for a given output.
- **Group out** groups all the outputs and displays all of the plots corresponding to each output on a single axis for a given input.
- **Array** plots each I/O channel in a separate axis (default).

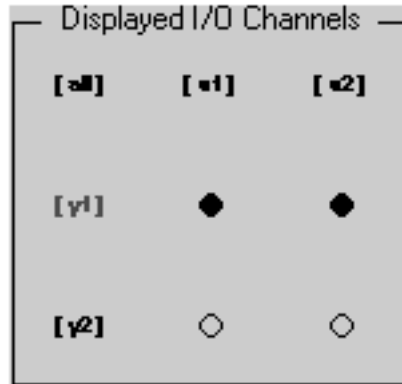
For this example, group all the responses in one axis by selecting **Group all** as shown in the figure above.

The overshoot of one of the channel's step responses is much larger than those of the other channels. However with all the plots on one axis, you may have difficulty identifying which I/O channel this response is associated with. By holding down the right mouse button on the response line with the large overshoot, you learn that this is the response of ssF8 from the Elevator input to the FlightPath output.

Note: When the text appears in the plot region, one of the **Displayed I/O Channels** selections is highlighted. This is the button associated with the I/O channel of the response you selected.

Specifying Which I/O Channels to Display

You can use the **Displayed I/O Channel** field shown below to plot only the Elevator to FlightPath response.



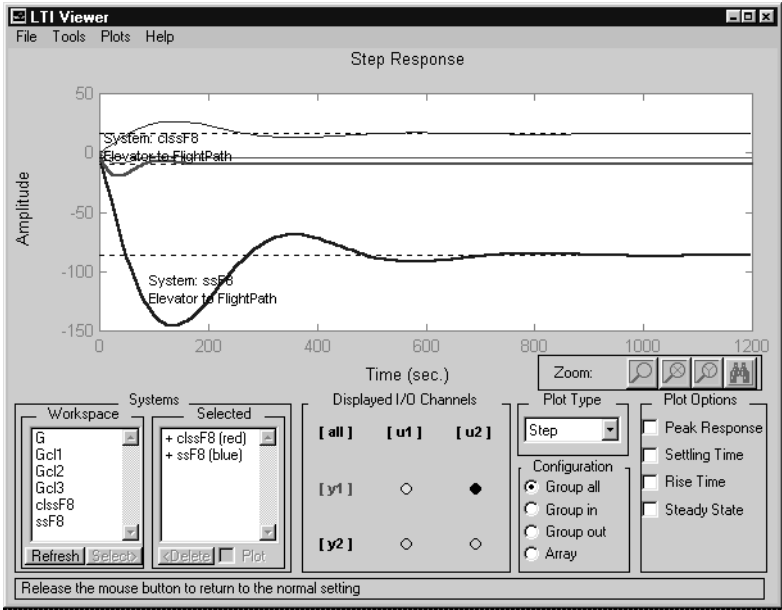
Each round button in the **Displayed I/O Channels** field represents one I/O channel of the LTI models in the **Selected** browser. (Because of this, all the systems in the **Selected** browser must have the same number of inputs and outputs.) To customize which I/O channels are actually shown in the plot, use the **Displayed I/O Channel** buttons. Do one of the following:

- Click the left mouse button on a single I/O channel button to display only that channel.
- Hold down the **Control** key while clicking on several I/O buttons to plot several channels.
- Hold down the left mouse button while dragging the cursor over a number of buttons in the **Displayed I/O Channel** field to plot the I/O channels associated with all these buttons.
- Click the left mouse button on any of the text headings to plot the associated channels.

For this example, first display only the I/O channel from the Elevator input to the FlightPath output. Again, holding the right mouse button down on the response plot with the large overshoot highlights the I/O channel button in the **u1** column and **y1** row. This is the I/O channel button associated with the Elevator to FlightPath channel. Click on it; now only the response of that channel is displayed (for both systems).

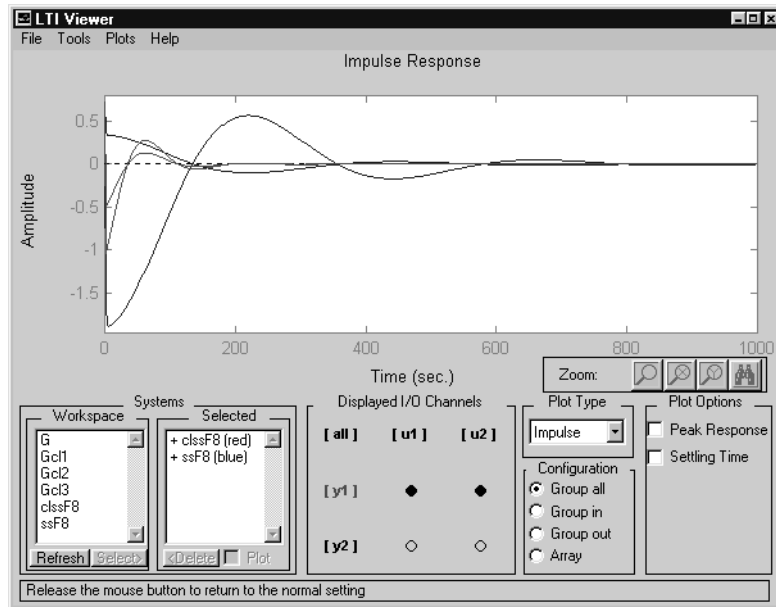
You can also easily display the responses from all the inputs to the FlightPath output. To do this, click on the **y1** text.

Once you have selected all the I/O buttons in the **y1** row, you can hold down the right mouse button on any of those I/O buttons to highlight the associated response plot. Try holding down the right mouse button on the **u1** to **y1** button. Your LTI Viewer should look like this:



Changing the Displayed MIMO Response Type

Again, you can use the **Plot Type** pop-up menu to change the displayed system response type. For this example, select **Impulse** from the **Plot Type** menu. Your LTI Viewer should now look like this:



Notice that only the selected I/O channels are displayed and the I/O channels are grouped on one axis.

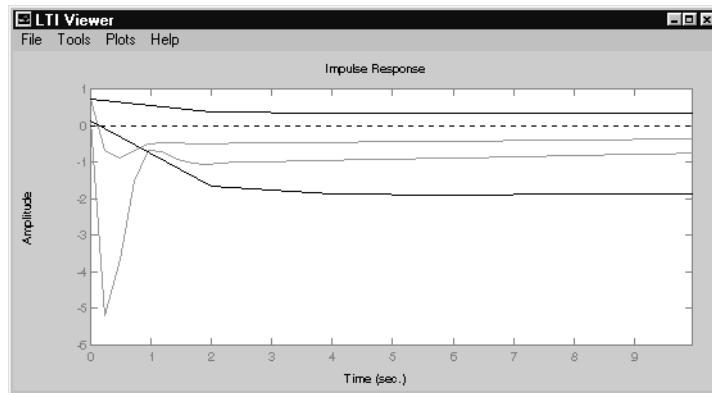
In general, the LTI Viewer overrides the default plotting behaviors of the LTI response functions and presents the plots of the I/O channels selected in the **Displayed I/O Channels** field in the configuration indicated in the **Configuration** field. In this example, instead of getting an array of axes with all the I/O channels plotted, you have only the FlightPath output channels displayed in a single axis.

Notice that the impulse response initially exhibits some large oscillations. Use the **Zoom** buttons to look more closely at those oscillations.

Remember, to use the **Zoom** buttons:

- 1 Press the zoom in-x button.
- 2 Click in the plot region and drag the cursor over the region containing the large oscillations, roughly 0-10 seconds.

Your plot should look something like this:



Note that when you perform a zoom, the response is not recalculated, only the axis limits are changed. The plot is not very smooth within this new time range, at the sampling rate used for the previous impulse response. By using the additional LTI Viewer controls located in the **Response Preferences** and **Linestyle Preferences** windows, you can recalculate the impulse response using a final time of 10 seconds and smooth out the curves.

Accessing Additional LTI Viewer Controls

Two preferences windows contain additional controls for manipulating the LTI system responses. You can access these controls from the **Tools** menu. The **Tools** menu gives you several options, including:

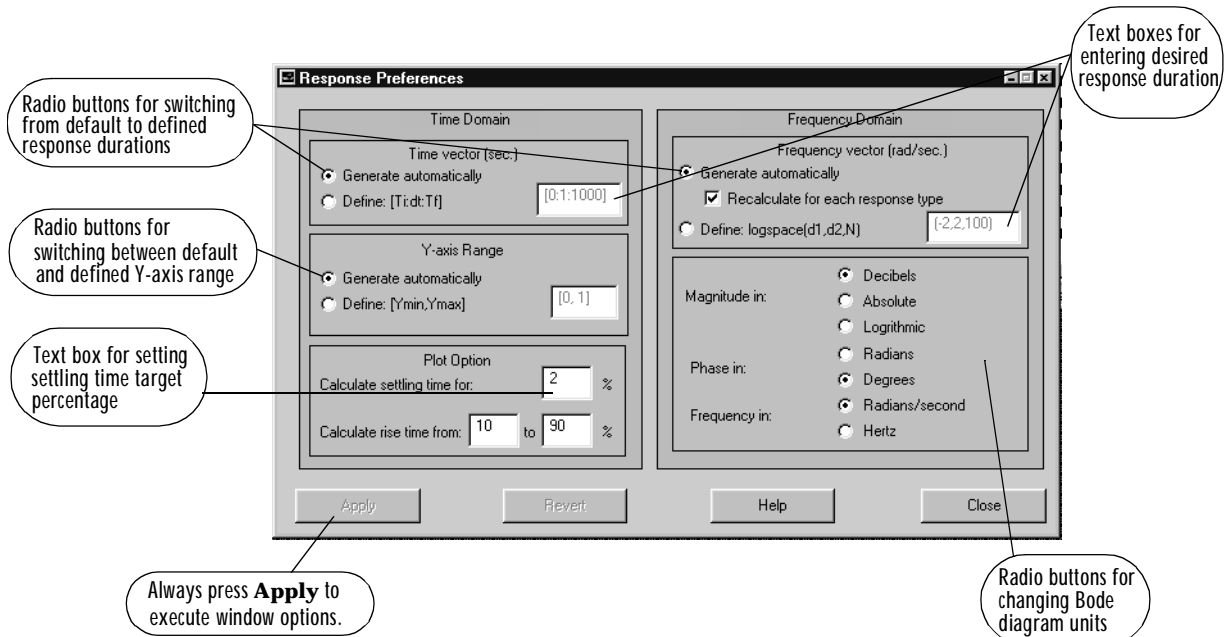
- Viewer Controls - to hide/show the LTI Viewer controls
- Response Preferences - to open a Response Preferences window
- Linestyle Preferences - to open a Linestyle Preferences window

The **Response Preferences** window allows you to set various parameters of the actual response calculations. The **Linestyle Preferences** window provides options for changing the plot styles of the resulting response plots.

For the MIMO example, choose the **Response Preferences** option and redo the impulse response using a final time of 10 seconds.

Response Preferences

When you select **Response Preferences** from the **Tools** menu, the **Response Preferences** window shown below opens.



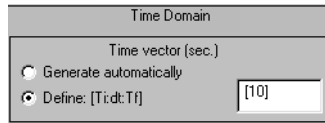
A single **Response Preferences** window controls every LTI Viewer operating in the current MATLAB session. You can use the fields in this window to:

- Enter the duration for time responses or frequency range for frequency functions.
- Choose units for plotting the magnitude, phase, and frequency of Bode diagrams.
- Specify y-axis limits for time domain responses.
- Set target percentages for settling time calculations.

Setting Response Durations

To get a smoother impulse response curve in the region from 0 to 10 seconds, you want to override the default time range and reset the final time to 10

seconds. You can use the **Time vector (sec.)** section of the **Time Domain** field shown below to do this.



The screenshot shows a dialog box titled "Time Domain". Inside, there is a section labeled "Time vector (sec.)". This section contains two radio buttons: "Generate automatically" (which is unselected) and "Define: [Ti:dt:Tf]" (which is selected). To the right of the "Define" radio button is a text input field containing the value "[10]".

The **Time vector (sec.)** field accepts one, two, or three arguments, separated by colons and surrounded by square brackets.

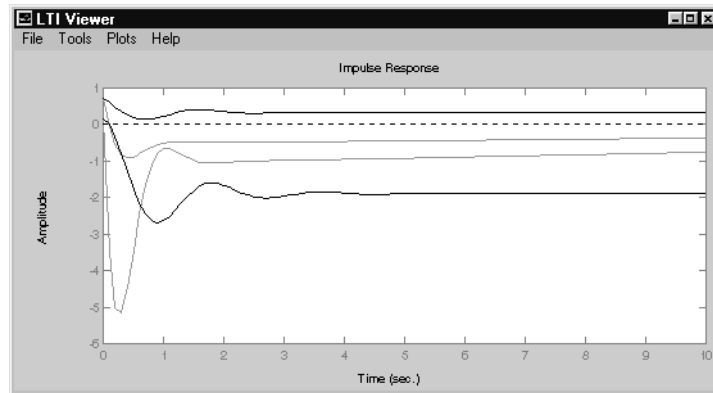
- [Tf] specifies only the final time.
- [Ti : Tf] specifies the initial and final time.
- [Ti : dt: Tf] specifies the initial and final time and provides the incremental step dt to use when generating the time vector.

Originally, the **Generate automatically** radio button is selected and the LTI Viewer automatically determines the time vector to use for the response. To override the default setting:

- 1 Select the **Define** radio button.
- 2 Enter the desired final time Tf or the new time vector as [Ti : dt: Tf] as described previously.

For this example, check the **Define** button and enter a final time of 10 seconds. Press the **Apply** button to map the change you made in the **Response**

Preferences window to the LTI Viewer. Your impulse response should now look like this:



You should also notice that the step response shown in the original LTI Viewer used in the SISO example now has a final time of 10 seconds as well. Remember, even though you opened the **Response Preferences** window from the MIMO example LTI Viewer, it controls every open LTI Viewer.

You can use the **Y-axis Range** section of the **Time Domain** field or the **Frequency vector (rad/sec.)** section in the **Frequency Domain** field to override their default settings in exactly the same manner, with the exception of the format of the text entered in the associated editable text box. Enter the y -axis limit as a row vector of the lower and upper axis limit, and the frequency vector as if it is the input argument of the `logspace` function.

The **Frequency vector (rad/sec.)** field also provides you with the option to recalculate a new frequency vector for each frequency response type. When this checkbox is selected along with **Generate automatically**, a new frequency vector and response is calculated each time you toggle between different frequency responses, e.g. from Bode to Nyquist. If you deselect the **Recalculate for each response type** checkbox, the frequency vector used to calculate the previous frequency response is used and the frequency response data is simply converted to the new response type.

Note: Whenever you override the default settings for any of these quantities, the values you enter are used on each plot and during every applicable response calculation.

Specifying Settling Time Target Percentage

Consider the SISO example started on page 4-11. Since the current **Response Preferences** window controls the SISO example LTI Viewer, you can use it to change the target percentage used in the settling time calculation. By default, the LTI Viewer defines the settling time as the time after which the response remains within an envelope of 2% of the steady state.

Use the editable text box in the **Plot Option** section of the **Time Domain** field shown below to change the percentage for the settling time envelope.

Plot Option

Calculate settling time for: 5 %

Calculate rise time from: 10 to 90 %

For this example, enter 5 in the **Plot Option** editable text box and press the **Apply** button. The SISO step response indicates the new settling time and holding the left mouse button down on the marker, as described previously, shows the 5% settling time is 4.86 seconds.

Changing Bode Diagram Units

In addition to providing options for specifying the frequencies used in the frequency responses, the **Frequency Domain** field allows you to choose the units used when plotting Bode diagrams. You can use the radio buttons in the **Frequency Domain** field shown below to override the default units used to plot Bode diagrams.

Magnitude in: ☒ Decibels ☐ Absolute ☐ Logarithmic

Phase in: ☒ Degrees ☐ Radians ☐ Radians/second

Frequency in: ☒ Hertz ☐ Radians/second ☐ Degrees

By default, the LTI Viewer plots:

- Magnitude in decibels
- Phase in degrees
- Frequency in radians per second

For this example, replot the Bode diagram of the SISO satellite model, using frequency in Hertz. To do this:

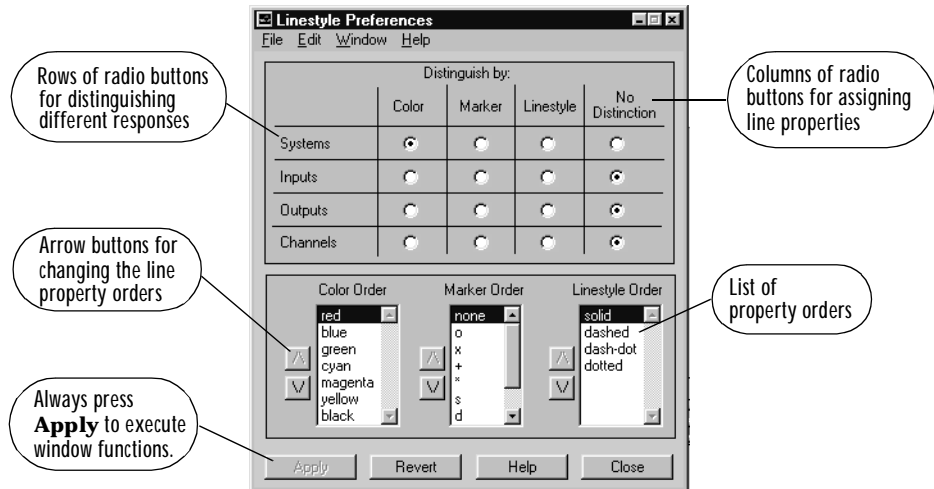
- 1 Select the **Hertz** radio button in the **Frequency in** section.
- 2 Press the **Apply** button.
- 3 Change the **Plot Type** pop-up menu on the SISO example LTI Viewer to **Bode**.

Now, return to the MIMO example of page 4-19. Note, the displayed impulse response contains two red lines for the responses of `cl ssF8` and two blue lines for the responses of `ssF8`. In this configuration, you may have a difficult time determining which of these lines represents the response from a particular input. You can use the additional controls in the **Linestyle Preferences** window to change the plot styles of the response lines and render the current plots easier to interpret. To do this:

- 1 Open the **Tools** menu.
- 2 Select **Linestyle Preferences**.

Linestyle Preferences

After selecting **Linestyle Preferences** from the **Tools** menu, the **Linestyle Preferences** window shown below opens.



As with the **Response Preferences** window, once a **Linestyle Preferences** window is initialized, it controls every LTI Viewer open in the current MATLAB session. You can use the fields in this window to:

- Select the line property used to distinguish different systems, inputs, outputs, or I/O channels.
- Change the order in which the line properties are applied.

The settings you make in the **Linestyle Preferences** window override any plot styles you may have entered in the original `lti view` command.

Try using the **Linestyle Preferences** window to change the line properties (color, marker, and linestyle) of the displayed MIMO impulse response. In particular, distinguish the responses of the two systems using different colors and the responses from different inputs using different line styles.

Altering the Response Plot Line Style Properties

In the **Linestyle Preferences** window, you can use the radio buttons in the **Distinguish by** field to vary a line property across different systems, inputs, outputs, or channels.

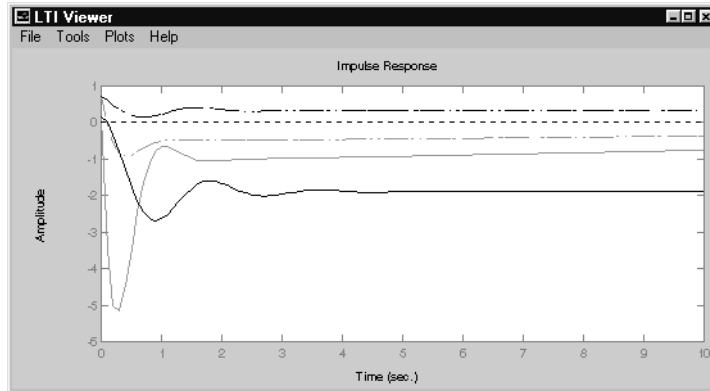
When you open the **Linestyle Preferences** window, the radio button in the **Systems** row and **Color** column of the **Distinguish by** field is selected and the radio buttons in the **No Distinction** column are selected in the remaining rows. This is the default setting for the plot styles used in the response plots.

To distinguish the responses from different inputs using different linestyles, select the radio button in the **Inputs** row and **Linestyle** column. As soon as you select this radio button, the previously selected radio button in the **Inputs** row is turned off, as shown in the figure below.

Distinguish by:				
	Color	Marker	Linestyle	No Distinction
Systems	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Inputs	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>
Outputs	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Channels	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>

The radio buttons are mutually exclusive along each row and each column with the exception of the **No Distinction** column. In other words, you can use only one line property to distinguish the different systems, inputs, outputs, or channels, and that same property cannot be applied to any other row of the

Linestyle Preferences window. Press the **Apply** button and your response plots should now look like this:

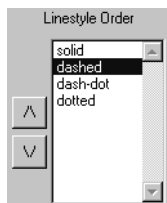


Determining the Line Property Order

You can determine which input is represented by each linestyle by referring to the line property order listboxes. The three listboxes tell you the default order in which each of the line properties will be applied. Look at the **Linestyle Order** listbox and notice that the responses from the first input will be plotted with a solid line, and the second input with a dashed line.

Changing the Line Property Order

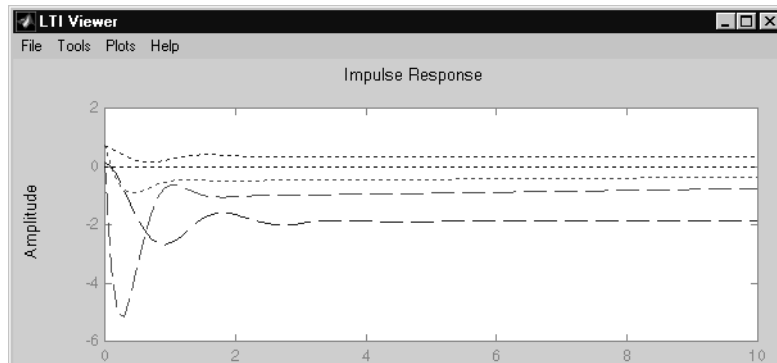
If, instead, you want to plot the response from the first input with dashed lines and those from the second input with dotted lines, you can use the up and down arrows to the left of each listbox to reorder the entries in the listboxes. The figure below shows the up and down arrows and the **Linestyle Order** listbox.



To change any listbox order:

- 1 Select the line property you want to move in the list.
- 2 Press the up and down arrows to the left of that listbox to move that property in the desired direction.

For this example, highlight **dashed** in the **Linestyle Order** listbox and press the up arrow to the left of the box once. Next, highlight **dotted** and press the same up arrow twice. Finally, press the **Apply** button and your response plots should look like this:



Now, you can easily identify the responses due to each input and associate them with the correct model.

Simulink LTI Viewer

If you have Simulink, you can use the *Simulink LTI Viewer*, a version of the LTI Viewer that performs linear analysis on any portion of a Simulink model.

The Simulink LTI Viewer features:

- Drag-and-drop blocks that identify the location for the inputs and outputs of the portion of a Simulink model you want to analyze.
- The ability to specify the operating conditions about which the Simulink model is linearized for analysis in the LTI Viewer.
- Access to all time and frequency response tools featured in the LTI Viewer.
- The ability to compare a set of (linearized) models obtained by varying either the operating conditions or some model parameter values.

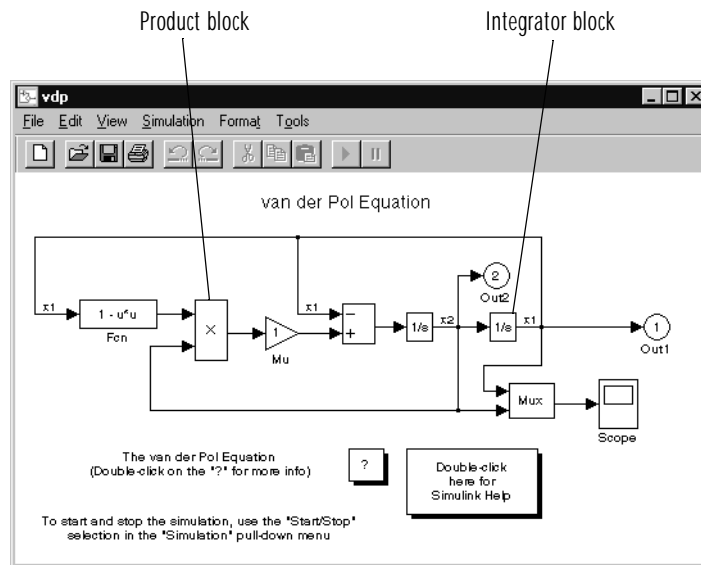
Note: The Simulink LTI Viewer in Version 4.1 of the Control System Toolbox only works on continuous-time Simulink models.

Using the Simulink LTI Viewer

To learn about the Simulink LTI Viewer, we will perform some analysis on a Simulink model for a van der Pol oscillator. To open this model, type:

```
vdp
```

at the MATLAB prompt. This brings up the following diagram:



Notice that the title of this Simulink model is **vdp**, and that it contains static nonlinearities.

A Sample Analysis Task

Suppose you want to:

- Analyze the Bode plot of the linear response between the input x_2 to the Product block and the output x_1 of the second Integrator block.
- Determine the effect of changing the value of the Gain block labeled μ on this response.

The basic procedure for carrying out this type of analysis is outlined below:

- 1 Open the Simulink LTI Viewer.
- 2 Specify your *analysis model*:
 - a Specify the portion of the Simulink model you want to analyze. This involves using special Simulink blocks to locate the inputs and outputs of this analysis model on your Simulink diagram.
 - b Set the operating conditions for linear analysis (optional). If your Simulink model includes nonlinear components, the Simulink LTI Viewer linearizes the model around the specified operation point. The default operating conditions have all state and input values set to zero.
 - c Modify any Simulink model block parameters (optional).
- 3 Perform linear analysis with the Simulink LTI Viewer:
 - a Import a linearized analysis model to the Simulink LTI Viewer.
 - b Analyze the Bode plot.
 - c Specify a second analysis model by changing the value of the Gain block, μ .
 - d Import the second linearized analysis model, and compare the Bode plots of the two linearized analysis models.
- 4 Save the analysis models for future use.

In the remaining sections of this chapter, we explain how to carry out each of these steps on the van der Pol oscillator example.

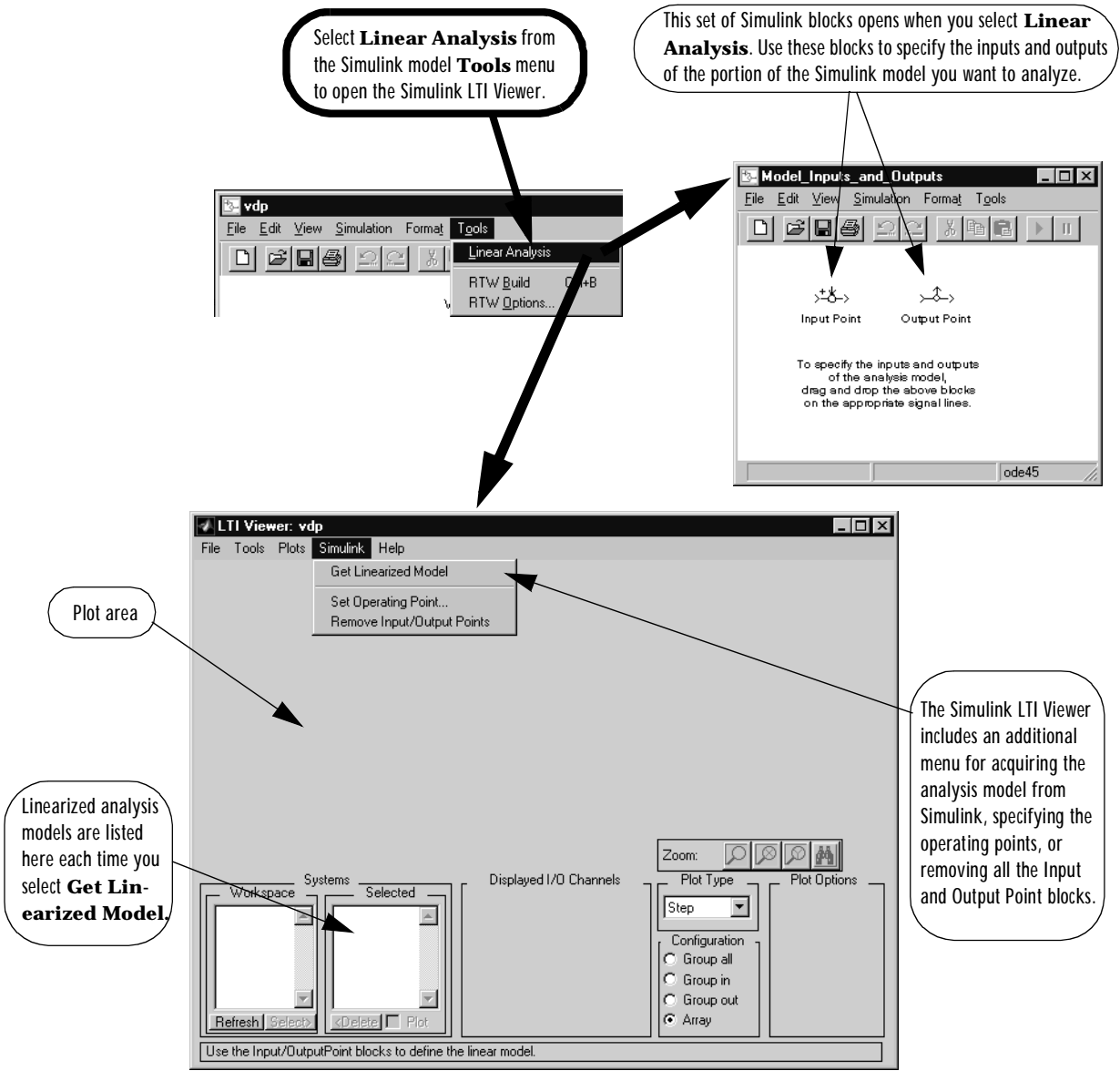
Opening the Simulink LTI Viewer

To open a Simulink LTI Viewer linked to the **vdP** Simulink model:

- 1 Go to the **Tools** menu on the Simulink model.
- 2 Select **Linear Analysis**.

When you select **Linear Analysis**, two new windows open: an LTI Viewer window and a Simulink diagram called **Model_Inputs_and_Outputs** containing two blocks: Input Point and Output Point.

The following figure depicts how to open the Simulink LTI Viewer.



The Version 4.1 Simulink LTI Viewer differs from the *regular* LTI Viewer, in that:

- The title bar shows the name of the Simulink model to which it is linked.
- It only operates on continuous-time models.
- The menu bar contains an additional menu called **Simulink** that contains the following items:
 - **Get Linearized Model** linearizes the Simulink model and imports the resulting linearized analysis model to the LTI Viewer. Each time you select this menu item, a new version of the linearized analysis model appears in the **Selected Browser** and is added to the Simulink LTI Viewer workspace.
 - **Set Operating Points** allows you to set or reset the operating conditions. Each time you set these, you must select **Get Linearized Model** to import the new linearized analysis model to your LTI Viewer.
 - **Remove Input/Output Points** clears all Input Point and Output Point blocks from the diagram.

Specifying the Simulink Model Portion for Analysis

To specify the portion of the Simulink model you want to analyze, mark its input and output signals on the Simulink model using the Input Point and Output Point blocks in the **Model_Inputs_and_Outputs** window. This defines an input/output relationship that is linearized and analyzed by the LTI Viewer.

Adding Input Point or Output Point Blocks to the Diagram

To designate the input and output signals of your analysis model, insert Input Point and Output Point blocks on the corresponding signal lines in your Simulink diagram.

For example, to insert an Input Point block on the Simulink model:

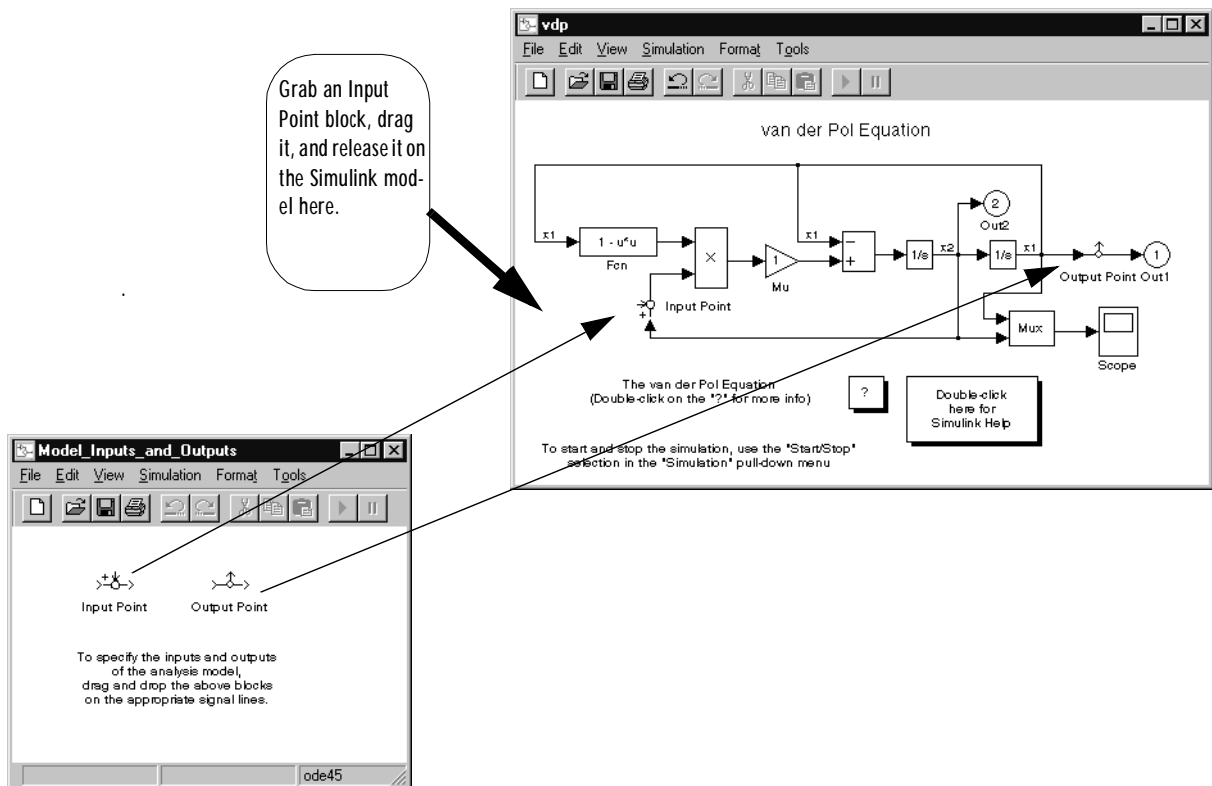
- 1 Grab the Input Point block in the **Model_Inputs_and_Outputs** window by clicking on the block and holding the mouse button down.
- 2 Drag the block to your Simulink model and place it over the line associated with the desired signal.

- 3 Release the mouse button. The block should automatically connect to the line.
- 4 If the block fails to connect (this may occur, for example, when the line is too short), resize the line and double-click on the block to force the connection.

To set up the analysis model for the **vdp** Simulink model:

- 1 Insert an Input Point block on the line labeled x2 going into the Product block.
- 2 Insert an Output Point block on the line entering the Outport block, Out 1.

This results in the following diagram.



Keep the following in mind when using the Input Point and Output Point blocks to specify analysis models:

- You must place at least one Input Point block and one Output Point block somewhere in the diagram in order to specify an analysis model.
- You can place the Input Point and Output Point blocks on any line in the Simulink model associated with a *scalar* signal.
- You can insert Input Point and Output Point blocks at different levels of a Simulink model hierarchy.
- There is no limit on the number of these blocks you can use.

Removing Input Points and Output Points

There are two ways you can remove Input Point or Output Point blocks from the Simulink model:

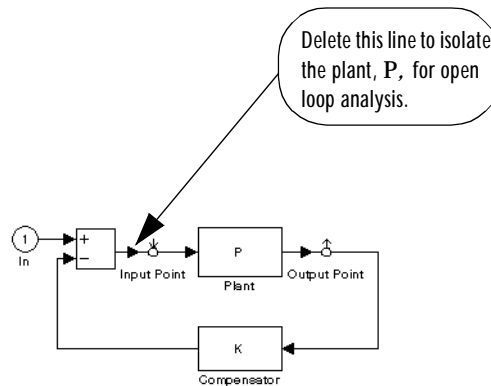
- 1 **One by one:** Select the Input Point or Output Point block you want to remove and delete it as you would any other Simulink block.
- 2 **All at once:** To remove all Input Point and Output Point blocks, select **Remove Input/Output Points** from the **Simulink** menu in the LTI Viewer.

When you delete an Input Point or an Output Point block, the signal lines coming into and out of this block are automatically reconnected.

Specifying Open- Versus Closed-Loop Analysis Models

Placing the Input Point and Output Point blocks on your Simulink model does not break any connection or isolate any component. As a result, the Simulink LTI Viewer performs closed-loop analysis whenever your diagram contains feedback loops. This may sometimes lead to counter-intuitive results, as is illustrated by the next example.

Consider the following simple diagram:



Based on the location of the Input Point and Output Point blocks, you might think that the analysis model specified by these blocks is simply the plant model, P . However, due to the feedback loop, this analysis model is actually the closed-loop transfer function $P/(1 + PK)$.

If you want to analyze the (open-loop) plant P instead, you need to open the loop, for example, by deleting the line between the Sum and Input Point blocks.

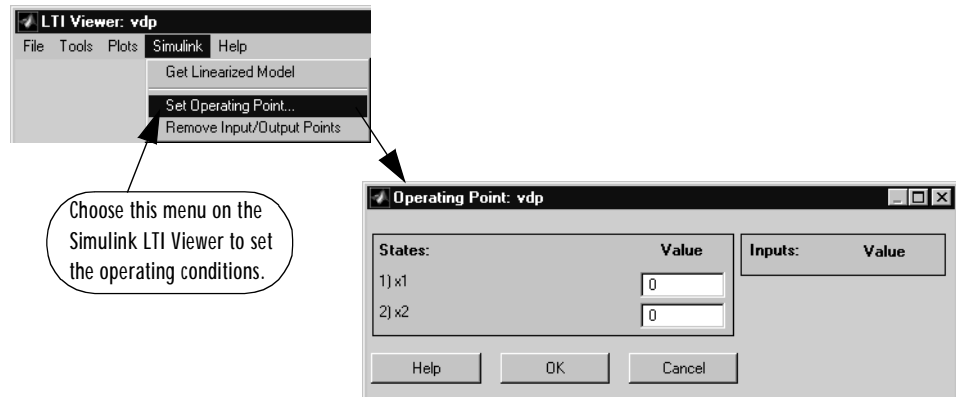
Setting the Operating Conditions

If you have nonlinear components in your Simulink model, the Simulink LTI Viewer automatically linearizes them when you select **Get Linearized Model**. You have the option to linearize about the operating conditions of your choice. The default state and input values for this linearization are zero.

If you want your analysis model to be linearized about nonzero operating conditions, follow these steps *before* selecting **Get Linearized Model**:

- 1 Select **Set Operating Point...** in the **Simulink** menu on the LTI Viewer tool bar. This opens the **Operating Point** window.
- 2 Specify the operating conditions for each state and input listed in the **Operating Point** window.
- 3 Press **OK** when finished.

For this example, we use the default settings shown in the figure below



Note the following:

- The inputs listed on the **Operating Point** window correspond to the Inport blocks on your Simulink model.
- All states and inputs in the Simulink diagram are listed in this window, not just those associated with your analysis model.
- If you want to change the operating conditions, you need only change those values associated with your analysis model.
- The values listed in the **Operating Point** window remain in effect throughout your Simulink LTI Viewer session unless you change these.

Modifying the Block Parameters

You have the option of modifying any of your Simulink model block parameters, such as Gain block gain values or LTI block poles and zeros, before you import your analysis model into the Simulink LTI Viewer.

Performing Linear Analysis

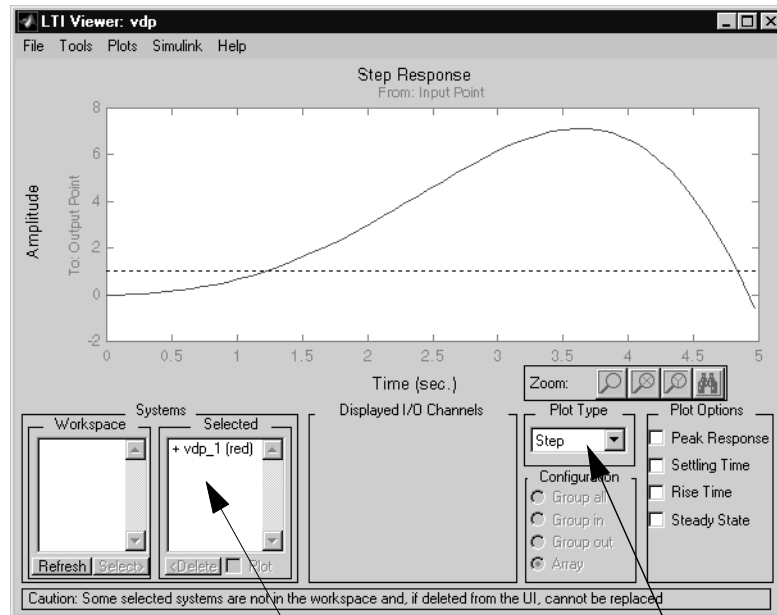
Once you have specified your analysis model, you are ready to analyze it with the Simulink LTI Viewer.

Let's use the Simulink LTI Viewer to compare the Bode plots of two different linearized analysis models. The procedure for carrying out this analysis on the van der Pol example involves:

- Importing a linearized analysis model to the Simulink LTI Viewer.
- Analyzing the Bode plot of the linearized analysis model.
- Specifying another analysis model.
- Importing the second linearized analysis model to compare the Bode plots of both linearized analysis models.

Importing a Linearized Analysis Model to the LTI Viewer

To linearize and load your first analysis model for the van der Pol example into the LTI Viewer, select **Get Linearized Model** on the **Simulink** menu on the LTI Viewer. This produces the following response plot:



The model name appears under **Selected** whenever you select the **Get Linearized Model** menu item under **Simulink**.

The LTI Viewer displays the selected response plot for all of the **Selected** models listed. Step response is the default.

Each time you select **Get Linearized Model** in the LTI Viewer's **Simulink** menu this menu item:

- Linearizes your analysis model and imports it for analysis with LTI Viewer.
- Lists the linearized model in the **Selected Browser**.

Analyzing the Bode Plot of the Linearized Analysis Model

The default response type displayed by the LTI Viewer is the step response. To analyze the Bode plot for this model, go to the pull-down tab under **Plot Type** in the lower region of the LTI Viewer, and change the plot type to **Bode**.

Specifying Another Analysis Model

Once an analysis model is specified on a Simulink diagram, you can specify other analysis models from the same Simulink diagram in one of the following three ways:

- Modify any of the model parameters.
- Change the operating conditions.
- Change the location of any of the Input Point or Output Point blocks.

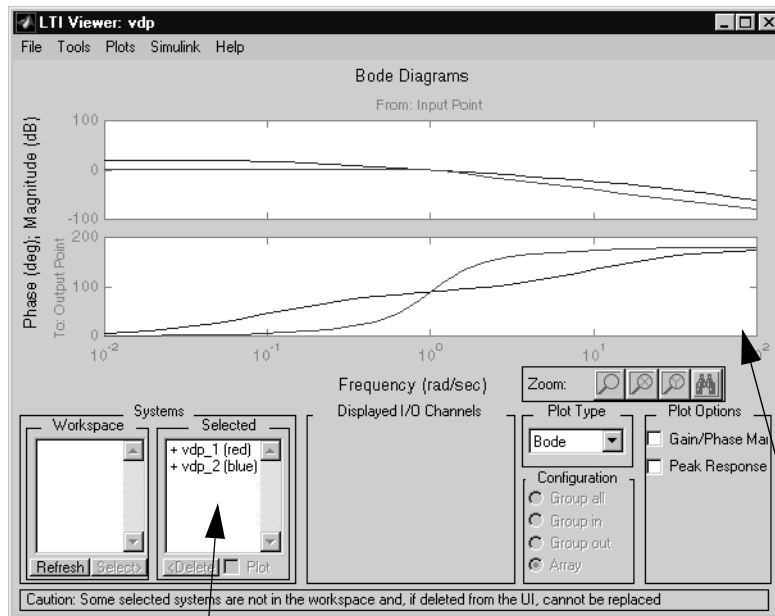
For the van der Pol example, we want to create a second analysis model by modifying a model parameter. Specifically, we modify the Gain block labeled **M₁**. To do this:

- Return to the Simulink model, and open the **M₁** Gain block by double-clicking on it.
- Change the gain to 10; click on **Apply**.

Comparing the Bode Plots of the Two Linearized Analysis Models

Having just specified a new analysis model, let's load its linearization into the LTI Viewer and compare the Bode plots of the two models. To do this, reselect the **Get Linearized Model** menu item under **Simulink** on the LTI Viewer.

As is shown below, the linearized model for the new value of M_1 appears as the last item in the **Selected** list, and the Bode plots for both models are displayed:



After reselecting **Get Linearized Model**, the **Selected** browser contains two model names with different version numbers.

The Bode plots for both models are displayed in different colors.

As you might expect, changing the gain alters the Bode plot.

Notice the following about the models listed in the **Selected** browser:

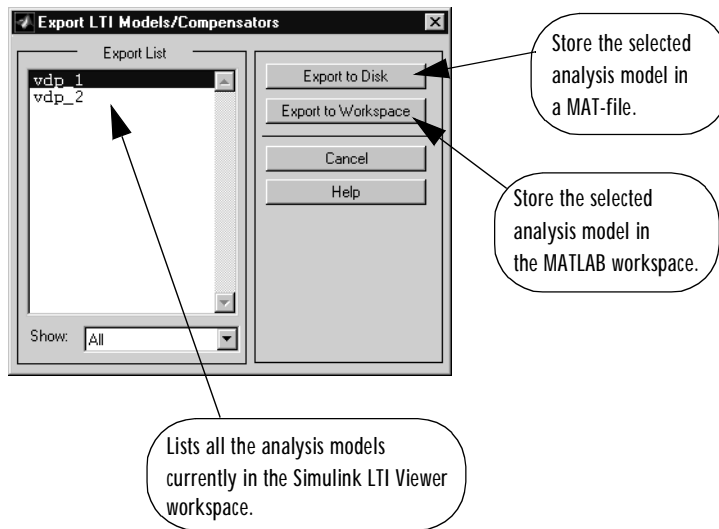
- The names reflect the title of the Simulink model.
- The version number is incremented every time **Get Linearized Model** is selected.
- The LTI Viewer simultaneously displays the response plots of all of the models listed with a "+" in the **Selected** list. (See also "Interpreting the Selected Browser" on page 4-9 for more detail.)

Note: Don't forget that the LTI Viewer analyzes only the input/output relationship between the Input Point blocks and the Output Point blocks you have placed on your Simulink model.

Saving Analysis Models

The analysis models obtained each time you select **Get Linearized Model** are stored only in the Simulink LTI Viewer workspace. You can save these models into the main MATLAB workspace by selecting **Export** from the Simulink LTI Viewer **File** menu.

Selecting **Export** opens the window shown below:



To export analysis models from the LTI Viewer workspace:

- 1 Highlight the models you want to save in the **Export List**. (See also the instructions for multiselection of models on page 4-10.)
- 2 Save the selected models by clicking on the appropriate button on this GUI. You can export the linearized models to either:
 - a A MAT-file
 - b The MATLAB workspace

Note: If you save models to a MAT-file, you are prompted to name the file. The variable names contained in that file are the same as those you selected from the **Export List**. The variable names of each model you save to the MATLAB workspace are also the same as those listed in the **Export List**. It's up to you to modify the names of these variables after you've saved them.

Control Design Tools

System Modeling	5-3
Root Locus Design	5-4
Pole Placement	5-6
State-Feedback Gain Selection	5-6
State Estimator Design	5-6
Pole Placement Tools	5-7
LQG Design	5-9
Optimal State-Feedback Gain	5-10
Kalman State Estimator	5-10
LQG Regulator	5-11
LQG Design Tools	5-11

We use the term control system *design* to refer to the process of selecting feedback gains in a closed-loop control system. Most design methods are iterative, combining parameter selection with analysis, simulation, and physical insight.

The Control System Toolbox offers functions to:

- Model the open-loop system.
- Design the control system gains using either classical root locus techniques or modern pole placement and LQG techniques.
- Close the loop for simulation and validation purposes.

System Modeling

The Control System Toolbox provides a number of functions to help with the model building process. This includes functions to perform general parallel and series connections (`parallel` and `series`), I/O concatenation (`append`), feedback connections (`feedback` and `star`), and general block-diagram modeling (`append` and `connect`). These functions are useful to model open- and closed-loop systems. In addition, functions are provided to generate random state-space models (`rss` and `drss`) and second-order models (`ord2`).

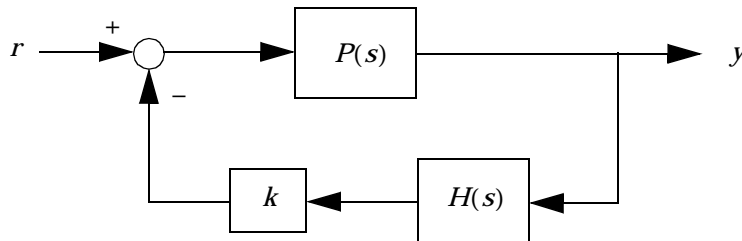
Model Building	
<code>append</code>	Group systems by appending inputs and outputs.
<code>augstate</code>	Augment output by appending states.
<code>connect</code>	Block diagram modeling.
<code>drss</code>	Generate random discrete state-space model.
<code>feedback</code>	Feedback connection.
<code>ord2</code>	Generate second-order model.
<code>pade</code>	Pade approximation of time delays.
<code>parallel</code>	Generalized parallel connection.
<code>rss</code>	Generate random continuous state-space model.
<code>series</code>	Generalized series connection.
<code>star</code>	Star product (LFT interconnection).

You can also use Simulink and the command `linmod` to derive linearized models of complex block diagrams.

Root Locus Design

The root locus method is used to describe the trajectories of the closed-loop poles of a feedback system as one parameter varies over a continuous range of values. Typically, the root locus method is used to tune a feedback gain so as to specify the closed-loop poles of a SISO control system.

Consider, for example, the tracking loop



where $P(s)$ is the plant, $H(s)$ is the sensor dynamics, and k is a scalar gain to be adjusted. The closed-loop poles are the roots of

$$q(s) = 1 + k P(s)H(s)$$

The root locus technique consists of plotting the closed-loop pole trajectories in the complex plane as k varies. You can use this plot to identify the gain value associated with a given set of closed-loop poles on the locus.

The command `rl tool` opens the Root Locus Design GUI. In addition to plotting the root locus, the Root Locus Design GUI can be used to design a compensator interactively to meet some system design specifications.

The Root Locus Design GUI can be used to:

- Analyze the root locus plot for a SISO LTI control system
- Specify feedback compensator parameters: poles, zeros, and gain
- Examine how the compensator parameters change the root locus, as well as various closed-loop system responses (step response, Bode plot, Nyquist plot, or Nichols chart)

Chapter 6 provides more detail on the Root Locus Design GUI.

If you are interested in just the root locus plot, use the command `rl locus`. This command takes one argument: a SISO model of the open loop system, created with `ss`, `tf`, or `zpk`. In the tracking loop depicted on the previous page, this model would represent $P(s)H(s)$. You can also use the function `rl ocfind` to select a point on the root locus plot and determine the corresponding gain k .

The following table summarizes the commands for root locus design:

Root Locus Design	
<code>pzmap</code>	Pole-zero map.
<code>rl tool</code>	Root Locus Design GUI.
<code>rl ocfind</code>	Interactive root locus gain selection.
<code>rl locus</code>	Evans root locus plot.
<code>sgrid</code>	Continuous ω_r ζ grid for root locus.
<code>zgrid</code>	Discrete ω_r ζ grid for root locus.

Pole Placement

The closed-loop pole locations have a direct impact on time response characteristics such as rise time, settling time, and transient oscillations. This suggests the following method for tuning the closed-loop behavior:

- 1 Based on the time response specifications, select desirable locations for the closed-loop poles.
- 2 Compute feedback gains that achieve these locations.

This design technique is known as pole placement.

Pole placement requires a state-space model of the system (use `ss` to convert other LTI models to state space). In continuous time, this model should be of the form

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

where u is the vector of control inputs and y is the vector of measurements. Designing a dynamic compensator for this system involves two steps: state-feedback gain selection, and state estimator design.

State-Feedback Gain Selection

Under state feedback $u = -Kx$, the closed-loop dynamics are given by

$$\dot{x} = (A - BK) x$$

and the closed-loop poles are the eigenvalues of $A - BK$. Using pole placement algorithms, you can compute a gain matrix K that assigns these poles to any desired locations in the complex plane (provided that (A, B) is controllable).

State Estimator Design

You cannot implement the state-feedback law $u = -Kx$ unless the full state x is measured. However, you can construct a state estimate \hat{x} such that the

law $u = -K\hat{\xi}$ retains the same pole assignment properties. This is achieved by designing a state estimator (or observer) of the form

$$\dot{\hat{\xi}} = A\hat{\xi} + Bu + L(y - C\hat{\xi} - Du)$$

The estimator poles are the eigenvalues of $A - LC$, which can be arbitrarily assigned by proper selection of the estimator gain matrix L . As a rule of thumb, the estimator dynamics should be faster than the controller dynamics (eigenvalues of $A - BK$).

Replacing x by its estimate $\hat{\xi}$ in $u = -Kx$ yields the dynamic output-feedback compensator:

$$\begin{aligned}\dot{\hat{\xi}} &= [A - LC - (B - LD)K]\hat{\xi} + Ly \\ u &= -K\hat{\xi}\end{aligned}$$

Note that the resulting closed-loop dynamics are

$$\begin{bmatrix} \dot{x} \\ \dot{e} \end{bmatrix} = \begin{bmatrix} A - BK & BK \\ 0 & A - LC \end{bmatrix} \begin{bmatrix} x \\ e \end{bmatrix}, \quad e = x - \hat{\xi}$$

Hence, you actually assign all closed-loop poles by independently placing the eigenvalues of $A - BK$ and $A - LC$.

Pole Placement Tools

The Control System Toolbox contains functions to:

- Compute gain matrices K and L that achieve the desired closed-loop pole locations
- Form the state estimator and dynamic compensator using these gains

Pole Placement	
acker	SISO pole placement.
estim	Form state estimator given estimator gain.
place	MIMO pole placement.
reg	Form output-feedback compensator given state-feedback and estimator gains.

The function `acker` is limited to SISO systems and should only be used for systems with a small number of states. The function `place` is a more general and numerically robust alternative to `acker`.

Caution: Pole placement can be badly conditioned if you choose unrealistic pole locations. In particular, you should avoid

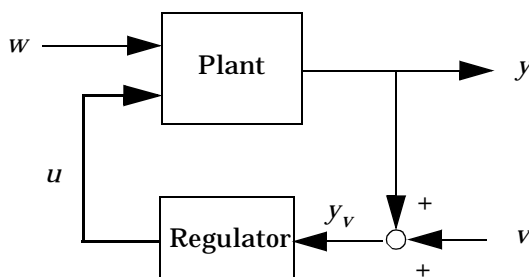
- Placing multiple poles at the same location
- Moving poles that are weakly controllable or observable. This typically requires high gain, which in turn makes the entire closed-loop eigenstructure very sensitive to perturbations.

Finally, the fewer control inputs, the more challenging pole placement tends to be.

LQG Design

Linear-Quadratic-Gaussian (LQG) control is a modern state-space technique for designing optimal dynamic regulators. It enables you to trade off regulation performance and control effort, and to take into account process and measurement noise. Like pole placement, LQG design requires a state-space model of the plant (use `ss` to convert other LTI models to state space). This section focuses on the continuous-time case (see related Reference pages for details on discrete-time LQG design).

LQG design addresses the following regulation problem:



The goal is to regulate the output y around zero. The plant is driven by the process noise w and the controls u , and the regulator relies on the noisy measurements $y_v = y + v$ to generate these controls. The plant state and measurement equations are of the form

$$\begin{aligned} \dot{x} &= Ax + Bu + Gw \\ y_v &= Cx + Du + Hw + v \end{aligned}$$

and both w and v are modeled as white noise.

The LQG regulator consists of an optimal state-feedback gain and a Kalman state estimator. You can design these two components independently as shown next.

Optimal State-Feedback Gain

In LQG control, the regulation performance is measured by a quadratic performance criterion of the form

$$J(u) = \int_0^{\infty} \{x^T Q x + 2x^T N u + u^T R u\} dt$$

The weighting matrices Q, N, R are user specified and define the trade-off between regulation performance (how fast $x(t)$ goes to zero) and control effort.

The first design step seeks a state-feedback law $u = -Kx$ that minimizes the cost function $J(u)$. The minimizing gain matrix K is obtained by solving an algebraic Riccati equation. This gain is called the *LQ-optimal* gain.

Kalman State Estimator

As for pole placement, the LQ-optimal state feedback $u = -Kx$ is not implementable without full state measurement. However, we can derive a state estimate \hat{x} such that $u = -K\hat{x}$ remains optimal for the output-feedback problem. This state estimate is generated by the Kalman filter

$$\dot{\hat{x}} = A\hat{x} + Bu + L(y_v - C\hat{x} - Du)$$

with inputs u (controls) and y_v (measurements). The noise covariance data

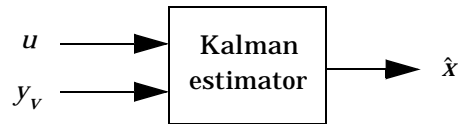
$$E(ww^T) = Q_n, \quad E(vv^T) = R_n, \quad E(wv^T) = N_n$$

determines the Kalman gain L through an algebraic Riccati equation.

The Kalman filter is an optimal estimator when dealing with Gaussian white noise. Specifically, it minimizes the asymptotic covariance

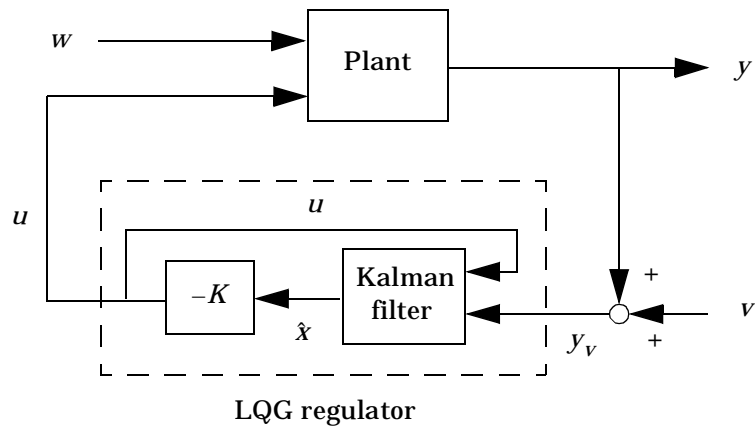
$$\lim_{t \rightarrow \infty} E((x - \hat{x})(x - \hat{x})^T)$$

of the estimation error $x - \hat{x}$.



LQG Regulator

To form the LQG regulator, simply connect the Kalman filter and LQ-optimal gain K as shown below:



This regulator has state-space equations:

$$\dot{\hat{x}} = [A - LC - (B - LD)K] \hat{x} + Ly_v$$

$$u = -K\hat{x}$$

LQG Design Tools

The Control System Toolbox contains functions to perform the three LQG design steps outlined above. These functions cover both continuous and

discrete problems as well as the design of discrete LQG regulators for continuous plants.

LQG Design	
care	Solve continuous-time algebraic Riccati equations.
dare	Solve discrete-time algebraic Riccati equations.
dlqr	LQ-optimal gain for discrete systems.
kalman	Kalman estimator.
kalmd	Discrete Kalman estimator for continuous plant.
lqgreg	Form LQG regulator given LQ gain and Kalman filter.
lqr	LQ-optimal gain for continuous systems.
lqrd	Discrete LQ gain for continuous plant.
lqry	LQ-optimal gain with output weighting.

See the case study on page 7-30 for an example of LQG design. You can also use the functions `kalman` and `kalmd` to perform Kalman filtering; see the case study on page 7-49 for details.

The Root Locus Design GUI

Introduction	6-2
A Servomechanism Example	6-4
Controller Design Using the Root Locus Design GUI . .	6-6
Opening the Root Locus Design GUI	6-6
Importing Models into the Root Locus Design GUI	6-7
Changing the Gain Set Point and Zooming	6-13
Displaying System Responses	6-20
Designing the Compensator to Meet Specifications	6-23
Saving the Compensator and Models	6-36
Additional Root Locus Design GUI Features	6-38
Specifying Design Models: General Concepts	6-38
Getting Help with the Root Locus Design GUI	6-39
Erasing Compensator Poles and Zeros	6-40
Listing Poles and Zeros	6-41
Printing the Root Locus	6-42
Drawing a Simulink Diagram	6-43
Converting Between Continuous and Discrete Models	6-44
Clearing Data	6-45
References	6-46

Introduction

The root locus method is used to describe the trajectories in the complex plane of the closed-loop poles of a SISO feedback system as one parameter (usually a gain) varies over a continuous range of values.

Along with the MATLAB command line function `rl locus`, the Control System Toolbox provides the Root Locus Design Graphical User Interface (GUI) for implementing root locus methods on single-input/single-output (SISO) LTI models defined using `zpk`, `tf`, or `ss`.

In addition to plotting root loci, the Root Locus Design GUI is an interactive design tool that can be used to:

- Analyze the root locus plot for a SISO LTI control system
- Specify the parameters of a feedback compensator: poles, zeros, and gain
- Examine how changing the compensator parameters effects changes in the root locus and various closed-loop system responses (step response, Bode plot, Nyquist plot, or Nichols chart)

This chapter explains how to use the Root Locus Design GUI, in part, through an example involving an electrohydraulic servomechanism comprised of an electrohydraulic amplifier, a valve, and a ram. These types of servomechanisms can be made quite small, and are used for position control. Details on the modeling of electrohydraulic position control mechanisms can be found in [1].

After an explanation of the servomechanism control system, the following operations on the Root Locus Design GUI are covered in the section, “Controller Design Using the Root Locus Design GUI” on page 6-6:

- Opening the Root Locus Design GUI
- Importing models into the Root Locus Design GUI
- Changing the gain set point and zooming
- Displaying system responses
- Designing the compensator to meet specifications
 - Specifying the design region boundaries on the root locus
 - Placing compensator poles and zeros

- Editing the compensator pole and zero locations
- Activating the LTI Viewer
- Saving the compensator and models to the workspace or the disk

Other important features listed below and not covered through this example are described in the section, “Additional Root Locus Design GUI Features” on page 6-38:

- Specifying design models: general concepts
- Getting help with the Root Locus Design GUI
- Erasing compensator poles and zeros
- Listing poles and zeros
- Printing the root locus
- Drawing a Simulink diagram of the closed-loop model
- Converting between continuous and discrete models
- Clearing data

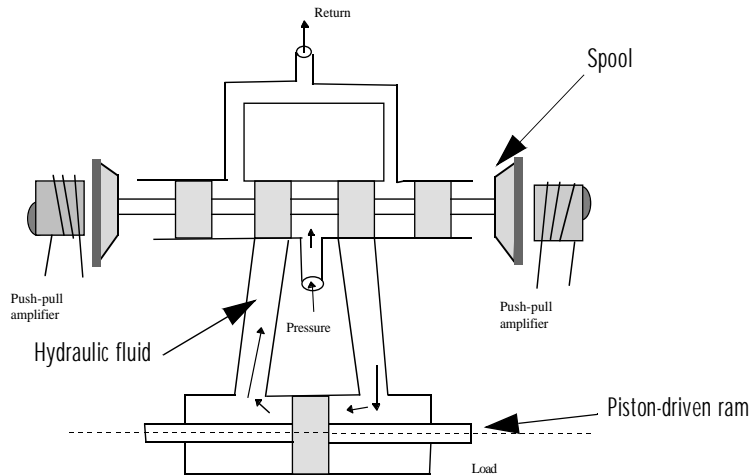
A Servomechanism Example

A simple version of an electrohydraulic servomechanism model consists of

- A push-pull amplifier (a pair of electromagnets)
- A sliding spool in a vessel of high pressure hydraulic fluid
- Valve openings in the vessel to allow for fluid to flow
- A central chamber with a piston-driven ram to deliver force to a load
- A symmetrical fluid return vessel

The force on the spool is proportional to the current in the electromagnet coil. As the spool moves, the valve opens, allowing the high pressure hydraulic fluid to flow through the chamber. The moving fluid forces the piston to move in the opposite direction of the spool. In [1], linearized models for the electromagnetic amplifier, the valve spool dynamics, and the ram dynamics are derived, and a detailed description of this type of servomechanism is provided.

A schematic of this servomechanism is depicted below.



If you want to use this servomechanism for position control, you can use the input voltage to the electromagnet to control the ram position. When measurements of the ram position are available, you can use feedback for the ram position control.

A closed-loop model for the electrohydraulic valve position control can be set up as follows:

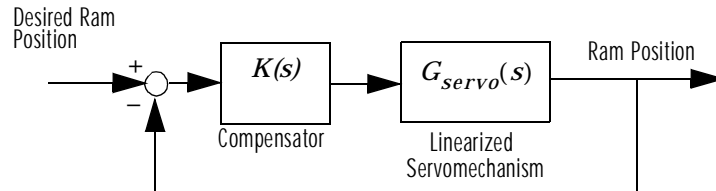


Figure 6-1: Feedback Control for an Electrohydraulic Servomechanism

$K(s)$ represents the compensator for you to design. This compensator can be either a gain or a more general LTI system.

A linearized plant model for the electrohydraulic position control mechanism is given by:

$$G_{servo}(s) = \frac{4 \times 10^7}{s(s + 250)(s^2 + 40s + 9 \times 10^4)}$$

For this example, you want to design a controller so that the step response of the closed-loop system meets the following specifications:

- The two-percent settling time is less than 0.05 seconds.
- The maximum overshoot is less than 5 percent.

For details on how these specifications are defined, see [2].

In the remainder of this chapter you learn how to use the Root Locus Design GUI. In the process, you design a controller to meet these specifications.

Controller Design Using the Root Locus Design GUI

In this section, we use the servomechanism example to describe some of the main Root Locus Design GUI features and operations:

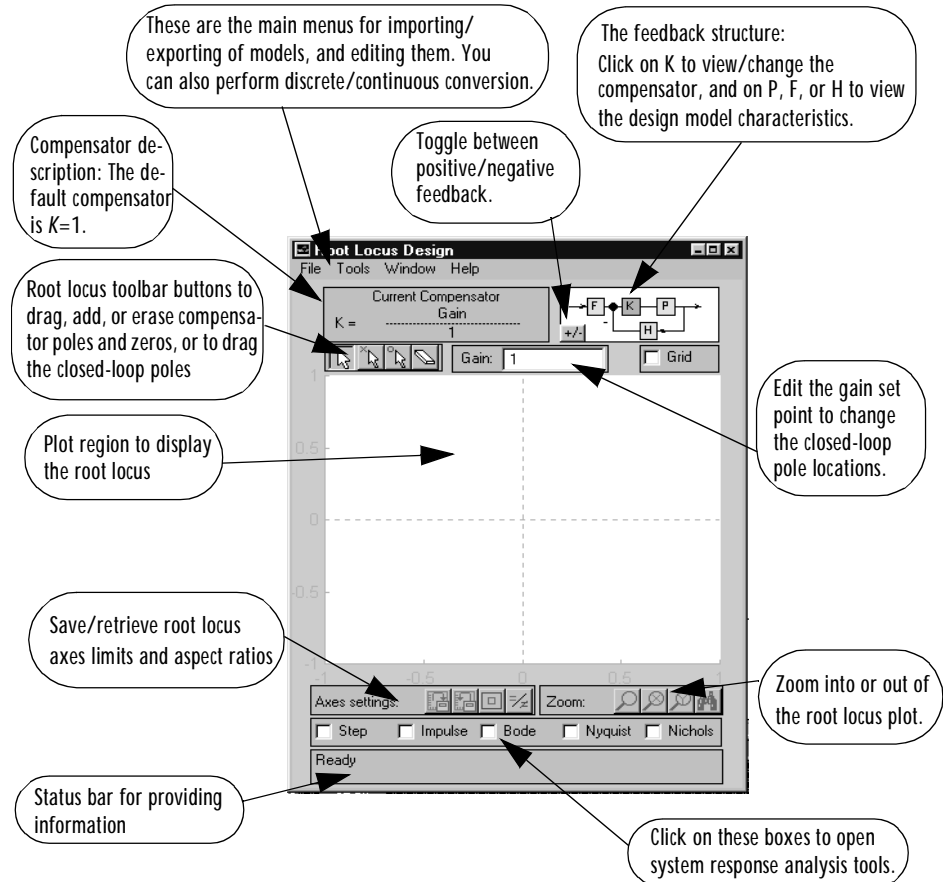
- Opening the Root Locus Design GUI
- Importing models into the Root Locus Design GUI
 - Opening the **Import LTI Design Model** window
 - Choosing a feedback structure
 - Specifying the design model
- Changing the gain set point and zooming
 - Dragging closed-loop poles to change the gain set point
 - Zooming
 - Storing and retrieving axes limits
- Displaying system responses
- Designing the compensator to meet specifications
 - Specifying the design region boundaries on the root locus
 - Placing compensator poles and zeros
 - Placing compensator poles and zeros using the root locus toolbar
 - Editing compensator pole and zero locations
 - Activating the LTI Viewer
- Saving the compensator and models to the workspace or the disk

Opening the Root Locus Design GUI

To open the Root Locus Design GUI, at the MATLAB prompt type

```
rltool
```

This brings up the following GUI:



Importing Models into the Root Locus Design GUI

The Root Locus Design GUI operates on SISO LTI models constructed using either `tf`, `zpk`, or `ss` (for detail on creating models, see “Creating LTI Models” in Chapter 2).

There are four ways to import SISO LTI models into the Root Locus Design GUI:

- Load a model from the MATLAB workspace.
- Load a model from a MAT-file on your disk.
- Load SISO LTI blocks from an open or saved Simulink diagram.
- Create models using `tf`, `ss`, or `zpk` within the GUI.

You can also use any combination of these methods to import models for root locus analysis and design. However, before you can import a model into the Root Locus Design GUI from the MATLAB workspace, you must have at least one SISO LTI model loaded into your workspace. Similar requirements hold for loading models from the disk or an open Simulink diagram.

For this example, we import our servomechanism model for root locus analysis from the MATLAB workspace. You can find a zero-pole-gain model for $G_{servo}(s)$ in a set of LTI models provided in the file `LTIView.MAT`.

To load these LTI models, at the MATLAB prompt type

```
load LTIView
```

The model for the position control system is contained in the variable `Gservo`. To view the information on this model, at the MATLAB prompt type

```
Gservo
```

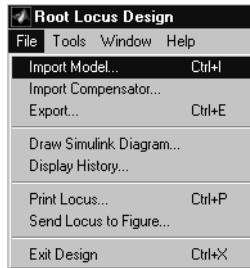
Now that a model for $G_{servo}(s)$ is loaded into the workspace, you can begin your root locus analysis and design for this example.

There are three steps involved in importing a model for our example covered in this section:

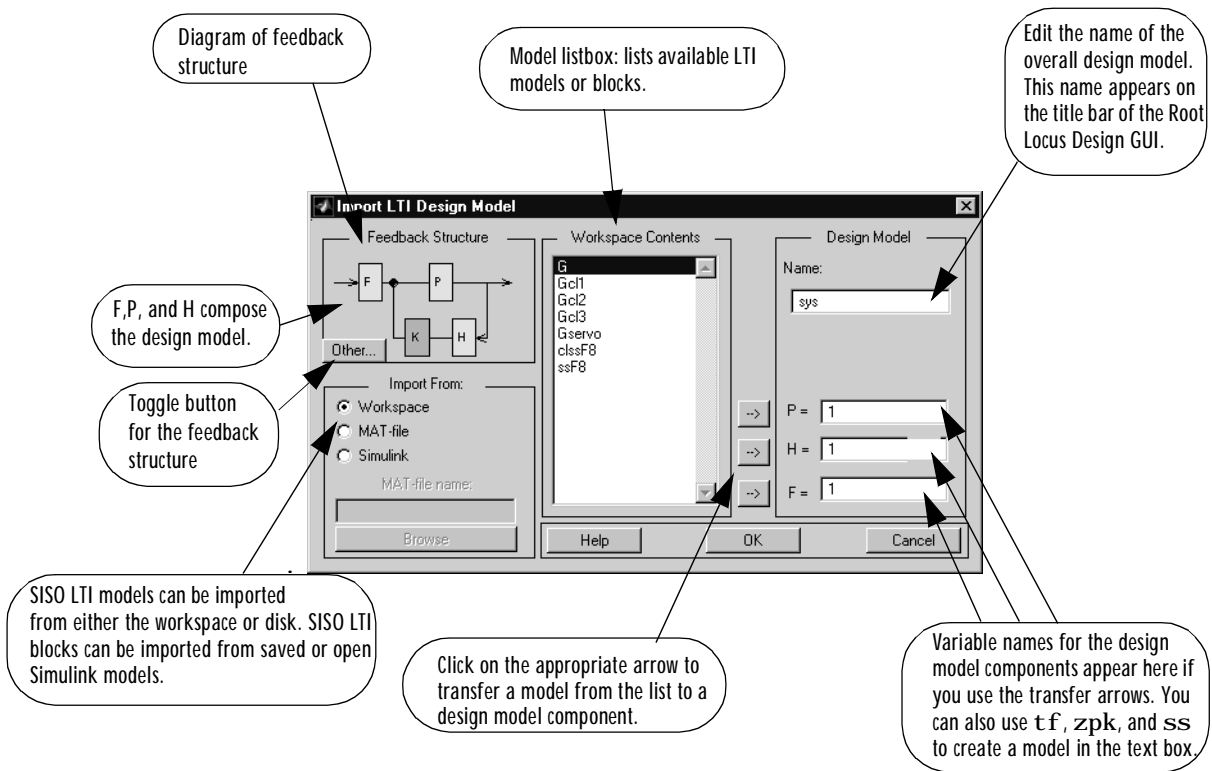
- Opening the **Import LTI Design Model** window
- Choosing a feedback structure
- Specifying the design model

Opening the Import LTI Design Model Window

To import the linearized electrohydraulic servomechanism model into the Root Locus Design GUI, first open the **Import LTI Design Model** window. To do this, select the **Import Model** menu item in the **File** menu.

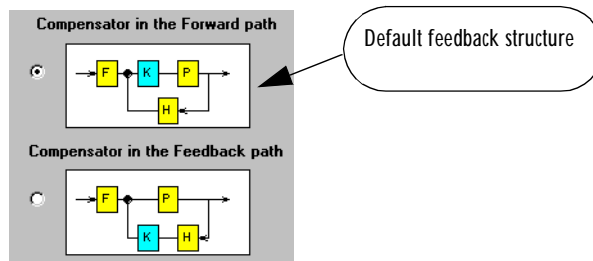


The **Import LTI Design Model** window that appears on your screen is as follows:



Choosing a Feedback Structure

The Root Locus design tool can be applied to SISO LTI systems whose feedback structure is in one of the following two configurations:



The **Feedback Structure** portion of the **Import LTI Design Model** window shows the current selection for the closed-loop structure. The **Other** button toggles the location of the compensator between the two configurations shown above. For this example you want the compensator in the forward path.

Specifying the Design Model

The SISO LTI models in either feedback configuration are coded as follows:

- **F** represents a pre-filter.
- **P** is the plant model.
- **H** is the sensor dynamics.
- **K** is the compensator to be designed.

In terms of the GUI design procedure, once they are set, **F**, **P**, and **H** are *fixed* in the feedback structure. This triple, along with the choice of feedback structure, is referred to throughout this chapter as the *design model*.

The default values for **F**, **P**, **H**, and **K** are all 1.

When you specify your design model, in addition to **F**, **P**, **H**, and the feedback structure, you can specify the design model name. To name the design model, click in the editable text box in the **Import LTI Design Model** window below **Name**, and enter the name you want for the design model. For this example, change the design model name to Gservo.

To specify **F**, **P**, and **H** for this example, we use the **Workspace** radio button, which is the default selection. A list of the LTI objects in your workspace appears in the model listbox under **Workspace Contents**.

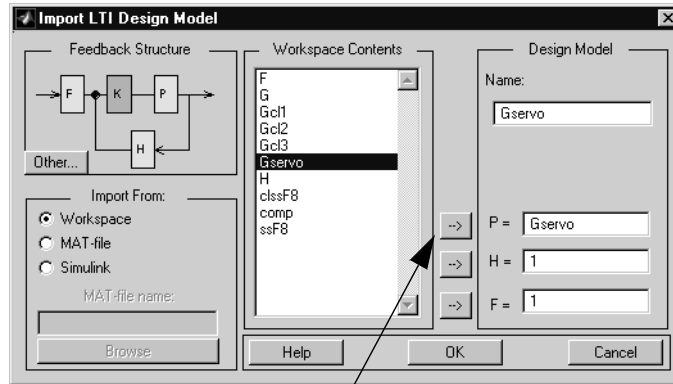
In general, to import design model components from the workspace to the GUI:

- 1 Select a given model in the **Workspace Contents** listbox to be loaded into either **F**, **P**, or **H**.
- 2 Click on the arrow buttons next to the design model component you want to specify.

To specify the design model components for this servomechanism example:

- 1 Load the linearized servomechanism model **Gservo** into the plant **P**, by first selecting it and then selecting the arrow button.

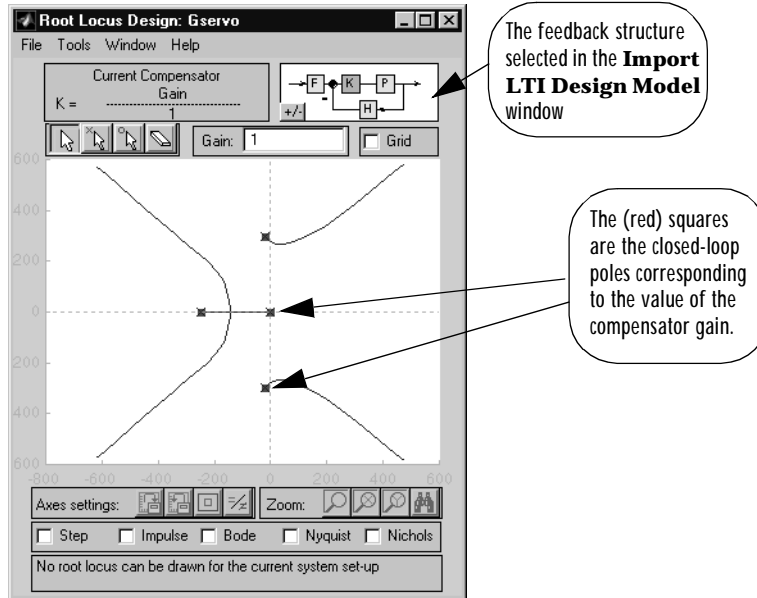
The **Import LTI Design Model** window looks like this after you specify both the plant, **P**, and the model name as **Gservo**.



Use the arrow buttons to transfer selected models from the listbox to the design model components, in this case, to the plant, **P**.

- 2 Press the **OK** button.

The root locus of the design model is displayed in the plot region of the GUI. Your Root Locus Design GUI looks like this:



Notice that the design model name appears in the title bar.

Changing the Gain Set Point and Zooming

The *gain set point* is the value of the gain you apply to the compensator to determine the closed-loop poles. This value appears in the **Gain** text box on the GUI.

In this section, we use some of the basic functions of the Root Locus Design GUI to analyze a root locus in the plot region. We cover how to change the gain set point, along with the closed-loop pole locations, and how to use the GUI zoom tools.

Let's begin by seeing how much gain you can apply to the compensator and still retain stability of the closed-loop system.

The red squares on each branch of the root locus mark the locations of the closed-loop poles associated with the gain set point. The system becomes

unstable if the gain set point is increased so as to place any of the closed-loop poles in the right-half of the complex plane.

You can test the limits of stability by either

- Dragging the closed-loop poles around the locus to change the gain set point
- Changing the gain set point manually

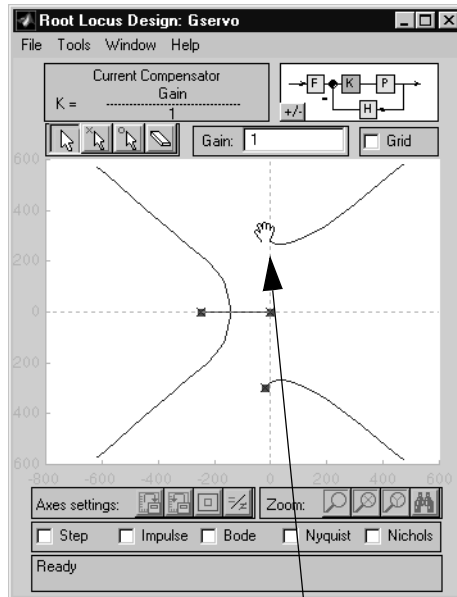
In this example we do the former.

Dragging Closed-loop Poles to Change the Gain Set Point

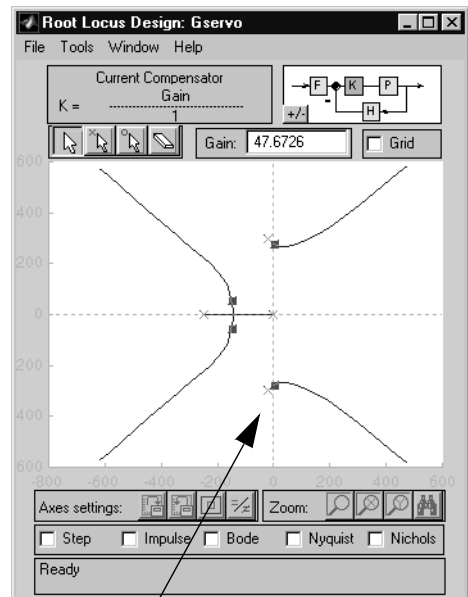
To see by how much the gain can be increased while maintaining the closed-loop poles in the left-half of the complex plane:

- 1 Move the mouse pointer over a red square marking one of the complex poles nearest the imaginary axis. Notice how the pointer becomes a hand.
- 2 Grab the closed-loop pole by holding down the left mouse button when the hand appears.
- 3 Drag the pole close to the imaginary axis.
- 4 Release the mouse button. The gain changes as the closed-loop set point is recomputed.

The following two figures capture this procedure.



As you get close to a closed-loop pole on the locus, the mouse pointer becomes a hand.



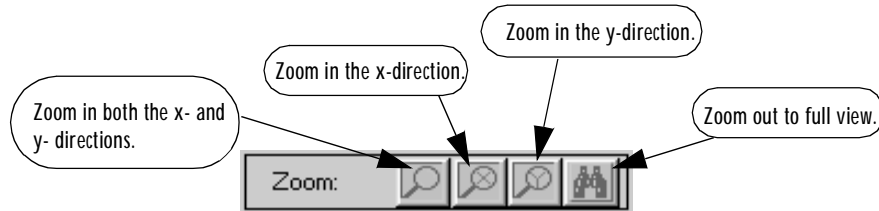
The poles appear to be on the imaginary axis.

The right-most pair of closed-loop poles seem to be on the imaginary axis. Actually, they are only close. Let's use the zoom controls to improve this result.

Zooming

You can use the **Zoom** controls on the lower right of the Root Locus Design GUI to zoom in on a region of the locus, or zoom out to show the entire locus.

The **Zoom** controls are shown below:



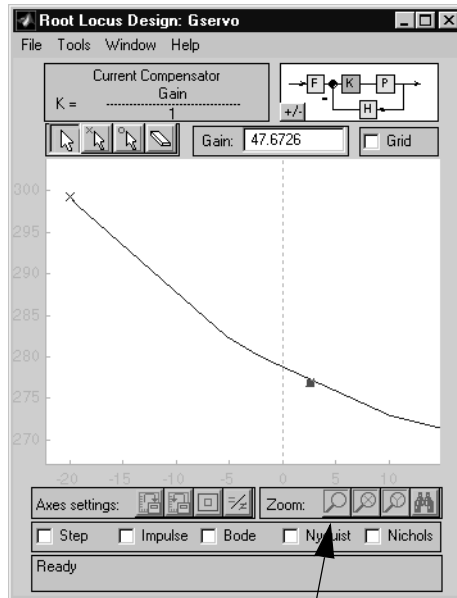
Once it is selected, you can operate any of the first three **Zoom** buttons in one of two ways:

- Use your mouse to *rubberband* around the area on the plot region you want to focus on. Rubberbanding involves clicking and holding the mouse down, while dragging the mouse pointer around the region of interest on the screen.
- Click in the plot region in the vicinity on which you want to focus.

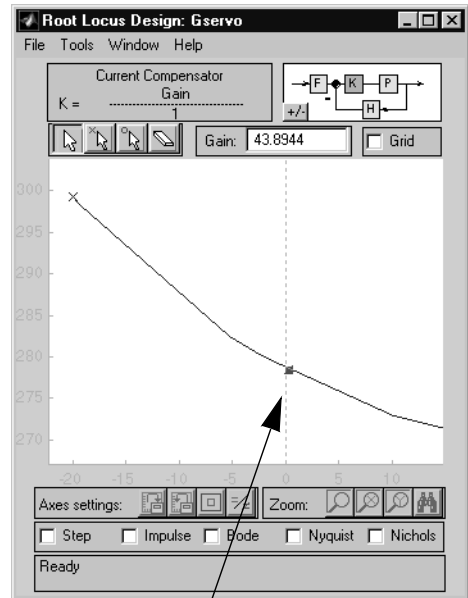
To move the closed-loop poles closer to the imaginary axis on your root locus by zooming with the rubberband method:

- 1 Zoom in X-Y by selecting the left-most **Zoom** button.
- 2 Use your mouse to rubberband the region of the imaginary axis near the closed-loop pole there.
- 3 Readjust the closed-loop pole position by grabbing it with the mouse and moving it until it rests on the imaginary axis.
- 4 Zoom out by clicking on the fourth **Zoom** button, (the icon is a pair of binoculars).

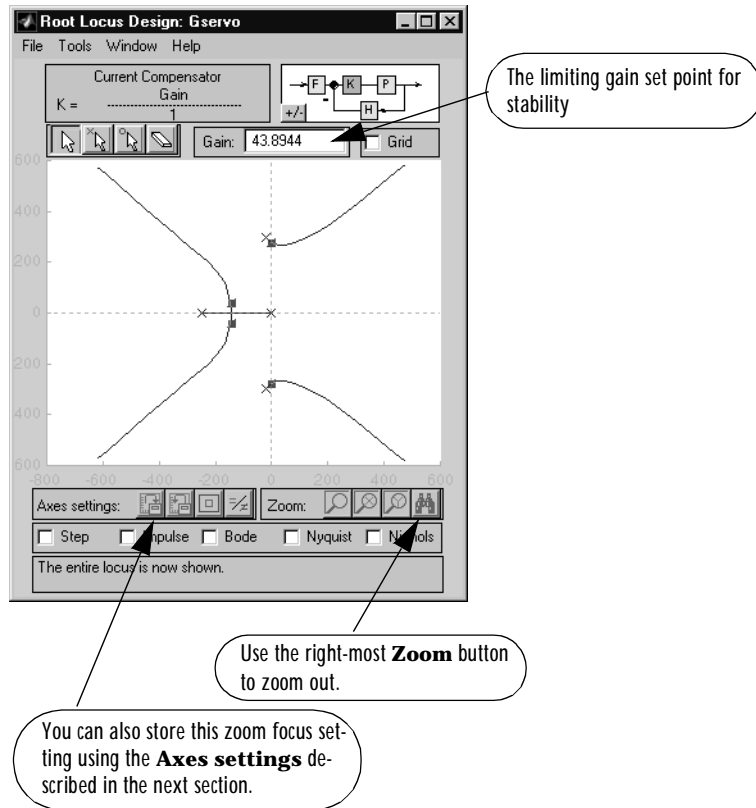
The use of zooming to find the limits of stability on the root locus is depicted in the following three figures:



After you drag that pole close to the imaginary axis, use the left-most **Zoom** button to zoom in.



Move the closed-loop pole to the imaginary axis to fine-tune your gain adjustment.



The critical gain value for stability is approximately 43.9.

Note: You can also test how much the gain can be increased while maintaining stability by arbitrarily applying different values of the gain in the text area next to **Gain** (pressing the **Enter** key after entering each value) until one pair of complex poles reaches the imaginary axis.

After using the zoom tools on the GUI, you may want to store one set of zoom focus settings so that you may later return to focus on that same region of the locus.

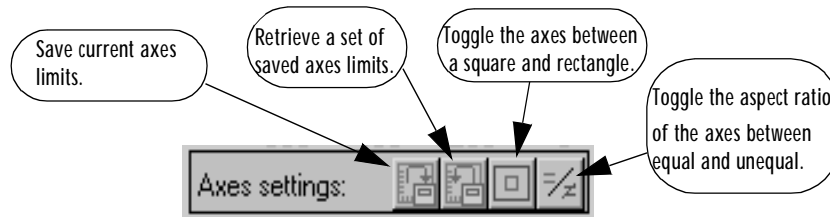
Storing and Retrieving Axes Limits

You can store and change axes limits in two ways:

- Using the **Axes settings** toolbar
- Using the **Set Axes Preferences** menu item from the **Tools** menu

For more information on square axes and/or equal axes, type `help axis` at the command line.

To use the **Axes settings** toolbar to save the current axes limits, select the left-most button shown below.

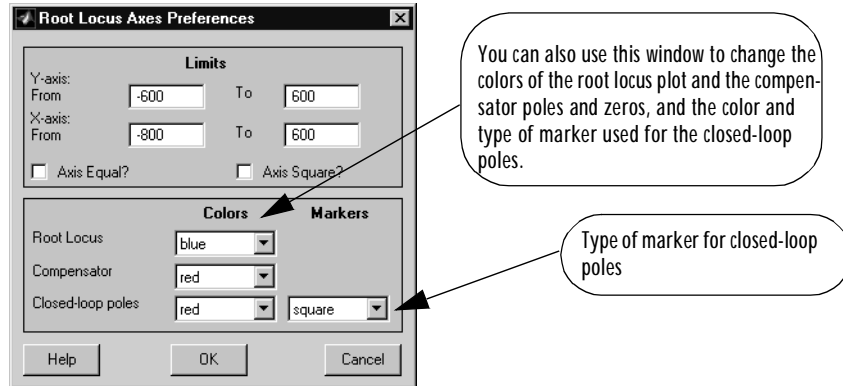


If you change the axes limits by zooming, you can always return to the saved axes limits by selecting the second **Axes settings** button.

To try out these tools:

- Select the left-most **Axes settings** button.
- Zoom in on a region of the root locus.
- Select the second **Axes settings** button.

You can also set or revise axes limits and other axes preferences in the **Root Locus Axes Preferences** window. To open this window, select **Set Axes Preferences** from the **Tools** menu.



If you have already stored axes limits using **Axes settings**, these limits appear in this window in the **Limits** field. You can reset these limits by typing in new values. To apply any changes to the entries in this window, click on **OK** after making the changes.

Displaying System Responses

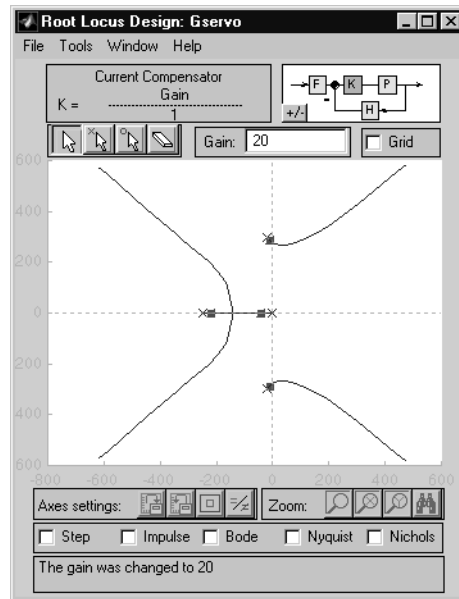
Before you design your compensator, you may want to conduct some response analysis of your closed-loop system evaluated at a fixed value of the gain.

You can access some of the system response analysis capabilities of the LTI Viewer (see Chapter 4) through the checkboxes located in the lower portion of the Root Locus Design GUI. Checking one or several of these boxes opens individual LTI Viewer windows that display the corresponding plots. The LTI Viewer windows that open are dynamically linked to the Root Locus Design GUI: any changes made to the system on the GUI that affect the gain set point will effect a corresponding change on the displayed LTI Viewer plots.

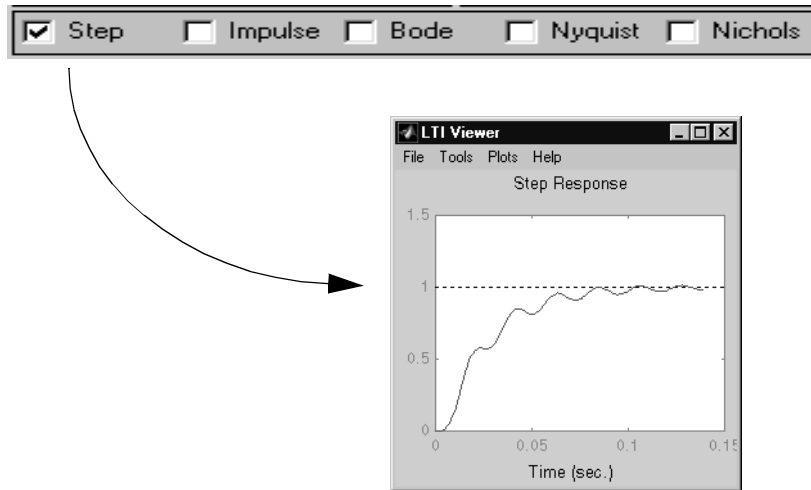
For this example, we want to know if the step response meets our design criteria. The current value for the gain set point is about 44. Clearly this value of the gain would test the limits of stability of our system. Let's choose a reasonable value for the gain (say, 20) and look at the step response.

Change the gain to 20 by editing the text box next to **Gain**, and pressing the **Enter** key. Notice that the locations of the closed-loop poles on the root locus are recalculated for the new gain set point.

Your Root Locus Design GUI looks like this now:



Check the **Step** box, as shown below.



Once you check the **Step** box, a small version of an LTI Viewer window opens. Clearly, this closed-loop response does not meet the desired settling time requirement (.05 seconds or less).

Note: The LTI Viewer window that you open from the Root Locus Design GUI is not fully operational as an LTI Viewer, but you can make it so. If you want to operate its features fully, you have to select the **Viewer Controls** menu item in the **Tools** menu on the LTI Viewer (see also “Activating the LTI Viewer” on page 6-35).

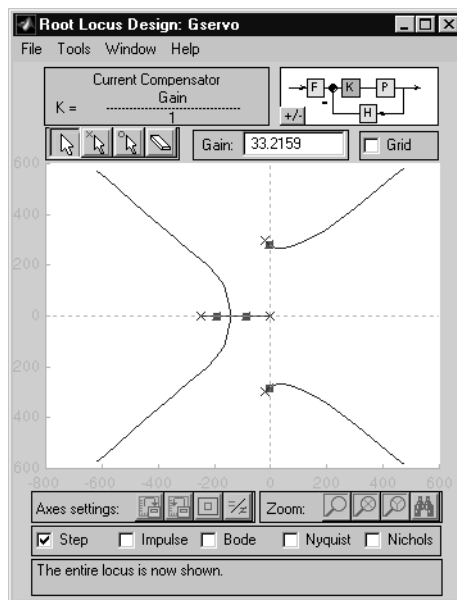
Caution: If you activate an LTI Viewer window, you lose the link between that LTI Viewer and the Root Locus Design GUI.

Designing the Compensator to Meet Specifications

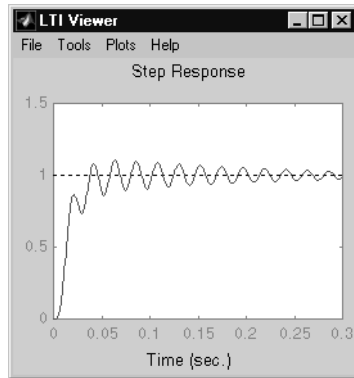
Since the current closed-loop system doesn't meet both of the required design specifications, let's first try increasing the gain. To aim to increase the gain to 33 or so:

- Hold your mouse button down as you click on any of the red squares.
- Drag the square in the direction of increasing gain, without allowing any of the closed-loop poles to enter the right half of the complex plane. If, when you start to move the red square, you see the gain value in the **Gain** box decreasing, drag it in the opposite direction.
- Release the mouse button when the gain is near 33.

Your GUI looks like this:.



The step response plot on the dynamically linked LTI Viewer automatically updates when you release the mouse button:



As you may have noticed, the response time decreases with increasing gain, while the overshoot increases. Here we no longer meet the overshoot requirement. Since this gain is already relatively large, it's likely that we will not be able to meet both design requirements using only a gain for the compensator. This conjecture is supported when you specify the design region boundaries on the root locus for these design requirements. We do this in the next subsection.

Specifying Design Region Boundaries on the Root Locus

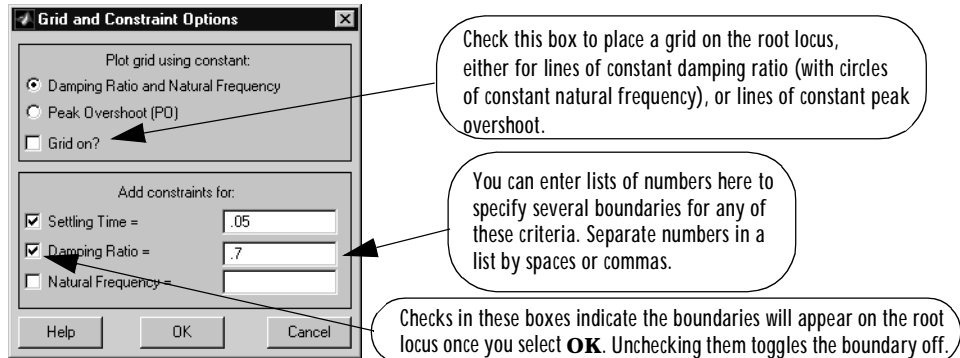
If, as in our example, your design criteria are specified in terms of step response characteristics, you may want to use the grid and boundary constraint options that are accessible from the **Add Grid/Boundary** menu item in the **Tools** menu on the Root Locus Design GUI. These options allow you to use second order system design criteria to inscribe the boundaries of design region directly on the root locus plot, or apply a grid to the plot.

Note: The boundaries you apply to the Root Locus Design GUI based on LTI system design criteria (settling time, damping ratio, natural frequency, and peak overshoot) are computed relative to second-order systems only. Therefore, for higher order systems, these boundaries provide approximations to the design region.

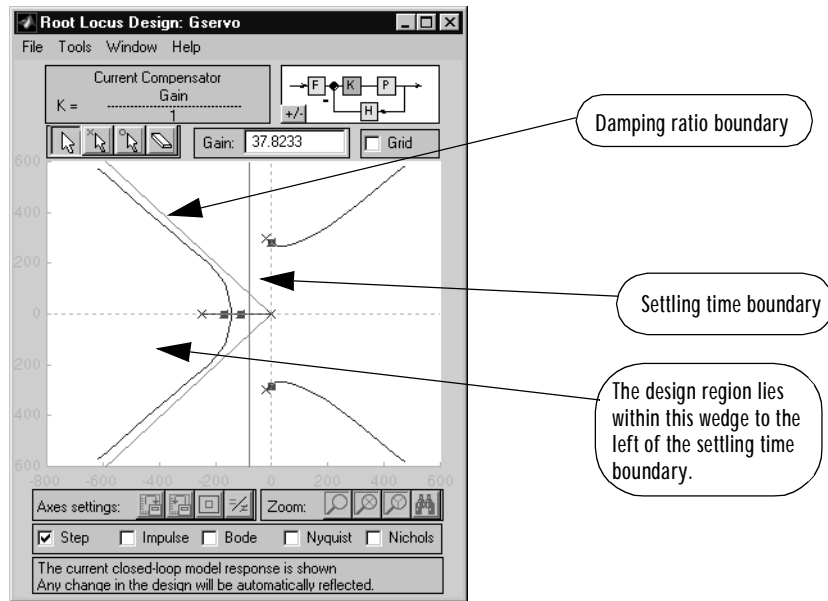
Let's place approximate design region boundaries on our root locus plot based on our design specifications. To do so, select the **Add Grid/Boundary** menu item in the **Tools** menu.

Our design specifications require that the (2 percent) settling time be less than .05 seconds, and the maximum overshoot less than 5 percent. For second-order systems, the overshoot requirement can be translated directly as a requirement on the damping ratio of about .7 (see [2]).

After you enter these values in the appropriate text fields for our specifications, your **Grid and Constraint Options** window looks like this:



After you press **OK**, the Root Locus Design GUI calculates and displays the specified boundaries:



Clearly, not all four branches of the root locus are within the design region. Therefore, we can try to add poles and zeros to our compensator and see if we can meet the design criteria.

Placing Compensator Poles and Zeros

There are three types of parameters specifying the compensator:

- Poles
- Zeros
- Gain

Once you have the gain specified, you can add poles or zeros to the compensator. You can add poles and zeros to the compensator (or remove them) in two ways:

- Use buttons on the root locus toolbar section of the GUI for pole/zero placement.
- Use the **Edit Compensator** menu item on the **Tools** menu.

The root locus toolbar is convenient for *graphically* placing compensator poles and zeros so that you meet the design specifications for a given gain set point. This method may require a certain amount of trial and error.

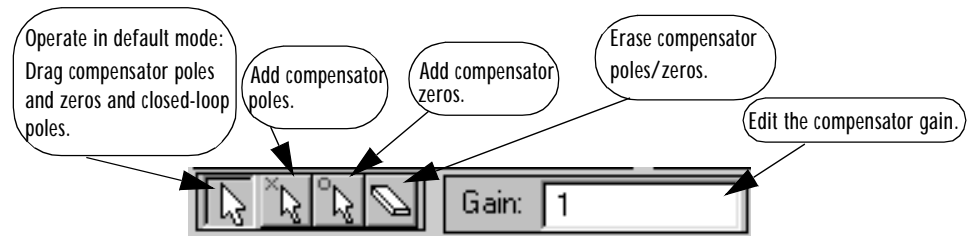
You can use the **Edit Compensator** menu item to:

- Fine-tune compensator parameter values for design implementation.
- Revise or implement an existing compensator design.

For this example we first use the root locus toolbar to place compensator poles and zeros on the root locus, and then use the **Edit Compensator** menu item to set the compensator pole and zero locations for a specific compensator solution.

Placing Compensator Poles and Zeros Using the Root Locus Toolbar

The root locus toolbar is located on the left side of the GUI, above the plot region. The figure below describes the root locus toolbar buttons:



The default mode for the toolbar is the *drag* mode. In this mode, you can:

- Click on a specific location on the root locus to place a closed-loop pole there (and consequently reassign the gain set point).
- Drag any of the closed-loop poles along its branch of the root locus (also reassigning the gain set point).
- Drag any of the compensator poles or zeros around the complex plane to change the compensator.

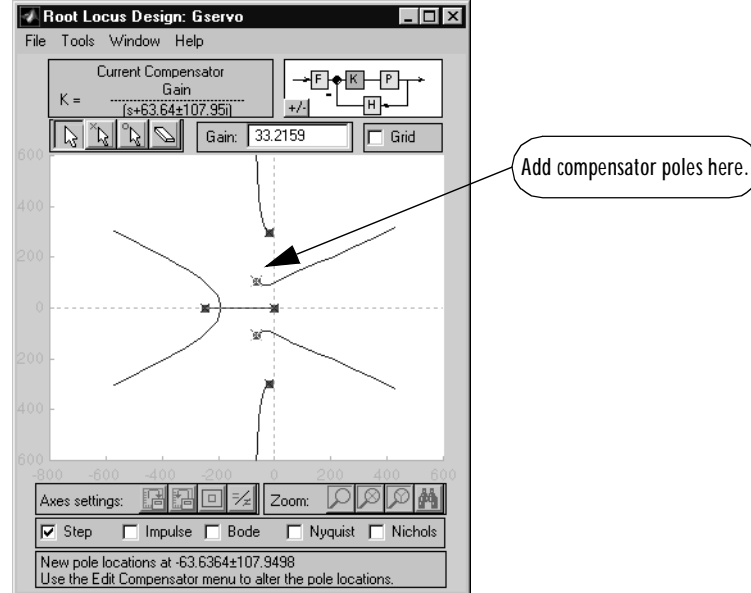
To add a complex conjugate compensator pole pair on the root locus plot:

- 1 Select the *add pole* button (the second button in the root locus toolbar).
- 2 Click on the plot region where you would like to add one of the complex poles.

Some of the features of using the root locus toolbar to add a pole are:

- While the *add pole* button is depressed, the cursor changes to an arrow with an **x** at its tip whenever it is over the plot region. This indicates the toolbar is in the “add pole” mode.
- After you add the poles, the *add pole* button pops back up and the default *drag* mode is restored. The added compensator poles appear in a different color. The default color is red.
- The LTI Viewer response plots change as soon as the pole is added.
- The text displayed in the **Current Compensator** region of the GUI now displays the new pair of poles.

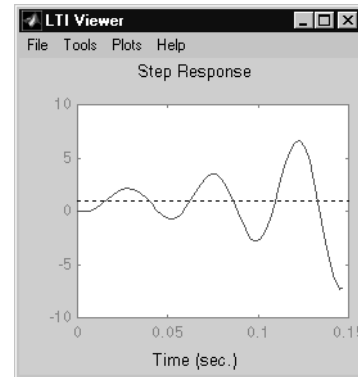
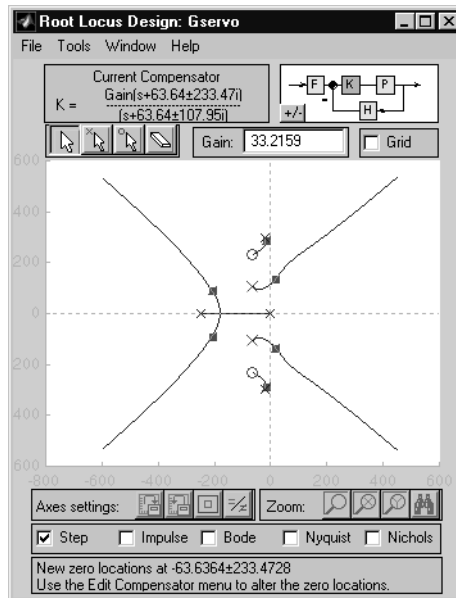
Try placing a pair of complex poles just above the right-most real closed-loop pole. The resulting root locus plot looks like this:



Similarly, to add a pair of complex zeros to the compensator:

- 1 Select the *add zero* button from the root locus toolbar (third button from the left).
- 2 Click on the plot region where you would like to add one of the complex zeros.

Try adding a pair of complex zeros just to the left of and a little bit below the complex closed-loop poles closest to the imaginary axis. The resulting root locus and step response plots are as follows:



Notice that the step response is unstable. Change the gain to 9.7. The resulting step response is stable, but still doesn't meet the design criteria:



As you can see, the compensator design process can involve some trial and error. You can try dragging the compensator poles, compensator zeros, or the closed-loop poles around the root locus until you meet the design criteria. Alternatively, you can edit the compensator using the solution we provide in the next section.

Note: You can follow the same procedure to add a single real pole or zero to the compensator by clicking on the real axis.

Editing Compensator Pole and Zero Locations

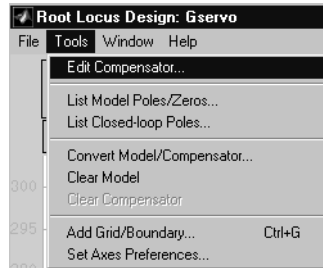
In this section we use the **Edit Compensator** window to design a compensator with the following characteristics:

- Gain: 9.7
- Poles: $-110 \pm 140i$
- Zeros: $-70 \pm 270i$

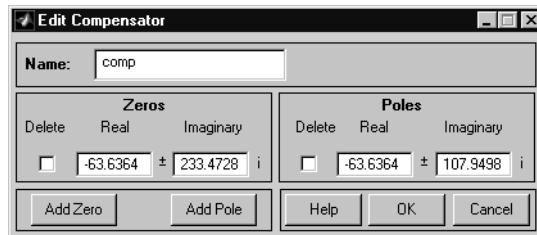
You can access the **Edit Compensator** window in one of three ways:

- Double-click on any of the **Current Compensator** text.
- Click on the (blue) compensator block in the feedback structure on the GUI.
- Select **Edit Compensator** from the **Tools** menu.

This figure shows how to choose this menu item on the GUI:



With either method, you open the **Edit Compensator** window shown in the figure below.

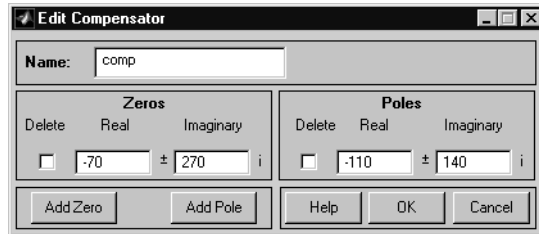


You can use the **Edit Compensator** window to:

- Edit the locations of compensator poles and zeros.
- Add compensator poles and zeros.
- Delete compensator poles and zeros.
- Change the name of the compensator (This name is used when exporting the compensator).

For this example, edit the poles and zeros to be at $-110 \pm 140i$, and $-70 \pm 270i$, respectively.

Your GUI looks like this:



If, in addition, you want to add poles to the compensator:

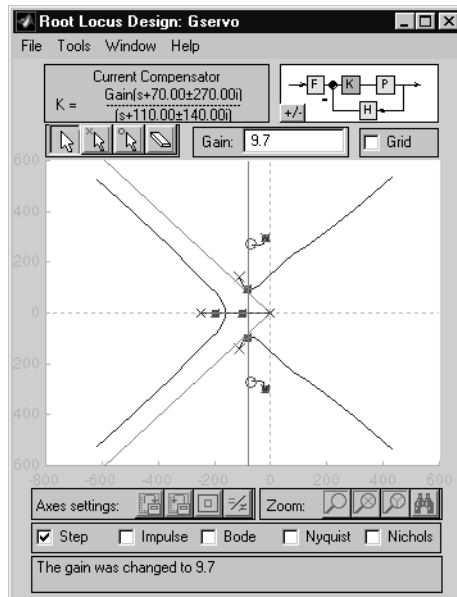
- 1 Click on **Add Pole**. A new editable text field appears.
- 2 Enter the new pole location in the text field.

The procedure is the same for adding zeros, only use the **Add Zero** button in place of the **Add Pole** button. To erase any poles or zeros, select the **Delete** checkbox.

Before closing this window by pressing **OK**, you can also optionally change the name of the compensator.

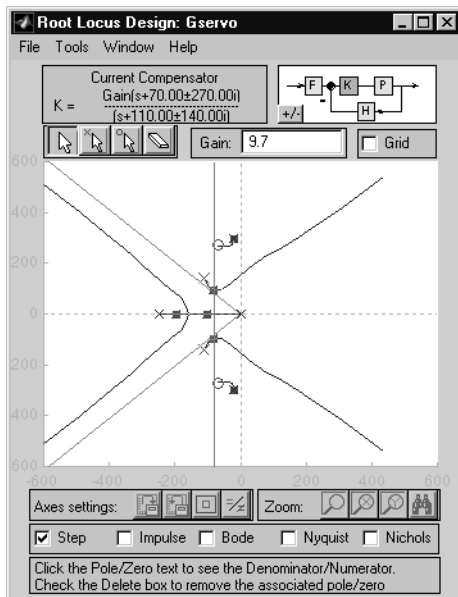
You can also erase poles or zeros on the root locus using the erase button on the root locus toolbar. See “Erasing Compensator Poles and Zeros” on page 6-41

With the gain set point at 9.7, your root locus looks as follows:

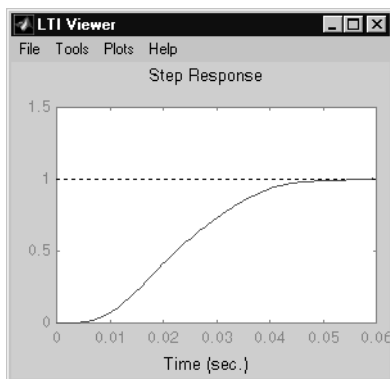


Note: Whenever the numerator or denominator are too long for the **Current Compensator** field, a default **numK** or **denK** is displayed. To display the actual numerator or denominator, resize the Root Locus Design GUI.

To check where the closed-loop roots are with respect to the boundary of the design region, let's zoom in a bit:



We don't have all of the closed-loop poles in the design region, but let's see if we meet the step response specifications. The updated LTI Viewer plot of the step response looks like this:



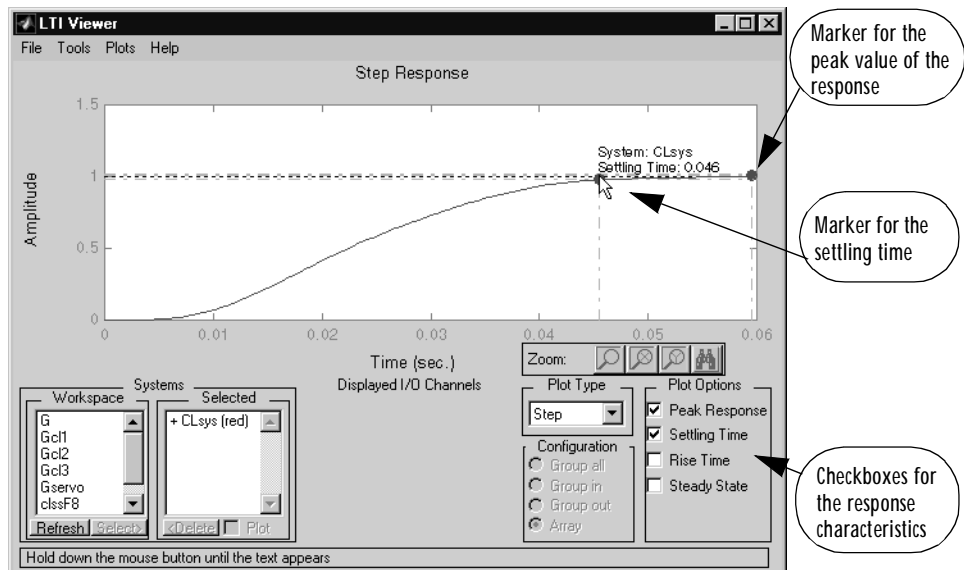
The step response looks good. However, to be certain that you've met your design specifications, you can activate the full LTI Viewer at this point.

Activating the LTI Viewer

To activate the LTI Viewer window and check the response specifications:

- 1 Click on the **Viewer Controls** menu item in the **Tools** menu on the LTI Viewer.
- 2 Check **Settling Time** and **Peak Response**.
- 3 Set your mouse pointer to the settling time marker and click to display the value.
- 4 Set your mouse pointer to the peak response marker and click to display the value.

The LTI Viewer will look like this:



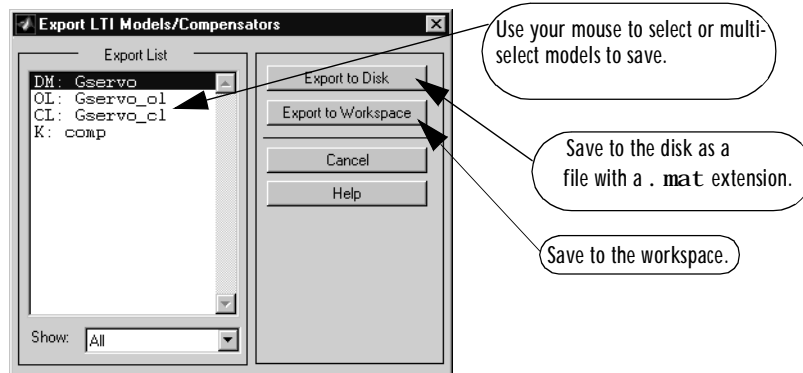
As you can see, the settling time is less than .05 seconds, and the overshoot is less than 5 percent. You've met the design specifications and you're done. You didn't need your closed-loop poles to be entirely in the design region, because

you actually have a sixth-order system with some fast dynamics, as opposed to having only a second-order system.

Note: If, after activating this LTI Viewer and breaking the link with the Root Locus Design GUI, you want to continue to view a step response plot that is linked to the Root Locus Design GUI, just check the **Step** box on the Root Locus Design GUI once more.

Saving the Compensator and Models

Now that you have successfully designed your compensator, you may want to save your design parameters for future implementation. You can do this by selecting **Export** from the **File** menu on the Root Locus Design GUI. The window shown below opens.



The variable listed in the **Export List** on the **Export LTI Models/Compensators** GUI are either previously named by you (on the **List Model Poles/Zeros** or the **Edit Compensator** windows) or have default names. They are coded as follows:

- DM: The design model
- OL: The open-loop model
- CL: The closed-loop model
- K: The compensator

To export your compensator to the workspace:

- 1 Select the compensator in the **Export List**.
- 2 Click on the **Export to Workspace** button.

The **Export LTI Model/Compensators** window is closed when you click on one of the export buttons. If you go to the MATLAB prompt and type

```
who
```

the compensator is now in the workspace, in the variable named `comp`.

Then type

```
comp
```

to see that this variable is stored in `zpk` format.

Additional Root Locus Design GUI Features

This section describes several features of the Root Locus Design GUI not covered in the servomechanism example. These are listed as follows:

- Specifying design models: general concepts
 - Creating models manually within the GUI
 - Designating the model source
- Getting help with the Root Locus Design GUI
- Erasing compensator poles and zeros
- Listing poles and zeros
- Printing the root locus
- Drawing a Simulink diagram of the closed-loop model
- Converting between continuous and discrete models
- Clearing data

Specifying Design Models: General Concepts

In this section we provide general concepts for specifying the design model by using one or both of the following methods:

- Creating models manually within the GUI
- Designating the model source
 - The MATLAB workspace
 - A MAT-file
 - An open or saved Simulink model

Creating Models Manually Within the GUI

You can create models for root locus analysis *manually* in the **Import LTI Design Model** window using `tf`, `ss`, or `zpk`. To do so, just edit the text boxes next to **P**, **F**, or **H** using any of these MATLAB expressions. You can also use a scalar number to specify **P**, **F**, or **H**.

Designating the Model Source

The source of your design model data is indicated in the **Import From** field shown below.



If you are loading your models from MAT-files or Simulink models saved on your disk, you are prompted to enter the name of the file in the editable text box below the **Import From** radio buttons. You can also select the **Browse** button to search for and select the file. The **Simulink** radio button also allows you to load SISO LTI blocks from an open Simulink model, by entering the name of the model in the text box.

When you select the model source, its contents are shown in the model listbox located in the central portion of the **Import LTI Design Model** window. The model listbox contains the following data:

- For **Workspace** or **MAT-file**: All LTI models in the workspace, or in the selected MAT-file
- For **Simulink**: All the LTI blocks in the selected Simulink diagram

Note: If you want to load models saved in more than one MAT-file, load these into the MATLAB workspace *before* selecting **Import Model** on the Root Locus Design GUI.

Getting Help with the Root Locus Design GUI

You can obtain instructions on how to use the Root Locus Design GUI either using the **Help** menu, from the status bar, or by using the tooltips.

Using the Help Menu

Click on the **Help** menu and you find that it contains five menu items:

- **Main Help**
- **Edit Compensator**
- **Convert Model**
- **Add Grid/Boundary**
- **Set Axis Preferences**

The first menu item, **Main Help**, opens a help window that describes how to use the controls located on the Root Locus Design GUI. The remaining menu items provide additional information on the features you can access from the **Tools** menu.

Using the Status Bar for Help

The status bar at the bottom of the Root Locus Design GUI:

- Provides you with information, hints, and error messages as you proceed through your design.
- Lets you know if you have tried to undertake an action the GUI is not capable of, or if a GUI operation you have performed has successfully been completed.
- Provides information about the location of newly placed compensator poles and zeros, as well as the damping ratio, natural frequency, and location of poles and zeros as you drag them.

Tooltips

You can obtain simple reminders (tooltips) on how to use the Root Locus Design GUI by moving your mouse and putting the cursor over one of these features. For example, if you put the cursor over the **Step** checkbox, its tooltip, **Closed-loop step response**, appears just below the button. When a tooltip is available, a small bubble containing information about the feature you selected appears. This bubble disappears when you move the mouse again.

Erasing Compensator Poles and Zeros

You can delete any compensator poles and zeros in one of two ways:

- Check the associated **Delete** box on the **Edit Compensator...** window obtained from the **Tools** menu
- Click on the erase button (fourth from left on the root locus toolbar) and click on the pole or zero to delete

After using the root locus toolbar erase button to delete a pole or zero, the following occurs:

- The erase button pops up and the default *drag* mode is restored.
- The zero or pole you erased (and, if applicable, its conjugate) is removed from the root locus and from the **Current Compensator** text.
- The root locus plot and any linked LTI Viewer response plots are recalculated.

Listing Poles and Zeros

At any point when the Root Locus Design GUI is open, you can view:

- The closed-loop pole locations associated with the current **Gain** setting
- The poles, zeros, and gain of each design model component

To view the current closed-loop poles:

- Select **List Closed-Loop Poles** from the **Tools** menu. If you select this menu, a window listing the closed-loop poles associated with the current gain set point opens.
- Click on the **OK** button to close the window.

Note: You cannot edit the closed-loop pole locations listed in this window.

To view the design model poles and zeros, you can either:

- Select the **List Model Poles/Zeros** item from the **Tools** menu
- Click on any of the design model blocks in the **Feedback Structure**. These are the yellow **F**, **P**, and **H** blocks

The **List Model Poles/Zeros** window opens. For each component of the design model, this window tells you:

- The component name
- The type of LTI model that the component is (that is, ss, zpk, or tf)
- The component's pole and zero locations
- The associated numerator and denominator of the model transfer function

Of these four features, only the name of each component of the design model is editable. To edit a model name:

- 1 Click in the appropriate editable text box.
- 2 Type in the new name.
- 3 Press the **OK** button.

Note: The names you enter in this window are only used when you generate a Simulink diagram of the closed-loop structure.

To list the denominator, (respectively, the numerator, including the gain) associated with a given component of the design model, in the **List Model Poles/Zeros** window, click on **Poles**, (respectively, **Zeros**) of that component. When you do, a window providing the selected information opens.

Printing the Root Locus

You can print the locus the GUI is displaying by:

- 1 Select **Print Locus** from the **File** menu.
- 2 Press the **OK** button on the printer dialog that appears.

When you select **Print Locus** a new MATLAB figure window containing the locus appears. This is the figure that is printed when you press **OK**.

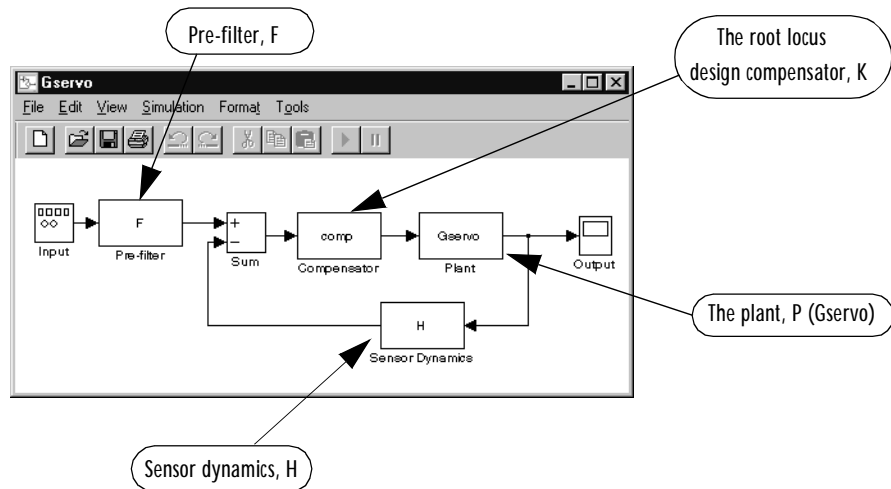
Note: You can also generate this root locus MATLAB figure window by selecting **Send Locus to Figure** from the **File** menu. This figure remains open until you close it. You can use the Scribe tools on it, or manipulate it in other ways.

Drawing a Simulink Diagram

If you have Simulink, you can use the Root Locus Design GUI to automatically draw a Simulink diagram of the closed-loop structure. To do this:

- 1 Select **Draw Simulink Diagram** from the **File** menu.
- 2 Answer **Yes** to the subsequent dialog box question to confirm that you want the design model data to be stored in the workspace using its current names, or answer **No** to this question, if you want to abort drawing the diagram.

If you answered **Yes**, a Simulink diagram such as this appears.



The Simulink diagram is linked to the workspace, not to the Root Locus Design GUI. If you change your compensator design in the Root Locus Design GUI, you

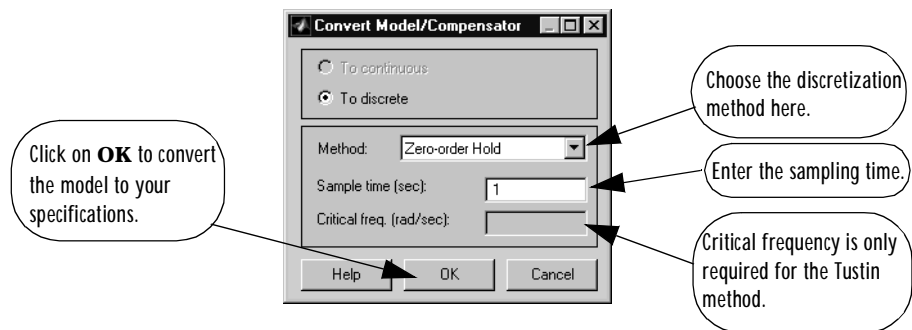
must export it to the workspace in order to reflect these changes in the Simulink diagram.

Converting Between Continuous and Discrete Models

The Root Locus Design GUI can be applied to either discrete or continuous-time systems, and you can convert between the two types of systems at any time during the design process. However, the design model and compensator must both be either continuous or discrete.

To obtain the discrete equivalent of your current compensator design, select **Convert Model/Compensator** from the **Tools** menu.

The **Convert Model/Compensator** window, shown below, opens.



Once you select a discretization method, sampling time, and critical frequency (if required), click on the **OK** button. The **Convert Model/Compensator** window closes and the discretization is performed. At this point:

- The design model and compensator are discretized.
- The **Current Compensator** text displays a system using the complex variable 'z'.
- The root locus for the discrete-time system representing the converted continuous-time system is plotted.
- Any linked LTI Viewer response plots are updated.
- The grids and boundaries are converted into the discrete plane.

You can also use the **Convert Model/Compensator** GUI to retrieve the continuous system, or to re-sample the discrete system using a different sampling time. These options are selected from the radio buttons at the top of this GUI.

Clearing Data

You can clear the design model and/or compensator from the Root Locus Design GUI using options available in the **Tools** menu:

- **Clear Model:** removes the plant, pre-filter, and sensor dynamics and replaces them all with unity gains.
- **Clear Compensator:** removes the compensator poles and zeros, and resets the gain to one.

To test these features, select **Clear Model**. Notice that your compensator is not altered, and its poles and zeros remain plotted in the root locus plot region of the GUI.

Now, select **Clear Compensator**. The Root Locus Design GUI returns to the state it was in when you first opened it and you are ready to start a new design.

References

- [1] Clark, R.N., *Control System Dynamics*, Cambridge University Press, 1996.
- [2] Dorf, R.C. and R.H. Bishop, *Modern Control Systems*, 8th Edition, Addison Wesley, 1997.

Design Case Studies

Yaw Damper for a Jet Transport	7-3
Open-Loop Analysis	7-6
Root Locus Design	7-9
Washout Filter Design	7-14
 Hard-Disk Read/Write Head Controller	 7-19
 LQG Regulation	 7-30
Process and Disturbance Models	7-30
LQG Design for the x-Axis	7-33
LQG Design for the y-Axis	7-41
Cross-Coupling Between Axes	7-42
MIMO LQG Design	7-46
 Kalman Filtering	 7-49
Discrete Kalman Filter	7-49
Steady-State Design	7-50
Time-Varying Kalman Filter	7-56
Time-Varying Design	7-57
References	7-62

This chapter contains four detailed case studies of control system design and analysis using the Control System Toolbox:

- | | |
|--------------|---|
| Case Study 1 | A yaw damper for a jet transport aircraft that illustrates the classical design process. |
| Case Study 2 | A hard-disk read/write head controller that illustrates classical digital controller design. |
| Case Study 3 | LQG regulation of the beam thickness in a steel rolling mill. |
| Case Study 4 | 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`, `disksdemo.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 Jet Transport

This case study demonstrates the tools for classical control design by stepping through the design of a yaw damper for a jet transport aircraft.

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

$$A = \begin{bmatrix} -0.0558 & -0.9968 & 0.0802 & 0.0415 \\ 0.5980 & -0.1150 & -0.0318 & 0 \\ -3.0500 & 0.3880 & -0.4650 & 0 \\ 0 & 0.0805 & 1.0000 & 0 \end{bmatrix};$$

$$B = \begin{bmatrix} 0.0729 & 0.0001 \\ -4.7500 & 1.2300 \\ 1.5300 & 10.6300 \\ 0 & 0 \end{bmatrix};$$

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

$$D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix};$$

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'};
output = {'yaw rate' '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	-0.05580	-0.99680	0.08020	0.04150
yaw	0.59800	-0.11500	-0.03180	0
roll	-3.05000	0.38800	-0.46500	0
phi	0	0.08050	1.00000	0

```
b =
```

	rudder	aileron
beta	0.07290	0.00010
yaw	-4.75000	1.23000
roll	1.53000	10.63000
phi	0	0

```
c =
```

	beta	yaw	roll	phi
yaw rate	0	1.00000	0	0
bank angle	0	0	0	1.00000

```
d =
```

	rudder	aileron
yaw rate	0	0
bank angle	0	0

Continuous-time system.

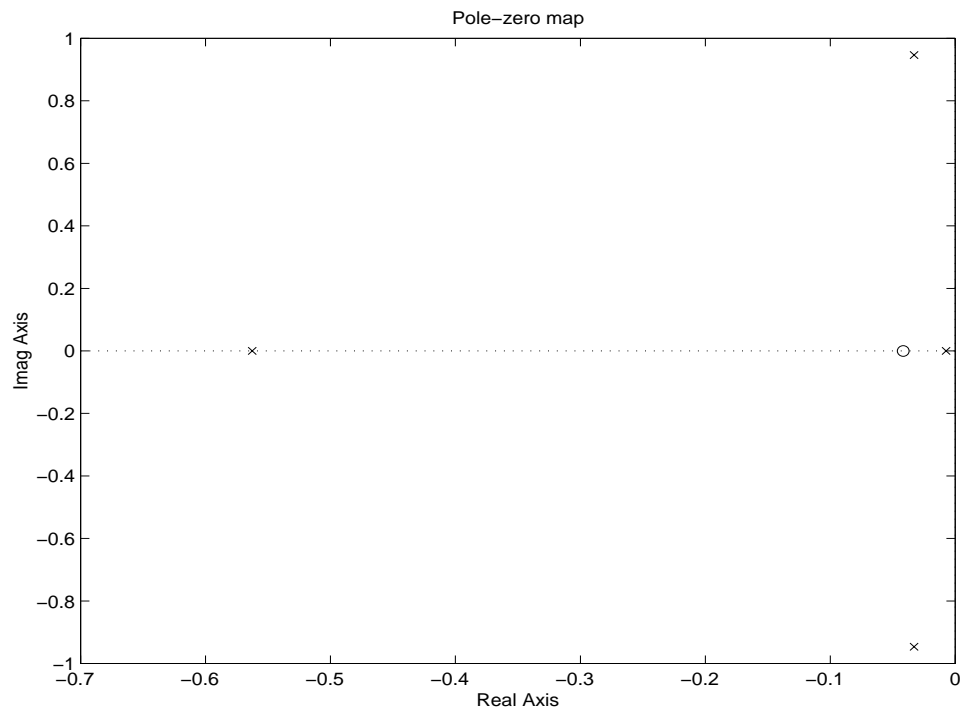
The model has two inputs and two outputs. The units are radians for `beta` (sideslip angle) and `phi` (bank angle) and radians/sec for `yaw` (yaw rate) and `roll` (roll rate). The rudder and aileron deflections are in degrees.

Compute the open-loop eigenvalues and plot them in the s -plane:

```
damp(sys)
```

Ei genval ue	Dampi ng	Freq. (rad/s)
-7.28e-03	1.00e+00	7.28e-03
-5.63e-01	1.00e+00	5.63e-01
-3.29e-02 + 9.47e-01i	3.48e-02	9.47e-01
-3.29e-02 - 9.47e-01i	3.48e-02	9.47e-01

```
pzmap(sys)
```



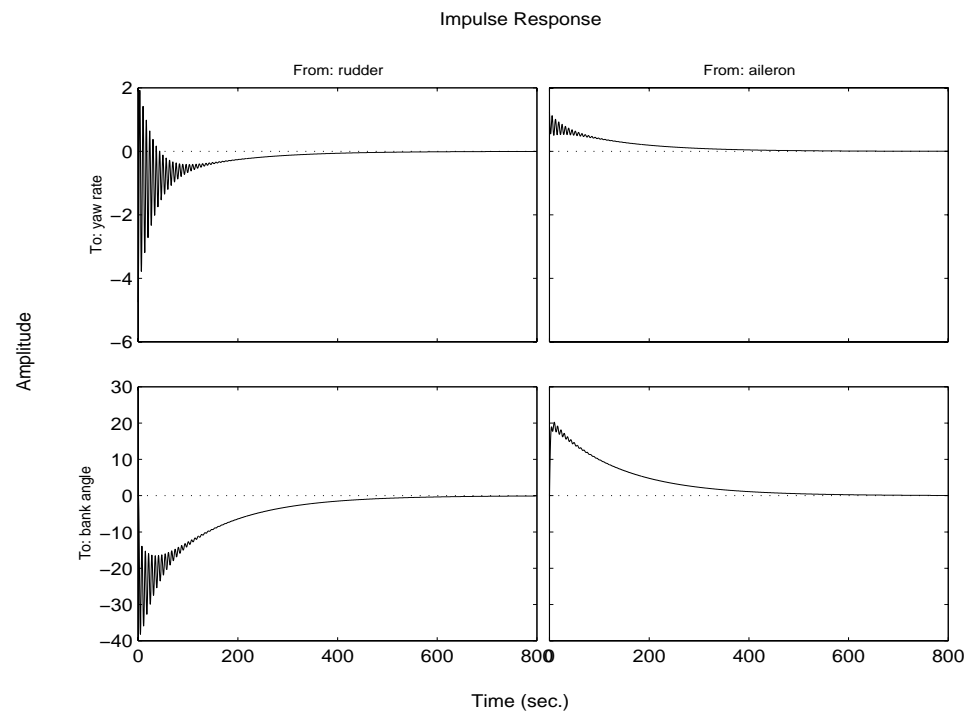
This model has one pair of lightly damped poles. They correspond to what is called the “Dutch roll mode.” You need to design a compensator that increases the damping of these poles.

The design specification is to provide a damping ratio $\zeta > 0.35$ with natural frequency $\omega_n < 1$ rad/sec. Next we show how you can design this compensator using classical methods.

Open-Loop Analysis

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

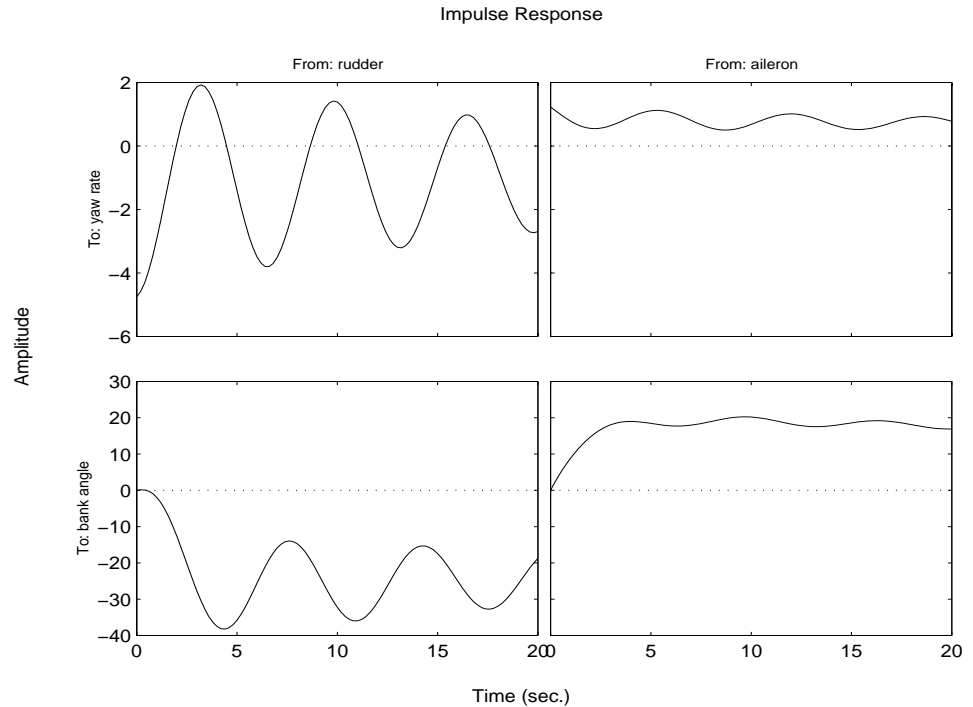
```
impz(sys)
```



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:

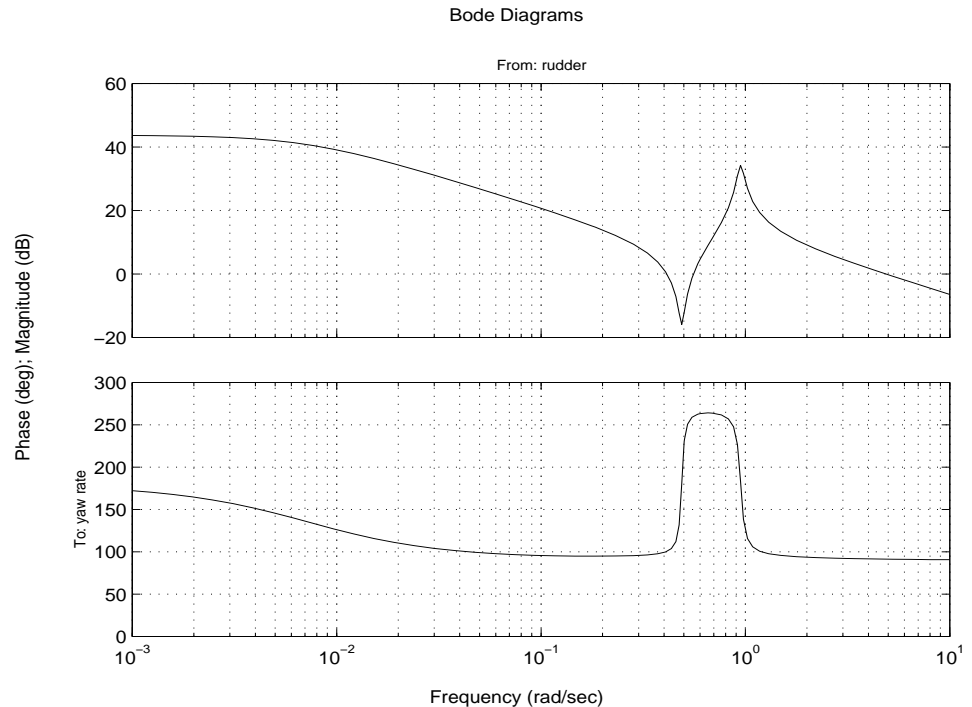
```
i mpul se(sys, 20)
```



Look at the plot from aileron (input 2) to bank angle (output 2). 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 (input 1 to output 1):

```
bode(sys(1, 1))
```



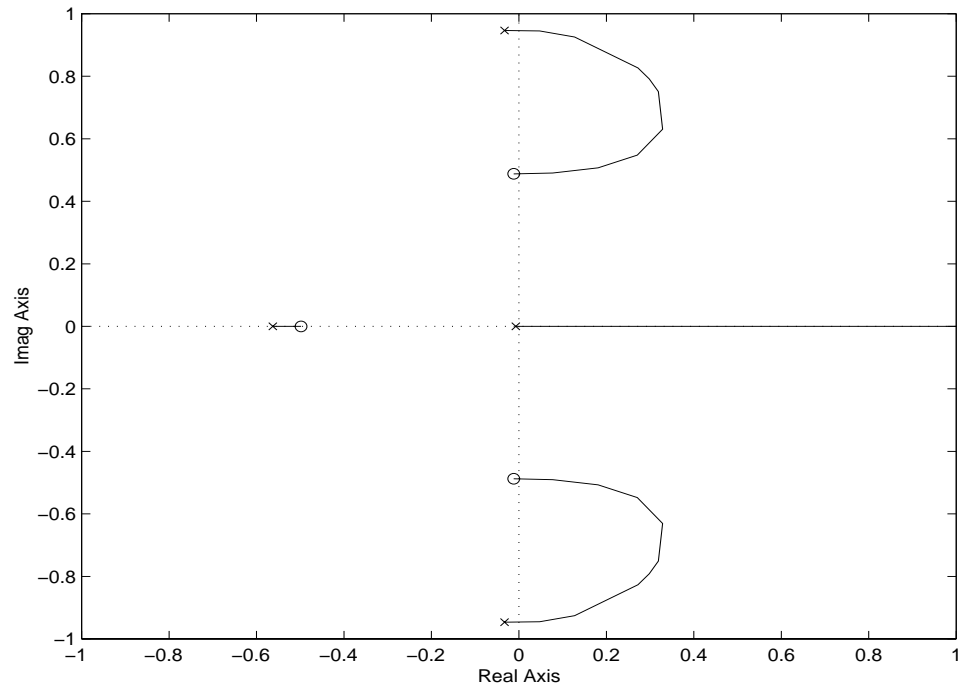
From this Bode diagram, you can see that the rudder has significant effect around the lightly damped Dutch roll mode (that is, near $\omega = 1$ rad/sec). To make the design easier, select the subsystem from rudder to yaw rate:

```
% Select system with input 1 and output 1
sys11 = sys(1, 1)
```

Root Locus Design

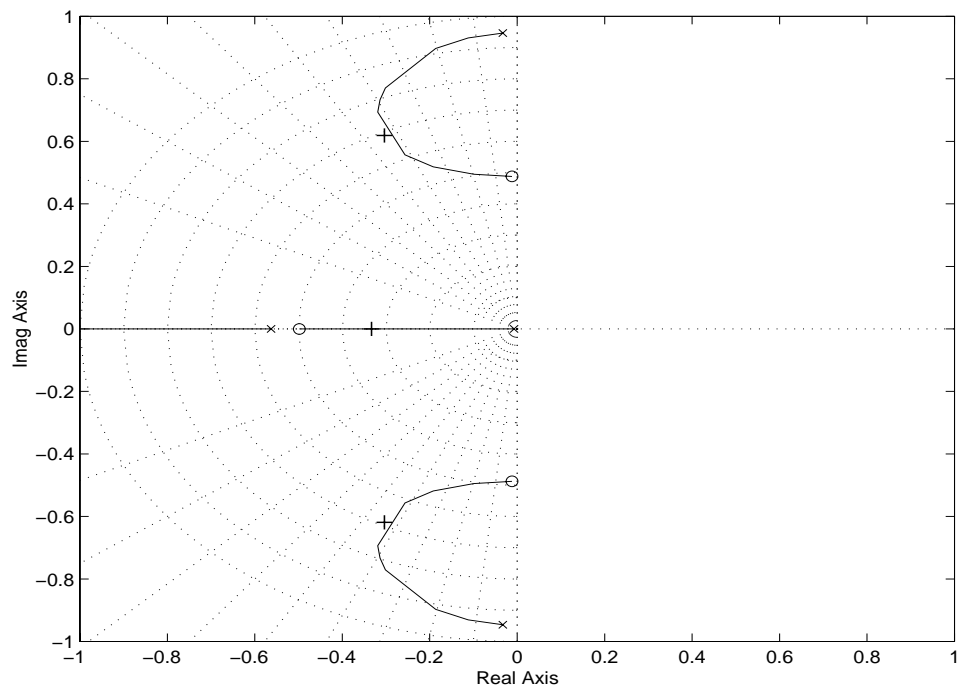
Since the simplest compensator is a static gain, first try to determine appropriate gain values using the root locus technique:

```
% Plot the root locus for the (1, 1) 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:

```
rl locus(-sys11)
sgrid
```



This looks better. Just using simple feedback can achieve a damping ratio of $\zeta = 0.45$. You can graphically select some pole locations and determine the corresponding gain with `rl ocfind`:

```
[k, poles] = rl ocfind(-sys11)
```

The '+' marks on the previous figure show a possible selection. The corresponding gain and closed-loop poles are:

k

k =

2.7580e-01

damp(poles)

Ei genval ue	Dampi ng	Freq. (rad/s)
-1.01e+00	1.00e+00	1.01e+00
-3.04e-01 + 6.19e-01i	4.41e-01	6.89e-01
-3.04e-01 - 6.19e-01i	4.41e-01	6.89e-01
-3.33e-01	1.00e+00	3.33e-01

Next, form the closed-loop system so that you can analyze this design:

cl11 = feedback(sys11, -k) % negative feedback by default

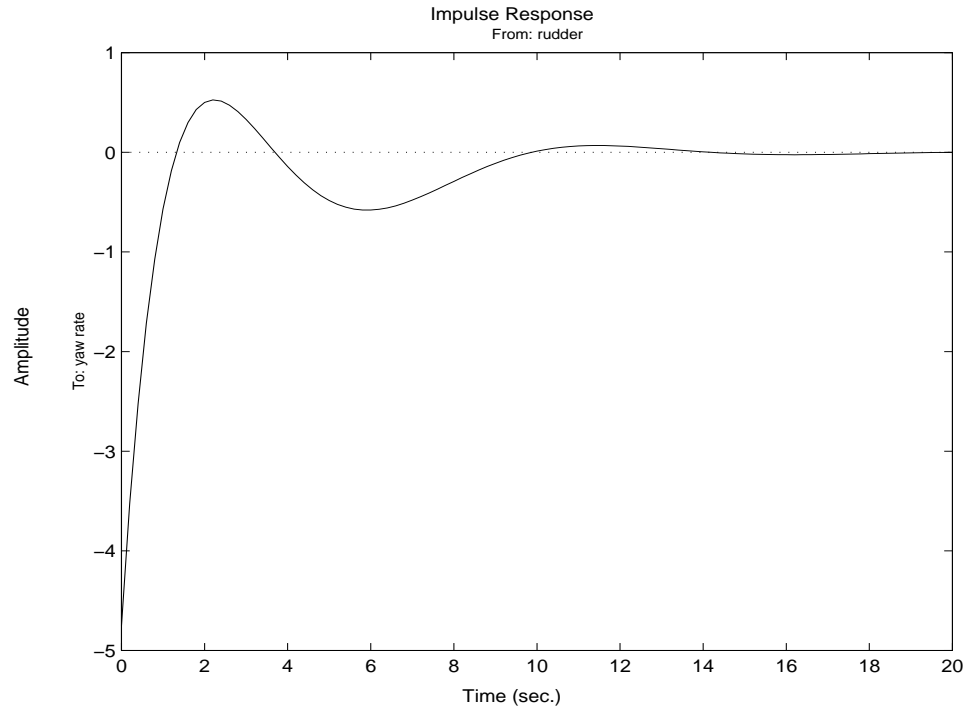
The closed-loop poles should match the ones chosen above (and they do):

damp(cl11)

Ei genval ue	Dampi ng	Freq. (rad/s)
-3.33e-01	1.00e+00	3.33e-01
-3.04e-01 + 6.19e-01i	4.41e-01	6.89e-01
-3.04e-01 - 6.19e-01i	4.41e-01	6.89e-01
-1.01e+00	1.00e+00	1.01e+00

Plot the closed-loop impulse response for a duration of 20 seconds:

```
impz(cl 11, 20)
```



The response settles quickly and does not oscillate much.

Now close the loop on the original 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:

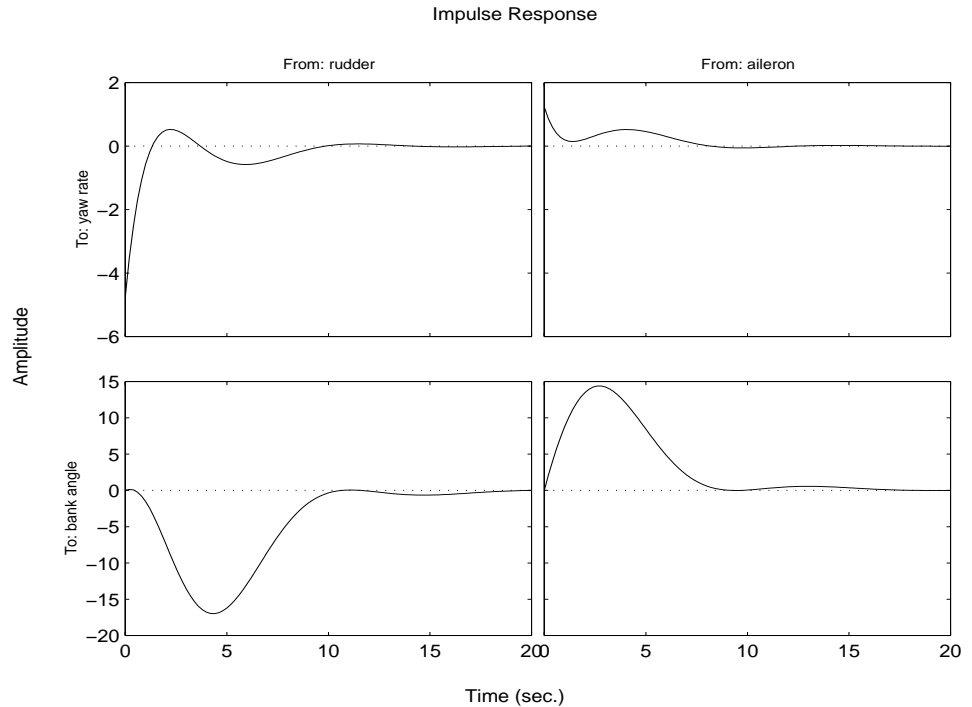
```
cl loop = feedback(sys, -k, 1, 1)
damp(cl loop)    % closed-loop poles
```

and MATLAB responds with:

Ei genval ue	Dampi ng	Freq. (rad/s)
-3. 33e-01	1. 00e+00	3. 33e-01
-3. 04e-01 + 6. 19e-01i	4. 41e-01	6. 89e-01
-3. 04e-01 - 6. 19e-01i	4. 41e-01	6. 89e-01
-1. 01e+00	1. 00e+00	1. 01e+00

Plot the MIMO impulse response:

`i mpul se(cl oop, 20)`



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 $a = 0.333$ for a time constant of three seconds and use the root locus technique to select the filter gain k . First specify the fixed part $s/(s + a)$ of the washout by

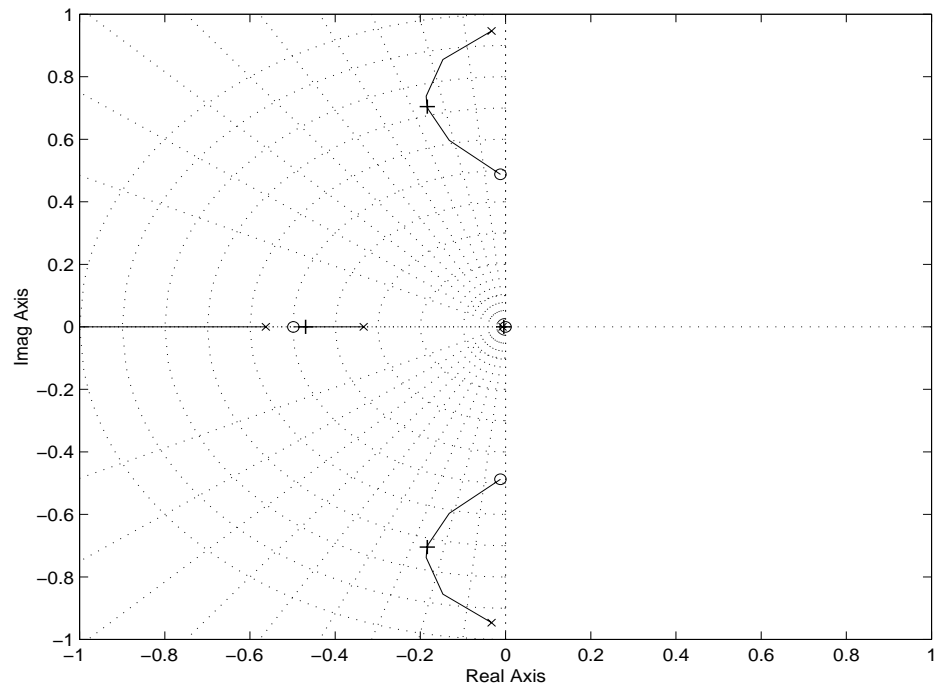
$$H = \text{zpk}(0, -0.333, 1)$$

Connect the washout in series with the design model `sys11` (relation between input 1 and output 1) to obtain the open-loop model

$$\text{ol loop} = H * \text{sys11}$$

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

```
rl locus(-ol oop)
sgrid
```



Now the maximum damping is $\zeta = 0.25$. You can select the gain providing maximum damping graphically by

```
[k, poles] = rlocfind(-ol oop)
```

The selected pole locations are marked by '+' on the root locus above. The resulting gain value and closed-loop dynamics are:

k

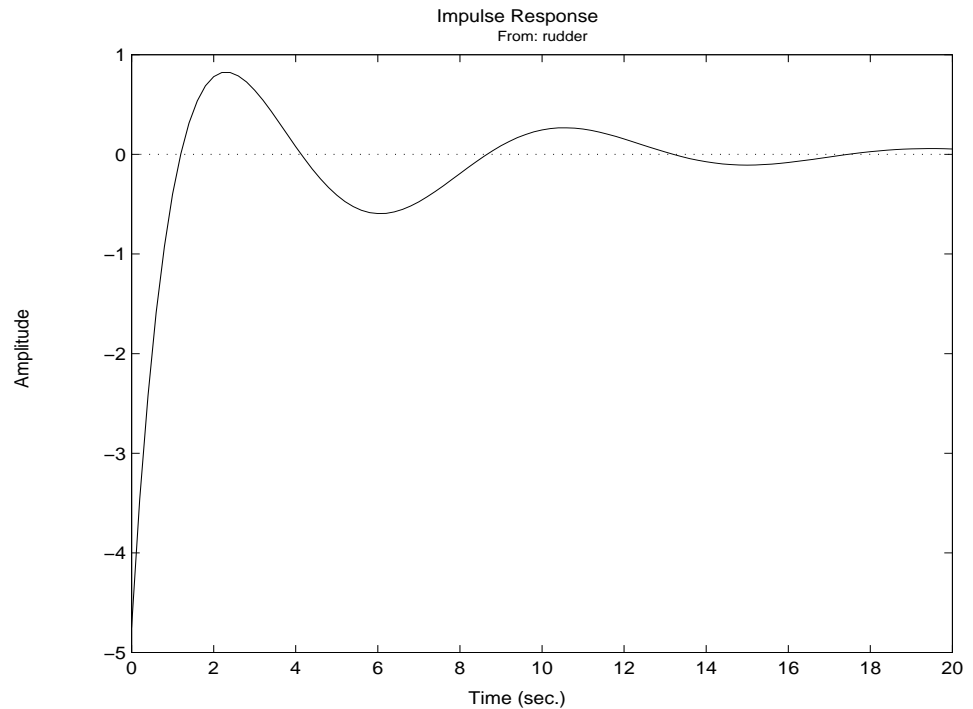
k =
2.2253e-01

damp(poles)

Eigenvalue	Damping	Freq. (rad/s)
-1.19e+00	1.00e+00	1.19e+00
-1.84e-01 + 7.05e-01i	2.52e-01	7.28e-01
-1.84e-01 - 7.05e-01i	2.52e-01	7.28e-01
-4.69e-01	1.00e+00	4.69e-01
-4.15e-03	1.00e+00	4.15e-03

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

```
cl11 = feedback(ol oop, -k);
impulse(cl11, 20)
```



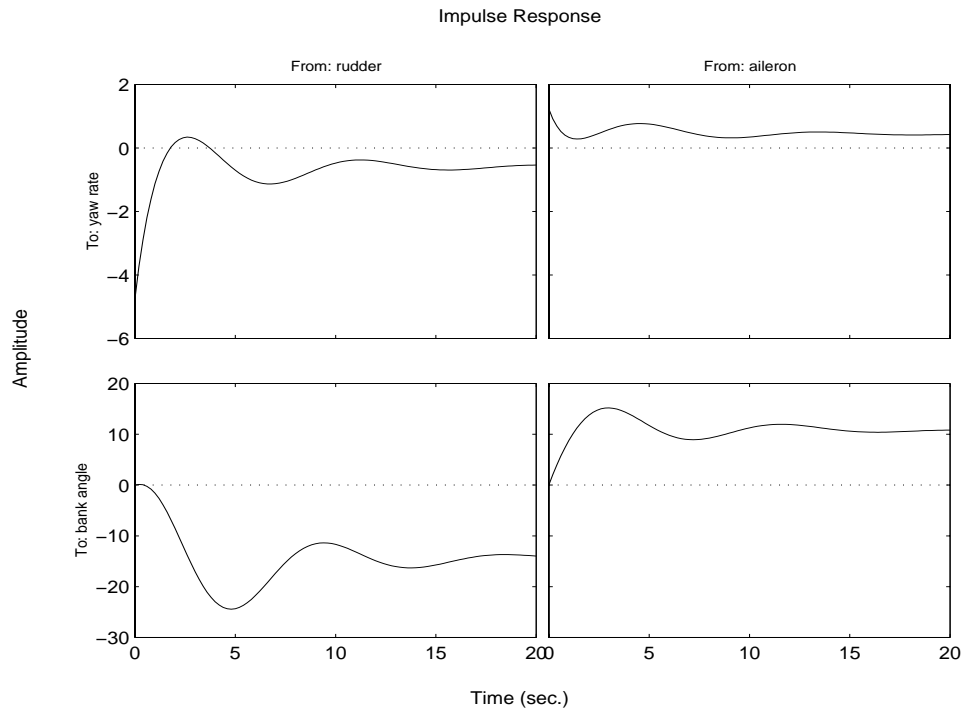
The response settles nicely but has less damping than your previous design. Finally, you can verify that the washout filter has fixed the spiral mode problem. First form the complete washout filter $kH(s)$ (washout + gain):

$$\text{WOF} = -k * H$$

Then close the loop around the first I/O pair of the MIMO model `sys` and simulate the impulse response:

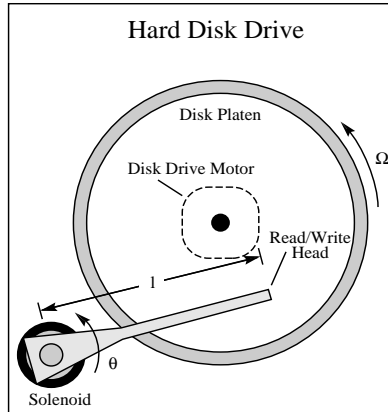
```
cl loop = feedback(sys, WOF, 1, 1);

% Final closed-loop impulse response
impul se(cl loop, 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. 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.

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
```

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')
```

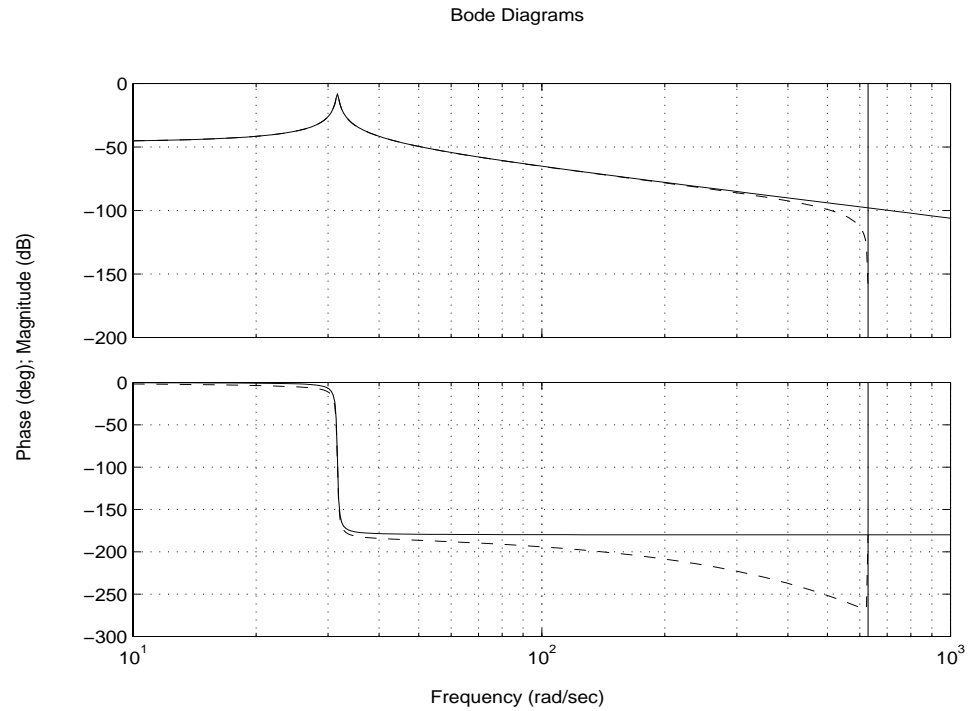
MATLAB responds:

```
Transfer function:
6.233e-05 z + 6.229e-05
-----
z^2 - 1.973 z + 0.998

Sampling time: 0.005
```

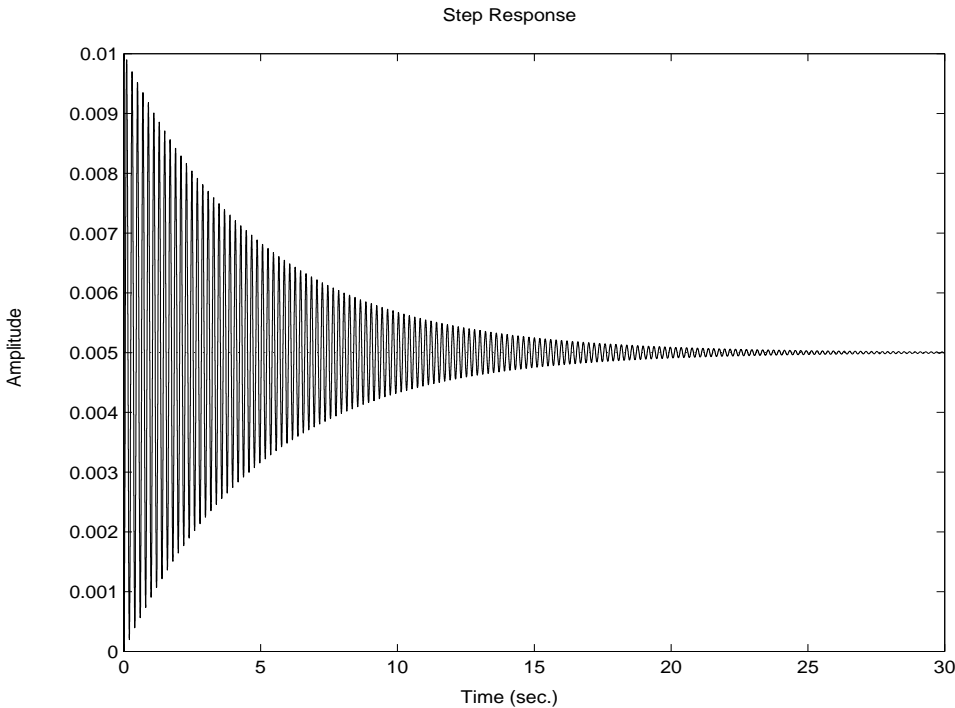
You can compare the Bode plots of the continuous and discretized models by:

```
bode(H, '-', Hd, '--')
```



To analyze the discrete system, plot its step response:

```
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)
```

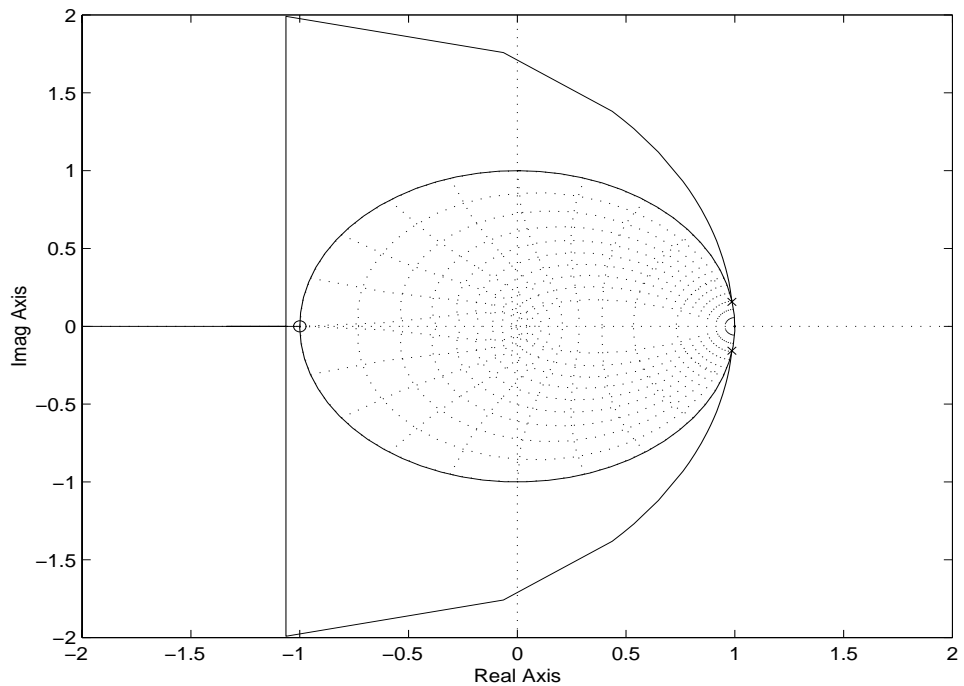
MATLAB responds:

Ei genval ue	Magni tude	Equi v. Dampi ng	Equi v. 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.

The simplest compensator is just a gain, so try the root locus technique to select an appropriate feedback gain:

`rlocus(Hd)`

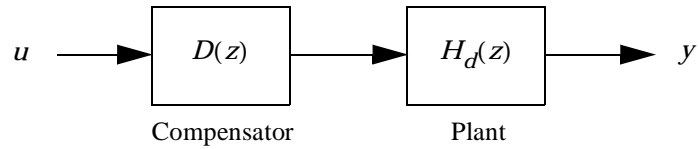


As shown in the root locus, the poles quickly leave the unit circle and go unstable. You need to introduce some lead or a compensator with some zeros. Try the compensator

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

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

The corresponding open-loop model



is obtained by the series connection:

```

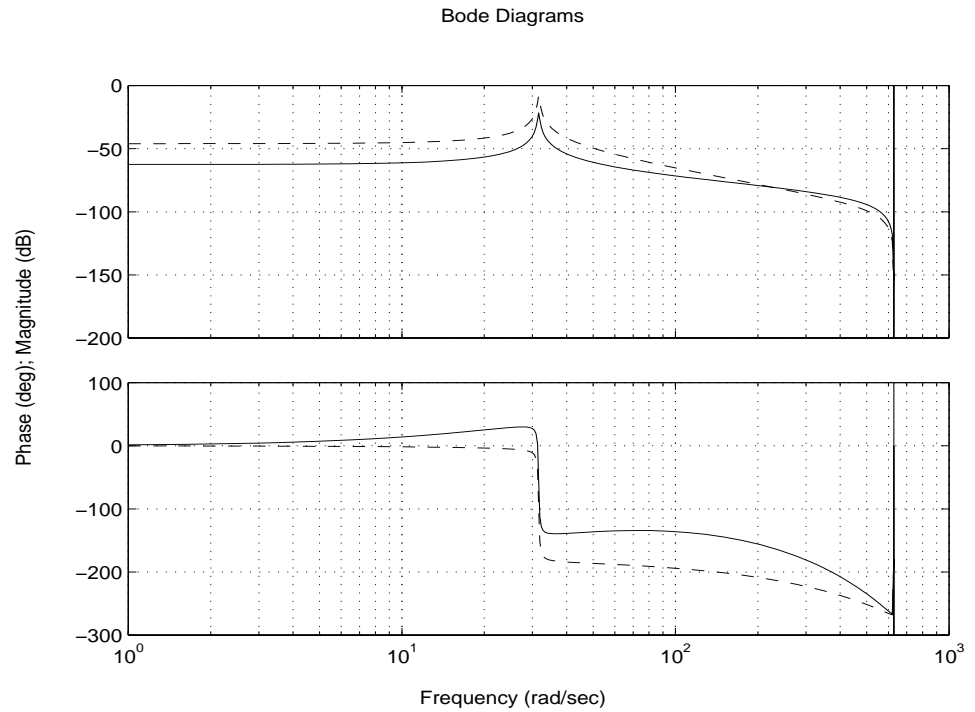
D = zpk(0.85, 0, 1, Ts)
ol oop = Hd * D
  
```

Now see how this compensator modifies the open-loop frequency response:

```

bode(Hd, ' - - ', ol oop, ' - ')
  
```

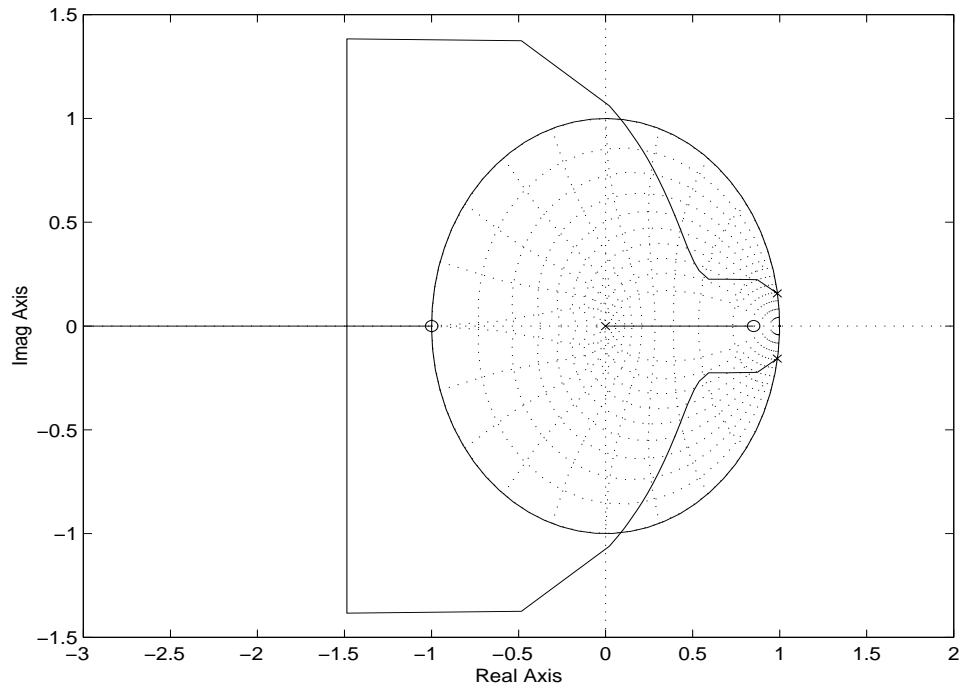
The plant response is the dashed line and the open-loop response with the compensator is the solid line.



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:

```
rl locus(ol oop)
zgrid
```



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). This plot shows a set of poles '+' selected using `rl ocfind`. At the MATLAB prompt, type:

```
[k, poles] = rl ocfind(ol oop)
k
```

MATLAB responds:

```
k =
    4.1179e+03
```

Type:

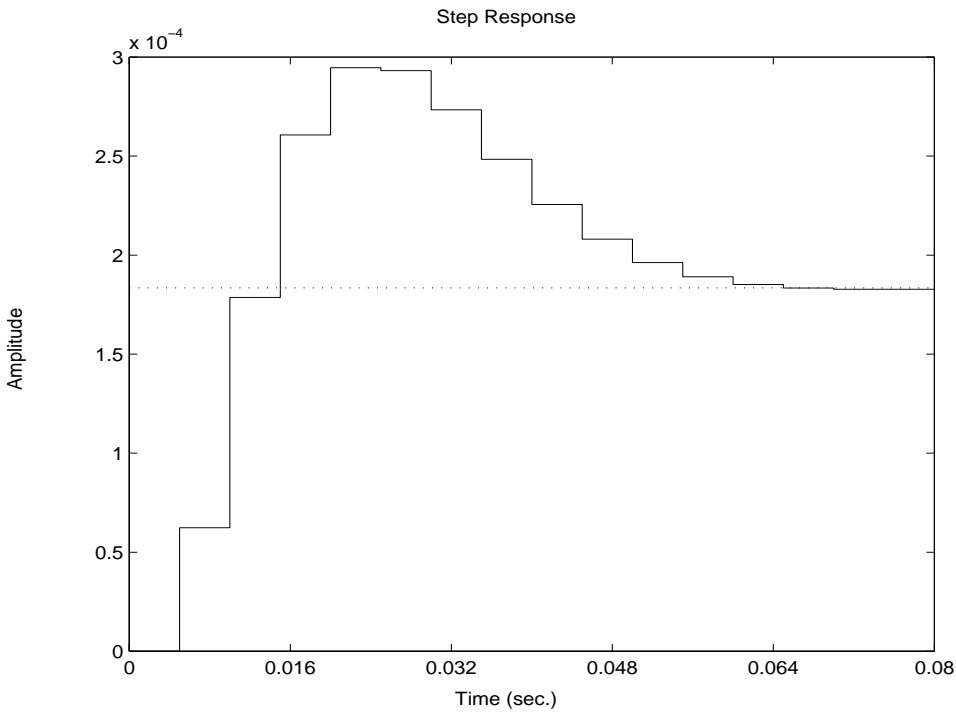
ddamp(pol es, Ts)

MATLAB responds:

Ei genval ue	Magni tude	Equi v. Dampi ng	Equi v. Freq.
5. 75e-01 + 2. 33e-01i	6. 21e-01	7. 78e-01	1. 23e+02
5. 75e-01 - 2. 33e-01i	6. 21e-01	7. 78e-01	1. 23e+02
5. 66e-01	5. 66e-01	1. 00e+00	1. 14e+02

To analyze this design, form the closed-loop system and plot the closed-loop step response:

```
cl oop = feedback(ol oop, k);  
step(cl oop)
```

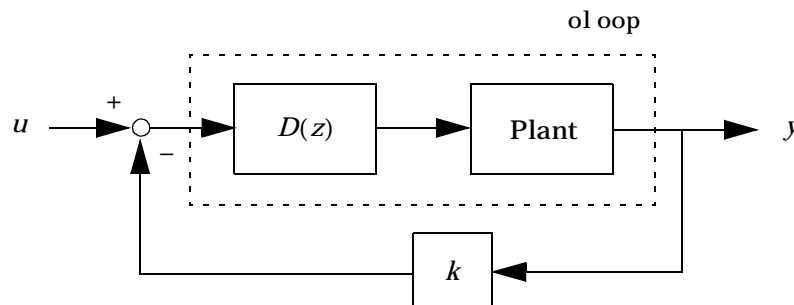


The response is fast and settles in about 14 samples, that is, $14 \times T_s = 0.07$ second. Thus, your disk drive has a seek time of about 0.07 second. This is slow by today's standards, but you also started with a very lightly damped system.

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:

```
ol k = k * ol oop;
```



Next apply `margin` to this open-loop model. Type:

```
[Gm, Pm, Wcg, Wcp] = margin(ol k);
Margins = [Gm Wcg Pm Wcp]
```

and MATLAB responds with:

```
Margins =
    3.7809   295.3172   43.1686   106.4086
```

Type:

```
% Gain margin in dB
20*log10(Gm)
```

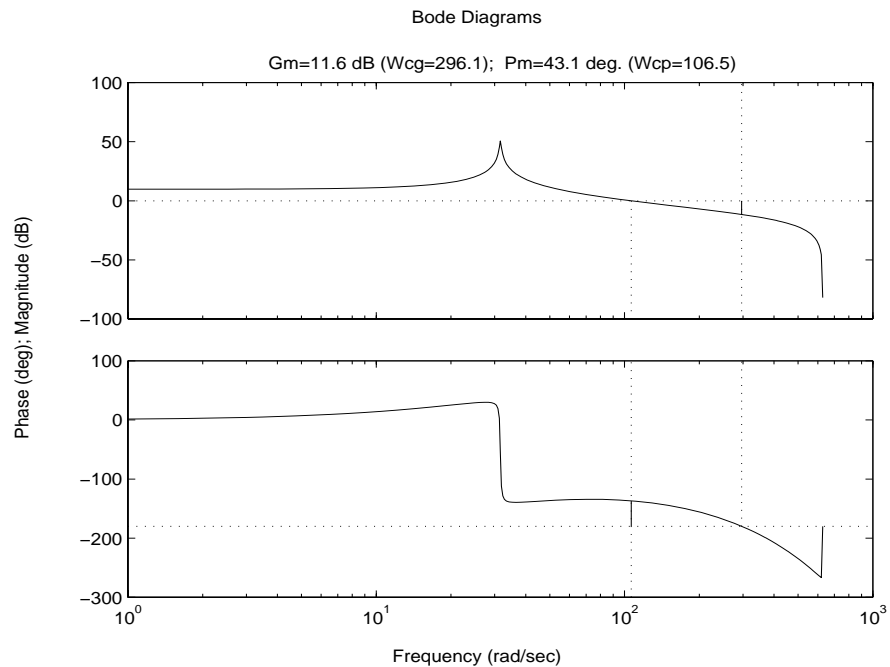
and MATLAB responds with:

```
ans =  
    11.5760
```

Type:

```
% Alternative graphical display of margins  
margin(ol k)
```

The last 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

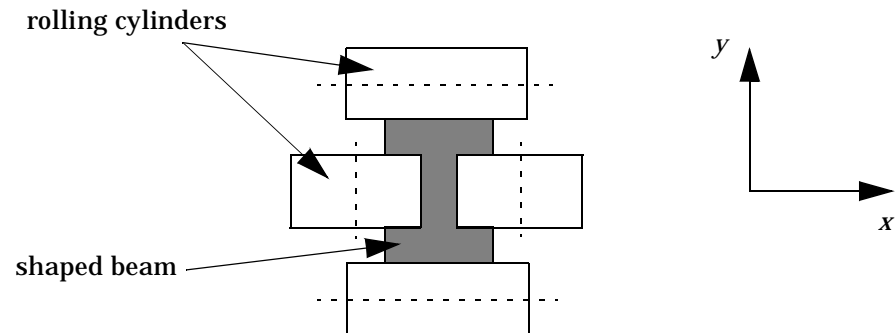
This case study demonstrates the use of the LQG design tools in a process control application. The goal is to regulate the horizontal and vertical thickness of the beam produced by a hot steel rolling mill. This example is adapted from [1]. The full plant model is MIMO and the example shows the advantage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

`mill demo`

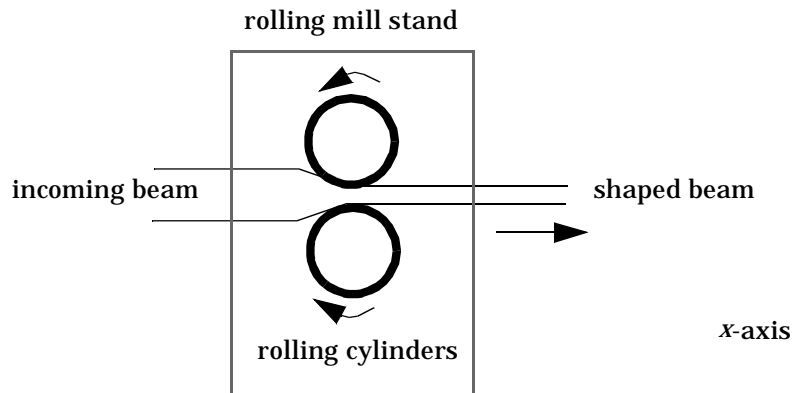
at the command line to run this demonstration interactively.

Process and Disturbance Models

The rolling mill is used to shape rectangular beams of hot metal. The desired outcoming 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 stem from:

- 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.

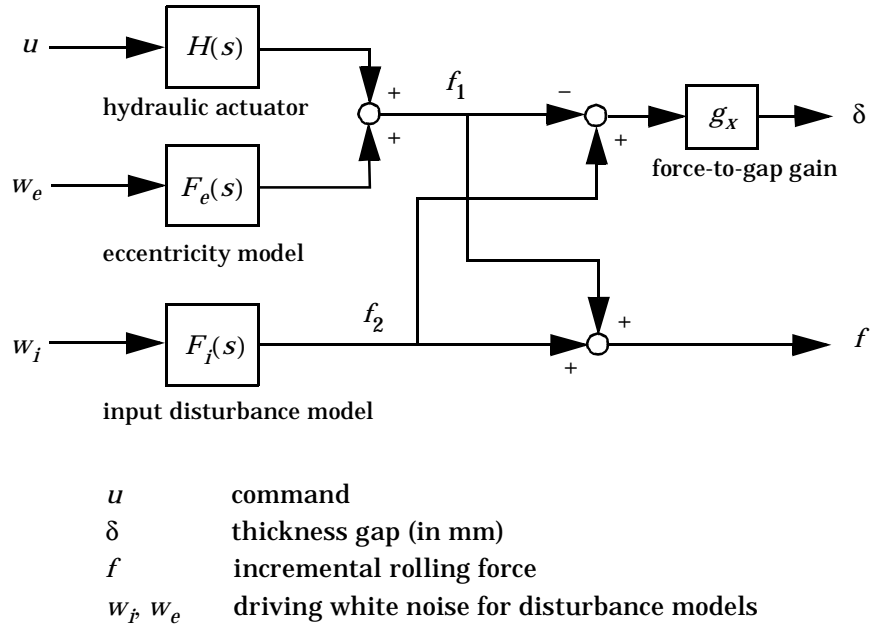


Figure 7-1: Open-loop model for x- or y-axis

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 Figure 7-1 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- i x'
```

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)
```

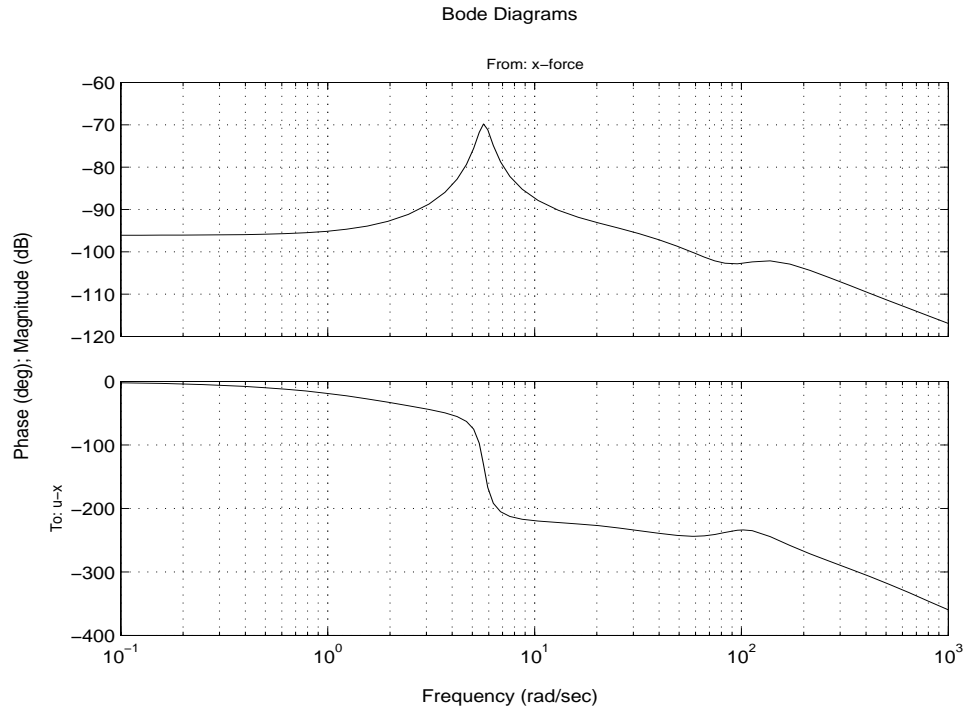
Finally, connect the state-feedback gain `kx` and state estimator `estx` to form the LQG regulator:

```
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° at low frequencies, the feedback loop then adequately reacts by increasing the hydraulic force to offset the thickness increase. Now consider the effect of eccentricity. Eccentricity causes fluctuations in the roll gap (gap between the rolling cylinders). When the roll gap is minimal, the rolling force increases and the beam thickness diminishes. The hydraulic force must then be reduced (negative force feedback) to restore the desired thickness. This is exactly what the LQG regulator does as its phase drops to -180° near the natural frequency of the eccentricity disturbance (6 rad/sec).

Next, compare the open- and closed-loop responses from disturbance to thickness gap. Use feedback to close the loop. To help specify the feedback connection, look at the I/O names of the plant Px and regulator Regx:

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

Regx. outputname
ans =
    'u- x'

Px. outputname
ans =
    'x- gap'
    'x- force'

Regx. inputname
ans =
    'x- force'
```

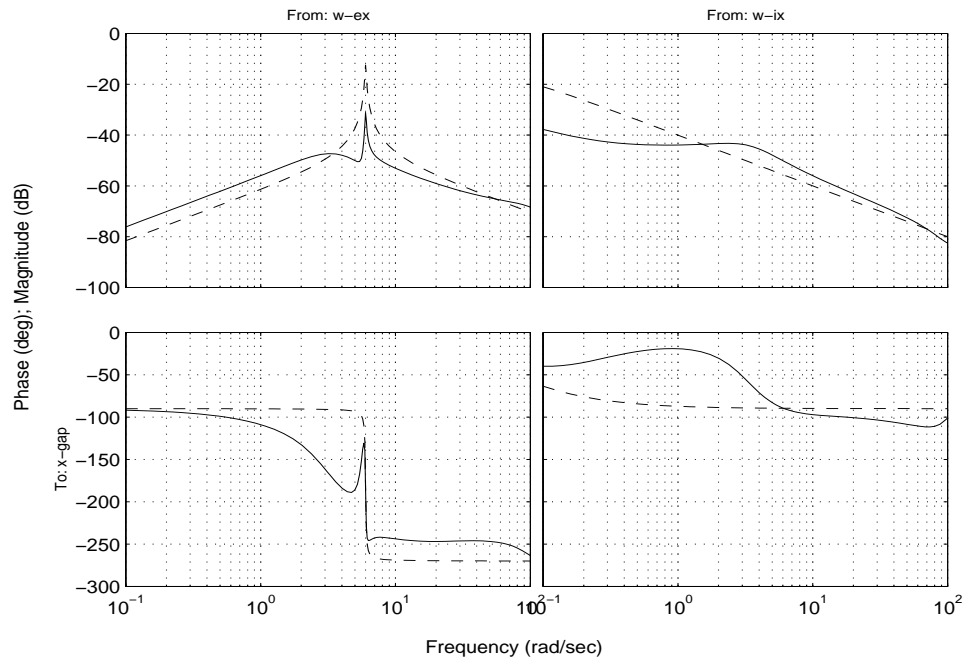
This indicates that you must connect the first input and second output of Px to the regulator:

```
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})
```

Bode Diagrams



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.

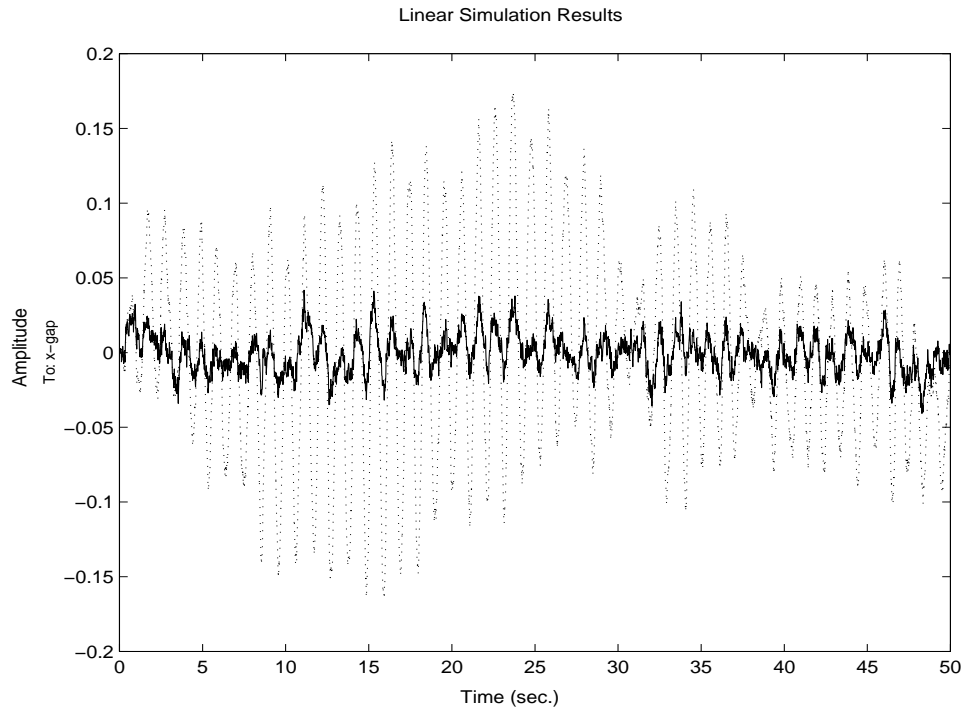
Finally, use `lsim` to simulate the open- and closed-loop time responses to the white noise inputs w_{ex} and w_{ix} . Choose $dt=0.01$ as sampling period for the

simulation, and derive equivalent discrete white noise inputs for this sampling rate:

```
dt = 0.01
t = 0:dt:50 % time samples

% Generate unit t-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. weighting
set(Pydes, 'outputn', {'y-gap*' 'y-force'})
ky = lqry(Pydes(1, 1), 1, 1e-4)

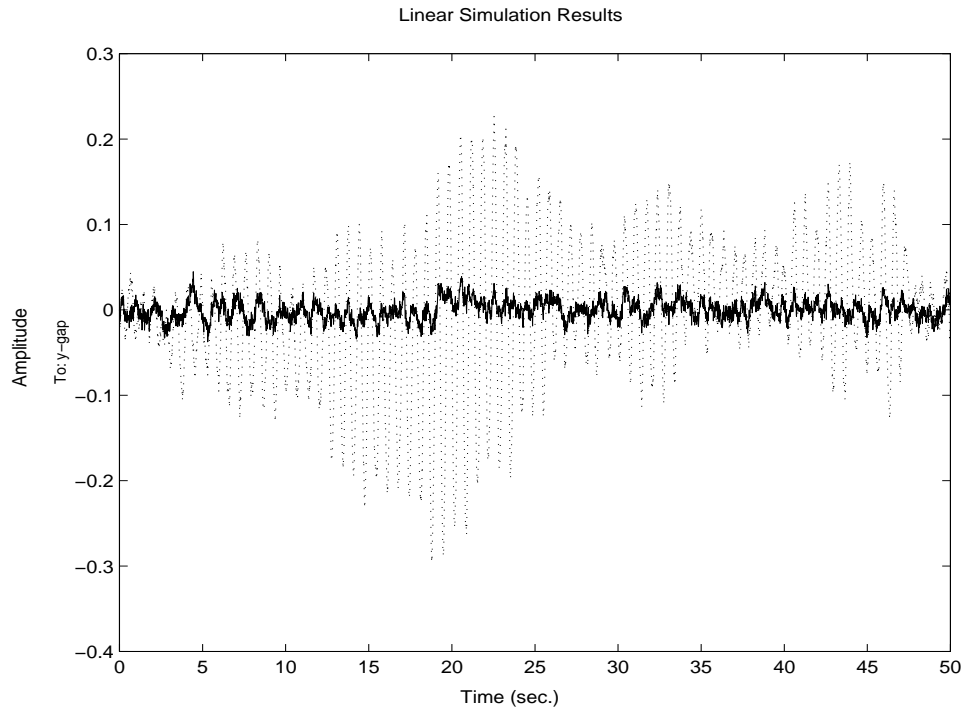
% Kalman estimator design
esty = kalman(Pydes(2, :), eye(2), 1e3)

% 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), ': ', cl y(1, 2:3), '- ', wy, t)
```



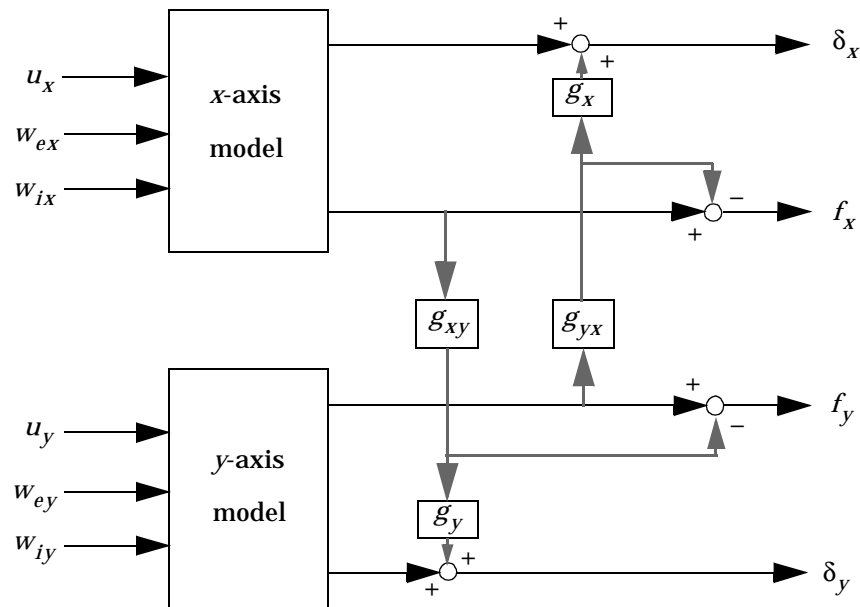
The dotted lines correspond to the open-loop response. The simulation results are comparable to those for the x -axis.

Cross-Coupling Between Axes

The x/y thickness regulation is a MIMO problem in essence. 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, the 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 coupling between axes is as follows:



$$g_{xy} = 0.1$$

$$g_{yx} = 0.4$$

Figure 7-2: Coupling between the x - and y -axes

Accordingly, the thickness gaps and rolling forces are related to the outputs $\bar{\delta}_x, \bar{f}_x, \dots$ 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 Figure 7-2, 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:

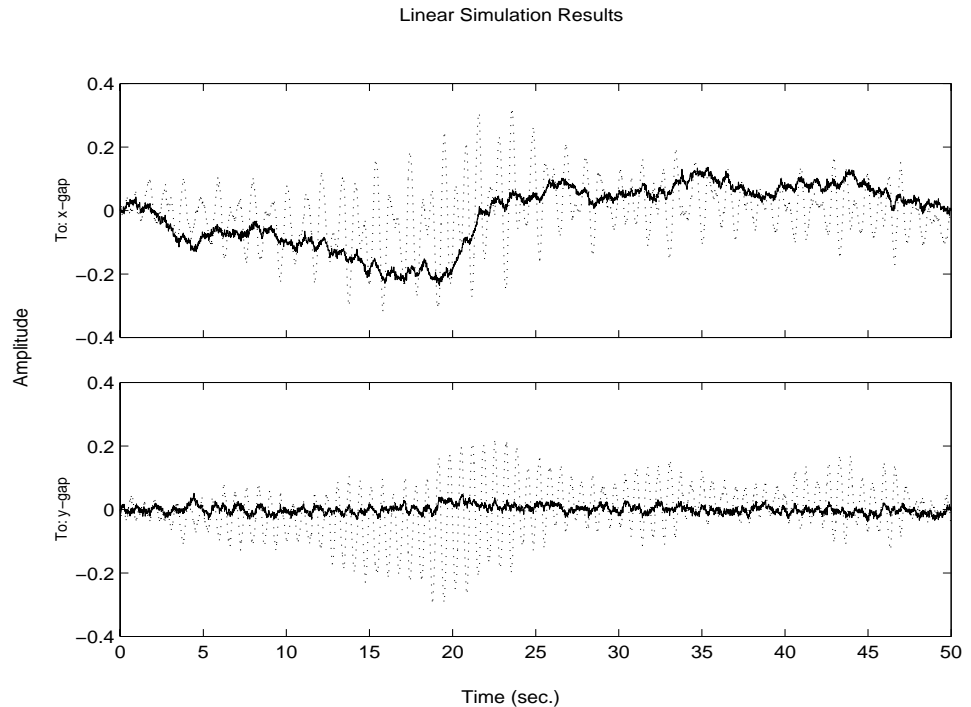
```
gxy = 0.1; gyx = 0.4;
CCmat = [eye(2) [0 gyx*gx; gxy*gy 0] ; zeros(2) [1 -gyx; -gxy 1]]
Pc = CCmat * P
Pc.outputname = P.outputname
```

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 w_x (for the x -axis) and w_y (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 P_c derived above. The model inputs and outputs are

```
Pc. inputname
```

```
ans =
    'u- x'
    'u- y'
    'w- ex'
    'w- i x'
    'w_ey'
    'w_i y'
```

```
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 `RegMI M0` has two inputs and two outputs:

```
RegMI M0. inputname
```

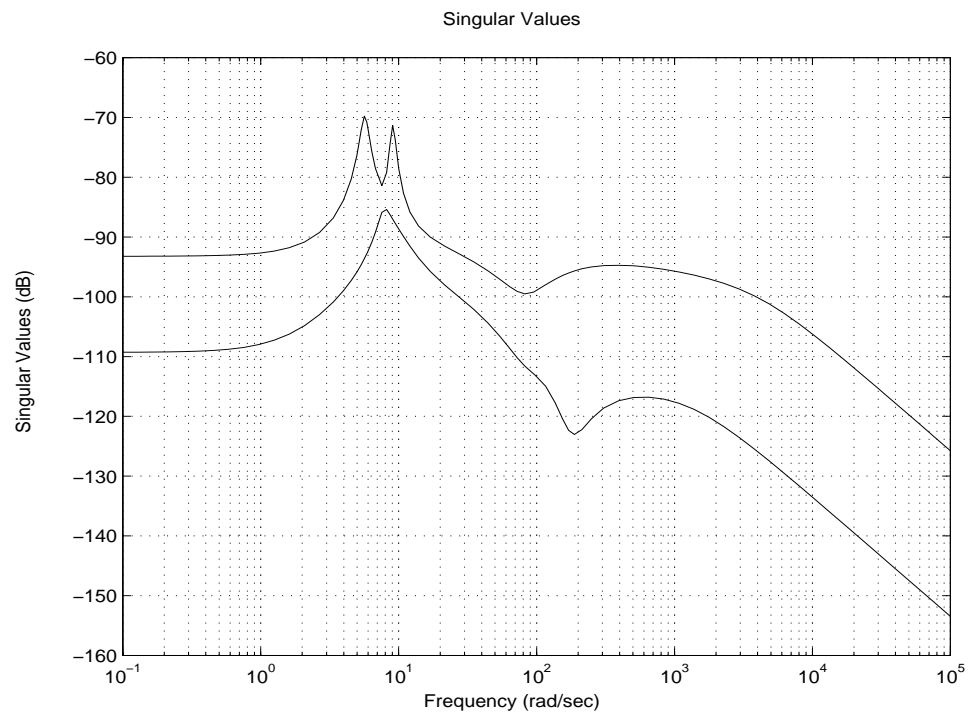
```
ans =  
    'x-force'  
    'y-force'
```

```
RegMI M0. outputname
```

```
ans =  
    'u-x'  
    'u-y'
```

Plot its singular value response (principal gains):

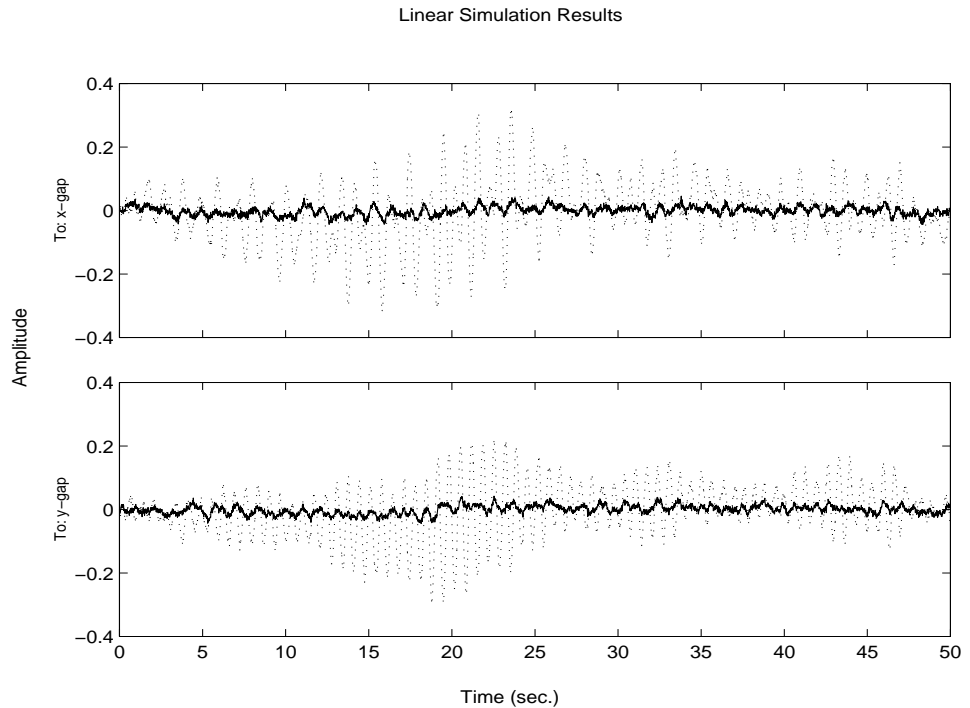
```
sigma(RegMI M0)
```



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

$$\begin{aligned}x[n+1] &= Ax[n] + B(u[n] + w[n]) \\ y[n] &= Cx[n]\end{aligned}$$

with additive Gaussian noise $w[n]$ on the input $u[n]$ and data

$$A = \begin{bmatrix} 1.1269 & -0.4940 & 0.1129 \\ 1.0000 & 0 & 0 \\ 0 & 1.0000 & 0 \end{bmatrix};$$

$$B = \begin{bmatrix} -0.3832 \\ 0.5919 \\ 0.5191 \end{bmatrix};$$

$$C = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix};$$

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:

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

$$E(w[n]w[n]^T) = Q, \quad E(v[n]v[n]^T) = R$$

You can combine the time and measurement update equations into one state-space model (the Kalman filter):

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

This filter generates an optimal estimate $\hat{y}[n|n]$ of $y[n]$. Note that the filter state is $\hat{x}[n|n-1]$.

Steady-State Design

You can design the steady-state Kalman filter described above with the function `kalman`. First specify the plant model with the process noise:

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] + Bw[n] && \text{(state equation)} \\ y[n] &= Cx[n] && \text{(measurement equation)} \end{aligned}$$

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');
```

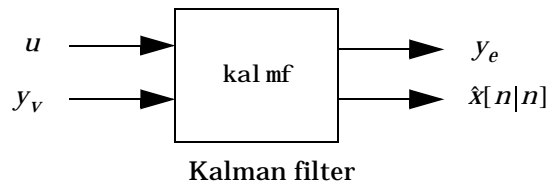
Assuming that $Q = R = 1$, you can now design the discrete Kalman filter by

```
Q = 1; R = 1;
[kal mf, L, P, M] = kal man(Pl ant, Q, R);
```

This returns a state-space model `kal mf` of the filter as well as the innovation gain

```
M
M =
    3.7980e-01
    8.1732e-02
   -2.5704e-01
```

The inputs of `kal mf` are u and y_v , and its outputs are the plant output and state estimates $y_e = \hat{y}[n|n]$ and $\hat{x}[n|n]$.



Because you are interested in the output estimate y_e , keep only the first output of `kal mf`. Type:

```
kal mf = kal mf(1, :);
kal mf
```

MATLAB responds with:

```
a =
```

	x1_e	x2_e	x3_e
x1_e	0. 76830	-0. 49400	0. 11290
x2_e	0. 62020	0	0
x3_e	-0. 08173	1. 00000	0

```
b =
```

	u	y
x1_e	-0. 38320	0. 35860
x2_e	0. 59190	0. 37980
x3_e	0. 51910	0. 08173

```
c =
```

	x1_e	x2_e	x3_e
y_e	0. 62020	0	0

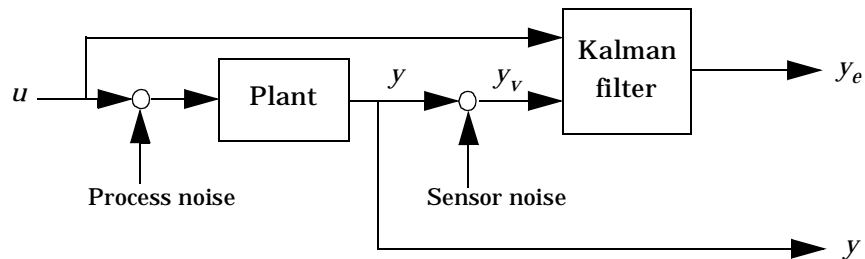
```
d =
```

	u	y
y_e	0	0. 37980

Sampling time: unspecified
Discrete-time system.

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.

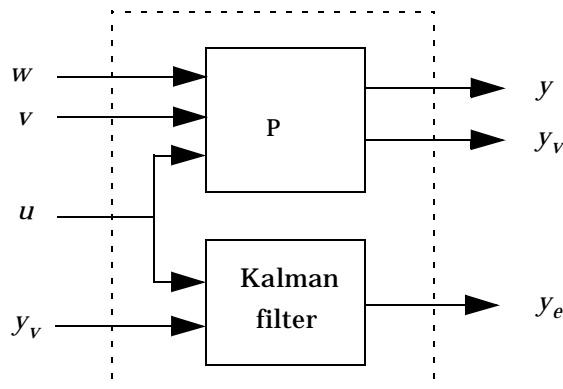
The block diagram below shows how to generate both true and filtered outputs.



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.input

ans =
    'w'
    'v'
    'u'

SimModel.output

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);
```

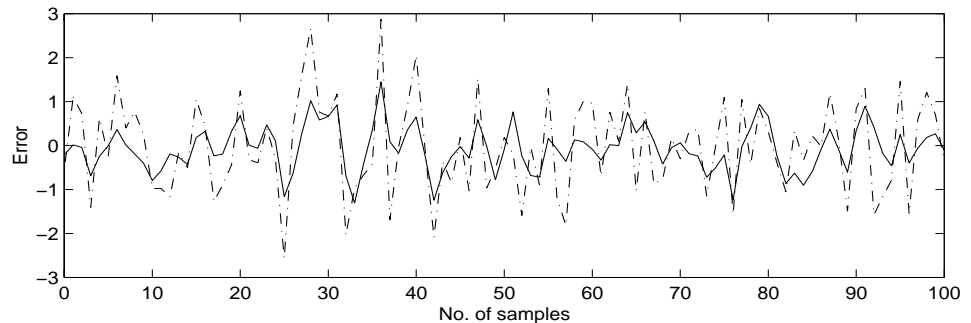
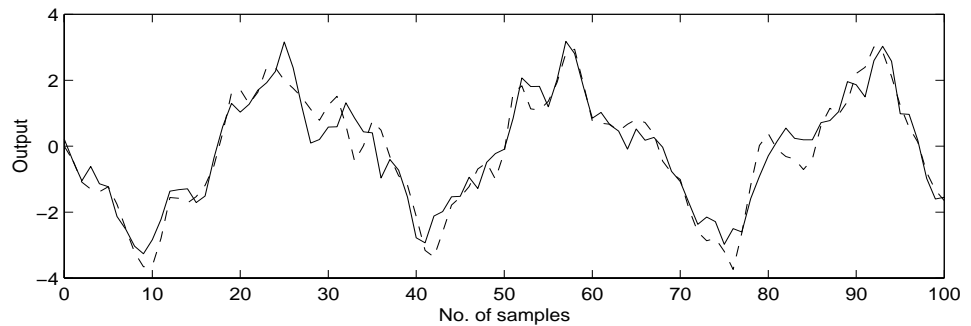
Now simulate with `lsim`:

```
[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')
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

$$\text{MeasErrCov}$$

$$\text{MeasErrCov} = 1.1138$$

while the error covariance after filtering (estimation error) is only

$$\text{EstErrCov}$$

$$\text{EstErrCov} = 0.2722$$

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

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

the time-varying Kalman filter is given by the 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 on page 7-49, and the notation

$$Q[n] = E(w[n]w[n]^T)$$

$$R[n] = E(v[n]v[n]^T)$$

$$P[n|n] = E(\{x[n] - x[n|n]\}\{x[n] - x[n|n]\}^T)$$

$$P[n|n-1] = E(\{x[n] - x[n|n-1]\}\{x[n] - x[n|n-1]\}^T)$$

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 = + v;                % v = measurement noise
```

Given the initial conditions

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

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';

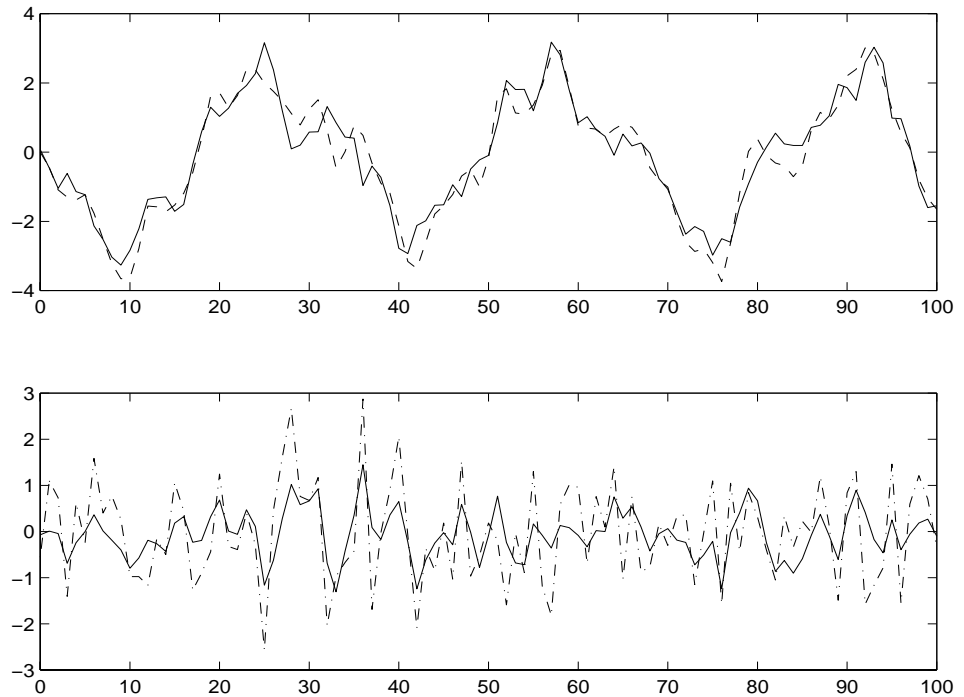
    % Time update
    x = A*x + B*u(i);          % x[n+1|n]
    P = A*P*A' + B*Q*B';       % P[n+1|n]
end

```

You can now compare the true and estimated output graphically:

```
subplot(211), plot(t, y, '--', t, ye, '-')
subplot(212), plot(t, y-yv, '-.', t, y-ye, '-')

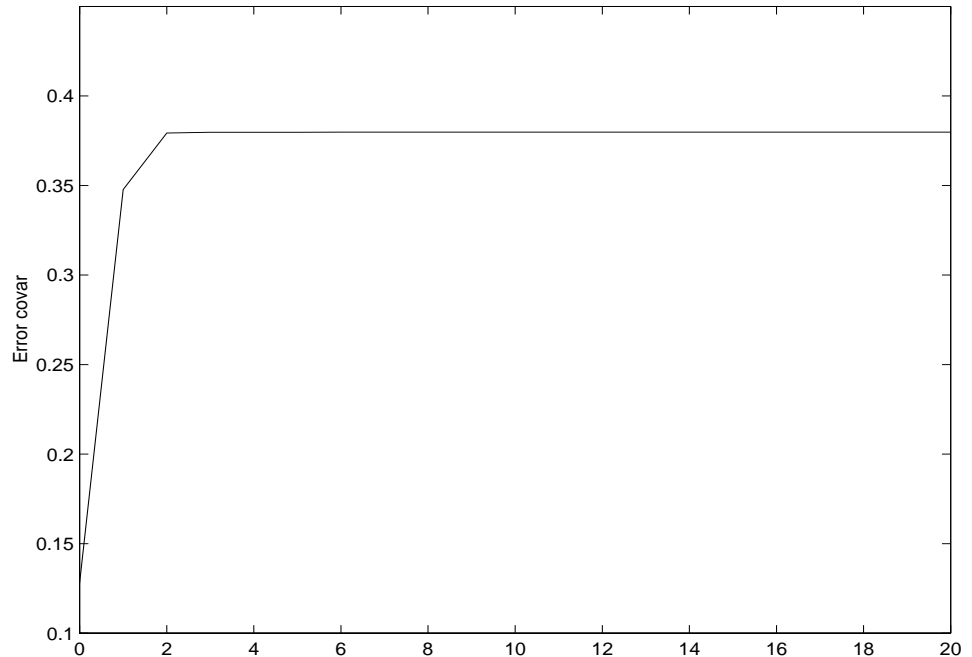
```



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):

```
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)
```

MATLAB responds:

```
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.

Reliable Computations

Conditioning and Numerical Stability	8-4
Conditioning	8-4
Numerical Stability	8-6
Choice of LTI Model	8-8
State Space	8-8
Transfer Function	8-8
Zero-Pole-Gain Models	8-14
Scaling	8-15
Summary	8-17
References	8-18

When working with low-order SISO models ($n < 5$), 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 $Ax = b$ with:

```
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 $\log_{10}(\text{cond}(A))$ 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 $1/\text{sqrt}(\text{eps})$, caution is advised for subsequent computations. For IEEE arithmetic, the machine precision, `eps`, is about 2.2×10^{-16} , and $1/\text{sqrt}(\text{eps}) = 6.7 \times 10^8$.

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:

$$A = \text{diag}(\text{ones}(1, n-1), 1);$$

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 $\lambda_1, \dots, \lambda_n$ 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):

$$A = \text{triu}(\text{ones}(n));$$

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

$$A = \begin{bmatrix} 0.001 & 1.000 \\ 1.000 & -1.000 \end{bmatrix} \quad b = \begin{bmatrix} 1.000 \\ 0.000 \end{bmatrix}$$

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 uses three types of LTI models:

- 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 MATLAB's 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] to illustrate. Consider the transfer function:

$$H(s) = \frac{1}{(s+1)(s+2)\dots(s+20)} = \frac{1}{s^{20} + 210s^{19} + \dots + 20!}$$

The A matrix of the companion realization of $H(s)$ is

$$A = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \vdots & 1 \\ -20! & \dots & \dots & \dots & -210 \end{bmatrix}$$

Despite the benign looking poles of the system (at $-1, -2, \dots, -20$) 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]
```

It has eigenvectors and eigenvalues:

```
[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).

Computing eigenvalues and eigenvectors, the eigenvectors have changed to:

```
[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
```

and 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 A_c) 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. And the problem generally gets far worse for higher-order systems, so 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 parsecs 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 :

$$u = N_u u_n \quad x = N_x x_n \quad y = N_y y_n$$

so the normalized system is

$$\begin{aligned} x_n &= A_n x_n + B_n u_n \\ y_n &= C_n x_n + D_n u_n \end{aligned}$$

where

$$\begin{aligned} A_n &= N_x^{-1} A N_x & B_n &= N_x^{-1} B N_u \\ C_n &= N_y^{-1} C N_x & D_n &= N_y^{-1} D N_u \end{aligned}$$

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*.

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] Bryson, A.E., and Y.C. Ho, *Applied Optimal Control*, Hemisphere Publishing, 1975. pp. 328-338.
- [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., "Efficient Multivariable Frequency Response Computations," *IEEE Transactions on Automatic Control*, Vol. AC-26, No. 2, April 1981, pp. 407-408.
- [5] 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.
- [6] Wilkinson, J.H., *Rounding Errors in Algebraic Processes*, Prentice-Hall, 1963.
- [7] Wilkinson, J.H., *The Algebraic Eigenvalue Problem*, Oxford University Press, 1965.

Reference

This chapter contains detailed descriptions of all Control System Toolbox functions. It begins with a list of functions grouped by subject area and continues with the reference entries in alphabetical order. Information is also available through the online Help facility.

Category Tables

LTI Models	
dss	Create descriptor state-space model.
filt	Create discrete filter with DSP convention.
get	Query LTI model properties.
set	Set LTI model properties.
ss	Create state-space model.
ssdata, dssdata	Retrieve state-space data.
tf	Create transfer function.
tfdata	Retrieve transfer function data.
zpk	Create zero-pole-gain model.
zpkdata	Retrieve zero-pole-gain data.
Model Characteristics	
class	Model type ('tf', 'zpk', or 'ss').
isa	True if LTI model is of specified type.
isct	True for continuous-time models.
isdt	True for discrete-time models.
isempty	True for empty LTI models.
isproper	True for proper LTI models.
isssiso	True for SISO models.
size	Input/output/state dimensions.

Model Conversions	
c2d	Continuous- to discrete-time conversion.
d2c	Discrete- to continuous-time conversion.
d2d	Resampling of discrete-time models.
pade	Pade approximation of input delays.
resi due	Partial fraction expansion (see <i>Using MATLAB</i>).
ss	Conversion to state space.
tf	Conversion to transfer function.
zpk	Conversion to zero-pole-gain.

Model Order Reduction	
bal real	Input/output balancing.
mi nreal	Minimal realization or pole/zero cancellation.
modred	State deletion in I/O balanced realization.

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.

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

Model Building	
append	Group systems by appending inputs and outputs.
augstate	Augment output by appending states.
connect	Block diagram modeling.
conv	Convolution of two polynomials (see <i>Using MATLAB</i>).
drmodel, drss	Generate random discrete model.
feedback	Feedback connection.
ord2	Generate second-order model.
pade	Pade approximation of time delays.
parallel	Generalized parallel connection.
rmodel, rss	Generate random continuous model.
series	Generalized series connection.
star	Star product (LFT interconnection).

Time Response	
<code>filter</code>	Simulation of discrete SISO filter (see <i>Using MATLAB</i>).
<code>gensig</code>	Input signal generator.
<code>impz</code>	Impulse response.
<code>initial</code>	Initial condition response.
<code>lsim</code>	Simulation of response to arbitrary inputs.
<code>ltiview</code>	LTI Viewer for linear response analysis.
<code>step</code>	Step response.

Frequency Response	
<code>bode</code>	Bode plot.
<code>evalfr</code>	Response at single complex frequency.
<code>freqresp</code>	Response over frequency grid.
<code>linspace</code>	Vector of evenly spaced frequencies.
<code>logspace</code>	Vector of logarithmically spaced frequencies.
<code>ltiview</code>	LTI Viewer for linear response analysis.
<code>margin</code>	Gain and phase margins.
<code>grid</code>	Grid lines for Nichols plot.
<code>nichols</code>	Nichols plot.
<code>nyquist</code>	Nyquist plot.
<code>sigma</code>	Singular value plot.

Root Locus Design	
pzmap	Pole-zero map.
rl ocfind	Interactive root locus gain selection.
rl locus	Evans root locus.
rl tool	Root Locus Design GUI.
sgrid	Continuous ω_n, ζ grid for root locus.
zgrid	Discrete ω_n, ζ grid for root locus.

Pole Placement	
acker	SISO pole placement.
place	MIMO pole placement.
estim	Form state estimator given estimator gain.
reg	Form output-feedback compensator given state-feedback and estimator gains.

LQG Design	
lqr	LQ-optimal gain for continuous systems.
dlqr	LQ-optimal gain for discrete systems.
lqry	LQ-optimal gain with output weighting.
lqrd	Discrete LQ gain for continuous plant.
kalman	Kalman estimator.
kalmd	Discrete Kalman estimator for continuous plant.
lqgreg	Form LQG regulator given LQ gain and Kalman filter.

Equation Solvers	
care	Solve continuous-time algebraic Riccati equations.
dare	Solve discrete-time algebraic Riccati equations.
lyap	Solve continuous-time Lyapunov equations.
dlap	Solve discrete-time Lyapunov equations.

Graphical User Interfaces for Linear Response Analysis and Design	
lti view	LTI Viewer for linear response analysis.
rltool	Root Locus Design GUI.

acker

Purpose Pole placement design for single-input systems

Syntax `k = acker(a, b, p)`

Description Given the single-input system

$$\dot{x} = Ax + bu$$

and a vector `p` of desired closed-loop pole locations, `acker` uses Ackermann's formula [1] to calculate a gain vector `k` such that the state feedback $u = -kx$ places the closed-loop poles at the locations `p`. In other words, the eigenvalues of $A - bk$ match the entries of `p` (up to ordering).

You can also use `acker` for estimator gain selection by transposing the `a` matrix and substituting `c'` for `b`:

$$l = \text{acker}(a', c', p)'$$

Limitations `acker` is limited to single-input systems and the pair (A, b) must be controllable.

Note that this method is not numerically reliable and starts to break down rapidly for problems of order greater than 5 or for weakly controllable systems. See `place` for a more general and reliable alternative.

See Also

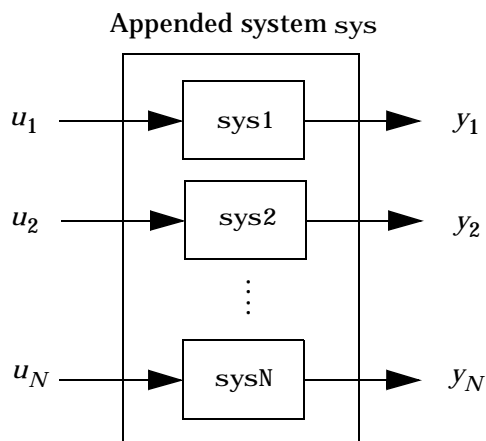
<code>place</code>	Pole placement design
<code>lqr</code>	Optimal LQ regulator
<code>rl locus</code> , <code>rl ocfind</code>	Root locus design

Reference [1] Kailath, T., *Linear Systems*, Prentice-Hall, 1980, p. 201.

Purpose Group LTI models by appending their inputs and outputs

Syntax `sys = append(sys1, sys2, . . . , sysN)`

Description `append` appends the inputs and outputs of the LTI models `sys1,...,sysN` to form the augmented model:



For systems with transfer functions $H_1(s), \dots, H_N(s)$, the resulting system `sys` has the block-diagonal transfer function

$$\begin{bmatrix} H_1(s) & 0 & \dots & 0 \\ 0 & H_2(s) & \dots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \dots & 0 & H_N(s) \end{bmatrix}$$

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) , `append(sys1, sys2)` produces the state-space model:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

Arguments

The input arguments `sys1`, ..., `sysN` can be LTI models of any type. Regular matrices are also accepted as a representation of static gains, but there should be at least one LTI object in the input list. The LTI models should be either all continuous, or all discrete with the same sample time. When appending models of different types, the resulting type is determined by the precedence rules (see page 2-3 for details).

There is no limitation on the number of inputs.

Example

The commands

```
sys1 = tf(1, [1 0])
sys2 = ss(1, 2, 3, 4)
sys = append(sys1, 10, sys2)
```

produce the state-space model

`sys`

`a =`

	x1	x2
x1	0	0
x2	0	1.00000

`b =`

	u1	u2	u3
x1	1.00000	0	0
x2	0	0	2.00000

c =		x1	x2
	y1	1. 00000	0
	y2	0	0
	y3	0	3. 00000

d =		u1	u2	u3
	y1	0	0	0
	y2	0	10. 00000	0
	y3	0	0	4. 00000

Continuous-time system.

See Also	parallel	Parallel connection
	series	Series connection
	feedback	Feedback connection
	connect	Modeling of block diagram interconnections

augstate

Purpose Augment state vector by appending outputs

Syntax `asys = augstate(sys)`

Description Given a state-space model `sys` with equations

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

(or their discrete-time counterpart), `augstate` appends the states x to the outputs y to form the model

$$\dot{x} = Ax + Bu$$

$$\begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} x + \begin{bmatrix} D \\ 0 \end{bmatrix} u$$

This command prepares the plant so that you can use the `feedback` command to close the loop on a full-state feedback $u = -Kx$.

Limitation Because `augstate` is only meaningful for state-space models, it cannot be used with transfer functions or zero-pole-gain models.

See Also

<code>parallel</code>	Parallel connection
<code>series</code>	Series connection
<code>feedback</code>	Feedback connection

Purpose	Input/output balancing of state-space realizations
Syntax	<pre>sysb = bal real (sys) [sysb, g, T, Ti] = bal real (sys)</pre>
Description	<p><code>sysb = bal real (sys)</code> produces a balanced realization <code>sysb</code> of the LTI model <code>sys</code> with equal and diagonal controllability and observability gramians (see <code>gram</code> for a definition of gramian). <code>bal real</code> handles both continuous and discrete systems. If <code>sys</code> is not a state-space model, it is first and automatically converted to state space using <code>ss</code>.</p> <p><code>[sysb, g, T, Ti] = bal real (sys)</code> also returns the vector <code>g</code> containing the diagonal of the balanced gramian, the state similarity transformation $x_b = Tx$ used to convert <code>sys</code> to <code>sysb</code>, and the inverse transformation $Ti = T^{-1}$.</p> <p>If the system is normalized properly, the diagonal <code>g</code> of the joint gramian can be used to reduce the model order. Because <code>g</code> reflects the combined controllability and observability of individual states of the balanced model, you can delete those states with a small <code>g(i)</code> while retaining the most important input-output characteristics of the original system. Use <code>modred</code> to perform the state elimination.</p>
Example	<p>Consider the zero-pole-gain model</p> <pre>sys = zpk([-10 -20.01], [-5 -9.9 -20.1], 1)</pre> <p>Zero/pole/gain:</p> <pre>(s+10) (s+20.01) ----- (s+5) (s+9.9) (s+20.1)</pre> <p>A state-space realization with balanced gramians is obtained by</p> <pre>[sysb, g] = bal real (sys)</pre> <p>The diagonal entries of the joint gramian are</p> <pre>g'</pre> <pre>ans =</pre> <pre>1.0062e-01 6.8039e-05 1.0055e-05</pre>

which indicates that the last two states of sysb are weakly coupled to the input and output. You can then delete these states by

```
sysr = modred(sysb, [2 3], 'del')
```

to obtain the following first-order approximation of the original system:

```
zpk(sysr)
```

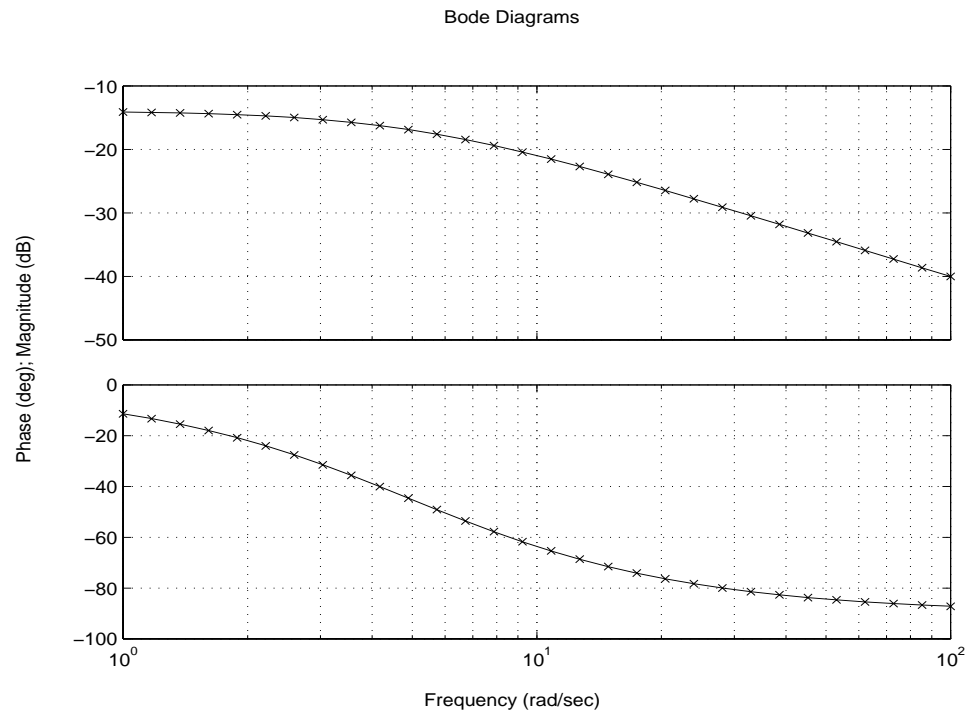
Zero/pole/gain:

1.0001

(s+4.97)

Compare the Bode responses of the original and reduced-order models:

```
bode(sys, '-', sysr, 'x')
```



Algorithm

Consider the model

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

with controllability and observability gramians W_c and W_o . The state coordinate transformation $\bar{x} = Tx$ produces the equivalent model

$$\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu$$

$$y = CT^{-1}\bar{x} + Du$$

and transforms the gramians to

$$\bar{W}_c = TW_cT^T, \quad \bar{W}_o = T^{-T}W_oT^{-1}$$

The function `bal real` computes a particular similarity transformation T such that

$$\bar{W}_c = \bar{W}_o = \text{diag}(g)$$

See [1,2] for details on the algorithm.

Limitations

The LTI model `sys` must be stable. In addition, controllability and observability are required for state-space models.

See Also

<code>gram</code>	Controllability and observability gramians
<code>modred</code>	Model order reduction
<code>mi nreal</code>	Minimal realizations

References

- [1] Laub, A.J., M.T. Heath, C.C. Paige, and R.C. Ward, "Computation of System Balancing Transformations and Other Applications of Simultaneous Diagonalization Algorithms," *IEEE Trans. Automatic Control*, AC-32 (1987), pp. 115–122.
- [2] Moore, B., "Principal Component Analysis in Linear Systems: Controllability, Observability, and Model Reduction," *IEEE Transactions on Automatic Control*, AC-26 (1981), pp. 17–31.
- [3] Laub, A.J., "Computation of Balancing Transformations," *Proc. ACC*, San Francisco, Vol.1, paper FA8-E, 1980.

bode

Purpose Bode frequency response of LTI systems

Syntax

```
bode(sys)
bode(sys, w)

bode(sys1, sys2, ..., sysN)
bode(sys1, sys2, ..., sysN, w)
bode(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')

[ mag, phase, w ] = bode(sys)
```

Description `bode` computes the magnitude and phase of the frequency response of LTI systems. When invoked without left-hand arguments, `bode` produces a Bode plot on the screen. Bode plots are used to analyze system properties such as the gain margin, phase margin, DC gain, bandwidth, disturbance rejection, and stability.

`bode(sys)` plots the Bode response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. In the MIMO case, `bode` produces an array of Bode plots, each plot showing the Bode response of one particular I/O channel. The frequency range is determined automatically based on the system poles and zeros.

`bode(sys, w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin, wmax]`, set `w = {wmin, wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in radians/sec.

`bode(sys1, sys2, ..., sysN)` or `bode(sys1, sys2, ..., sysN, w)` plots the Bode responses of several LTI models on a single figure. All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous and discrete systems. This syntax is useful to compare the Bode responses of multiple systems.

`bode(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')` further specifies which color, linestyle, and/or marker should be used to plot each system. For example,

```
bode(sys1, 'r--', sys2, 'gx')
```

uses red dashed lines for the first system `sys1` and green 'x' markers for the second system `sys2`.

When invoked with left-hand arguments,

```
[mag, phase, w] = bode(sys)
[mag, phase] = bode(sys, w)
```

return the magnitude and phase (in degrees) of the frequency response at the frequencies `w` (in rad/sec). The outputs `mag` and `phase` are 3-D arrays with the frequency as the last dimension (see “Arguments” below for details). You can convert the magnitude to decibels by

```
magdb = 20*log10(mag).
```

Arguments

The output arguments `mag` and `phase` are 3-D arrays with dimensions

(number of outputs) \times (number of inputs) \times (length of `w`)

For SISO systems, `mag(1, 1, k)` and `phase(1, 1, k)` give the magnitude and phase of the response at the frequency $\omega_k = w(k)$:

$$\begin{aligned}\text{mag}(1, 1, k) &= |h(j\omega_k)| \\ \text{phase}(1, 1, k) &= \angle h(j\omega_k)\end{aligned}$$

MIMO systems are treated as arrays of SISO systems and the magnitudes and phases are computed for each SISO entry h_{ij} independently (h_{ij} is the transfer function from input j to output i). The values `mag(i, j, k)` and `phase(i, j, k)` then characterize the response of h_{ij} at the frequency `w(k)`:

$$\begin{aligned}\text{mag}(i, j, k) &= |h_{ij}(j\omega_k)| \\ \text{phase}(i, j, k) &= \angle h_{ij}(j\omega_k)\end{aligned}$$

Example

You can plot the Bode response of the continuous SISO system

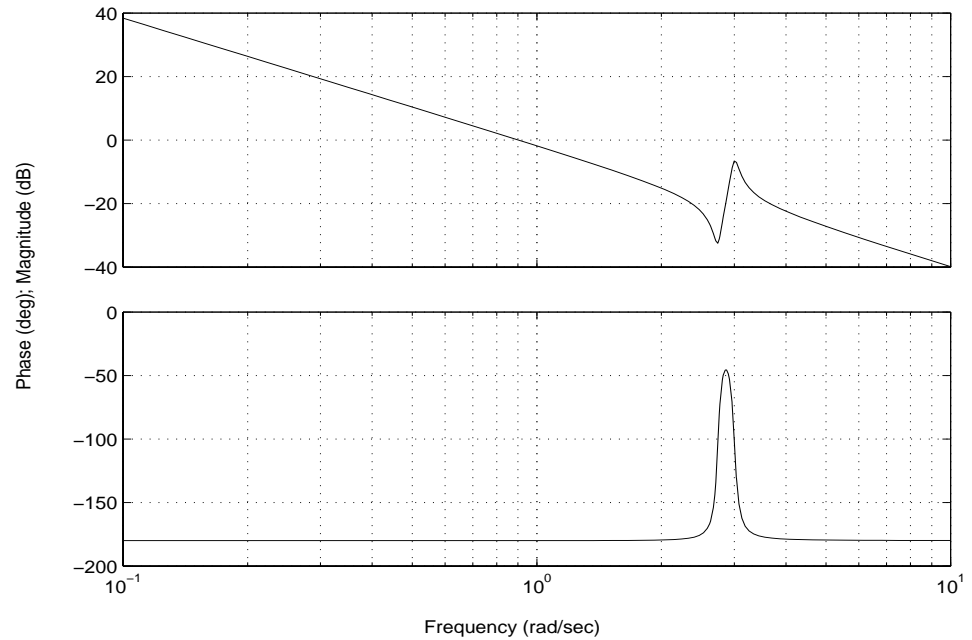
$$H(s) = \frac{s^2 + 0.1s + 7.5}{s^4 + 0.12s^3 + 9s^2}$$

bode

by typing

```
g = tf([1 0.1 7.5], [1 0.12 9 0 0]);  
bode(g)
```

Bode Diagrams

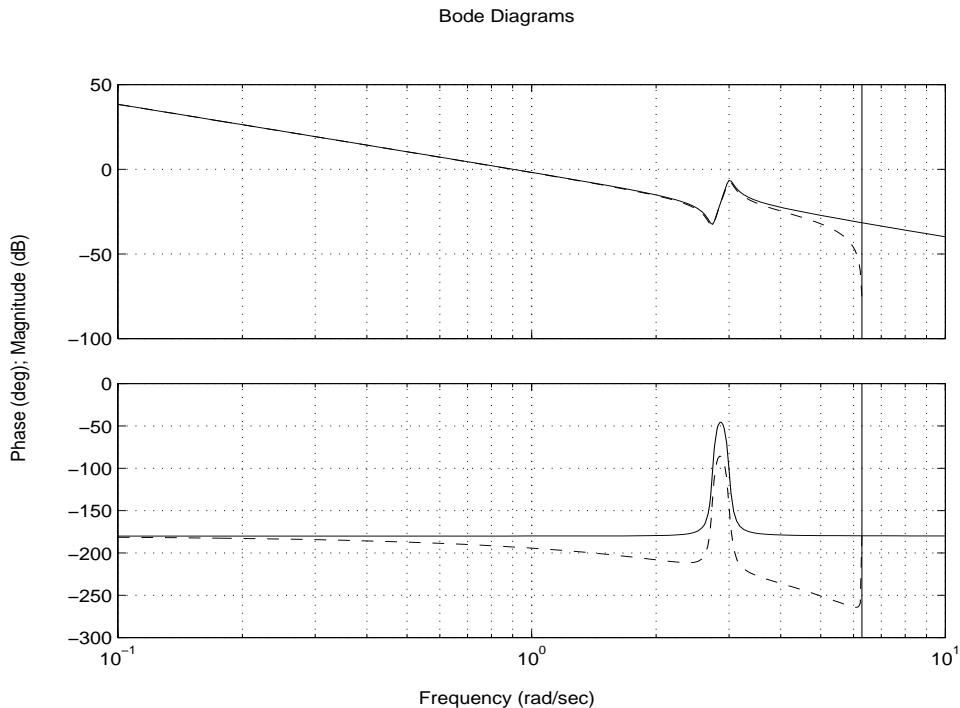


To plot the response on a wider frequency range, for example, from 0.1 to 100 rad/sec, type

```
bode(g, {0.1 , 100})
```

You can also discretize this system using zero-order hold and the sample time $T_s = 0.5$ second, and compare the continuous and discretized responses by typing:

```
gd = c2d(g, 0.5)
bode(g, 'r', gd, 'b--')
```



Algorithm

For continuous-time systems, bode computes the frequency response by evaluating the transfer function $H(s)$ on the imaginary axis $s = j\omega$. Only positive frequencies ω are considered. For state-space models, the frequency response is

$$D + C(j\omega - A)^{-1}B, \quad \omega \geq 0$$

When numerically safe, A is diagonalized for maximum speed. Otherwise, A is reduced to upper Hessenberg form and the linear equation $(j\omega - A)X = B$

is solved at each frequency point, taking advantage of the Hessenberg structure. The reduction to Hessenberg form provides a good compromise between efficiency and reliability. See [1] for more details on this technique.

For discrete-time systems, the frequency response is obtained by evaluating the transfer function $H(z)$ on the unit circle. To facilitate interpretation, the upper-half of the unit circle is parametrized as:

$$z = e^{j\omega T_s}, \quad 0 \leq \omega \leq \omega_N = \frac{\pi}{T_s}$$

where T_s is the sample time and ω_N is called the *Nyquist frequency*. The equivalent “continuous-time frequency” ω is then used as the x -axis variable. Because

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

is periodic with period $2\omega_N$, bode plots the response only up to the Nyquist frequency ω_N . If the sample time is unspecified, the default value $T_s = 1$ is assumed.

Diagnostics

If the system has a pole on the $j\omega$ axis (or unit circle in the discrete case) and w happens to contain this frequency point, the gain is infinite, $j\omega I - A$ is singular, and bode produces the warning message:

Singularity in freq. response due to jw-axis or unit circle pole.

See Also

- | | |
|----------|--------------------------------------|
| lti view | LTI system viewer |
| nyquist | Nyquist plot |
| nichols | Nichols plot |
| sigma | Singular value plot |
| freqresp | Frequency response computation |
| evalfr | Response at single complex frequency |

References

[1] Laub, A.J., “Efficient Multivariable Frequency Response Computations,” *IEEE Transactions on Automatic Control*, AC-26 (1981), pp. 407–408.

Purpose Discretize continuous-time systems

Syntax

```
sysd = c2d(sys, Ts)
sysd = c2d(sys, Ts, method)
```

Description `sysd = c2d(sys, Ts)` discretizes the continuous-time LTI model `sys` using zero-order hold on the inputs and a sample time of `Ts` seconds.

`sysd = c2d(sys, Ts, method)` gives access to alternative discretization schemes. The string *method* selects the discretization method among the following:

'zoh'	Zero-order hold. The control inputs are assumed piecewise constant over the sampling period <code>Ts</code> .
'foh'	Triangle approximation (modified first-order hold, see [1], p. 151). The control inputs are assumed piecewise linear over the sampling period <code>Ts</code> .
'tustin'	Bilinear (Tustin) approximation.
'prewarp'	Tustin approximation with frequency prewarping.
'matched'	Matched pole-zero method. See [1], p. 147.

Refer to page 2-39 for more detail on these discretization methods.

`c2d` supports MIMO systems (except for the 'matched' method) as well as LTI models with input delays ('zoh' and 'foh' methods only).

Example Consider the system

$$H(s) = \frac{s-1}{s^2 + 4s + 5}$$

with input delay $T_d = 0.35$ second. To discretize this system using the triangle approximation with sample time $T_s = 0.1$ second, type

```
H = tf([1 -1], [1 4 5], 'td', 0.35)
Hd = c2d(H, 0.1, 'foh')
```

MATLAB responds:

Transfer function:

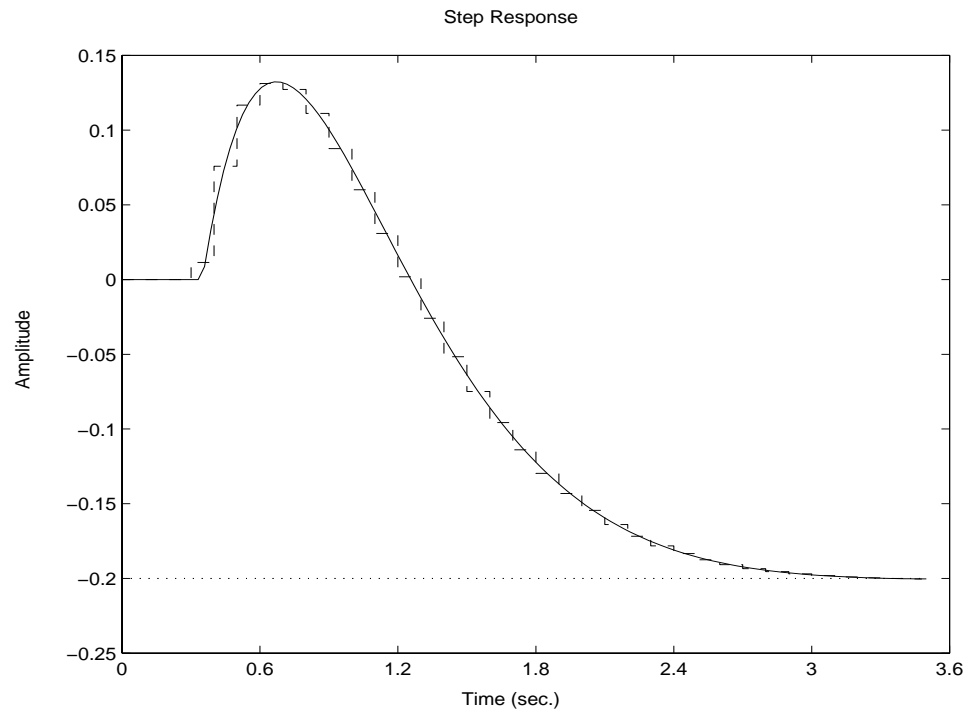
$$0.0115 z^3 + 0.0456 z^2 - 0.0562 z - 0.009104$$

$$z^6 - 1.629 z^5 + 0.6703 z^4$$

Sampling time: 0.1

The next command compares the continuous and discretized step responses:

`step(H, '-', Hd, '--')`



See Also

d2c
d2d

Discrete to continuous conversion
Resampling of discrete systems

References

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

canon

Purpose Compute canonical state-space realizations

Syntax
`csys = canon(sys, 'type')`
`[csys, T] = canon(sys, 'type')`

Description `canon` computes a canonical state-space model for the continuous or discrete LTI system `sys`. Two types of canonical forms are supported:

Modal Form

`csys = canon(sys, 'modal')` returns a realization `csys` in modal form, that is, where the real eigenvalues appear on the diagonal of the A matrix and the complex conjugate eigenvalues appear in 2-by-2 blocks on the diagonal of A . For a system with eigenvalues $(\lambda_1, \sigma \pm j\omega, \lambda_2)$, the modal A matrix is of the form

$$\begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}$$

Companion Form

`csys = canon(sys, 'companion')` produces a companion realization of `sys` where the characteristic polynomial of the system appears explicitly in the rightmost column of the A matrix. For a system with characteristic polynomial

$$p(s) = s^n + a_1 s^{n-1} + \dots + a_{n-1} s + a_n$$

the corresponding companion A matrix is

$$A = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 & -a_n \\ 1 & 0 & 0 & \dots & 0 & -a_{n-1} \\ 0 & 1 & 0 & \dots & \vdots & \vdots \\ \vdots & 0 & \dots & \dots & \vdots & \vdots \\ 0 & \dots & \dots & 1 & 0 & -a_2 \\ 0 & \dots & \dots & 0 & 1 & -a_1 \end{bmatrix}$$

For state-space models `sys`,

```
[csys, T] = canon(a, b, c, d, 'type')
```

also returns the state coordinate transformation T relating the original state vector x and the canonical state vector x_c :

$$x_c = Tx$$

This syntax returns $T=[]$ when `sys` is not a state-space model.

Algorithm

Transfer functions or zero-pole-gain models are first converted to state space using `ss`.

The transformation to modal form uses the matrix P of eigenvectors of the A matrix. The modal form is then obtained as

$$\begin{aligned}\dot{x}_c &= P^{-1}APx_c + P^{-1}Bu \\ y &= CPx_c + Du\end{aligned}$$

The state transformation T returned is the inverse of P .

The reduction to companion form uses a state similarity transformation based on the controllability matrix [1].

Limitations

The modal transformation requires that the A matrix be diagonalizable. A sufficient condition for diagonalizability is that A has no repeated eigenvalues.

The companion transformation requires that the system be controllable from the first input. The companion form is often poorly conditioned for most state-space computations; avoid using it when possible.

See Also

<code>ctrb</code>	Controllability matrix
<code>ctrbf</code>	Controllability canonical form
<code>ss2ss</code>	State similarity transformation

References

[1] Kailath, T. *Linear Systems*, Prentice-Hall, 1980.

care

Purpose

Solve continuous-time algebraic Riccati equations (CARE)

Syntax

`[X, L, G, rr] = care(A, B, Q)`

`[X, L, G, rr] = care(A, B, Q, R, S, E)`

`[X, L, G, report] = care(A, B, Q, ..., 'report')`

`[X1, X2, L, report] = care(A, B, Q, ..., 'implicit')`

Description

`[X, L, G, rr] = care(A, B, Q)` computes the unique solution X of the algebraic Riccati equation:

$$Ric(X) = A^T X + XA - XBB^T X + Q = 0$$

such that $A - BB^T X$ has all its eigenvalues in the open left-half plane. The matrix X is symmetric and called the *stabilizing* solution of $Ric(X) = 0$. Also returned are

- The eigenvalues L of $A - BB^T X$
- The gain matrix $G = B^T X$
- The relative residual rr defined by

$$rr = \frac{\|Ric(X)\|_F}{\|X\|_F}$$

`[X, L, G, rr] = care(A, B, Q, R, S, E)` solves the more general CARE:

$$Ric(X) = A^T XE + E^T XA - (E^T XB + S)R^{-1}(B^T XE + S^T) + Q = 0$$

Here the gain matrix is $G = R^{-1}(B^T XE + S^T)$ and the “closed-loop” eigenvalues are $L = \text{eig}(A - B^*G, E)$.

Two additional syntaxes are provided to help develop applications such as H_∞ -optimal control design:

`[X, L, G, report] = care(A, B, Q, ..., 'report')`

turns off the error messages when the solution X fails to exist and returns a failure report instead.

The value of report is

- -1 when the associated Hamiltonian pencil has eigenvalues on or very near the imaginary axis (failure)
- -2 when there is no finite solution, i.e., $X = X_2 X_1^{-1}$ with X_1 singular (failure)
- The relative residual rr defined above when the solution exists (success)

Alternatively,

```
[X1, X2, L, report] = care(A, B, Q, ..., 'implicit')
```

also turns off error messages but now returns X in implicit form as

$$X = X_2 X_1^{-1}$$

Note that this syntax returns report=0 when successful.

Examples

Example 1

Given

$$A = \begin{bmatrix} -3 & 2 \\ 1 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad C = \begin{bmatrix} 1 & -1 \end{bmatrix} \quad R = 3$$

you can solve the Riccati equation

$$A^T X + X A - X B R^{-1} B^T X + C^T C = 0$$

by

```
a = [-3 2; 1 1]
b = [0 ; 1]
c = [1 -1]
r = 3
[x, l, g] = care(a, b, c' * c, r)
```

This yields the solution

x

$x =$

$$\begin{bmatrix} 0.5895 & 1.8216 \\ 1.8216 & 8.8188 \end{bmatrix}$$

You can verify that this solution is indeed stabilizing by comparing the eigenvalues of a and $a-b^*g$:

$[\text{eig}(a) \quad \text{eig}(a-b^*g)]$

$\text{ans} =$

$$\begin{bmatrix} -3.4495 & -3.5026 \\ 1.4495 & -1.4370 \end{bmatrix}$$

Finally, note that the closed-loop eigenvalues $\text{eig}(a-b^*g)$ coincides with the output l :

l

$l =$

$$\begin{bmatrix} -3.5026 \\ -1.4370 \end{bmatrix}$$

Example 2

To solve the H_∞ -like Riccati equation

$$A^T X + XA + X(\gamma^{-2} B_1 B_1^T - B_2 B_2^T) X + C^T C = 0$$

rewrite it in the care format as

$$A^T X + XA - \underbrace{X \begin{bmatrix} B_1 & B_2 \end{bmatrix}}_B \underbrace{\begin{bmatrix} -\gamma^{-2} I & 0 \\ 0 & I \end{bmatrix}}_R^{-1} \begin{bmatrix} B_1^T \\ B_2^T \end{bmatrix} X + C^T C = 0$$

You can now compute the stabilizing solution X by

```
B = [ B1 , B2 ]
m1 = size(B1, 2)
m2 = size(B2, 2)
R = [-g^2*eye(m1) zeros(m1, m2) ; zeros(m2, m1) eye(m2) ]

X = care(A, B, C' * C, R)
```

Algorithm

care implements the algorithms described in [1]. It works with the Hamiltonian matrix when R is well-conditioned and $E = I$, and uses the extended Hamiltonian pencil and QZ algorithm otherwise.

Limitations

The (A, B) pair must be stabilizable (that is, all unstable modes are controllable). In addition, the associated Hamiltonian matrix or pencil must have no eigenvalue on the imaginary axis. Sufficient conditions for this to hold are (Q, A) detectable when $S = 0$ and $R > 0$, or

$$\begin{bmatrix} Q & S \\ S^T & R \end{bmatrix} > 0$$

See Also

dare	Solve discrete-time Riccati equations
lyap	Solve continuous-time Lyapunov equations

References

[1] W.F. Arnold, III and A.J. Laub, "Generalized Eigenproblem Algorithms and Software for Algebraic Riccati Equations," *Proc. IEEE*, 72 (1984), pp. 1746–1754.

connect

Purpose Derive state-space model from block diagram description

Syntax `sysc = connect(sys, Q, inputs, outputs)`

Description Complex dynamical systems are often given in block diagram form. For systems of even moderate complexity, it can be quite difficult to find the state-space model required in order to bring certain analysis and design tools into use. Starting with a block diagram description, you can use `append` and `connect` to construct a state-space model of the system.

First, use

```
sys = append(sys1, sys2, . . . , sysN)
```

to specify each block `sysj` in the diagram and form a block-diagonal, *unconnected* LTI model `sys` of the diagram.

Next, use

```
sysc = connect(sys, Q, inputs, outputs)
```

to connect the blocks together and derive a state-space model `sysc` for the overall interconnection. The arguments `Q`, `inputs`, and `outputs` have the following purpose:

- The matrix `Q` indicates how the blocks on the diagram are connected. It has a row for each input of `sys`, where the first element of each row is the input number. The subsequent elements of each row specify where the block input gets its summing inputs; negative elements indicate minus inputs to the summing junction. For example, if input 7 gets its inputs from the outputs 2, 15, and 6, where the input from output 15 is negative, the corresponding row of `Q` is `[7 2 -15 6]`. Short rows can be padded with trailing zeros (see example below).
- Given `sys` and `Q`, `connect` computes a state-space model of the interconnection with the same inputs and outputs as `sys` (that is, the concatenation of all block inputs and outputs). The index vectors `inputs` and `outputs` then indicate which of the inputs and outputs in the large *unconnected* system are external inputs and outputs of the block diagram. For example, if the external inputs are inputs 1, 2, and 15 of `sys`, and the

external outputs are outputs 2 and 7 of sys, then inputs and outputs should be set to

```
inputs = [1 2 15];
outputs = [2 7];
```

The final model sysc has these particular inputs and outputs.

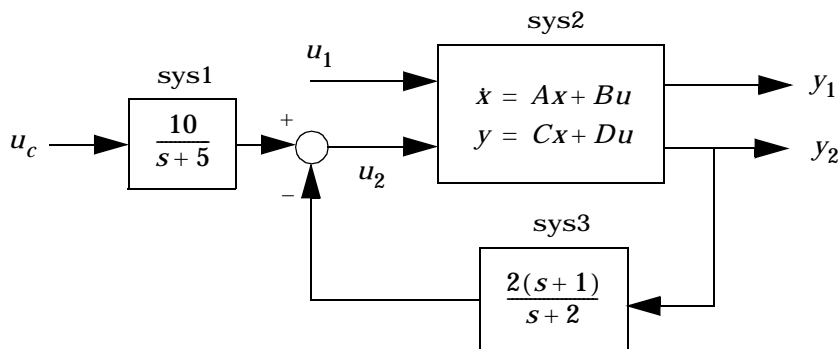
Since it is easy to make a mistake entering all the data required for a large model, be sure to verify your model in as many ways as you can. Here are some suggestions:

- Make sure the poles of the unconnected model sys match the poles of the various blocks in the diagram.
- Check that the final poles and DC gains are reasonable.
- Plot the step and bode responses of sysc and compare them with your expectations.

If you need to work extensively with block diagrams, Simulink is a much easier and more comprehensive tool for model building.

Example

Consider the following block diagram:



Given the matrices of the state-space model `sys2`:

```
A = [ -9.0201  17.7791
       -1.6943   3.2138 ];
B = [ -0.5112   0.5362
       -0.002  -1.8470];
C = [ -3.2897   2.4544
       -13.5009  18.0745];
D = [ -0.5476  -0.1410
       -0.6459   0.2958 ];
```

first define the three blocks as individual LTI models:

```
sys1 = tf(10, [1 5], 'inputname', 'uc')
sys2 = ss(A, B, C, D, 'inputname', {'u1' 'u2'}, ...
          'outputname', {'y1' 'y2'})
sys3 = zpk(-1, -2, 2)
```

and append these blocks to form the unconnected model `sys`:

```
sys = append(sys1, sys2, sys3)
```

This produces the block-diagonal model

`sys`

`a =`

	x1	x2	x3	x4
x1	-5	0	0	0
x2	0	-9.0201	17.779	0
x3	0	-1.6943	3.2138	0
x4	0	0	0	-2

`b =`

	uc	u1	u2	?
x1	4	0	0	0
x2	0	-0.5112	0.5362	0
x3	0	-0.002	-1.847	0
x4	0	0	0	1.4142

```

c =
      x1      x2      x3      x4
      ?      2.5      0      0      0
    y1      0    -3.2897    2.4544      0
    y2      0   -13.501    18.075      0
      ?      0      0      0   -1.4142

```

```

d =
      uc      u1      u2      ?
      ?      0      0      0      0
    y1      0   -0.5476   -0.141      0
    y2      0   -0.6459    0.2958      0
      ?      0      0      0      2

```

Continuous-time system.

Note that the ordering of the inputs and outputs is the same as the block ordering you chose. Unnamed inputs or outputs are denoted by ?.

To derive the overall block diagram model from sys, specify the interconnections and the external inputs and outputs. You need to connect outputs 1 and 4 into input 3 (u2), and output 3 (y2) into input 4. The interconnection matrix Q is therefore

```

Q = [3 1 -4
     4 3 0];

```

Note that the second row of Q has been padded with a trailing zero. The block diagram has two external inputs uc and u1 (inputs 1 and 2 of sys), and two external outputs y1 and y2 (outputs 2 and 3 of sys). Accordingly, set inputs and outputs to:

```

inputs = [1 2];
outputs = [2 3];

```

You can now obtain a state-space model for the overall interconnection by:

```
sysc = connect (sys, Q, inputs, outputs)
```

a =

	x1	x2	x3	x4
x1	- 5	0	0	0
x2	0. 84223	0. 076636	5. 6007	0. 47644
x3	- 2. 9012	- 33. 029	45. 164	- 1. 6411
x4	0. 65708	- 11. 996	16. 06	- 1. 6283

b =

	uc	u1
x1	4	0
x2	0	- 0. 076001
x3	0	- 1. 5011
x4	0	- 0. 57391

c =

	x1	x2	x3	x4
y1	- 0. 22148	- 5. 6818	5. 6568	- 0. 12529
y2	0. 46463	- 8. 4826	11. 356	0. 26283

d =

	uc	u1
y1	0	- 0. 66204
y2	0	- 0. 40582

Continuous-time system.

Note that the inputs and outputs are as desired.

See Also

append	Append LTI systems
series	Series connection
parallel	Parallel connection
feedback	Feedback connection
minimal	Minimal state-space realization

References

[1] Edwards, J.W., "A Fortran Program for the Analysis of Linear Continuous and Sampled-Data Systems," *NASA Report TM X56038*, Dryden Research Center, 1976.

covar

Purpose Output and state covariance of a system driven by white noise

Syntax `[P, Q] = covar(sys, W)`

Description `covar` calculates the stationary covariance of the output y of an LTI model `sys` driven by Gaussian white noise inputs w . This function handles both continuous- and discrete-time cases.

`P = covar(sys, W)` returns the steady-state output response covariance

$$P = E(yy^T)$$

given the noise intensity:

$$E(w(t)w(\tau)^T) = W\delta(t-\tau) \quad (\text{continuous time})$$

$$E(w[k]w[l]^T) = W\delta_{kl} \quad (\text{discrete time})$$

`[P, Q] = covar(sys, W)` also returns the steady-state state covariance

$$Q = E(xx^T)$$

when `sys` is a state-space model (otherwise `Q` is set to `[]`).

Example Compute the output response covariance of the discrete SISO system

$$H(z) = \frac{2z+1}{z^2+0.2z+0.5}, \quad T_s = 0.1$$

due to Gaussian white noise of intensity $W = 5$. Type:

```
sys = tf([2 1], [1 0.2 0.5], 0.1);  
p = covar(sys, 5)
```

and MATLAB returns:

```
p =  
30.3167
```


You can compare with simulation results:

```
randn('seed', 0)
w = sqrt(5)*randn(1, 1000); % 1000 samples

% Simulate response to w with LSIM
y = lsim(sys, w);

% Compute covariance of y values
psim = sum(y .* y) / length(w);
```

This yields

```
psim =
    32.6269
```

The two covariance values p and psim do not agree perfectly due to the finite simulation horizon.

Algorithm

Transfer functions and zero-pole-gain models are first converted to state space with `ss`.

For continuous-time state-space models

$$\begin{aligned} \dot{x} &= Ax + Bw \\ y &= Cx + Dw \end{aligned}$$

Q is obtained by solving the Lyapunov equation

$$AQ + QA^T + BWB^T = 0$$

The output response covariance P is finite only when $D = 0$ and then $P = CQC^T$.

In discrete time, the state covariance solves the discrete Lyapunov equation

$$AQA^T - Q + BWB^T = 0$$

and P is given by

$$P = CQC^T + DWD^T$$

Note that P is well defined for nonzero D in the discrete case.

Limitations	The state and output covariances are defined for <i>stable</i> systems only. For continuous systems, the output response covariance P is finite only when the D matrix is zero (strictly proper system).	
See Also	lyap	Solver for continuous-time Lyapunov equations
	dl yap	Solver for discrete-time Lyapunov equations
References	[1] Bryson, A.E., and Ho, Y.C., <i>Applied Optimal Control</i> , Hemisphere Publishing, 1975, pp. 458-459.	

Purpose Form the controllability matrix

Syntax `Co = ctrb(A, B)`
`Co = ctrb(sys)`

Description `ctrb` computes the controllability matrix for state-space systems. For an n -by- n matrix A and an n -by- m matrix B , `ctrb(A, B)` returns the controllability matrix:

$$Co = \begin{bmatrix} B & AB & A^2B & \dots & A^{n-1}B \end{bmatrix}$$

where Co has n rows and nm columns.

`Co = ctrb(sys)` calculates the controllability matrix of the state-space LTI object `sys`. This syntax is equivalent to executing

`Co = ctrb(sys.A, sys.B)`

The system is controllable if Co has full rank n .

Example Check if the system with:

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

is controllable. Type:

`Co=ctrb(A, B);`

`% Number of uncontrollable states`
`unco=length(A)-rank(Co)`

and MATLAB returns:

`unco =`
`1`

Limitations

The calculation of C_0 may be ill-conditioned with respect to inversion. An indication of this can be seen from the simple example:

$$A = \begin{bmatrix} 1 & \delta \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ \delta \end{bmatrix}$$

This pair is controllable if $\delta \neq 0$ but if $\delta < \sqrt{\text{eps}}$, where *eps* is the relative machine precision, then it is easily seen that `ctrb` returns:

$$\begin{bmatrix} B & AB \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ \delta & \delta \end{bmatrix}$$

from which an erroneous conclusions would be drawn. For cases like these, it is better to determine the controllability of a system using `ctrbf`.

See Also

`ctrbf`
`obsv`

Compute the controllability staircase form
Compute the observability matrix

Purpose	Compute the controllability staircase form
Syntax	$[Abar, Bbar, Cbar, T, k] = ctrbf(A, B, C)$ $[Abar, Bbar, Cbar, T, k] = ctrbf(A, B, C, tol)$
Description	<p>If the controllability matrix of (A, B) has rank $r \leq n$, where n is the size of A, then there exists a similarity transformation such that:</p>

$$\bar{A} = TAT^T, \quad \bar{B} = TB, \quad \bar{C} = CT^T$$

where T is unitary and the transformed system has a *staircase* form with the uncontrollable modes, if any, in the upper left corner:

$$\bar{A} = \begin{bmatrix} A_{uc} & 0 \\ A_{21} & A_c \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} 0 \\ B_c \end{bmatrix}, \quad \bar{C} = \begin{bmatrix} C_{nc} & C_c \end{bmatrix}$$

where $(A_c B_c)$ is controllable, all eigenvalues of A_{uc} are uncontrollable, and

$$C_c(sI - A_c)^{-1}B_c = C(sI - A)^{-1}B.$$

$[Abar, Bbar, Cbar, T, k] = ctrbf(A, B, C)$ decomposes the state-space system represented by A , B , and C into the controllability staircase form, $Abar$, $Bbar$, and $Cbar$, described above. T is the similarity transformation matrix and k is a vector of length n , where n is the order of the system represented by A . Each entry of k represents the number of controllable states factored out during each step of the transformation matrix calculation. The number of nonzero elements in k indicates how many iterations were necessary to calculate T , and $\text{sum}(k)$ is the number of states in A_c , the controllable portion of $Abar$.

$ctrbf(A, B, C, tol)$ uses the tolerance tol when calculating the controllable/uncontrollable subspaces. When the tolerance is not specified, it defaults to $10*n*\text{norm}(A, 1)*\text{eps}$.

Example

Compute the controllability staircase form for

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and locate the uncontrollable mode.

$$[Abar, Bbar, Cbar, T, k] = ctrbf(A, B, C)$$

$$Abar = \begin{bmatrix} -3.0000 & 0 \\ -3.0000 & 2.0000 \end{bmatrix}$$

$$Bbar = \begin{bmatrix} 0.0000 & 0.0000 \\ 1.4142 & -1.4142 \end{bmatrix}$$

$$Cbar = \begin{bmatrix} -0.7071 & 0.7071 \\ 0.7071 & 0.7071 \end{bmatrix}$$

$$T = \begin{bmatrix} -0.7071 & 0.7071 \\ 0.7071 & 0.7071 \end{bmatrix}$$

$$k = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

The decomposed system Abar shows an uncontrollable mode located at -3 and a controllable mode located at 2 .

See also the function `mi nreal`, which uses `ctrbf` to find the minimal realization of a system.

Algorithm	ctrbf is an M-file that implements the Staircase Algorithm of [1].	
See Also	ctrb	Form the controllability matrix
	mi nreal	Minimum realization and pole-zero cancellation
References	[1] Rosenbrock, M.M., <i>State-Space and Multivariable Theory</i> , John Wiley, 1970.	

Purpose Convert discrete-time LTI models to continuous time

Syntax

```
sysc = d2c(sysd)  
sysc = d2c(sysd, method)
```

Description d2c converts LTI models from discrete to continuous time using one of the following conversion methods:

'zoh'	Zero-order hold on the inputs. The control inputs are assumed piecewise constant over the sampling period.
'tustin'	Bilinear (Tustin) approximation to the derivative.
'prewarp'	Tustin approximation with frequency prewarping.
'matched'	Matched pole-zero method of [1] (for SISO systems only).

The string *method* specifies the conversion method. If *method* is omitted then zero-order hold ('zoh') is assumed. See “Continuous/Discrete Conversions” on page 2-39 of this manual and reference [1] for more details on the conversion methods.

Example Consider the discrete-time model with transfer function

$$H(z) = \frac{z - 1}{z^2 + z + 0.3}$$

and sample time $T_s = 0.1$ second. You can derive a continuous-time zero-order-hold equivalent model by typing:

```
Hc = d2c(H)
```

Discretizing the resulting model Hc with the zero-order hold method (this is the default method) and sampling period $T_s = 0.1$ gives back the original discrete model $H(z)$. To see this, type:

```
c2d(Hc, 0.1)
```

To use the Tustin approximation instead of zero-order hold, type

```
Hc = d2c(H, 'tustin')
```


As with zero-order hold, the inverse discretization operation

```
c2d(Hc, 0.1, 'tustin')
```

gives back the original $H(z)$.

Algorithm

The 'zoh' conversion is performed in state space and relies on the matrix logarithm (see `logm` in *Using MATLAB*).

Limitations

The Tustin approximation is not defined for systems with poles at $z = -1$ and is ill-conditioned for systems with poles near $z = -1$.

The zero-order hold method cannot handle systems with poles at $z = 0$. In addition, the 'zoh' conversion increases the model order for systems with negative real poles, [2]. This is necessary because the matrix logarithm maps real negative poles to complex poles. As a result, a discrete model with a single pole at $z = -0.5$ would be transformed to a continuous model with a single *complex* pole at $\log(-0.5) \approx -0.6931 + j\pi$. Such a model is not meaningful because of its complex time response.

To ensure that all complex poles of the continuous model come in conjugate pairs, d2c replaces negative real poles $z = -\alpha$ with a pair of complex conjugate poles near $-\alpha$. The conversion then yields a continuous model with higher order. For example, the discrete model with transfer function

$$H(z) = \frac{z + 0.2}{(z + 0.5)(z^2 + z + 0.4)}$$

and sample time 0.1 second is converted by typing:

```
Ts = 0.1
H = zpkm(-0.2, -0.5, 1, Ts) * tf(1, [1 1 0.4], Ts)
Hc = d2c(H)
```

MATLAB responds with:

Warning: System order was increased to handle real negative poles.

Zero/pole/gain:

```
-33.6556 (s-6.273) (s^2 + 28.29s + 1041)
-----
(s^2 + 9.163s + 637.3) (s^2 + 13.86s + 1035)
```

Converting H_c back to discrete time by typing:

`c2d(Hc, Ts)`

yielding

Zero/pole/gain:

$(z+0.5) (z+0.2)$

 $(z+0.5)^2 (z^2 + z + 0.4)$

Sampling time: 0.1

This discrete model coincides with $H(z)$ after canceling the pole/zero pair at $z = -0.5$.

See Also

`c2d`

Continuous- to discrete-time conversion

`d2d`

Resampling of discrete models

`logm`

Matrix logarithm

References

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

[2] Kollár, I., G.F. Franklin, and R. Pintelon, "On the Equivalence of z-domain and s-domain Models in System Identification," *Proceedings of the IEEE Instrumentation and Measurement Technology Conference*, Brussels, Belgium, June, 1996, Vol. 1, pp. 14-19.

Purpose	Resample discrete-time LTI models or add input delays
Syntax	<pre>sys1 = d2d(sys, Ts) sys1 = d2d(sys, [], Nd)</pre>
Description	<p><code>sys1 = d2d(sys, Ts)</code> resamples the discrete-time LTI model <code>sys</code> to produce an equivalent discrete-time model <code>sys1</code> with the new sample time <code>Ts</code> (in seconds). The resampling assumes zero-order hold on the inputs and is equivalent to consecutive <code>d2c</code> and <code>c2d</code> conversions:</p> $\text{sys1} = \text{c2d}(\text{d2c}(\text{sys}), \text{Ts})$ <p><code>sys1 = d2d(sys, [], Nd)</code> adds input delays to the discrete model <code>sys</code>. The delay times must be integer multiples <code>Nd</code> of the sample time. A scalar integer <code>Nd</code> specifies a uniform input delay. A vector of integer <code>Nd</code> specifies independent delays for each input channel (the length of <code>Nd</code> must then match the number of inputs). The delays give rise to additional poles at $z = 0$ and the resulting LTI model <code>sys1</code> is delay free.</p>
Example	<p>Consider the zero-pole-gain model</p> $H(z) = \frac{z - 0.7}{z - 0.5}$ <p>with sample time 0.1 second. You can resample this model at 0.05 second by typing:</p> <pre>H = zp(0.7, 0.5, 1, 0.1) H2 = d2d(H, 0.05)</pre> <p>MATLAB responds with:</p> <pre>Zero/pole/gain: (z-0.8243) ----- (z-0.7071) Sampling time: 0.05</pre>

Note that the inverse resampling operation, performed by typing `d2d(H2, 0.1)`, yields back the initial model $H(z)$:

```
Zero/pole/gain:
(z-0.7)
-----
(z-0.5)
```

Sampling time: 0.1

You can also delay the input of $H(z)$ by three sampling periods using the syntax:

```
hd3 = d2d(H, [], 3)
```

which yields:

```
Zero/pole/gain:
(z-0.7)
-----
z^3 (z-0.5)
```

Sampling time: 0.1

Note that the integer 3 means three times the current sample time, not 3 seconds.

See Also

`c2d`
`d2c`

Continuous- to discrete-time conversion
Discrete- to continuous-time conversion

Purpose	Compute damping factors and natural frequencies
Syntax	$[W_n, Z] = \text{damp}(\text{sys})$ $[W_n, Z, P] = \text{damp}(\text{sys})$
Description	<p>damp calculates the damping factor and natural frequencies of the poles of an LTI model sys. When invoked without lefthand arguments, a table of the eigenvalues in increasing frequency, along with their damping factors and natural frequencies, is displayed on the screen.</p> <p>$[W_n, Z] = \text{damp}(\text{sys})$ returns column vectors W_n and Z containing the natural frequencies ω_n and damping factors ζ of the poles of sys. For discrete-time systems with poles z and sample time T_s, damp computes “equivalent” continuous-time poles s by solving</p> $z = e^{sT_s}$ <p>The values W_n and Z are then relative to the continuous-time poles s. Both W_n and Z are empty if the sample time is unspecified.</p> <p>$[W_n, Z, P] = \text{damp}(\text{sys})$ returns an additional vector P containing the (true) poles of sys. Note that P coincides with the result of $P = \text{pole}(\text{sys})$.</p>
Example	<p>Compute and display the eigenvalues, natural frequencies, and damping factors of the continuous transfer function:</p> $H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$ <p>Type:</p> $H = \text{tf}([2 \ 5 \ 1], [1 \ 2 \ 3])$ <p>an MATLAB returns:</p> <p>Transfer function:</p> $\frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$

damp

Type

damp(H)

and MATLAB returns:

Ei genval ue	Dampi ng	Freq. (rad/s)
−1. 00e+000 + 1. 41e+000i	5. 77e−001	1. 73e+000
−1. 00e+000 − 1. 41e+000i	5. 77e−001	1. 73e+000

See Also

eig	Calculate eigenvalues and eigenvectors
esort, dsort	Sort system poles
tzero	Compute (transmission) zeros
pole	Compute system poles
pzmap	Pole-zero map

Purpose Solve discrete-time algebraic Riccati equations (DARE)

Syntax $[X, L, G, rr] = \text{dare}(A, B, Q, R)$
 $[X, L, G, rr] = \text{dare}(A, B, Q, R, S, E)$

$[X, L, G, \text{report}] = \text{dare}(A, B, Q, \dots, ' \text{report}')$
 $[X1, X2, L, \text{report}] = \text{dare}(A, B, Q, \dots, ' \text{implicit}')$

Description $[X, L, G, rr] = \text{dare}(A, B, Q, R)$ computes the unique solution X of the discrete-time algebraic Riccati equation

$$Ric(X) = A^T X A - X - A^T X B (B^T X B + R)^{-1} B^T X A + Q = 0$$

such that the “closed-loop” matrix

$$A_{cl} = A - B (B^T X B + R)^{-1} B^T X A$$

has all its eigenvalues inside the unit disk. The matrix X is symmetric and called the *stabilizing* solution of $Ric(X) = 0$. Also returned are

- The eigenvalues L of A_{cl}
- The gain matrix

$$G = (B^T X B + R)^{-1} B^T X A$$

- The relative residual rr defined by

$$rr = \frac{\|Ric(X)\|_F}{\|X\|_F}$$

$[X, L, G, rr] = \text{dare}(A, B, Q, R, S, E)$ solves the more general DARE:

$$A^T X A - E^T X E - (A^T X B + S)(B^T X B + R)^{-1} (B^T X A + S^T) + Q = 0$$

The corresponding gain matrix and “closed-loop” eigenvalues are

$$G = (B^T X B + R)^{-1} (B^T X A + S^T)$$

and $L = \text{eig}(A - B * G, E)$.

Two additional syntaxes are provided to help develop applications such as H_∞ -optimal control design:

`[X, L, G, report] = dare(A, B, Q, ..., 'report')`

turns off the error messages when the solution X fails to exist and returns a failure report instead. The value of report is

- -1 when the associated symplectic pencil has eigenvalues on or very near the unit circle (failure)
- -2 when there is no finite solution, that is, $X = X_2 X_1^{-1}$ with X_1 singular (failure)
- The relative residual rr defined above when the solution exists (success)

Alternatively,

`[X1, X2, L, report] = dare(A, B, Q, ..., 'implicit')`

also turns off error messages but now returns X in implicit form as

$$X = X_2 X_1^{-1}$$

Note that this syntax returns report=0 when successful.

Algorithm

dare implements the algorithms described in [1]. It uses the QZ algorithm to deflate the extended symplectic pencil and compute its stable invariant subspace.

Limitations

The (A, B) pair must be stabilizable (that is, all eigenvalues of A outside the unit disk must be controllable). In addition, the associated symplectic pencil must have no eigenvalue on the unit circle. Sufficient conditions for this to hold are (Q, A) detectable when $S = 0$ and $R > 0$, or

$$\begin{bmatrix} Q & S \\ S^T & R \end{bmatrix} > 0$$

See Also

care
dl yap

Solve continuous-time Riccati equations
Solve discrete-time Lyapunov equations

References

- [1] Arnold, W.F., III and A.J. Laub, "Generalized Eigenproblem Algorithms and Software for Algebraic Riccati Equations," *Proc. IEEE*, 72 (1984), pp. 1746–1754.

dcgain

Purpose Compute low frequency (DC) gain of LTI system

Syntax `k = dcgain(sys)`

Description `k = dcgain(sys)` computes the DC gain `k` of the LTI model `sys`.

Continuous Time

The continuous-time DC gain is the transfer function value at the frequency $s = 0$. For state-space models with matrices (A, B, C, D) , this value is

$$K = D - CA^{-1}B$$

Discrete Time

The discrete-time DC gain is the transfer function value at $z = 1$. For state-space models with matrices (A, B, C, D) , this value is

$$K = D + C(I - A)^{-1}B$$

The DC gain is infinite for systems with integrators.

Example To compute the DC gain of the MIMO transfer function

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

type:

```
H = [1 tf([1 -1],[1 1 3]) ; tf(1,[1 1]) tf([1 2],[1 -3])]
dcgain(H)
```

and MATLAB returns:

```
ans =
    1.0000    -0.3333
    1.0000    -0.6667
```

See Also

`norm`
`eval fr`

LTI system norms
Evaluates frequency response at single frequency

Purpose Design linear-quadratic (LQ) state-feedback regulator for discrete-time plant

Syntax $[K, S, e] = \text{dlqr}(a, b, Q, R)$
 $[K, S, e] = \text{dlqr}(a, b, Q, R, N)$

Description $[K, S, e] = \text{dlqr}(a, b, Q, R, N)$ calculates the optimal gain matrix K such that the state-feedback law

$$u[n] = -Kx[n]$$

minimizes the quadratic cost function

$$J(u) = \sum_{n=1}^{\infty} (x[n]^T Q x[n] + u[n]^T R u[n] + 2x[n]^T N u[n])$$

for the discrete-time state-space model

$$x[n+1] = Ax[n] + Bu[n]$$

The default value $N=0$ is assumed when N is omitted.

In addition to the state-feedback gain K , dlqr returns the solution S of the associated discrete-time Riccati equation

$$A^T S A - S - (A^T S B + N)(B^T X B + R)^{-1} (B^T S A + N^T) + Q = 0$$

and the closed-loop eigenvalues $e = \text{eig}(a - b * K)$. Note that K is derived from S by

$$K = (B^T X B + R)^{-1} (B^T S A + N^T)$$

Limitations The problem data must satisfy:

- the pair (A, B) is stabilizable
- $R > 0$ and $Q - NR^{-1}N^T \geq 0$
- $(Q - NR^{-1}N^T, A - BR^{-1}N^T)$ has no unobservable mode on the unit circle

See Also

lqr
lqry
lqrd
lqgreg
dare

State-feedback LQ regulator for continuous plant
State-feedback LQ regulator with output weighting
Discrete LQ regulator for continuous plant
LQG regulator
Solve discrete Riccati equations

dlyap

Purpose Solve discrete-time Lyapunov equations

Syntax `X = dlyap(A, Q)`

Description `dlyap` solves the discrete-time Lyapunov equation

$$A^T X A - X + Q = 0$$

where A and Q are n -by- n matrices.

The solution X is symmetric when Q is symmetric, and positive definite when Q is positive definite and A has all its eigenvalues inside the unit disk.

Diagnostics The discrete-time Lyapunov equation has a (unique) solution if the eigenvalues $\alpha_1, \alpha_2, \dots, \alpha_n$ of A satisfy

$$\alpha_i \alpha_j \neq 1 \quad \text{for all pairs } (i, j)$$

If this condition is violated, `dlyap` produces the error message

Solution does not exist or is not unique.

See Also `lyap` Solve continuous Lyapunov equations
`covar` Covariance of system response to white noise

Purpose	Generate stable random discrete test models
Syntax	<pre>sys = drss(n) sys = drss(n, p) sys = drss(n, p, m) [num, den] = drmodel(n) [A, B, C, D] = drmodel(n) [A, B, C, D] = drmodel(n, p, m)</pre>
Description	<p><code>sys = drss(n)</code> produces a random nth order stable model with one input and one output, and returns the model in the state-space object <code>sys</code>.</p> <p><code>drss(n, p)</code> produces a random nth order stable model with one input and p outputs.</p> <p><code>drss(n, m, p)</code> generates a random nth order stable model with m inputs and p outputs.</p> <p>In all cases, the discrete model returned by <code>drss</code> has an unspecified sampling time and is in state-space form. To generate transfer function or zero-pole-gain systems, convert <code>sys</code> using <code>tf</code> or <code>zpk</code>.</p> <p><code>drmodel(n)</code> produces a random nth order stable model and returns either the transfer function numerator <code>num</code> and denominator <code>den</code> or the state-space matrices <code>A</code>, <code>B</code>, <code>C</code>, and <code>D</code>, based on the number of output arguments. The resulting model always has one input and one output.</p> <p><code>[A, B, C, D] = drmodel(n, m, p)</code> produces a random nth order stable state-space model with m inputs and p outputs.</p>

Example Generate a random discrete LTI system with three states, two inputs, and two outputs.

```
sys = drss(3, 2, 2)

a =
      x1      x2      x3
x1  0.38630 -0.21458 -0.09914
x2 -0.23390 -0.15220 -0.06572
x3 -0.03412  0.11394 -0.22618

b =
      u1      u2
x1  0.98833  0.51551
x2  0         0.33395
x3  0.42350  0.43291

c =
      x1      x2      x3
y1  0.22595  0.76037  0
y2  0         0         0

d =
      u1      u2
y1  0         0.68085
y2  0.78333  0.46110

Sampling time: unspecified
Discrete-time system.
```

See Also `rmodel`, `rss` Generate stable random continuous test models
 `tf` Convert LTI systems to transfer functions form
 `zpk` Convert LTI systems to zero-pole-gain form

Purpose Sort discrete-time poles by magnitude

Syntax `s = dsort(p)`
`[s, ndx] = dsort(p)`

Description `dsort` sorts the discrete-time poles contained in the vector `p` in descending order by magnitude. Unstable poles appear first.

When called with one lefthand argument, `dsort` returns the sorted poles in `s`.

`[s, ndx] = dsort(p)` also returns the vector `ndx` containing the indices used in the sort.

Example Sort the following discrete poles:

```
p =
-0.2410 + 0.5573i
-0.2410 - 0.5573i
0.1503
-0.0972
-0.2590
```

```
s = dsort(p)
```

```
s =
-0.2410 + 0.5573i
-0.2410 - 0.5573i
-0.2590
0.1503
-0.0972
```

Limitations The poles in the vector `p` must appear in complex conjugate pairs.

See Also	<code>ei g</code>	Calculate eigenvalues and eigenvectors
	<code>esort, sort</code>	Sort system poles
	<code>tzero</code>	Compute (transmission) zeros
	<code>pol e</code>	Compute system poles
	<code>pzmap</code>	Pole-zero map

Purpose Specify descriptor state-space models

Syntax

```
sys = dss(a, b, c, d, e)
sys = dss(a, b, c, d, e, Ts)
sys = dss(a, b, c, d, e, lti sys)

sys = dss(a, b, c, d, e, 'Property1', Value1, ..., 'PropertyN', ValueN)
sys = dss(a, b, c, d, e, Ts, 'Property1', Value1, ..., 'PropertyN', ValueN)
```

Description `sys = dss(a, b, c, d, e)` creates the continuous-time descriptor state-space model

$$Ex = Ax + Bu$$

$$y = Cx + Du$$

The E matrix must be nonsingular. The output `sys` is an SS object storing the model data (see page 2-2). Note that `ss` produces the same type of object. If $D = 0$, you need not dimension `d` and can simply set `d` to the scalar 0 (zero).

`sys = dss(a, b, c, d, e, Ts)` creates the discrete-time descriptor model

$$Ex[n+1] = Ax[n] + Bu[n]$$

$$y[n] = Cx[n] + Du[n]$$

with sample time `Ts` (in seconds).

`sys = dss(a, b, c, d, e, lti sys)` creates a descriptor model with generic LTI properties inherited from the LTI model `lti sys` (including the sample time). See page 2-17 for an overview of generic LTI properties.

Any of the previous syntaxes can be followed by property name/property value pairs

'Property', Value

Each pair specifies a particular LTI property of the model, for example, the input names or some notes on the model history. See `set` and the example below for details.

Example

The command

```
sys = dss(1, 2, 3, 4, 5, 'td', 0.1, 'inputname', 'voltage', ...  
         'notes', 'Just an example')
```

creates the model

$$\begin{aligned}5\dot{x} &= x + 2u \\ y &= 3x + 4u\end{aligned}$$

with a 0.1 second input delay. The input is labeled 'voltage', and a note is attached to tell you that this is just an example.

See Also

dssdata	Retrieve A, B, C, D, E matrices of descriptor model
ss	Specify (regular) state-space models
set	Set properties of LTI models
get	Get properties of LTI models

dssdata

Purpose	Quick access to descriptor state-space data	
Syntax	<pre>[a, b, c, d, e] = dssdata(sys) [a, b, c, d, e, Ts, Td] = dssdata(sys)</pre>	
Description	<p>[a, b, c, d, e] = dssdata(sys) extracts the descriptor matrix data (A, B, C, D, E) from the state-space model <code>sys</code>. If <code>sys</code> is a transfer function or zero-pole-gain model, it is first converted to state space. Note that <code>dssdata</code> is then equivalent to <code>ssdata</code> because it always returns $E = I$.</p> <p>[a, b, c, d, e, Ts, Td] = dssdata(sys) also returns the sample time <code>Ts</code> and the input delay data <code>Td</code>. For continuous models, <code>Td</code> is a row vector with one entry per input channel (<code>Td(j)</code> indicates by how many seconds the jth input is delayed). For discrete models, <code>Td</code> is the empty matrix <code>[]</code> (see <code>d2d</code> for delays in discrete-time systems).</p> <p>You can access the remaining LTI properties of <code>sys</code> with <code>get</code> or by direct referencing, for example,</p> <pre>sys.notes</pre>	
See Also	<div>dss get ssdata tfdata</div>	<div>Specify descriptor state-space models Get properties of LTI models Quick access to state-space data Quick access to transfer function data Quick access to zero-pole-gain data</div>

Purpose	Compute the poles of an LTI model	
Syntax	<code>poles = eig(sys)</code>	
Description	The overloaded <code>eig</code> computes the poles of an LTI model. It is identical to the function <code>pole</code> .	
See Also	<code>pole</code> <code>damp</code> <code>esort, dsort</code> <code>tzero</code> <code>pzmap</code>	Compute system poles Damping and natural frequency of system poles Sort system poles Compute (transmission) zeros Pole-zero map

esort

Purpose Sort continuous-time poles by real part

Syntax `s = esort(p)`
`[s, ndx] = esort(p)`

Description `esort` sorts the continuous-time poles contained in the vector `p` by real part. Unstable eigenvalues appear first and the remaining poles are ordered by decreasing real parts.

When called with one lefthand argument, `esort` returns the sorted eigenvalues in `s`.

`[s, ndx] = esort(p)` also returns the vector `ndx` containing the indices used in the sort.

Example Sort the following continuous eigenvalues.

```
p
p =
-0.2410+ 0.5573i
-0.2410- 0.5573i
 0.1503
-0.0972
-0.2590
```

```
esort(p)
```

```
ans =
 0.1503
-0.0972
-0.2410+ 0.5573i
-0.2410- 0.5573i
-0.2590
```

Limitations The eigenvalues in the vector `p` must appear in complex conjugate pairs.

See Also

- `ei g`
- `dsort, sort`
- `tzero`
- `pol e`
- `pzmap`
- Calculate eigenvalues and eigenvectors
- Sort system poles
- Compute (transmission) zeros
- Compute system poles
- Pole-zero map

estim

Purpose Form state estimator given estimator gain

Syntax
`est = estim(sys, L)`
`est = estim(sys, L, sensors, known)`

Description `est = estim(sys, L)` produces a state/output estimator `est` given the plant state-space model `sys` and the estimator gain `L`. All inputs w of `sys` are assumed stochastic (process and/or measurement noise), and all outputs y are measured. The estimator `est` is returned in state-space form (SS object). For a continuous-time plant `sys` with equations

$$\dot{x} = Ax + Bw$$

$$y = Cx + Dw$$

`est` generates plant output and state estimates \hat{y} and \hat{x} by:

$$\dot{\hat{x}} = A\hat{x} + L(y - C\hat{x})$$

$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} \hat{x}$$

The discrete-time estimator has similar equations.

`est = estim(sys, L, sensors, known)` handles more general plants `sys` with both known inputs u and stochastic inputs w , and both measured outputs y and nonmeasured outputs z :

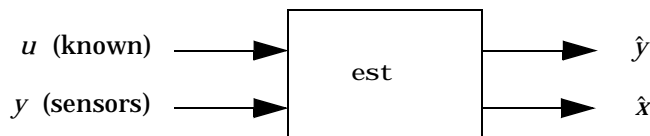
$$\dot{x} = Ax + B_1 w + B_2 u$$

$$\begin{bmatrix} \dot{z} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} x + \begin{bmatrix} D_{11} \\ D_{21} \end{bmatrix} w + \begin{bmatrix} D_{12} \\ D_{22} \end{bmatrix} u$$

The index vectors `sensors` and `known` specify which outputs y are measured and which inputs u are known. The resulting estimator `est` uses both u and y to produce the output and state estimates:

$$\dot{\hat{x}} = A\hat{x} + B_2u + L(y - C_2\hat{x} - D_{22}u)$$

$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C_2 \\ I \end{bmatrix} \hat{x} + \begin{bmatrix} D_{22} \\ 0 \end{bmatrix} u$$



`estim` handles both continuous- and discrete-time cases. You can use the functions `place` (pole placement) or `kalman` (Kalman filtering) to design an adequate estimator gain L . Note that the estimator poles (eigenvalues of $A - LC$) should be faster than the plant dynamics (eigenvalues of A) to ensure accurate estimation.

Example

Consider a state-space model `sys` with seven outputs and four inputs. Suppose you designed a Kalman gain matrix L using outputs 4, 7, and 1 of the plant as sensor measurements, and inputs 1, 4, and 3 of the plant as known (deterministic) inputs. You can then form the Kalman estimator by

```
sensors = [4, 7, 1];
known = [1, 4, 3];
est = estim(sys, L, sensors, known)
```

See the function `kalman` for direct Kalman estimator design.

See Also

<code>reg</code>	Form regulator given state-feedback and estimator gains
<code>place</code>	Pole placement
<code>kalman</code>	Design Kalman estimator

evalfr

Purpose Evaluate frequency response at a single (complex) frequency

Syntax `frsp = evalfr(sys, f)`

Description `frsp = evalfr(sys, f)` evaluates the transfer function of the LTI model `sys` at the complex number `f`. For state-space models with data (A, B, C, D) , the result is

$$H(f) = D + C(fI - A)^{-1}B$$

`evalfr` is a simplified version of `freqresp` meant for quick evaluation of the response at a single point. Use `freqresp` to compute the frequency response over a set of frequencies.

Example To evaluate the discrete-time transfer function

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

at $z = 1+j$, type

```
H = tf([1 -1], [1 1 1], -1)
z = 1+j
evalfr(H, z)
```

and MATLAB responds with:

```
ans =
    2.3077e-01 + 1.5385e-01i
```

Limitations The response is not finite when `f` is a pole of `sys`.

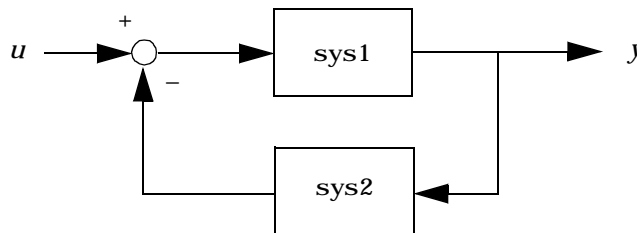
See Also	<code>freqresp</code>	Frequency response over a set of frequencies
	<code>bode</code>	Bode frequency response
	<code>sigma</code>	Singular value response

Purpose Feedback connection of two LTI models

Syntax

```
sys = feedback(sys1, sys2)
sys = feedback(sys1, sys2, sign)
sys = feedback(sys1, sys2, feedin, feedout, sign)
```

Description `sys = feedback(sys1, sys2)` returns an LTI model `sys` for the negative feedback interconnection



The closed-loop model `sys` has `u` as input vector and `y` as output vector. The LTI models `sys1` and `sys2` must be both continuous or both discrete with identical sample times. Precedence rules are used to determine the resulting model type (see page 2-3).

To apply positive feedback, use the syntax

```
sys = feedback(sys1, sys2, +1)
```

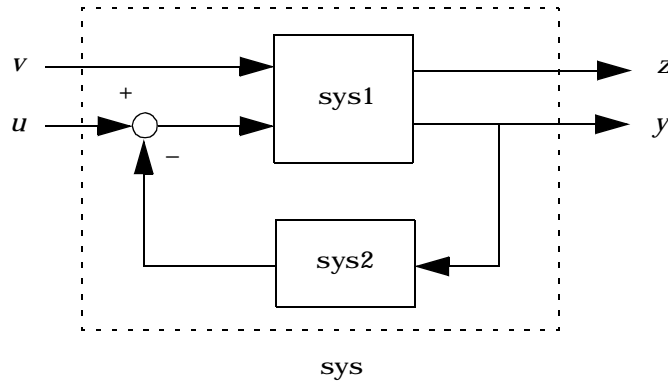
By default, `feedback(sys1, sys2)` assumes negative feedback and is equivalent to `feedback(sys1, sys2, -1)`.

Finally,

```
sys = feedback(sys1, sys2, feedin, feedout)
```

feedback

computes a closed-loop model `sys` for the more general feedback loop:



The vector `feedin` contains indices into the input vector of `sys1` and specifies which inputs `u` are involved in the feedback loop. Similarly, `feedout` specifies which outputs `y` of `sys1` are used for feedback. The resulting LTI model `sys` has the same inputs and outputs as `sys1` (with their order preserved). As before, negative feedback is applied by default and you must use

```
sys = feedback(sys1, sys2, feedin, feedout, +1)
```

to apply positive feedback.

For more complicated feedback structures, use `append` and `connect`.

Remark

You can specify static gains as regular matrices, for example,

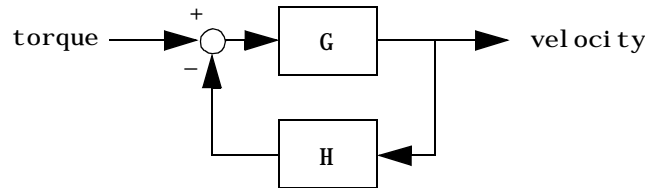
```
sys = feedback(sys1, 2)
```

However, at least one of the two arguments `sys1` and `sys2` should be an LTI object. For feedback loops involving two static gains `k1` and `k2`, use the syntax

```
sys = feedback(tf(k1), k2)
```

Examples

Example 1



To connect the plant

$$G(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

with the controller

$$H(s) = \frac{5(s+2)}{s+10}$$

using negative feedback, type

```
G = tf([2 5 1],[1 2 3], 'inputname', 'torque', ...
        'outputname', 'velocity');
H = zp(-2, -10, 5)
Cl oop = feedback(G, H)
```

and MATLAB returns

```
Zero/pole/gain from input "torque" to output "velocity":
0.18182 (s+10) (s+2.281) (s+0.2192)
-----
(s+3.419) (s^2 + 1.763s + 1.064)
```

The result is a zero-pole-gain model as expected from the precedence rules. Note that Cl oop inherited the input and output names from G.

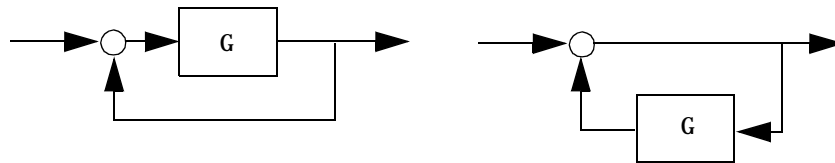
Example 2

Consider a state-space plant P with five inputs and four outputs and a state-space feedback controller K with two inputs and three outputs. To connect outputs 1, 3, and 4 of the plant to the controller inputs, and the controller outputs to inputs 4 and 2 of the plant, use

```
feedin = [4 2];  
feedout = [1 3 4];  
Cl loop = feedback(P, K, feedin, feedout)
```

Example 3

You can form the following negative-feedback loops



by

```
Cl loop = feedback(G, 1)    % left di agram  
Cl loop = feedback(1, G)    % right di agram
```

Limitations

The feedback connection should be free of algebraic loop. If D_1 and D_2 are the feedthrough matrices of sys1 and sys2 , this condition is equivalent to:

- $I + D_1 D_2$ nonsingular when using negative feedback
- $I - D_1 D_2$ nonsingular when using positive feedback.

See Also

star	Star product of LTI systems (LFT connection)
series	Series connection
parallel	Parallel connection
connect	Derive state-space model for block diagram interconnection

Purpose	Specify discrete transfer functions in DSP format
Syntax	<pre>sys = filt(num, den) sys = filt(num, den, Ts) sys = filt(M)</pre> <pre>sys = filt(num, den, 'Property1', Value1, ..., 'PropertyN', ValueN) sys = filt(num, den, Ts, 'Property1', Value1, ..., 'PropertyN', ValueN)</pre>
Description	<p>In digital signal processing (DSP), it is customary to write transfer functions as rational expressions in z^{-1} and to order the numerator and denominator terms in <i>ascending</i> powers of z^{-1}, for example,</p> $H(z^{-1}) = \frac{2 + z^{-1}}{1 + 0.4z^{-1} + 2z^{-2}}$ <p>The function <code>filt</code> is provided to facilitate the specification of transfer functions in DSP format.</p> <p><code>sys = filt(num, den)</code> creates a discrete-time transfer function <code>sys</code> with numerator(s) <code>num</code> and denominator(s) <code>den</code>. The sample time is left unspecified (<code>sys.Ts = -1</code>) and the output <code>sys</code> is a TF object.</p> <p><code>sys = filt(num, den, Ts)</code> further specifies the sample time <code>Ts</code> (in seconds).</p> <p><code>sys = filt(M)</code> specifies a static filter with gain matrix <code>M</code>.</p> <p>Any of the previous syntaxes can be followed by property name/property value pairs of the form:</p> <p style="padding-left: 40px;">'Property', Value</p> <p>Each pair specifies a particular LTI property of the model, for example, the input names or the transfer function variable. See page 2-18 and the <code>set</code> entry for additional information on LTI properties and admissible property values.</p>
Arguments	<p>For SISO transfer functions, <code>num</code> and <code>den</code> are row vectors containing the numerator and denominator coefficients ordered in ascending powers of z^{-1}. For example, <code>den = [1 0.4 2]</code> represents the polynomial $1 + 0.4z^{-1} + 2z^{-2}$.</p>

MIMO transfer functions are regarded as arrays of SISO transfer functions (one per I/O channel), each of which is characterized by its numerator and denominator. The input arguments `num` and `den` are then cell arrays of row vectors such that:

- `num` and `den` have as many rows as outputs and as many columns as inputs.
- Their (i, j) entries `num{i, j}` and `den{i, j}` specify the numerator and denominator of the transfer function from input j to output i .

If all SISO entries have the same denominator, you can also set `den` to the row vector representation of this common denominator. See also page 2-31 for alternative ways to specify MIMO transfer functions.

Remark

`filt` behaves as `tf` with the `Variable` property set to ' z^{-1} ' or ' q '. See `tf` entry below for details.

Example

Typing the commands

```
num = {1, [1 0.3]}
den = {[1 1 2], [5 2]}
H = filt(num, den, 'inputname', {'channel 1' 'channel 2'})
```

creates the two-input digital filter

$$H(z^{-1}) = \begin{bmatrix} \frac{1}{1 + z^{-1} + 2z^{-2}} & \frac{1 + 0.3z^{-1}}{5 + 2z^{-1}} \end{bmatrix}$$

with unspecified sample time and input names 'channel 1' and 'channel 2'.

See Also

<code>tf</code>	Create transfer functions
<code>zpk</code>	Create zero-pole-gain models
<code>ss</code>	Create state-space models

Purpose	Compute frequency response over grid of frequencies
Syntax	$H = \text{freqresp}(\text{sys}, w)$
Description	<p>$H = \text{freqresp}(\text{sys}, w)$ computes the frequency response of the LTI model sys at the real frequency points specified by the vector w. The frequencies must be in radians/sec. The frequency response is returned as a 3-D array H with the frequency as the last dimension (see “Arguments” below).</p> <p>In continuous time, the response at a frequency ω is the transfer function value at $s = j\omega$. For state-space models, this value is given by</p> $H(j\omega) = D + C(j\omega I - A)^{-1}B$ <p>In discrete time, the real frequencies $w(1), \dots, w(N)$ are mapped to points on the unit circle using the transformation</p> $z = e^{j\omega T_s}$ <p>where T_s is the sample time. The transfer function is then evaluated at the resulting z values. The default $T_s = 1$ is used for models with unspecified sample time.</p>
Arguments	<p>The output argument H is a 3-D array with dimensions:</p> $(\text{number of outputs}) \times (\text{number of inputs}) \times (\text{length of } w)$ <p>For SISO systems, $H(1, 1, k)$ gives the scalar response at the frequency $w(k)$. For MIMO systems, the frequency response at $w(k)$ is $H(:, :, k)$, a matrix with as many rows as outputs and as many columns as inputs.</p>
Example	<p>Compute the frequency response of</p> $P(s) = \begin{bmatrix} 0 & \frac{1}{s+1} \\ \frac{s-1}{s+2} & 1 \end{bmatrix}$

freqresp

at the frequencies $\omega = 1, 10, 100$. Type:

```
w = [1 10 100]
H = freqresp(P, w)
```

MATLAB returns:

```
H(:, :, 1) =
```

```
0 0.5000- 0.5000i
-0.2000+ 0.6000i 1.0000
```

```
H(:, :, 2) =
```

```
0 0.0099- 0.0990i
0.9423+ 0.2885i 1.0000
```

```
H(:, :, 3) =
```

```
0 0.0001- 0.0100i
0.9994+ 0.0300i 1.0000
```

The three displayed matrices are the values of $P(j\omega)$ for

$\omega = 1, \quad \omega = 10, \quad \omega = 100$

The third index in the 3-D array H is relative to the frequency vector w, so you can extract the frequency response at $\omega = 10$ rad/sec by

```
H(:, :, w==10)
```

```
ans =
```

```
0 0.0099- 0.0990i
0.9423+ 0.2885i 1.0000
```

Algorithm For transfer functions or zero-pole-gain models, freqresp evaluates the numerator(s) and denominator(s) at the specified frequency points. For continuous-time state-space models (A, B, C, D) , the frequency response is

$$D + C(j\omega - A)^{-1}B, \quad \omega = \omega_1, \dots, \omega_N$$

When numerically safe, A is diagonalized for fast evaluation of this expression at the frequencies $\omega_1, \dots, \omega_N$. Otherwise, A is reduced to upper Hessenberg form and the linear equation $(j\omega - A)X = B$ is solved at each frequency point, taking advantage of the Hessenberg structure. The reduction to Hessenberg form provides a good compromise between efficiency and reliability. See [1] for more details on this technique.

Diagnostics If the system has a pole on the $j\omega$ axis (or unit circle in the discrete-time case) and ω happens to contain this frequency point, the gain is infinite, $j\omega I - A$ is singular, and freqresp produces the warning message:

Singularity in freq. response due to jw-axis or unit circle pole.

See Also	evalfr	Response at single complex frequency
	bode	Bode plot
	nyquist	Nyquist plot
	nichols	Nichols plot
	sigma	Singular value plot
	ltiview	LTI system viewer

Reference [1] Laub, A.J., “Efficient Multivariable Frequency Response Computations,” *IEEE Transactions on Automatic Control*, AC-26 (1981), pp. 407–408.

gensig

Purpose Generate test input signals for `l sim`

Syntax
`[u, t] = gensig(type, tau)`
`[u, t] = gensig(type, tau, Tf, Ts)`

Description `[u, t] = gensig(type, tau)` generates a scalar signal `u` of class `type` and with period `tau` (in seconds). The following types of signals are available:

<code>type = 'sin'</code>	Sine wave.
<code>type = 'square'</code>	Square wave.
<code>type = 'pulse'</code>	Periodic pulse.

`gensig` returns a vector `t` of time samples and the vector `u` of signal values at these samples. All generated signals have unit amplitude.

`[u, t] = gensig(type, tau, Tf, Ts)` also specifies the time duration `Tf` of the signal and the spacing `Ts` between the time samples `t`.

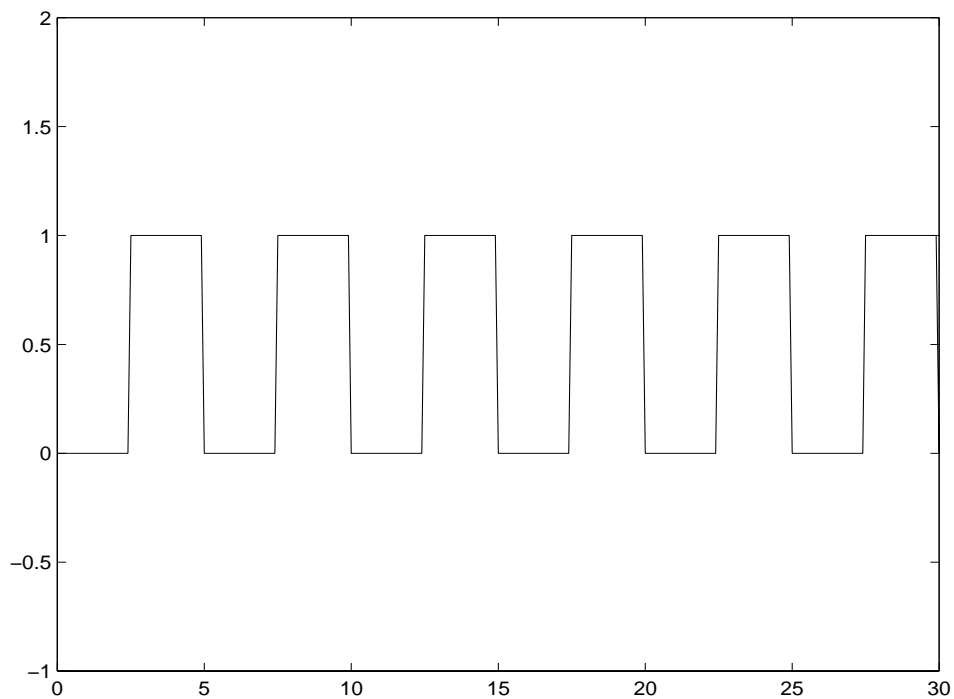
You can feed the outputs `u` and `t` directly to `l sim` and simulate the response of a single-input linear system to the specified signal. Since `t` is uniquely determined by `Tf` and `Ts`, you can also generate inputs for multi-input systems by repeated calls to `gensig`.

Example Generate a square wave with period 5 seconds, duration 30 seconds, and sampling every 0.1 second:

```
[u, t] = gensig('square', 5, 30, 0.1)
```

Plot the resulting signal:

```
plot(t, u)  
axis([0 30 -1 2])
```



See Also

`lsim`
`square`
`sawtooth`

Simulate response to arbitrary inputs
Square wave generator
Sawtooth and triangle wave generator

get

Purpose Access/query LTI property values

Syntax `Value = get(sys, 'PropertyName')`
`get(sys)`
`Struct = get(sys)`

Description `Value = get(sys, 'PropertyName')` returns the current value of the property *PropertyName* of the LTI model `sys`. The string *'PropertyName'* can be the full property name (for example, *'UserData'*) or any unambiguous case-insensitive abbreviation (for example, *'user'*). You can specify any generic LTI property, or any property specific to the model `sys` (see page 2-18 for details on generic and model-specific LTI properties).

`Struct = get(sys)` converts the TF, SS, or ZPK object `sys` into a standard MATLAB structure with the property names as field names and the property values as field values.

Without left-hand argument,

`get(sys)`

displays all properties of `sys` and their values.

Example Consider the discrete-time SISO transfer function defined by

```
h = tf(1, [1 2], 0.1, 'inputname', 'voltage', 'user', 'hello')
```

You can display all LTI properties of `h` by

```
get(h)
    num = {[0 1]}
    den = {[1 2]}
    Variable = 'z'
    Ts = 0.1
    Td = []
    InputName = {'voltage'}
    OutputName = {''}
    Notes = {}
    UserData = 'hello'
```

or query only about the numerator and sample time values by

```
get(h, 'num')

ans =
    [1x2 double]
```

and

```
get(h, 'ts')

ans =
    0.1000
```

Because the numerator data (num property) is always stored as a cell array, the first command evaluates to a cell array containing the row vector [0 1].

Remark

An alternative to the syntax

```
Value = get(sys, 'PropertyName')
```

is the structure-like referencing

```
Value = sys.PropertyName
```

For example,

```
sys.Ts
sys.a
sys.user
```

return the values of the sample time, *A* matrix, and UserData property of the (state-space) model sys.

See Also

set	Set/modify LTI properties
tfdata	Quick access to transfer function data
zpkdata	Quick access to zero-pole-gain data
ssdata	Quick access to state-space data

Purpose Controllability and observability gramians

Syntax
`Wc = gram(sys, 'c')`
`Wo = gram(sys, 'o')`

Description `gram` calculates controllability and observability gramians. You can use gramians to study the controllability and observability properties of state-space models and for model reduction [1,2]. They have better numerical properties than the controllability and observability matrices formed by `ctrb` and `obsv`.

Given the continuous-time state-space model

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

the controllability gramian is defined by

$$W_c = \int_0^{\infty} e^{A\tau} B B^T e^{A^T \tau} d\tau$$

and the observability gramian by

$$W_o = \int_0^{\infty} e^{A^T \tau} C^T C e^{A \tau} d\tau$$

The discrete-time counterparts are

$$W_c = \sum_{k=0}^{\infty} A^k B B^T (A^T)^k, \quad W_o = \sum_{k=0}^{\infty} (A^T)^k C^T C A^k$$

The controllability gramian is positive definite if and only if (A, B) is controllable. Similarly, the observability gramian is positive definite if and only if (C, A) is observable.

Use the commands

```
Wc = gram(sys, 'c')    % controllability gramian
Wo = gram(sys, 'o')    % observability gramian
```


to compute the gramians of a continuous or discrete system. The LTI model sys must be in state-space form.

Algorithm

The controllability gramian W_c is obtained by solving the continuous-time Lyapunov equation

$$AW_c + W_c A^T + BB^T = 0$$

or its discrete-time counterpart

$$AW_c A^T - W_c + BB^T = 0$$

Similarly, the observability gramian W_o solves the Lyapunov equation

$$A^T W_o + W_o A + C^T C = 0$$

in continuous time, and the Lyapunov equation

$$A^T W_o A - W_o + C^T C = 0$$

in discrete time.

Limitations

The A matrix must be stable (all eigenvalues have negative real part in continuous time, and magnitude strictly less than one in discrete time).

See Also

<code>bal real</code>	Gramian-based balancing of state-space realizations
<code>ctrb</code>	Controllability matrix
<code>obsv</code>	Observability matrix
<code>lyap</code> , <code>dl yap</code>	Lyapunov equation solvers

Reference

[1] Kailath, T., *Linear Systems*, Prentice-Hall, 1980.

impulse

Purpose Impulse response of LTI systems

Syntax

```
impulse(sys)
impulse(sys, t)

impulse(sys1, sys2, ..., sysN)
impulse(sys1, sys2, ..., sysN, t)
impulse(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')

[y, t, x] = impulse(sys)
```

Description `impulse` calculates the unit impulse response of a linear system. The impulse response is the response to a Dirac input $\delta(t)$ for continuous-time systems and to a unit pulse at $t = 0$ for discrete-time systems. Zero initial state is assumed in the state-space case. When invoked without left-hand arguments, this function plots the impulse response on the screen.

`impulse(sys)` plots the impulse response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. The impulse response of multi-input systems is the collection of impulse responses for each input channel. The duration of simulation is determined automatically to display the transient behavior of the response.

`impulse(sys, t)` sets the simulation horizon explicitly. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form

`t = 0:dt:Tfinal`

For discrete systems, the spacing `dt` should match the sample period. For continuous systems, `dt` becomes the sample time of the discretized simulation model (see “Algorithm”), so make sure to choose `dt` small enough to capture transient phenomena.

To plot the impulse responses of several LTI models `sys1,..., sysN` on a single figure, use

```
impulse(sys1, sys2, ..., sysN)
impulse(sys1, sys2, ..., sysN, t)
```

As with `bode` or `plot`, you can specify a particular color, linestyle, and/or marker for each system, for example,

```
impul se(sys1, 'y: ', sys2, 'g- -')
```

See page 3-11 and the `bode` entry for more details.

When invoked with lefthand arguments,

```
[y, t] = impul se(sys)
[y, t, x] = impul se(sys)      % for state-space models only
y = impul se(sys, t)
```

return the output response y , the time vector t used for simulation, and the state trajectories x (for state-space models only). No plot is drawn on the screen. For single-input systems, y has as many rows as time samples (length of t), and as many columns as outputs. In the multi-input case, the impulse responses of each input channel are stacked up along the third dimension of y . The dimensions of y are then

$$(\text{length of } t) \times (\text{number of outputs}) \times (\text{number of inputs})$$

and $y(:, :, j)$ gives the response to an impulse disturbance entering the j th input channel. Similarly, the dimensions of x are

$$(\text{length of } t) \times (\text{number of states}) \times (\text{number of inputs})$$

Example

To plot the impulse response of the second-order state-space model:

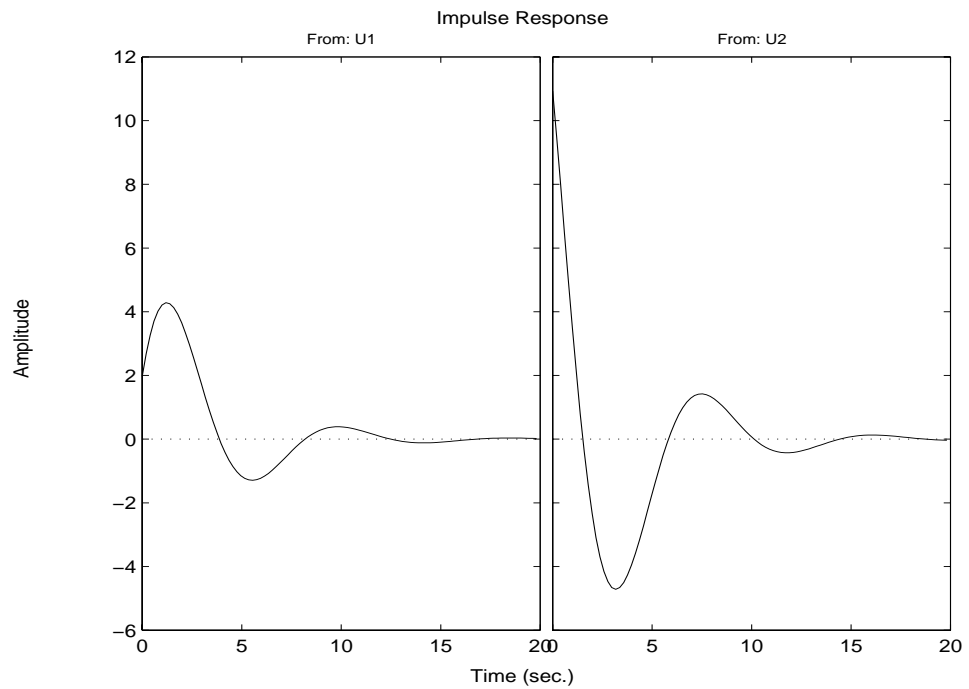
$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

impulse

use the commands:

```
a = [-0.5572  -0.7814; 0.7814  0];  
b = [ 1  -1; 0  2];  
c = [1.9691  6.4493];  
sys = ss(a, b, c, 0);  
impz(sys)
```



The left plot shows the impulse response of the first input channel, and the right plot shows the impulse response of the second input channel.

You can store the impulse response data in MATLAB arrays by

```
[y, t] = impulse(sys)
```

Because this system has two inputs, `y` is a 3-D array with dimensions

```
size(y)
```

```
ans =  
    101     1     2
```

(the first dimension is the length of `t`). The impulse response of the first input channel is then accessed by

```
y(:, :, 1)
```

Algorithm

Continuous-time models are first converted to state space. The impulse response of a single-input state-space model

$$\begin{aligned} \dot{x} &= Ax + bu \\ y &= Cx \end{aligned}$$

is equivalent to the undriven response with initial state b :

$$\begin{aligned} \dot{x} &= Ax, & x(0) &= b \\ y &= Cx \end{aligned}$$

To simulate this response, the system is discretized using zero-order hold on the inputs. The sampling period is chosen automatically based on the system dynamics, except when a time vector `t = 0:dt:Tf` is supplied (`dt` is then used as sampling period).

Limitations

The impulse response of a continuous system with nonzero D matrix is infinite at $t = 0$. `impz` ignores this discontinuity and returns the lower continuity value Cb at $t = 0$.

See Also

`ltiview`
`step`
`initial`
`lsim`

LTI system viewer
Step response
Free response to initial condition
Simulate response to arbitrary inputs

initial

Purpose Initial condition response of state-space models

Syntax

```
i n i t i a l (sys, x0)
i n i t i a l (sys, x0, t)

i n i t i a l (sys1, sys2, . . . , sysN, x0)
i n i t i a l (sys1, sys2, . . . , sysN, x0, t)
i n i t i a l (sys1, ' P l o t S t y l e 1' , . . . , sysN, ' P l o t S t y l e N' , x0)

[y, t, x] = i n i t i a l (sys, x0)
```

Description `i n i t i a l` calculates the undriven response of a state-space model to an initial condition on the states:

$$\begin{aligned} \dot{x} &= Ax, & x(0) &= x_0 \\ y &= Cx \end{aligned}$$

This function is applicable to either continuous- or discrete-time models. When invoked without lefthand arguments, `i n i t i a l` plots the initial condition response on the screen.

`i n i t i a l (sys, x0)` plots the response of `sys` to an initial condition `x0` on the states. `sys` can be any *state-space* model (continuous or discrete, SISO or MIMO, with or without inputs). The duration of simulation is determined automatically to reflect adequately the response transients.

`i n i t i a l (sys, x0, t)` explicitly sets the simulation horizon. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form

$$t = 0: dt: Tfinal$$

For discrete systems, the spacing `dt` should match the sample period. For continuous systems, `dt` becomes the sample time of the discretized simulation model (see `i m p u l s e`), so make sure to choose `dt` small enough to capture transient phenomena.

To plot the initial condition responses of several LTI models on a single figure, use

```
initial(sys1, sys2, ..., sysN, x0)
initial(sys1, sys2, ..., sysN, x0, t)
```

(see `impz` for details).

When invoked with lefthand arguments,

```
[y, t, x] = initial(sys, x0)
[y, t, x] = initial(sys, x0, t)
```

return the output response y , the time vector t used for simulation, and the state trajectories x . No plot is drawn on the screen. The array y has as many rows as time samples (length of t) and as many columns as outputs. Similarly, x has `length(t)` rows and as many columns as states.

Example

Plot the response of the state-space model

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

to the initial condition

$$x(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

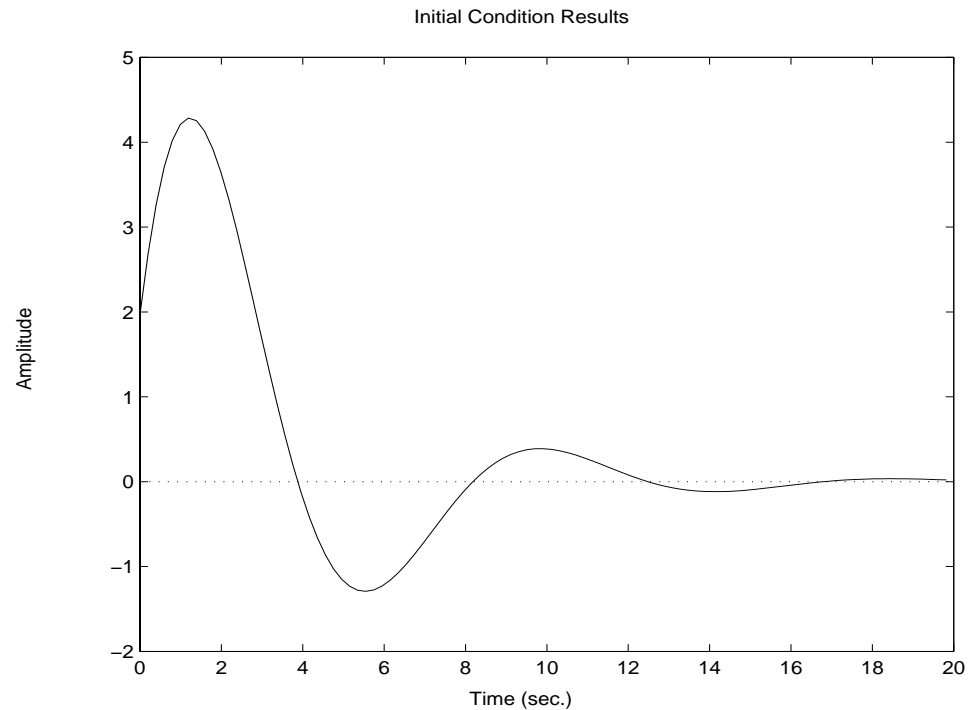
```
a = [-0.5572    -0.7814; 0.7814    0];
```

```
c = [1.9691    6.4493];
```

```
x0 = [1 ; 0]
```

```
sys = ss(a, [], c, []);
```

```
initial(sys, x0)
```



See Also

lti view
impz
step
lsim

LTI system viewer
Impulse response
Step response
Simulate response to arbitrary inputs

Purpose Invert LTI systems

Syntax `i sys = i nv(sys)`

Description `i nv` inverts the input/output relation

$$y = G(s)u$$

to produce the LTI system with transfer matrix $H(s) = G(s)^{-1}$:

$$u = H(s)y$$

This operation is defined only for square systems (same number of inputs and outputs) with an invertible feedthrough matrix D . `i nv` handles both continuous- and discrete-time systems.

Example Consider

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

At the MATLAB prompt, type:

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

to invert it and MATLAB returns:

Transfer function from input 1 to output...

#1: 1

#2: 0

Transfer function from input 2 to output...

-1

#1: -----

s + 1

#2: 1

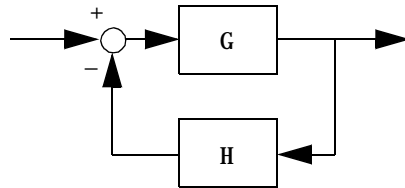
You can verify that

$$H * \text{inv}(H) = I$$

is the identity transfer function (static gain I).

Limitations

Do not use `inv` to model feedback connections such as:



While it seems reasonable to evaluate the corresponding closed-loop transfer function $(I + GH)^{-1}G$ as

$$\text{inv}(1 + g * h) * g$$

this typically leads to nonminimal closed-loop models. For example,

```
g = zpk([], 1, 1)
h = tf([2 1], [1 0])
clloop = inv(1 + g * h) * g
```

yields a third-order closed-loop model with an unstable pole-zero cancellation at $s = 1$:

clloop

Zero/pole/gain:

$$\frac{s(s-1)}{(s-1)(s^2 + s + 1)}$$

Use feedback or star to avoid such pitfalls:

`cl oop = feedback(g, h)`

Zero/pole/gain:

s

$(s^2 + s + 1)$

isct, isdt

Purpose	Determine whether an LTI model is continuous or discrete	
Syntax	<pre>boo = isct(sys) boo = isdt(sys)</pre>	
Description	<p><code>boo = isct(sys)</code> returns 1 (true) if the LTI model <code>sys</code> is continuous and 0 (false) otherwise. <code>sys</code> is continuous if its sample time is zero, that is, <code>sys.Ts=0</code>.</p> <p><code>boo = isdt(sys)</code> returns 1 (true) if <code>sys</code> is discrete and 0 (false) otherwise. Discrete-time LTI models have a nonzero sample time, except for empty models and static gains, which are regarded as either continuous or discrete as long as their sample time is not explicitly set to a nonzero value. Thus both</p> <pre>isct(tf(10)) isdt(tf(10))</pre> <p>are true. However, if you explicitly label a gain as discrete, for example, by typing</p> <pre>g = tf(10, 'ts', 0.01)</pre> <p><code>isct(g)</code> now returns false and only <code>isdt(g)</code> is true.</p>	
See Also	<pre>isa isempty isproper</pre>	<pre>Determine LTI model type True for empty LTI models True for proper LTI models</pre>

Purpose	Test if an LTI model is empty	
Syntax	<code>boo = isempty(sys)</code>	
Description	<code>isempty(sys)</code> returns 1 (true) if the LTI model <code>sys</code> has no input or no output, and 0 (false) otherwise.	
Example	<p>Both commands</p> <pre> isempty(tf) % tf by itself returns an empty transfer function isempty(ss(1, 2, [], [])) return 1 (true) while isempty(ss(1, 2, 3, 4)) returns 0 (false).</pre>	
See Also	<div> <div>size</div> <div>issiso</div> </div>	<div> <div>I/O dimensions of LTI system</div> <div>True for SISO systems</div> </div>

isproper

Purpose Test if an LTI model is proper

Syntax `boo = isproper(sys)`

Description `isproper(sys)` returns 1 (true) if the LTI model `sys` is proper and 0 (false) otherwise.

State-space models are always proper. SISO transfer functions or zero-pole-gain models are proper if the degree of their numerator is less than or equal to the degree of their denominator. MIMO transfer functions are proper if all their SISO entries are proper.

Example The following commands

```
isproper(tf([1 0], 1))      % transfer function s
isproper(tf([1 0], [1 1]))  % transfer function s/(s+1)
```

return false and true, respectively.

Purpose	Test if an LTI model is single-input/single-output (SISO)	
Syntax	<code>boo = issiso(sys)</code>	
Description	<code>issiso(sys)</code> returns 1 (true) if the LTI model <code>sys</code> is SISO and 0 (false) otherwise.	
See Also	<code>size</code> <code>isempty</code>	I/O dimensions of LTI system True for empty LTI models

kalman

Purpose Design continuous- or discrete-time Kalman estimator

Syntax `[kest, L, P] = kalman(sys, Qn, Rn, Nn)`
`[kest, L, P, M, Z] = kalman(sys, Qn, Rn, Nn) % discrete time only`
`[kest, L, P] = kalman(sys, Qn, Rn, Nn, sensors, known)`

Description `kalman` designs a Kalman state estimator given a state-space model of the plant and the process and measurement noise covariance data. The Kalman estimator is the optimal solution to the following continuous or discrete estimation problems.

Continuous-Time Estimation

Given the continuous plant

$$\begin{aligned} \dot{x} &= Ax + Bu + Gw && \text{(state equation)} \\ y_v &= Cx + Du + Hw + v && \text{(measurement equation)} \end{aligned}$$

with known inputs u and process and measurement white noise w, v satisfying

$$E(w) = E(v) = 0, \quad E(ww^T) = Q, \quad E(vv^T) = R, \quad E(wv^T) = N$$

construct a state estimate $\hat{x}(t)$ that minimizes the steady-state error covariance

$$P = \lim_{t \rightarrow \infty} E(\{x - \hat{x}\}\{x - \hat{x}\}^T)$$

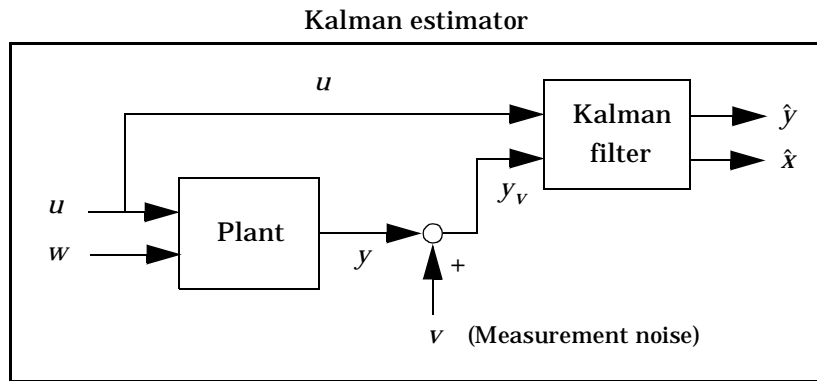
The optimal solution is the Kalman filter with equations

$$\begin{aligned} \dot{\hat{x}} &= A\hat{x} + Bu + L(y_v - C\hat{x} - Du) \\ \begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} &= \begin{bmatrix} C \\ I \end{bmatrix} \hat{x} + \begin{bmatrix} D \\ 0 \end{bmatrix} u \end{aligned}$$

where the filter gain L is determined by solving an algebraic Riccati equation. This estimator uses the known inputs u and the measurements y_v to generate

the output and state estimates \hat{y} and \hat{x} . Note that \hat{y} estimates the true plant output

$$y = Cx + Du + Hw$$



Discrete-Time Estimation

Given the discrete plant

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

and the noise covariance data

$$E(w[n]w[n]^T) = Q, \quad E(v[n]v[n]^T) = R, \quad E(w[n]v[n]^T) = N$$

the Kalman estimator has equations

$$\hat{x}[n+1|n] = A\hat{x}[n|n-1] + Bu[n] + L(y_v[n] - C\hat{x}[n|n-1] - Du[n])$$

$$\begin{bmatrix} \hat{y}[n|n] \\ \hat{x}[n|n] \end{bmatrix} = \begin{bmatrix} C(I-MC) \\ I-MC \end{bmatrix} \hat{x}[n|n-1] + \begin{bmatrix} (I-CM)D & CM \\ -MD & M \end{bmatrix} \begin{bmatrix} u[n] \\ y_v[n] \end{bmatrix}$$

and generates optimal “current” output and state estimates $\hat{y}[n|n]$ and $\hat{x}[n|n]$ using all available measurements including $y_v[n]$. The gain matrices L and M are derived by solving a discrete Riccati equation. The *innovation gain* M is used to update the prediction $\hat{x}[n|n-1]$ using the new measurement $y_v[n]$:

$$\hat{x}[n|n] = \hat{x}[n|n-1] + M \underbrace{(y_v[n] - C\hat{x}[n|n-1] - Du[n])}_{\text{innovation}}$$

Usage

`[kest, L, P] = kalman(sys, Qn, Rn, Nn)` returns a state-space model `kest` of the Kalman estimator given the plant model `sys` and the noise covariance data `Qn`, `Rn`, `Nn` (matrices Q , R , N above). `sys` must be a state-space model with matrices

$$A, \begin{bmatrix} B & G \end{bmatrix}, C, \begin{bmatrix} D & H \end{bmatrix}$$

The resulting estimator `kest` has $[u; y_v]$ as inputs and $[\hat{y}; \hat{x}]$ (or their discrete-time counterparts) as outputs. You can omit the last input argument `Nn` when $N = 0$.

The function `kalman` handles both continuous and discrete problems and produces a continuous estimator when `sys` is continuous, and a discrete estimator otherwise. In continuous time, `kalman` also returns the Kalman gain L and the steady-state error covariance matrix P . Note that P is the solution of the associated Riccati equation. In discrete time, the syntax

$$[\text{kest}, L, P, M, Z] = \text{kalman}(\text{sys}, Qn, Rn, Nn)$$

returns the filter gain L and innovations gain M , as well as the steady-state error covariances

$$P = \lim_{n \rightarrow \infty} E(e[n|n-1]e[n|n-1]^T), \quad e[n|n-1] = x[n] - x[n|n-1]$$

$$Z = \lim_{n \rightarrow \infty} E(e[n|n]e[n|n]^T), \quad e[n|n] = x[n] - x[n|n]$$

Finally, use the syntaxes

```
[kest, L, P] = kalman(sys, Qn, Rn, Nn, sensors, known)
[kest, L, P, M, Z] = kalman(sys, Qn, Rn, Nn, sensors, known)
```

for more general plants sys where the known inputs u and stochastic inputs w are mixed together, and not all outputs are measured. The index vectors sensors and known then specify which outputs y of sys are measured and which inputs u are known. All other inputs are assumed stochastic.

Example

See examples on page 1-19, page 7-33, and page 7-49.

Limitations

The plant and noise data must satisfy:

- (C, A) detectable
- $\bar{R} > 0$ and $\bar{Q} - \bar{N}\bar{R}^{-1}\bar{N}^T \geq 0$
- $(A - \bar{N}\bar{R}^{-1}C, \bar{Q} - \bar{N}\bar{R}^{-1}\bar{N}^T)$ has no uncontrollable mode on the imaginary axis (or unit circle in discrete time)

with the notation

$$\bar{Q} = GQG^T$$

$$\bar{R} = R + HN + N^TH^T + HQH^T$$

$$\bar{N} = G(QH^T + N)$$

See Also

<code>kalmd</code>	Discrete Kalman estimator for continuous plant
<code>estim</code>	Form estimator given estimator gain
<code>lqr</code>	Design state-feedback LQ regulator
<code>lqgreg</code>	Assemble LQG regulator

kalman

care

Solve continuous-time Riccati equations

dare

Solve discrete-time Riccati equations

Reference

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

Purpose	Design discrete Kalman estimator for continuous plant	
Syntax	<code>[kest, L, P, M, Z] = kalmd(sys, Qn, Rn, Ts)</code>	
Description	<p><code>kalmd</code> designs a discrete-time Kalman estimator that has response characteristics similar to a continuous-time estimator designed with <code>kalman</code>. This command is useful to derive a discrete estimator for digital implementation after a satisfactory continuous estimator has been designed.</p> <p><code>[kest, L, P, M, Z] = kalmd(sys, Qn, Rn, Ts)</code> produces a discrete Kalman estimator <code>kest</code> with sample time <code>Ts</code> for the continuous-time plant</p> $\dot{x} = Ax + Bu + Gw \quad (\text{state equation})$ $y_v = Cx + Du + v \quad (\text{measurement equation})$ <p>with process noise w and measurement noise v satisfying</p> $E(w) = E(v) = 0, \quad E(ww^T) = Q_n, \quad E(vv^T) = R_n, \quad E(wv^T) = 0$ <p>The estimator <code>kest</code> is derived as follows. The continuous plant <code>sys</code> is first discretized using zero-order hold with sample time <code>Ts</code> (see <code>c2d</code> entry), and the continuous noise covariance matrices Q_n and R_n are replaced by their discrete equivalents</p> $Q_d = \int_0^{T_s} e^{A\tau} G Q G^T e^{A^T \tau} d\tau$ $R_d = R / T_s$ <p>The integral is computed using the matrix exponential formulas in [2]. A discrete-time estimator is then designed for the discretized plant and noise. See <code>kalman</code> for details on discrete-time Kalman estimation.</p> <p><code>kalmd</code> also returns the estimator gains <code>L</code> and <code>M</code>, and the discrete error covariance matrices <code>P</code> and <code>Z</code> (see <code>kalman</code> for details).</p>	
Limitations	The discretized problem data should satisfy the requirements for <code>kalman</code> .	
See Also	<code>lqrd</code> <code>kalman</code>	Discrete LQ-optimal gain for continuous plant Design Kalman estimator

l qgreg

Assemble LQG regulator

Reference

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

[2] Van Loan, C.F., "Computing Integrals Involving the Matrix Exponential," *IEEE Trans. Automatic Control*, AC-15, October 1970.

Purpose Form LQG regulator given state-feedback gain and Kalman estimator

Syntax

```
rlqg = lqgreg(kest, k)
rlqg = lqgreg(kest, k, 'current')    % discrete-time only

rlqg = lqgreg(kest, k, controls)
```

Description lqgreg forms the LQG regulator by connecting the Kalman estimator designed with kalman and the optimal state-feedback gain designed with lqr, dlqr, or lqry. The LQG regulator minimizes some quadratic cost function that trades off regulation performance and control effort. This regulator is dynamic and relies on noisy output measurements to generate the regulating commands (see page 5-11 for details).

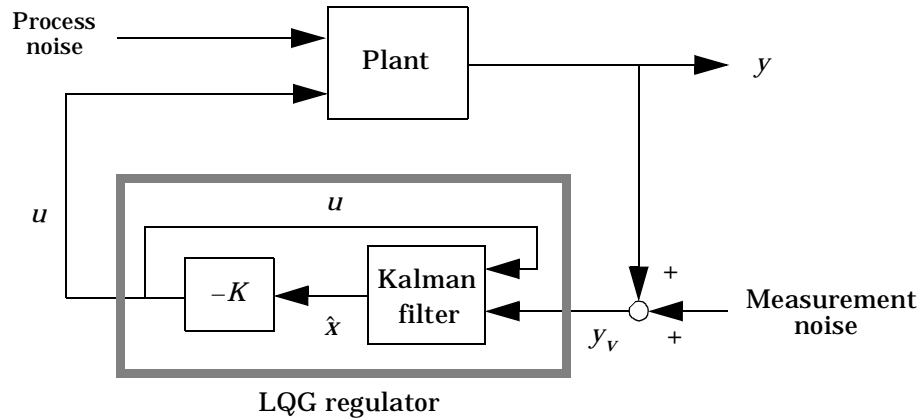
In continuous time, the LQG regulator generates the commands

$$u = -K\hat{x}$$

where \hat{x} is the Kalman state estimate. The regulator state-space equations are

$$\begin{aligned}\dot{\hat{x}} &= [A - LC - (B - LD)K]\hat{x} + Ly_v \\ u &= -K\hat{x}\end{aligned}$$

where y_v is the vector of plant output measurements (see kalman for background and notation). The diagram below shows this dynamic regulator in relation to the plant.



In discrete time, you can form the LQG regulator using either the prediction $\hat{x}[n|n-1]$ of $x[n]$ based on measurements up to $y_v[n-1]$, or the *current* state estimate $\hat{x}[n|n]$ based on all available measurements including $y_v[n]$. While the regulator

$$u[n] = -K\hat{x}[n|n-1]$$

is always well-defined, the “current” regulator

$$u[n] = -K\hat{x}[n|n]$$

is causal only when $I - KMD$ is invertible (see `kalman` for the notation). In addition, practical implementations of the current regulator should allow for the processing time required to compute $u[n]$ once the measurements $y_v[n]$ become available (this amounts to a time delay in the feedback loop).

Usage

`rlqg = lqgreg(kest, k)` returns the LQG regulator `rlqg` (a state-space model) given the Kalman estimator `kest` and the state-feedback gain matrix `k`. The same function handles both continuous- and discrete-time cases. Use consistent tools to design `kest` and `k`:

- Continuous regulator for continuous plant: use `lqr` or `lqry` and `kalman`.
- Discrete regulator for discrete plant: use `dlqr` or `lqrd` and `kalman`.
- Discrete regulator for continuous plant: use `lqrd` and `kalmd`.

In discrete time, `lqgreg` produces the regulator

$$u[n] = -K\hat{x}[n|n-1]$$

by default (see “Description”). To form the “current” LQG regulator instead, use

$$u[n] = -K\hat{x}[n|n]$$

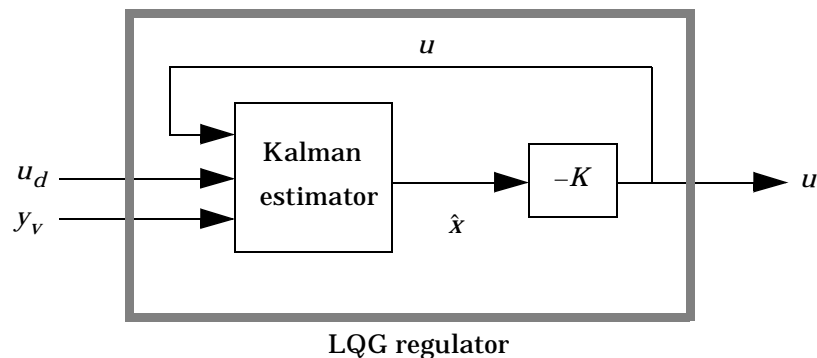
the syntax

```
rlqg = lqgreg(kest, k, 'current')
```

This syntax is meaningful only for discrete-time problems.

`rlqg = lqgreg(kest, k, controls)` handles estimators that have access to additional known plant inputs u_d . The index vector `controls` then specifies which estimator inputs are the controls u , and the resulting LQG regulator `rlqg` has u_d and y_v as inputs (see figure below).

Note: Always use *positive* feedback to connect the LQG regulator to the plant.



Example

See examples on page 1-19 and page 7-30.

Limitations The “current” discrete LQG regulator is not defined if $I - KMD$ is singular where M is the Kalman innovation gain (see kal man) and D is the feedthrough matrix from controls u to measurements y_v .

See Also	kal man	Kalman estimator design
	kal md	Discrete Kalman estimator for continuous plant
	l qr, dl qr	State-feedback LQ regulator
	l qry	LQ regulator with output weighting
	l qrd	Discrete LQ regulator for continuous plant
	reg	Form regulator given state-feedback and estimator gains

Purpose	Design linear-quadratic (LQ) state-feedback regulator for continuous plant
Syntax	$[K, S, e] = \text{lqr}(A, B, Q, R)$ $[K, S, e] = \text{lqr}(A, B, Q, R, N)$
Description	<p>$[K, S, e] = \text{lqr}(A, B, Q, R, N)$ calculates the optimal gain matrix K such that the state-feedback law</p> $u = -Kx$ <p>minimizes the quadratic cost function</p> $J(u) = \int_0^{\infty} (x^T Q x + u^T R u + 2x^T N u) dt$ <p>for the continuous-time state-space model</p> $\dot{x} = Ax + Bu$ <p>The default value $N=0$ is assumed when N is omitted.</p> <p>In addition to the state-feedback gain K, lqr returns the solution S of the associated Riccati equation</p> $A^T S + SA - (SB + N)R^{-1}(B^T S + N^T) + Q = 0$ <p>and the closed-loop eigenvalues $e = \text{eig}(A - B^*K)$. Note that K is derived from S by</p> $K = R^{-1}(B^T S + N^T)$
Limitations	<p>The problem data must satisfy:</p> <ul style="list-style-type: none"> • the pair (A, B) is stabilizable • $R > 0$ and $Q - NR^{-1}N^T \geq 0$ • $(Q - NR^{-1}N^T, A - BR^{-1}N^T)$ has no unobservable mode on the imaginary axis

See Also

[dlqr](#)

[lqry](#)

[lqrd](#)

[lqgreg](#)

[care](#)

[State-feedback LQ regulator for discrete plant](#)

[State-feedback LQ regulator with output weighting](#)

[Discrete LQ regulator for continuous plant](#)

[Form LQG regulator](#)

[Solve continuous Riccati equations](#)

Purpose Design discrete LQ regulator for continuous plant

Syntax $[K_d, S, e] = \text{lqrd}(A, B, Q, R, T_s)$
 $[K_d, S, e] = \text{lqrd}(A, B, Q, R, N, T_s)$

Description lqrd designs a discrete full-state-feedback regulator that has response characteristics similar to a continuous state-feedback regulator designed using lqr . This command is useful to design a gain matrix for digital implementation after a satisfactory continuous state-feedback gain has been designed.

$[K_d, S, e] = \text{lqrd}(A, B, Q, R, T_s)$ calculates the discrete state-feedback law

$$u[n] = -K_d x[n]$$

that minimizes a discrete cost function equivalent to the continuous cost function

$$J = \int_0^{\infty} (x^T Q x + u^T R u) dt$$

The matrices A and B specify the continuous plant dynamics

$$\dot{x} = Ax + Bu$$

and T_s specifies the sample time of the discrete regulator. Also returned are the solution S of the discrete Riccati equation for the discretized problem and the discrete closed-loop eigenvalues $e = \text{eig}(Ad - Bd * K_d)$.

$[K_d, S, e] = \text{lqrd}(A, B, Q, R, N, T_s)$ solves the more general problem with a cross-coupling term in the cost function:

$$J = \int_0^{\infty} (x^T Q x + u^T R u + 2x^T N u) dt$$

Algorithm The equivalent discrete gain matrix K_d is determined by discretizing the continuous plant and weighting matrices using the sample time T_s and the zero-order hold approximation.

With the notation

$$\begin{aligned}\Phi(\tau) &= e^{A\tau}, & A_d &= \Phi(T_s) \\ \Gamma(\tau) &= \int_0^\tau e^{A\eta} B d\eta, & B_d &= \Gamma(T_s)\end{aligned}$$

the discretized plant has equations

$$x[n+1] = A_d x[n] + B_d u[n]$$

and the weighting matrices for the equivalent discrete cost function are:

$$\begin{bmatrix} Q_d & N_d \\ N_d^T & R_d \end{bmatrix} = \int_0^{T_s} \begin{bmatrix} \Phi^T(\tau) & 0 \\ \Gamma^T(\tau) & I \end{bmatrix} \begin{bmatrix} Q & N \\ N^T & R \end{bmatrix} \begin{bmatrix} \Phi(\tau) & \Gamma(\tau) \\ 0 & I \end{bmatrix} d\tau$$

The integrals are computed using matrix exponential formulas due to Van Loan (see [2]). The plant is discretized using c2d and the gain matrix is computed from the discretized data using dl qr.

Limitations	The discretized problem data should meet the requirements for dl qr.	
See Also	dl qr	State-feedback LQ regulator for discrete plant
	l qr	State-feedback LQ regulator for continuous plant
	c2d	Discretization of LTI model
	kal md	Discrete Kalman estimator for continuous plant
References	[1] Franklin, G.F. J.D. Powell, and M.L. Workman, <i>Digital Control of Dynamic Systems</i> , Second Edition, Addison-Wesley, 1980, pp. 439–440	
	[2] Van Loan, C.F., “Computing Integrals Involving the Matrix Exponential,” <i>IEEE Trans. Automatic Control</i> , AC-15, October 1970.	

Purpose Linear-quadratic (LQ) state-feedback regulator with output weighting

Syntax [K, S, e] = lqry(sys, Q, R)
[K, S, e] = lqry(sys, Q, R, N)

Description Given the plant

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

or its discrete-time counterpart, lqry designs a state-feedback control

$$u = -Kx$$

that minimizes the quadratic cost function with output weighting

$$J(u) = \int_0^{\infty} (y^T Q y + u^T R u + 2 y^T N u) dt$$

(or its discrete-time counterpart). The function lqry is equivalent to lqr or dlqr with weighting matrices:

$$\begin{bmatrix} \bar{Q} & \bar{N} \\ \bar{N}^T & \bar{R} \end{bmatrix} = \begin{bmatrix} C^T & 0 \\ D^T & I \end{bmatrix} \begin{bmatrix} Q & N \\ N^T & R \end{bmatrix} \begin{bmatrix} C & D \\ 0 & I \end{bmatrix}$$

[K, S, e] = lqry(sys, Q, R, N) returns the optimal gain matrix K, the Riccati solution S, and the closed-loop eigenvalues e = eig(A - B*K). The state-space model sys specifies the continuous- or discrete-time plant data (A, B, C, D). The default value N=0 is assumed when N is omitted.

Example See page 7-33.

Limitations The data A, B, \bar{Q} , \bar{R} , \bar{N} must satisfy the requirements for lqr or dlqr.

See Also lqr State-feedback LQ regulator for continuous plant
dlqr State-feedback LQ regulator for discrete plant
kalman Kalman estimator design
lqreg Form LQG regulator

Purpose Simulate LTI model response to arbitrary inputs

Syntax

```
lsim(sys, u, t)
lsim(sys, u, t, x0)

lsim(sys1, sys2, ..., sysN, u, t)
lsim(sys1, sys2, ..., sysN, u, t, x0)
lsim(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN', u, t)

[y, t, x] = lsim(sys, u, t, x0)
```

Description `lsim` simulates the (time) response of continuous or discrete linear systems to arbitrary inputs. When invoked without lefthand arguments, `lsim` plots the response on the screen.

`lsim(sys, u, t)` produces a plot of the time response of the LTI model `sys` to the input time history `t,u`. The vector `t` specifies the time samples for the simulation and consists of regularly spaced time samples:

$$t = 0:dt:T_{final}$$

The matrix `u` must have as many rows as time samples (`length(t)`) and as many columns as system inputs. Each row `u(i, :)` specifies the input value(s) at the time sample `t(i)`.

The LTI model `sys` can be continuous or discrete, SISO or MIMO. In discrete time, `u` must be sampled at the same rate as the system (`t` is then redundant and can be omitted or set to the empty matrix). In continuous time, the time sampling $dt = t(2) - t(1)$ is used to discretize the continuous model. Automatic resampling is performed if `dt` is too large (undersampling) and may give rise to hidden oscillations (see “Algorithm”).

`lsim(sys, u, t, x0)` further specifies an initial condition `x0` for the system states. This syntax applies only to state-space models.

Finally,

```
lsim(sys1, sys2, ..., sysN, u, t)
```


simulates the responses of several LTI models to the same input history t, u and plots these responses on a single figure. As with `bode` or `plot`, you can specify a particular color, linestyle, and/or marker for each system, for example,

```
lsim(sys1, 'y:', sys2, 'g- -', u, t, x0)
```

The multisystem behavior is similar to that of `bode` or `step`.

When invoked with lefthand arguments,

```
[y, t] = lsim(sys, u, t)
[y, t, x] = lsim(sys, u, t)      % for state-space models only
[y, t, x] = lsim(sys, u, t, x0)  % with initial state
```

return the output response y , the time vector t used for simulation, and the state trajectories x (for state-space models only). No plot is drawn on the screen. The matrix y has as many rows as time samples (`length(t)`) and as many columns as system outputs. The same holds for x with “outputs” replaced by states. Note that the output t may differ from the specified time vector when the input data is undersampled (see “Algorithm”).

Example

Simulate and plot the response of the system

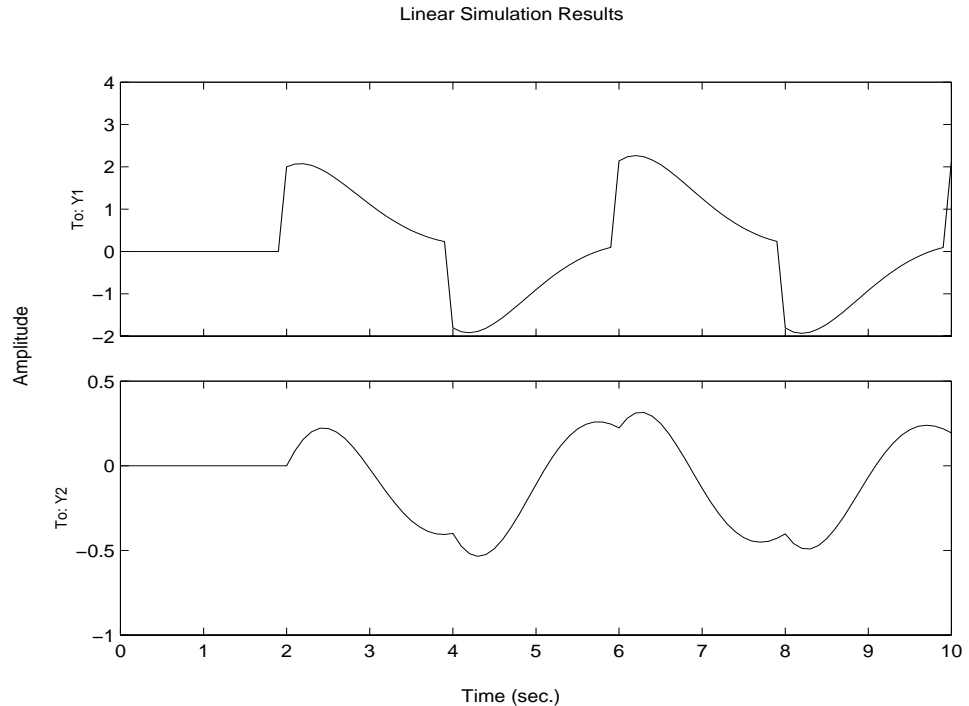
$$H(s) = \left[\begin{array}{c} \frac{2s^2 + 5s + 1}{s^2 + 2s + 3} \\ \frac{s - 1}{s^2 + s + 5} \end{array} \right]$$

to a square wave with period of four seconds. First generate the square wave with `gensig`. Sample every 0.1 second during 10 seconds:

```
[u, t] = gensig('square', 4, 10, 0.1)
```

Then simulate with `lsim`:

```
H = [tf([2 5 1],[1 2 3]) ; tf([1 -1],[1 1 5])]
lsim(H, u, t)
```



Algorithm

Discrete-time systems are simulated with `ltitr` (state space) or `filter` (transfer function and zero-pole-gain).

Continuous-time systems are discretized with `c2d` using either the 'zoh' or 'foh' method ('foh' is used for smooth input signals and 'zoh' for discontinuous signals such as pulses or square waves). By default, the sampling period is set to the spacing `dt` between the user-supplied time samples `t`. However, if `dt` is not small enough to capture intersample behavior, `lsim` selects a smaller sampling period and resamples the input data using linear interpolation for smooth signals and zero-order hold for square signals. The time vector returned by `lsim` is then different from the specified `t` vector.

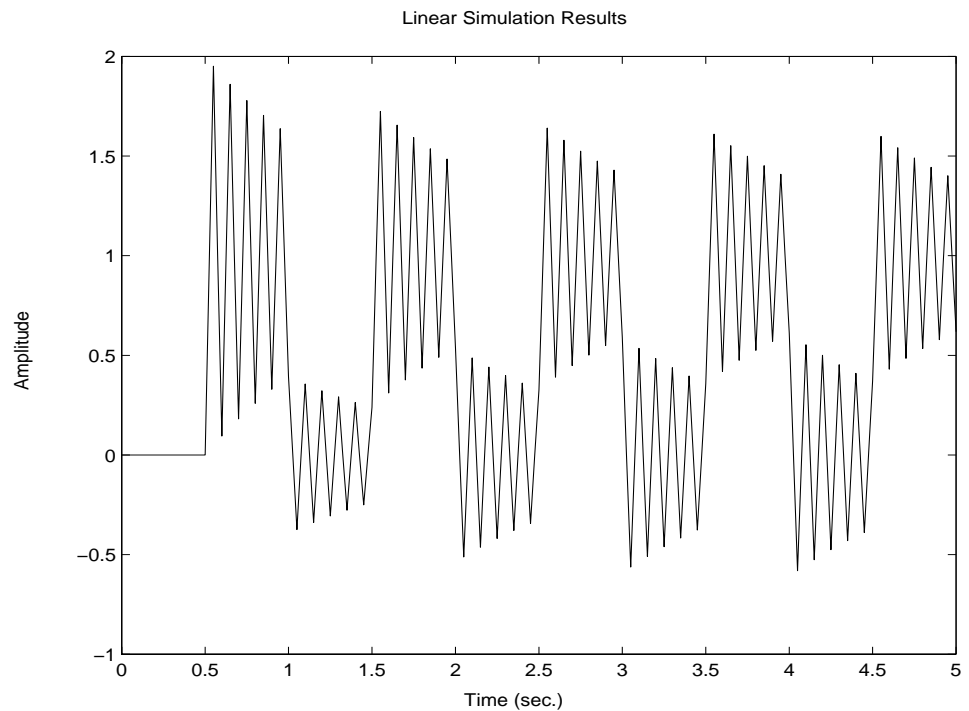
To illustrate why resampling is sometimes necessary, consider the second-order model

$$H(s) = \frac{\omega^2}{s^2 + 2s + \omega^2}, \quad \omega = 62.83$$

To simulate its response to a square wave with period 1 second, you can proceed as follows:

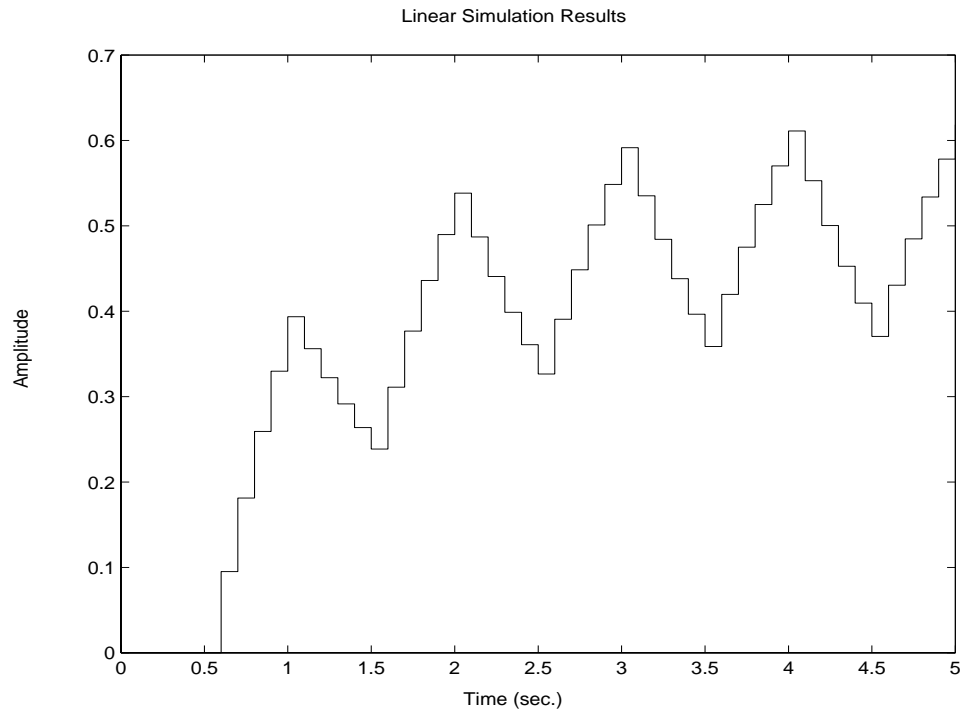
```
w2 = 62.83^2
h = tf(w2, [1 2 w2])

t = 0:0.1:5;           % vector of time samples
u = (rem(t, 1) >= 0.5); % square wave values
lsim(h, u, t)
```



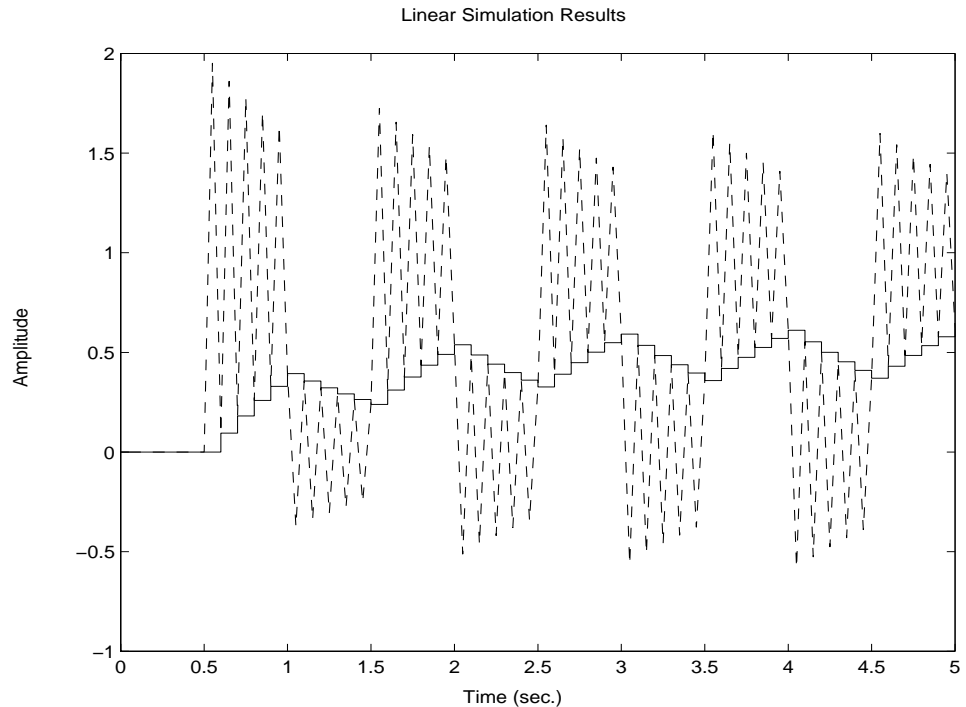
The response exhibits strong oscillations. Less obvious from this plot is the fact that `lsim` has resampled the input to reveal the oscillatory behavior. To see this, discretize $H(s)$ using the sampling period 0.1 second (spacing in your `t` vector) and simulate the response of the discretized model:

```
hd = c2d(h, 0.1)
lsim(hd, u, t)
```



The two responses look quite different. To clarify this discrepancy, superimpose the two plots by

```
lsim(h, 'b--', hd, 'r-', u, t)
```



The cause is now obvious: `hd` is undersampled and its response (solid line) masks the intersample oscillations of the continuous model $H(s)$.

By comparing the suggested sampling $dt=t(2)-t(1)$ against the system dynamics, `lsim` detects such undersampling and resamples the input to produce accurate continuous-time simulations.

See Also

`ltiview`
`step`
`impz`
`initial`
`gensig`

LTI system viewer
 Step response
 Impulse response
 Free response to initial condition
 Generate test input signals for `lsim`

ltiview

Purpose Initialize an LTI Viewer for LTI system response analysis

Syntax

```
ltiview
ltiview(plottype, sys)
ltiview(plottype, sys, extras)

ltiview(plottype, sys1, sys2, ... sysN)
ltiview(plottype, sys1, sys2, ... sysN, extras)
ltiview(plottype, sys1, PlotStyle1, sys2, PlotStyle2, ...)
```

Description `ltiview` when invoked without input arguments, initializes a new LTI Viewer for LTI system response analysis.

`ltiview(plottype, sys)` initializes an LTI Viewer containing the LTI response type indicated by *plottype* for the LTI model *sys*. The string *plottype* can be any of the following:

```
'step'
'impulse'
'initial'
'lsim'

'bode'
'nyquist'
'nichols'
'sigma'
```

`ltiview(plottype, sys, extras)` allows the additional input arguments supported by the various LTI model response functions to be passed to the `ltiview` command.

extras is one or more input arguments as specified by the function named in *plottype*. These arguments may be required or optional, depending on the type of LTI response. For example, if *plottype* is 'step' then *extras* may be the desired final time, *Tfinal*, as shown below:

```
ltiview('step', sys, Tfinal)
```

However, if *plottype* is 'initial', the extras arguments must contain the initial conditions *x0* and may contain other arguments, such as *Tfinal*:

```
ltiview('initial', sys, x0, Tfinal)
```

See the individual references pages of each possible *plottype* commands for a list of appropriate arguments for extras.

Finally,

```
ltiview(plottype, sys1, sys2, ... sysN)
ltiview(plottype, sys1, sys2, ... sysN, extras)
ltiview(plottype, sys1, PlotStyle1, sys2, PlotStyle2, ...)
```

initializes an LTI Viewer containing the responses of multiple LTI models, using the plot styles in *PlotStyle*, when applicable. See the individual reference pages of the LTI response functions for more information on specifying plot styles.

Example

See Chapter 4.

See Also

step	Step response
impulse	Impulse response
bode	Bode response
nyquist	Nyquist response
nichols	Nichols response
sigma	Singular value response
lsim	Simulate LTI model response to arbitrary inputs
initial	Response to initial condition

lyap

Purpose	Solve continuous-time Lyapunov equations	
Syntax	$X = \text{lyap}(A, Q)$ $X = \text{lyap}(A, B, C)$	
Description	<p><code>lyap</code> solves the special and general forms of the Lyapunov matrix equation. Lyapunov equations arise in several areas of control, including stability theory and the study of the RMS behavior of systems.</p> <p>$X = \text{lyap}(A, Q)$ solves the Lyapunov equation</p> $AX + XA^T + Q = 0$ <p>where A and Q are square matrices of identical sizes. The solution X is a symmetric matrix if Q is.</p> <p>$X = \text{lyap}(A, B, C)$ solves the generalized Lyapunov equation (also called Sylvester equation):</p> $AX + XB + C = 0$ <p>The matrices A, B, C must have compatible dimensions but need not be square.</p>	
Algorithm	<code>lyap</code> transforms the A and B matrices to complex Schur form, computes the solution of the resulting triangular system, and transforms this solution back [1].	
Limitations	<p>The continuous Lyapunov equation has a (unique) solution if the eigenvalues $\alpha_1, \alpha_2, \dots, \alpha_n$ of A and $\beta_1, \beta_2, \dots, \beta_n$ of B satisfy</p> $\alpha_i + \beta_j \neq 0 \quad \text{for all pairs } (i, j)$ <p>If this condition is violated, <code>lyap</code> produces the error message</p> <p>Solution does not exist or is not unique.</p>	
See Also	<code>dlap</code> <code>covar</code>	Solve discrete Lyapunov equations Covariance of system response to white noise

Reference

- [1] Bartels, R.H., and G.W. Stewart, "Solution of the Matrix Equation $AX + XB = C$," *Comm. of the ACM*, Vol. 15, No. 9, 1972.
- [2] Bryson, A.E., and Y.C. Ho, *Applied Optimal Control*, Hemisphere Publishing, 1975. pp. 328–338.

margin

Purpose	Compute gain and phase margins and associated crossover frequencies
Syntax	$[G_m, P_m, W_{cg}, W_{cp}] = \text{margin}(\text{sys})$ $[G_m, P_m, W_{cg}, W_{cp}] = \text{margin}(\text{mag}, \text{phase}, w)$ $\text{margin}(\text{sys})$
Description	<p><code>margin</code> calculates the gain margin, phase margin, and associated crossover frequencies of SISO open-loop models. The gain and phase margins indicate the relative stability of the control system when the loop is closed. When invoked without lefthand arguments, <code>margin</code> produces a Bode plot and displays the margins on this plot.</p> <p>The gain margin is the amount of gain increase required to make the loop gain unity at the frequency where the phase angle is -180°. In other words, the gain margin is $1/g$ if g is the gain at the -180° phase frequency. Similarly, the phase margin is the difference between the phase of the response and -180° when the loop gain is 1.0. The frequency at which the magnitude is 1.0 is called the unity-gain frequency or crossover frequency. It is generally found that gain margins of three or more combined with phase margins between 30 and 60 degrees result in reasonable trade-offs between bandwidth and stability.</p> <p>$[G_m, P_m, W_{cg}, W_{cp}] = \text{margin}(\text{sys})$ computes the gain margin G_m, the phase margin P_m, and the corresponding crossover frequencies W_{cg} and W_{cp}, given the SISO open-loop model <code>sys</code>. This function handles both continuous- and discrete-time cases. When faced with several crossover frequencies, <code>margin</code> returns the smallest gain and phase margins.</p> <p>$[G_m, P_m, W_{cg}, W_{cp}] = \text{margin}(\text{mag}, \text{phase}, w)$ derives the gain and phase margins from the Bode frequency response data (magnitude, phase, and frequency vector). Interpolation is performed between the frequency points to estimate the margin values. This approach is generally less accurate.</p> <p>When invoked without lefthand argument,</p> <p style="padding-left: 40px;"><code>margin(sys)</code></p> <p>plots the open-loop Bode response with the gain and phase margins marked by vertical lines.</p>

Example

You can compute the gain and phase margins of the open-loop discrete-time transfer function. Type:

```
hd = tf([0.04798 0.0464], [1 -1.81 0.9048], 0.1)
```

MATLAB responds with:

```
Transfer function:
  0.04798 z + 0.0464
-----
z^2 - 1.81 z + 0.9048
```

```
Sampling time: 0.1
```

Type:

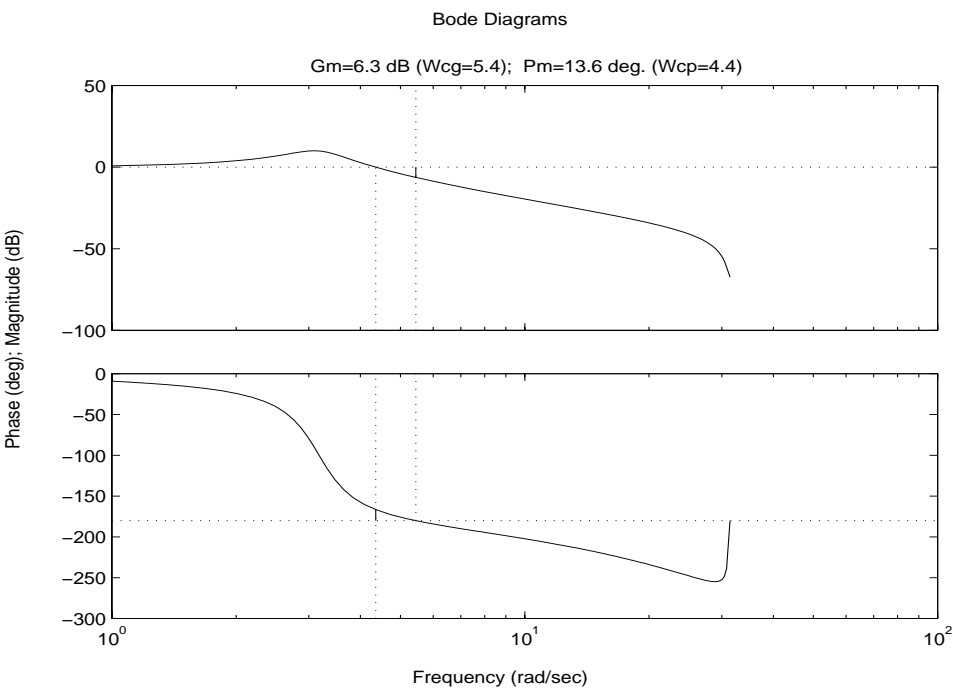
```
[Gm, Pm, Wcg, Wcp] = margin(hd);
[Gm, Pm, Wcg, Wcp]
```

and MATLAB returns:

```
ans =
    2.0517    13.5712    5.4374    4.3544
```

You can also display these margins graphically:

```
margin(hd)
```



Algorithm

The phase margin is computed using H_{∞} theory, and the gain margin by solving $H(j\omega) = \overline{H(j\omega)}$ for the frequency ω .

See Also

- | | |
|----------|-------------------------|
| lti view | LTI system viewer |
| bode | Bode frequency response |

Purpose	Minimal realization or pole-zero cancellation
Syntax	<pre>sysr = mi nreal (sys) sysr = mi nreal (sys, tol)</pre>
Description	<p><code>sysr = mi nreal (sys)</code> eliminates uncontrollable or unobservable states in state-space models, or canceling pole-zero pairs in transfer functions or zero-pole-gain models. The output <code>sysr</code> has minimal order and the same response characteristics as the original model <code>sys</code>.</p> <p><code>sysr = mi nreal (sys, tol)</code> specifies the tolerance used for state elimination or pole-zero cancellation. The default value is <code>tol = sqrt (eps)</code> and increasing this tolerance forces additional state deletions or pole-zero cancellations.</p>
Example	<p>The commands</p> <pre>g = zpk([], 1, 1) h = tf([2 1], [1 0]) cl oop = i nv(1+g*h) * g</pre> <p>produce the nonminimal zero-pole-gain model by typing <code>cl oop</code>:</p> <p>Zero/pole/gain:</p> $\frac{s}{(s-1)(s^2 + s + 1)}$ <p>To cancel the pole-zero pair at $s = 1$, type</p> <pre>cl oop = mi nreal (cl oop)</pre> <p>and MATLAB returns:</p> <p>Zero/pole/gain:</p> $\frac{s}{(s^2 + s + 1)}$
Algorithm	Pole-zero cancellation is a straightforward search through the poles and zeros looking for matches that are within tolerance. Transfer functions are first converted to zero-pole-gain form.

The state-space case is tackled with the staircase algorithm described in [1].

See Also bal real Gramian-based input/output balancing
 modred Model order reduction

Reference [1] Rosenbrock, M.M., *State-Space and Multivariable Theory*, John Wiley, 1970.

Purpose	Model order reduction
Syntax	<pre>rsys = modred(sys, elim) rsys = modred(sys, elim, 'mdc') rsys = modred(sys, elim, 'del')</pre>
Description	<p><code>modred</code> reduces the order of a continuous or discrete state-space model <code>sys</code>. This function is usually used in conjunction with <code>bal real</code>. Two order reduction techniques are available:</p> <p><code>rsys = modred(sys, elim)</code> or <code>rsys = modred(sys, elim, 'mdc')</code> produces a reduced-order model <code>rsys</code> with matching DC gain (or equivalently, matching steady state in the step response). The index vector <code>elim</code> specifies the states to be eliminated. The resulting model <code>rsys</code> has <code>length(elim)</code> fewer states. This technique consists of setting the derivative of the eliminated states to zero and solving for the remaining states.</p> <p><code>rsys = modred(sys, elim, 'del')</code> simply deletes the states specified by <code>elim</code>. While this method does not guarantee matching DC gains, it tends to produce better approximations in the frequency domain (see example below).</p> <p>If the state-space model <code>sys</code> has been balanced with <code>bal real</code> and the gramians have m small diagonal entries, you can reduce the model order by eliminating the last m states with <code>modred</code>.</p>

Example Consider the continuous fourth-order model

$$h(s) = \frac{s^3 + 11s^2 + 36s + 26}{s^4 + 14.6s^3 + 74.96s^2 + 153.7s + 99.65}$$

To reduce its order, first compute a balanced state-space realization with `bal real` by typing:

```
h = tf([1 11 36 26], [1 14.6 74.96 153.7 99.65])
[hb, g] = bal real (h)
g'
```

MATLAB returns:

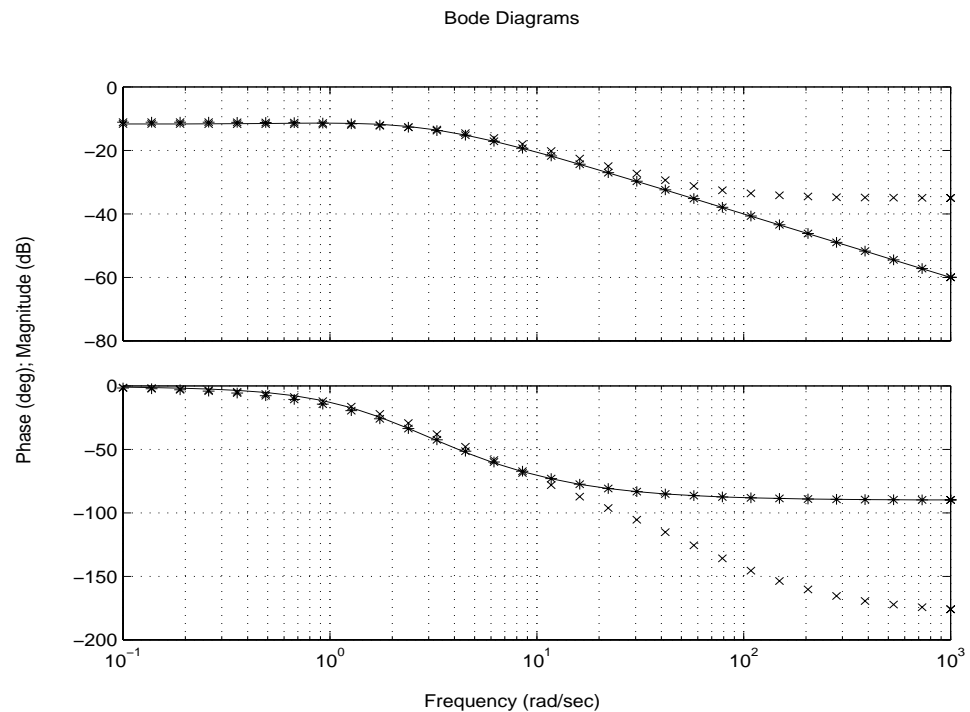
```
ans =  
1.3938e-01    9.5482e-03    6.2712e-04    7.3245e-06
```

The last three diagonal entries of the balanced gramians are small, so eliminate the last three states with `modred` using both matched DC gain and direct deletion methods:

```
hmdc = modred(hb, 2:4, 'mdc')  
hdel = modred(hb, 2:4, 'del')
```

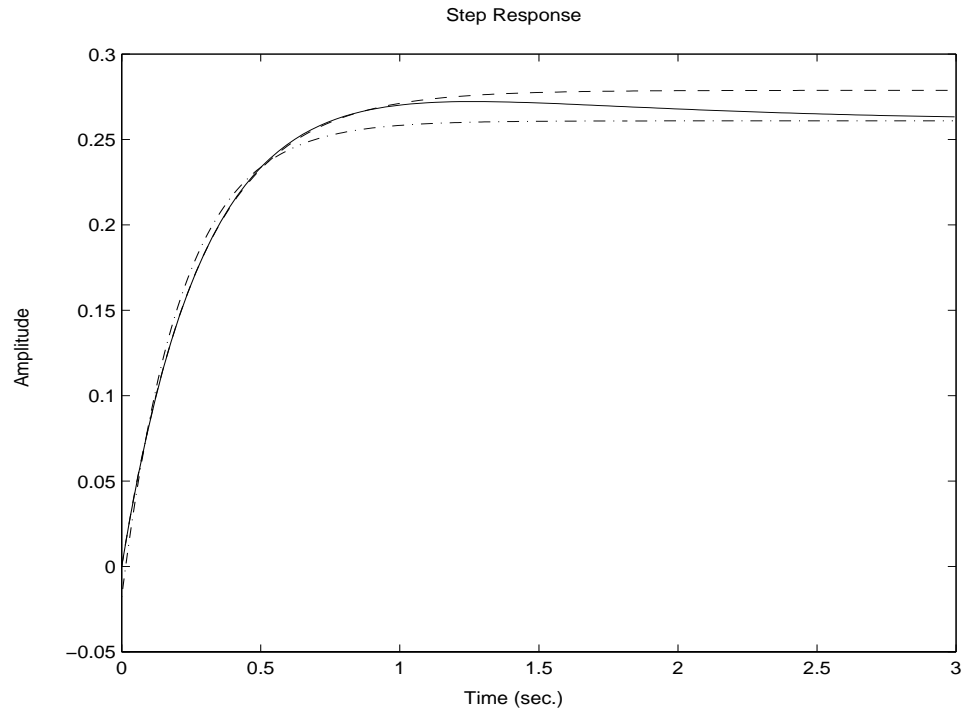
Both `hmdc` and `hdel` are first-order models. Compare their Bode responses against that of the original model $h(s)$:

```
bode(h, '-', hmdc, 'x', hdel, '*')
```



The reduced-order model `hdel` is clearly a better frequency-domain approximation of $h(s)$. Now compare the step responses:

```
step(h, '-', hmdc, '-.', hdel, '--')
```



While `hdel` accurately reflects the transient behavior, only `hmdc` gives the true steady-state response.

Algorithm

The algorithm for the matched DC gain method is as follows. For continuous-time models

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

the state vector is partitioned into x_1 , to be kept, and x_2 , to be eliminated:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} u$$

$$y = \begin{bmatrix} C_1 & C_2 \end{bmatrix} x + Du$$

Next, the derivative of x_2 is set to zero and the resulting equation is solved for x_2 . The reduced-order model is given by:

$$\dot{x}_1 = [A_{11} - A_{12}A_{22}^{-1}A_{21}]x_1 + [B_1 - A_{12}A_{22}^{-1}B_2]u$$

$$y = [C_1 - C_2A_{22}^{-1}A_{21}]x_1 + [D - C_2A_{22}^{-1}B_2]u$$

The discrete-time case is treated similarly by setting

$$x_2[n+1] = x_2[n]$$

Limitations

With the matched DC gain method, A_{22} must be invertible in continuous time, and $I - A_{22}$ must be invertible in discrete time.

See Also

bal real
mi nreal

Input/output balancing of state-space models
Minimal state-space realizations

Purpose	Superimpose a Nichols chart on a Nichols plot
Syntax	<code>ngrid</code>
Description	<p><code>ngrid</code> superimposes Nichols chart grid lines over the Nichols frequency response of a SISO LTI system. The range of the Nichols grid lines is set to encompass the entire Nichols frequency response.</p> <p>The chart relates the complex number $H/(1 + H)$ to H, where H is any complex number. For SISO systems, when H is a point on the open-loop frequency response, then</p> $\frac{H}{1 + H}$ <p>is the corresponding value of the closed-loop frequency response assuming unit negative feedback.</p> <p>If the current axis is empty, <code>ngrid</code> generates a new Nichols chart grid in the region -40 dB to 40 dB in magnitude and -360 degrees to 0 degrees in phase. If the current axis does not contain a SISO Nichols frequency response, <code>ngrid</code> returns a warning.</p>

Example Plot the Nichols response with Nichols grid lines for the system:

$$H(s) = \frac{-4s^4 + 48s^3 - 18s^2 + 250s + 600}{s^4 + 30s^3 + 282s^2 + 525s + 60}$$

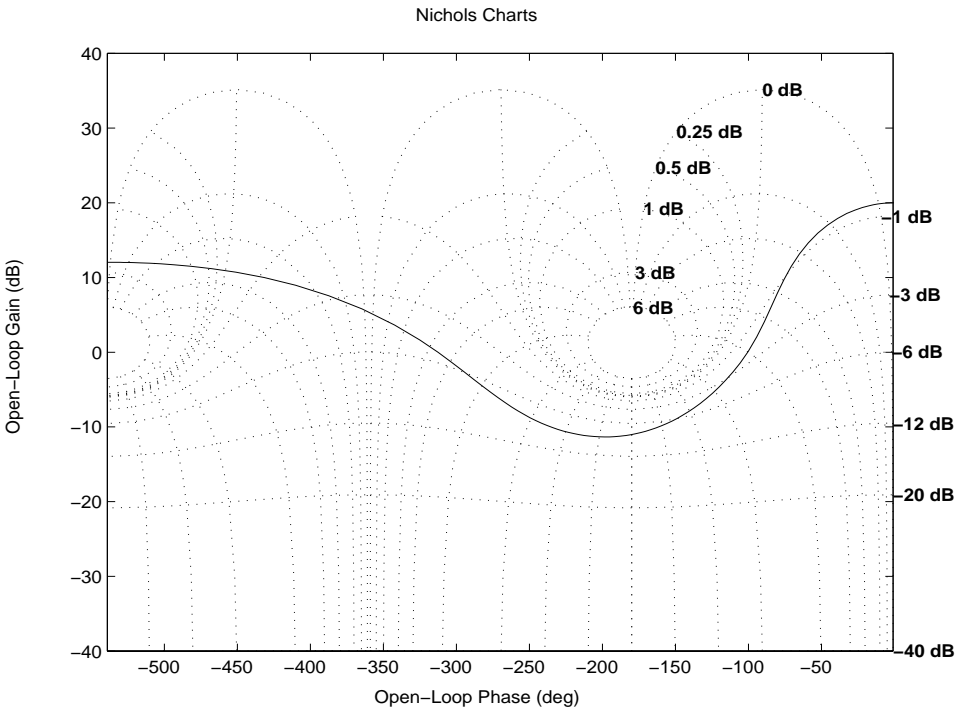
Type:

```
H = tf([-4 48 -18 250 600], [1 30 282 525 60])
```

MATLAB returns:

```
Transfer function:
- 4 s^4 + 48 s^3 - 18 s^2 + 250 s + 600
-----
s^4 + 30 s^3 + 282 s^2 + 525 s + 60
```

Type:
nichols(H)
ngrid



See Also nichols Nichols plots

Purpose

Nichols frequency response of LTI systems

Syntax

```

ni chol s(sys)
ni chol s(sys, w)

ni chol s(sys1, sys2, . . . , sysN)
ni chol s(sys1, sys2, . . . , sysN, w)
ni chol s(sys1, 'Pl otStyl e1' , . . . , sysN, 'Pl otStyl eN' )

[ mag, phase, w] = ni chol s(sys)

```

Description

`ni chol s` computes the frequency response of an LTI system and plots it in the Nichols coordinates. Nichols plots are useful to analyze open- and closed-loop properties of SISO systems, but offer little insight into MIMO control loops. Use `ngri d` to superimpose a Nichols chart on an existing SISO Nichols plot.

`ni chol s(sys)` produces a Nichols plot of the LTI model `sys`. This model can be continuous or discrete, SISO or MIMO. In the MIMO case, `ni chol s` produces an array of Nichols plots, each plot showing the response of one particular I/O channel. The frequency range and gridding are determined automatically based on the system poles and zeros.

`ni chol s(sys, w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmi n, wmax]`, set `w = { wmi n, wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `l ogspace` to generate logarithmically spaced frequency vectors. Frequencies should be specified in radians/sec.

`ni chol s(sys1, sys2, . . . , sysN)` or `ni chol s(sys1, sys2, . . . , sysN, w)` superimposes the Nichols plots of several LTI models on a single figure. All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous- and discrete-time systems. You can also specify a distinctive color, linestyle, and/or marker for each system plot with the syntax

```
ni chol s(sys1, 'Pl otStyl e1' , . . . , sysN, 'Pl otStyl eN' )
```

See `bode` for an example.

When invoked with lefthand arguments,

```
[mag, phase, w] = nichols(sys)
[mag, phase] = nichols(sys, w)
```

return the magnitude and phase (in degrees) of the frequency response at the frequencies w (in rad/sec). The outputs `mag` and `phase` are 3-D arrays similar to those produced by `bode` (see Reference page for `bode`). They have dimensions

$(\text{number of outputs}) \times (\text{number of inputs}) \times (\text{length of } w)$

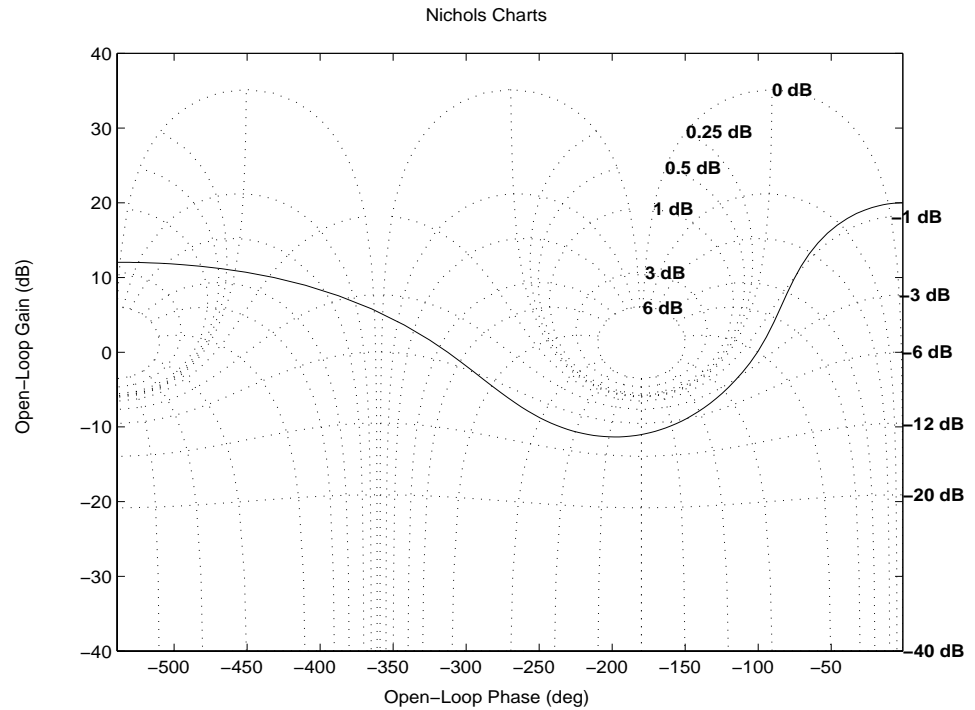
Example

Plot the Nichols response of the system

$$H(s) = \frac{-4s^4 + 48s^3 - 18s^2 + 250s + 600}{s^4 + 30s^3 + 282s^2 + 525s + 60}$$

```
num = [-4 48 -18 250 600];
den = [1 30 282 525 60];
H = tf(num, den)
```

```
nichols(H); grid
```



Algorithm See bode.

Limitations Same as for bode.

See Also

lti view	LTI system viewer
bode	Bode plot
nyquist	Nyquist plot
sigma	Singular value plot
freqresp	Frequency response computation
evalfr	Response at single complex frequency

norm

Purpose LTI system norms

Syntax

```
norm(sys)
norm(sys, 2)

norm(sys, inf)
norm(sys, inf, tol)
[norm, fpeak] = norm(sys)
```

Description norm computes the H_2 or L_∞ norm of a continuous- or discrete-time LTI model.

H_2 Norm

The H_2 norm of a stable continuous system with transfer function $H(s)$ is the root-mean-square of its impulse response, or equivalently

$$\|H\|_2 = \sqrt{\frac{1}{2\pi} \int_{-\infty}^{\infty} \text{Trace}(H(j\omega)^H H(j\omega)) d\omega}$$

This norm measures the steady-state covariance (or power) of the output response $y = Hw$ to unit white noise inputs w :

$$\|H\|_2^2 = \lim_{t \rightarrow \infty} E\{y(t)^T y(t)\} \quad , \quad E(w(t)w(\tau)^T) = \delta(t - \tau)I$$

Infinity Norm

The infinity norm is the peak gain of the frequency response, that is,

$$\|H(s)\|_\infty = \max_{\omega} |H(j\omega)| \quad (\text{SISO case})$$

$$\|H(s)\|_\infty = \max_{\omega} \sigma_{\max}(H(j\omega)) \quad (\text{MIMO case})$$

where $\sigma_{\max}(\cdot)$ denotes the largest singular value of a matrix.

The discrete-time counterpart is

$$\|H(z)\|_{\infty} = \max_{\theta \in [0, \pi]} \sigma_{\max}(H(e^{j\theta}))$$

Usage

`norm(sys)` or `norm(sys, 2)` returns the H_2 norm of the LTI model `sys`. This norm is infinite in the following cases:

- `sys` is unstable.
- `sys` is continuous and has a nonzero feedthrough (that is, nonzero gain at the frequency $\omega = \infty$).

Note that `norm(sys)` produces the same result as

```
sqrt(trace(covar(sys, 1)))
```

`norm(sys, inf)` computes the infinity norm of `sys`. This norm is infinite if `sys` has poles on the imaginary axis in continuous time, or on the unit circle in discrete time.

`norm(sys, inf, tol)` sets the desired relative accuracy on the computed infinity norm (the default value is `tol = 1e-2`).

`[ni nf, fpeak] = norm(sys, inf)` also returns the frequency `fpeak` where the gain achieves its peak value.

Example

Consider the discrete-time transfer function

$$H(z) = \frac{z^3 - 2.841z^2 + 2.875z - 1.004}{z^3 - 2.417z^2 + 2.003z - 0.5488}$$

with sample time 0.1 second. Compute its H_2 norm by typing:

```
H = tf([1 -2.841 2.875 -1.004], [1 -2.417 2.003 -0.5488], 0.1)
norm(H)
```

to which MATLAB responds:

```
ans =
    1.2438
```

and its infinity norm by typing:

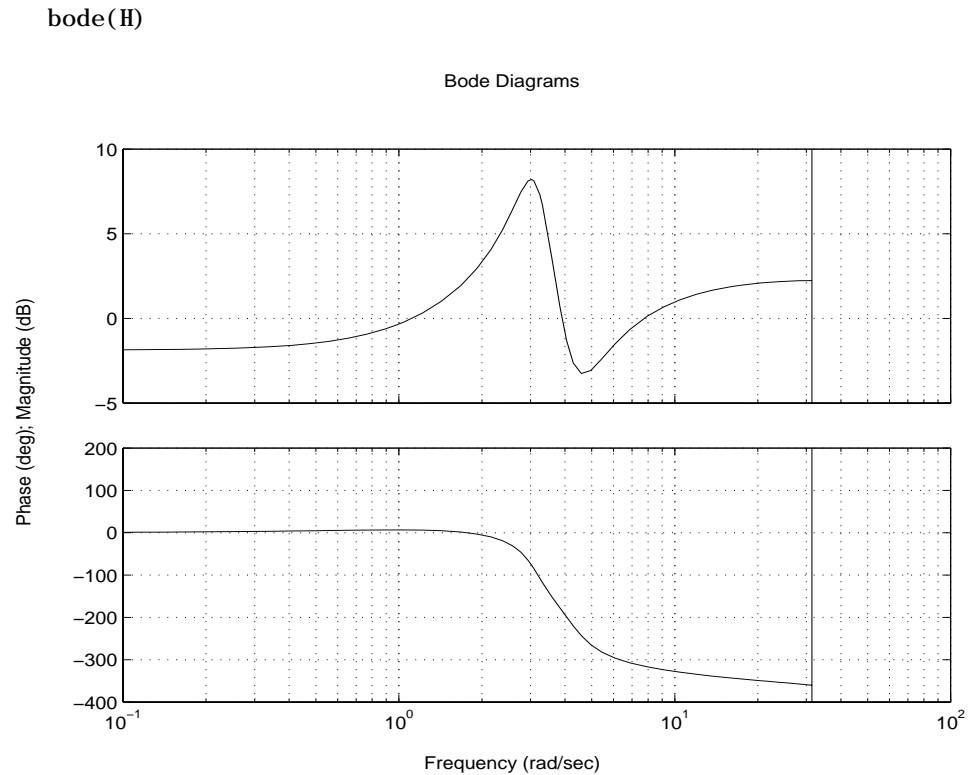
```
[ni nf, fpeak] = norm(H, inf)
```

to which MATLAB responds:

```
ni nf =  
2.5488
```

```
fpeak =  
3.0844
```

These values are confirmed by the Bode plot of $H(z)$:



The gain indeed peaks at approximately 3 rad/sec and its peak value in dB is found by typing:

```
20*log10(ninf)
```

MATLAB returns:

```
ans =  
8.1268
```

Algorithm

norm uses the same algorithm as covar for the H_2 norm, and the algorithm of [1] for the infinity norm. sys is first converted to state space.

See Also

bode	Bode plot
sigma	Singular value plot
freqresp	Frequency response computation

Reference

[1] Bruisma, N.A., and M. Steinbuch, "A Fast Algorithm to Compute the H_∞ -Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287–293.

nyquist

Purpose Nyquist frequency response of LTI systems

Syntax

```
nyquist(sys)
nyquist(sys, w)

nyquist(sys1, sys2, ..., sysN)
nyquist(sys1, sys2, ..., sysN, w)
nyquist(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')

[re, im, w] = nyquist(sys)
```

Description `nyquist` calculates the Nyquist frequency response of LTI systems. When invoked without lefthand arguments, `nyquist` produces a Nyquist plot on the screen. Nyquist plots are used to analyze system properties including gain margin, phase margin, and stability.

`nyquist(sys)` plots the Nyquist response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. In the MIMO case, `nyquist` produces an array of Nyquist plots, each plot showing the response of one particular I/O channel. The frequency points are chosen automatically based on the system poles and zeros.

`nyquist(sys, w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval $[w_{min}, w_{max}]$, set `w = {wmin, wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. Frequencies should be specified in rad/sec.

`nyquist(sys1, sys2, ..., sysN)` or `nyquist(sys1, sys2, ..., sysN, w)` superimposes the Nyquist plots of several LTI models on a single figure. All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous- and discrete-time systems. You can also specify a distinctive color, linestyle, and/or marker for each system plot with the syntax

```
nyquist(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')
```

See `bode` for an example.

When invoked with lefthand arguments,

```
[re, im, w] = nyquist(sys)
[re, im] = nyquist(sys, w)
```

return the real and imaginary parts of the frequency response at the frequencies w (in rad/sec). re and im are 3-D arrays with the frequency as last dimension (see “Arguments” below for details).

Arguments

The output arguments re and im are 3-D arrays with dimensions
 $(\text{number of outputs}) \times (\text{number of inputs}) \times (\text{length of } w)$

For SISO systems, the scalars $re(1, 1, k)$ and $im(1, 1, k)$ are the real and imaginary parts of the response at the frequency $\omega_k = w(k)$:

$$re(1, 1, k) = \text{Re}(h(j\omega_k))$$

$$im(1, 1, k) = \text{Im}(h(j\omega_k))$$

For MIMO systems with transfer function $H(s)$, $re(:, :, k)$ and $im(:, :, k)$ give the real and imaginary parts of $H(j\omega_k)$ (both arrays with as many rows as outputs and as many columns as inputs). Thus,

$$re(i, j, k) = \text{Re}(h_{ij}(j\omega_k))$$

$$im(i, j, k) = \text{Im}(h_{ij}(j\omega_k))$$

where h_{ij} is the transfer function from input j to output i .

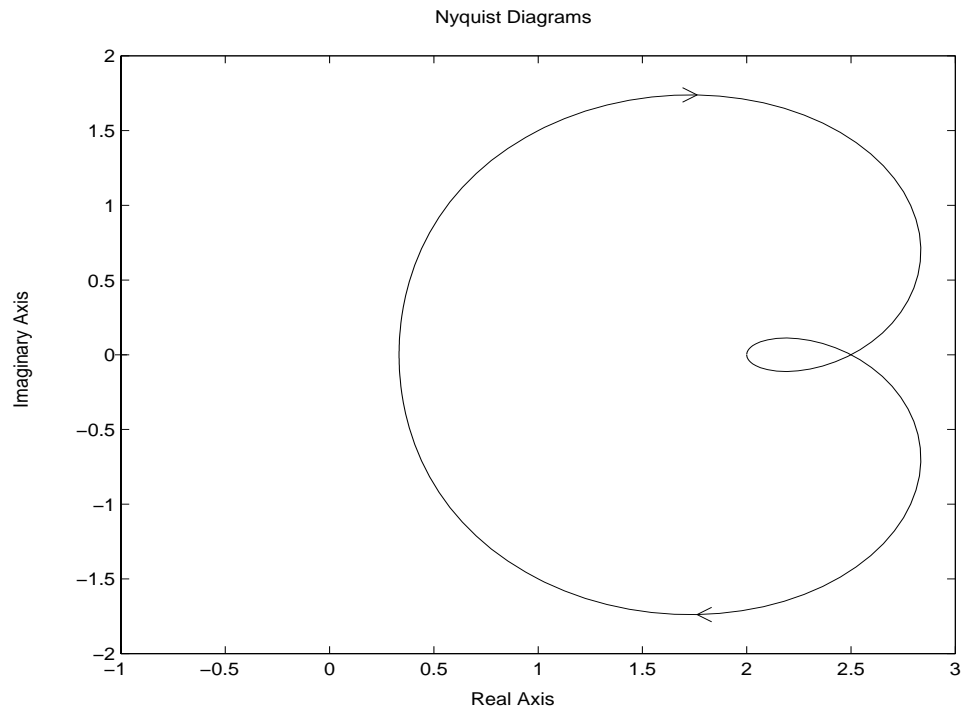
nyquist

Example

Plot the Nyquist response of the system

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
H = tf([2 5 1], [1 2 3])  
nyquist(H)
```



Limitations

See bode.

See Also

ltiview
bode
nichols
sigma
freqresp
evalfr

LTI system viewer
Bode plot
Nichols plot
Singular value plot
Frequency response computation
Response at single complex frequency

Purpose Form the observability matrix

Syntax `Ob = obsv(A, B)`
`Ob = obsv(sys)`

Description `obsv` computes the observability matrix for state-space systems. For an n -by- n matrix A and a p -by- n matrix C , `obsv(A, C)` returns the observability matrix:

$$Ob = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

with n rows and np columns.

`Ob = obsv(sys)` calculates the observability matrix of the state-space model `sys`. This syntax is equivalent to executing

`Ob = obsv(sys.A, sys.C)`

The model is observable if `Ob` has full rank n .

Example Determine if the pair

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

is observable. Type:

`Ob = obsv(A, C);`

`% Number of unobservable states`
`unob = length(A) - rank(Ob)`

obsv

MATLAB responds with:

```
unob =  
      0
```

See Also

[obsvf](#)

Compute the observability staircase form

Purpose	Compute the observability staircase form
Syntax	$[Abar, Bbar, Cbar, T, k] = \text{obsvf}(A, B, C)$ $[Abar, Bbar, Cbar, T, k] = \text{obsvf}(A, B, C, tol)$
Description	<p>If the observability matrix of (A, C) has rank $r \leq n$, where n is the size of A, then there exists a similarity transformation such that:</p> $\bar{A} = TAT^T, \quad \bar{B} = TB, \quad \bar{C} = CT^T$ <p>where T is unitary and the transformed system has a <i>staircase</i> form with the unobservable modes, if any, in the upper left corner:</p> $\bar{A} = \begin{bmatrix} A_{no} & A_{12} \\ 0 & A_o \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} B_{no} \\ B_o \end{bmatrix}, \quad \bar{C} = \begin{bmatrix} 0 & C_o \end{bmatrix}$ <p>where (C_o, A_o) is observable, and the eigenvalues of A_{no} are the unobservable modes.</p> <p>$[Abar, Bbar, Cbar, T, k] = \text{obsvf}(A, B, C)$ decomposes the state-space system with matrices A, B, and C into the observability staircase form $Abar$, $Bbar$, and $Cbar$, as described above. T is the similarity transformation matrix and k is a vector of length n, where n is the number of states in A. Each entry of k represents the number of observable states factored out during each step of the transformation matrix calculation [1]. The number of nonzero elements in k indicates how many iterations were necessary to calculate T, and $\text{sum}(k)$ is the number of states in A_o, the observable portion of $Abar$.</p> <p>$\text{obsvf}(A, B, C, tol)$ uses the tolerance tol when calculating the observable/unobservable subspaces. When the tolerance is not specified, it defaults to $10*n*\text{norm}(a, 1)*\text{eps}$.</p>

Example

Form the observability staircase form of

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

by typing:

```
[Abar, Bbar, Cbar, T, k] = obsvf(A, B, C)
```

MATLAB returns:

$$Abar = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$Bbar = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$Cbar = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$k = \begin{bmatrix} 2 & 0 \end{bmatrix}$$

Algorithm

obsvf is an M-file that implements the Staircase Algorithm of [1] by calling ctrbf and using duality.

See Also

obsv	Calculate the observability matrix
ctrbf	Compute the controllability staircase form

Reference

- [1] Rosenbrock, M.M., *State-Space and Multivariable Theory*, John Wiley, 1970.

ord2

Purpose Generate continuous second-order systems

Syntax [A, B, C, D] = ord2(wn, z)
[num, den] = ord2(wn, z)

Description [A, B, C, D] = ord2(wn, z) generates the state-space description (A, B, C, D) of the second-order system:

$$h(s) = \frac{1}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

given the natural frequency ω_n and damping factor ζ . Use `ss` to turn this description into a state-space object.

[num, den] = ord2(wn, z) returns the numerator and denominator of the second-order transfer function. Use `tf` to form the corresponding transfer function object.

Example Generate an LTI model of the second-order transfer function with damping factor $\zeta = 0.4$ and natural frequency $\omega_n = 2.4$ rad/sec:

Type:

```
[num, den] = ord2(2.4, 0.4)
```

MATLAB responds with:

```
num =  
    1  
den =  
    1.0000    1.9200    5.7600
```

Type:

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

MATLAB responds with:

```
Transfer function:  
    1  
-----  
s^2 + 1.92 s + 5.76
```

See Also

`rmodel`, `rss`
`ss`
`tf`

Generate random stable continuous models
Create a state-space LTI model
Create a transfer function LTI model

pade

Purpose Pade approximation of time delays

Syntax `[num, den] = pade(Td, N)`
`pade(Td, N)`

`sysx = pade(sys, N)`
`sysx = pade(sys, [N1, . . . , Nm])`

Description `pade` approximates time delays by rational LTI models. Such approximations are useful to model time delay effects such as transport and computation delays within the context of continuous-time systems. The Laplace transform of a time delay of T_d seconds is $\exp(-sT_d)$. This exponential transfer function is approximated by a rational transfer function using the Pade approximation formulas [1].

`[num, den] = pade(Td, N)` returns the N th-order (diagonal) Pade approximation of the continuous-time delay $\exp(-sT_d)$ in transfer function form. The row vectors `num` and `den` contain the numerator and denominator coefficients in descending powers of s . Both are N th-order polynomials.

When invoked without lefthand argument,

`pade(Td, N)`

plots the step and phase responses of the N th-order Pade approximation and compares them with the exact responses of the time delay `Td`. Note that the Pade approximation has unit gain at all frequencies.

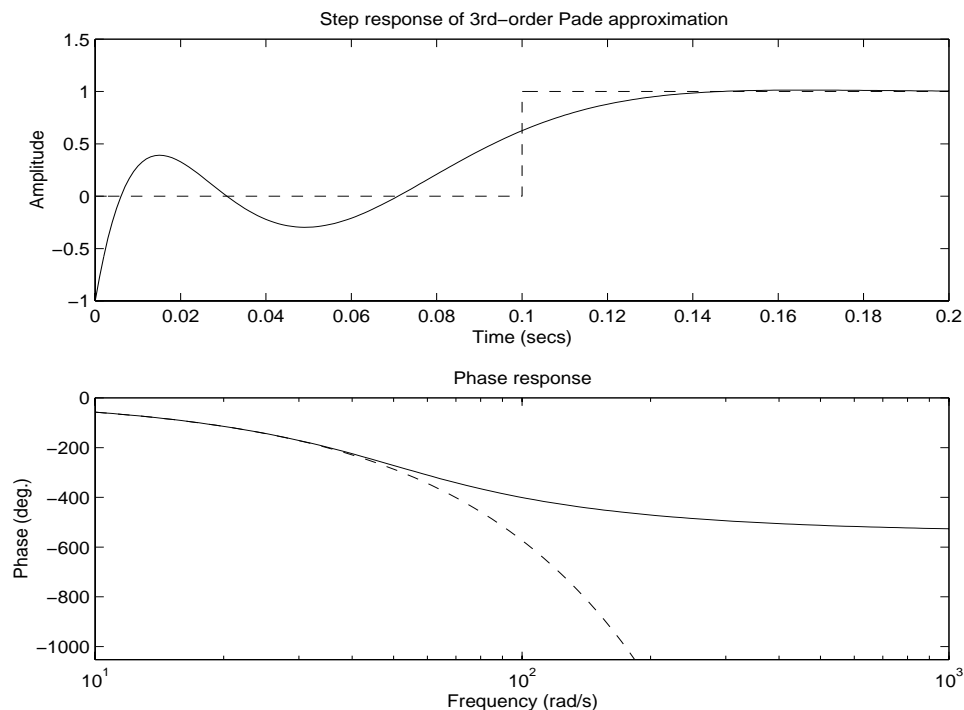
`sysx = pade(sys, N)` produces a delay-free approximation `sysx` of the continuous delay system `sys`. See page 2-37 for details on LTI models with input delays. All input delays are replaced by their N th-order Pade approximation. If `sys` is a multi-input system with m inputs, you can also specify an independent approximation order for each input channel by

`sysx = pade(sys, [N1, . . . , Nm])`

Example

Compute a third-order Pade approximation of a 0.1 second time delay and compare the time and frequency responses of the true delay and its approximation:

```
pade(0.1, 3)
```

**Limitations**

High-order Pade approximations produce transfer functions with clustered poles. Because such pole configurations tend to be very sensitive to perturbations, Pade approximations with order $N > 10$ should be avoided.

See Also

c2d Discretization of continuous system

Reference

[1] Golub, G. H. and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 557–558.

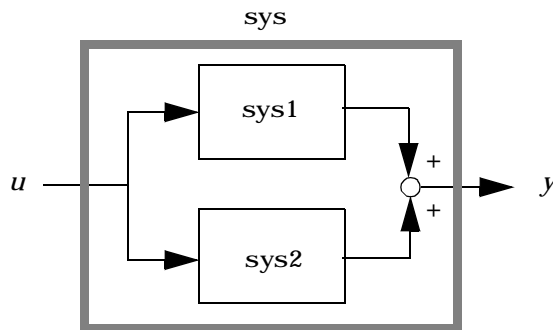
parallel

Purpose Parallel connection of two LTI models

Syntax
`sys = parallel(sys1, sys2)`
`sys = parallel(sys1, sys2, inp1, inp2, out1, out2)`

Description `parallel` connects two LTI models in parallel. This function accepts any type of LTI model. The two systems must be either both continuous or both discrete with identical sample time. Static gains are neutral and can be specified as regular matrices.

`sys = parallel(sys1, sys2)` forms the basic parallel connection shown below.

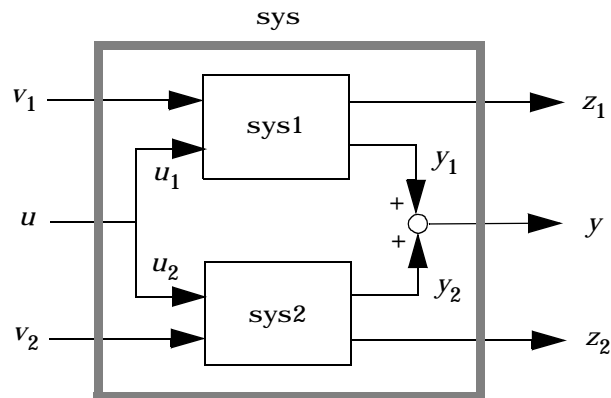


This command is equivalent to the direct addition

`sys = sys1 + sys2`

(See page 2-29 for details on LTI system addition.)

`sys = parallel(sys1, sys2, i np1, i np2, out 1, out 2)` forms the more general parallel connection:



The index vectors `i np1` and `i np2` specify which inputs u_1 of `sys1` and which inputs u_2 of `sys2` are connected. Similarly, the index vectors `out 1` and `out 2` specify which outputs y_1 of `sys1` and which outputs y_2 of `sys2` are summed. The resulting model `sys` has $[v_1 ; u ; v_2]$ as inputs and $[z_1 ; y ; z_2]$ as outputs.

Example

See page 7-53 for an example.

See Also

`append`
`series`
`feedback`

Append LTI systems
 Series connection
 Feedback connection

place

Purpose Pole placement design

Syntax $K = \text{place}(A, B, p)$
 $[K, \text{prec}, \text{message}] = \text{place}(A, B, p)$

Description Given the single- or multi-input system

$$\dot{x} = Ax + Bu$$

and a vector p of desired self-conjugate closed-loop pole locations, `place` computes a gain matrix K such that the state feedback $u = -Kx$ places the closed-loop poles at the locations p . In other words, the eigenvalues of $A - BK$ match the entries of p (up to the ordering).

$K = \text{place}(A, B, p)$ computes a feedback gain matrix K that achieves the desired closed-loop pole locations p , assuming all the inputs of the plant are control inputs. The length of p must match the row size of A . `place` works for multi-input systems and is based on the algorithm from [1]. This algorithm uses the extra degrees of freedom to find a solution that minimizes the sensitivity of the closed-loop poles to perturbations in A or B .

$[K, \text{prec}, \text{message}] = \text{place}(A, B, p)$ also returns `prec`, an estimate of how closely the eigenvalues of $A - BK$ match the specified locations p (`prec` measures the number of accurate decimal digits in the actual closed-loop poles). If some nonzero closed-loop pole is more than 10% off from the desired location, `message` contains a warning message.

You can also use `place` for estimator gain selection by transposing the A matrix and substituting C' for B :

$$L = \text{place}(A', C', p)'$$

Example Consider a state-space system (a, b, c, d) with two inputs, three outputs, and three states. You can compute the feedback gain matrix needed to place the closed-loop poles at $p = [1.1 \ 2.3 \ 5.0]$ by

$$p = [1.1 \ 2.3 \ 5.0];$$
$$K = \text{place}(a, b, p)$$

Algorithm

pl ace uses the algorithm of [1] which, for multi-input systems, optimizes the choice of eigenvectors for a robust solution. We recommend pl ace rather than acker even for single-input systems.

In high-order problems, some choices of pole locations result in very large gains. The sensitivity problems attached with large gains suggest caution in the use of pole placement techniques. See [2] for results from numerical testing.

See Also

lqr State-feedback LQ regulator design
rl locus, rlocfind Root locus design

Reference

- [1] Kautsky, J. and N.K. Nichols, "Robust Pole Assignment in Linear State Feedback," *Int. J. Control*, 41 (1985), pp. 1129–1155.
- [2] Laub, A.J. and M. Wette, *Algorithms and Software for Pole Assignment and Observers*, UCRL-15646 Rev. 1, EE Dept., Univ. of Calif., Santa Barbara, CA, Sept. 1984.

pole

Purpose	Compute the poles of an LTI system	
Syntax	<code>p = pole(sys)</code>	
Description	<code>pole</code> computes the poles <code>p</code> of the SISO or MIMO LTI model <code>sys</code> .	
Algorithm	<p>For state-space models, the poles are the eigenvalues of the A matrix, or the generalized eigenvalues of $A - \lambda E$ in the descriptor case.</p> <p>For SISO transfer functions or zero-pole-gain models, the poles are simply the denominator roots (see <code>roots</code>).</p> <p>For MIMO transfer functions (or zero-pole-gain models), the poles are a subset of the collection of poles for each SISO entry. It is numerically difficult to determine the exact order of a MIMO transfer function and to eliminate redundant poles. The function <code>pole</code> converts such transfer functions to state space and computes the eigenvalues of the resulting state-space model. While the state-space realization algorithm (<code>ss</code>) addresses part of the minimal-order issues, <code>pole</code> is not guaranteed to produce the minimal set of poles except in the SIMO and MIMO cases.</p>	
Limitations	<p>Multiple poles are numerically sensitive and cannot be computed to high accuracy. A pole λ with multiplicity m typically gives rise to a cluster of computed poles distributed on a circle with center λ and radius of order</p> $\rho \approx \text{eps}^{1/m}$	
See Also	<code>damp</code> <code>esort</code> , <code>dsort</code> <code>tzero</code> <code>pzmap</code>	Damping and natural frequency of system poles Sort system poles Compute (transmission) zeros Pole-zero map

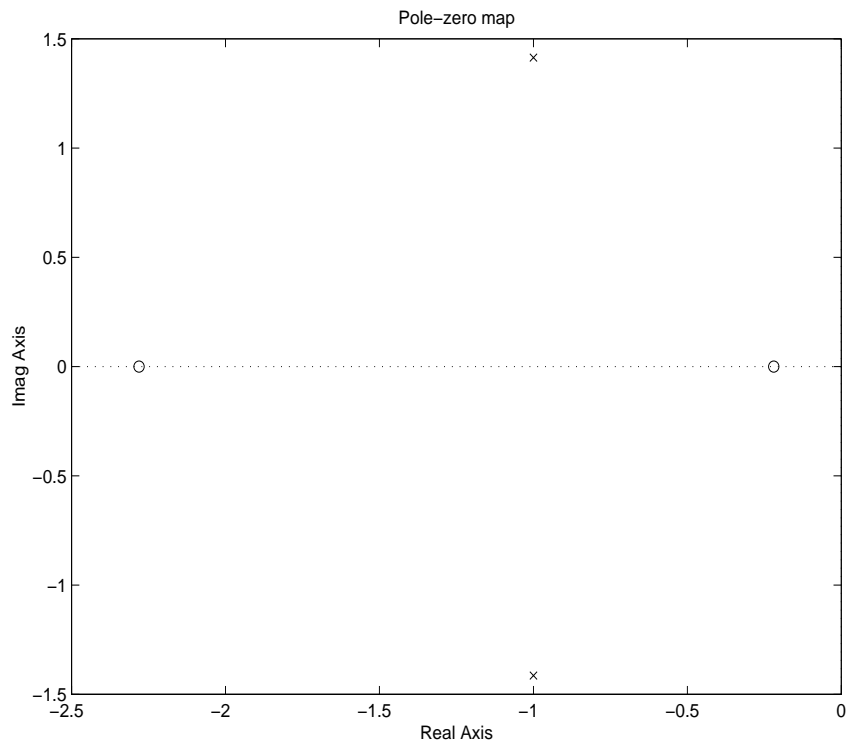
Purpose	Pole-zero map of LTI system
Syntax	<pre>pzmap(sys) [p, z] = pzmap(sys)</pre>
Description	<p><code>pzmap(sys)</code> plots the pole-zero map of the continuous- or discrete-time LTI model <code>sys</code>. For SISO systems, <code>pzmap</code> plots the transfer function poles and zeros. For MIMO systems, it plots the system poles and transmission zeros. The poles are plotted as x's and the zeros are plotted as o's.</p> <p>When invoked without lefthand arguments,</p> <pre>[p, z] = pzmap(sys)</pre> <p>returns the system poles and (transmission) zeros in the column vectors <code>p</code> and <code>z</code>. No plot is drawn on the screen.</p> <p>You can use the functions <code>sgri d</code> or <code>zgri d</code> to plot lines of constant damping ratio and natural frequency in the <i>s</i>- or <i>z</i>-plane.</p>

Example

Plot the poles and zeros of the continuous-time system:

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
H = tf([2 5 1],[1 2 3])  
pzmap(H)
```



Algorithm

pzmap uses a combination of pole and tzero.

See Also

pole	Compute system poles
tzero	Compute system (transmission) zeros
damp	Damping and natural frequency of system poles
esort, dsort	Sort system poles

rl locus
sgri d, zgri d

Root locus
Plot lines of constant damping and natural frequency

Purpose Form regulator given state-feedback and estimator gains

Syntax
`rsys = reg(sys, K, L)`
`rsys = reg(sys, K, L, sensors, known, control s)`

Description `rsys = reg(sys, K, L)` forms a dynamic regulator or compensator `rsys` given a state-space model `sys` of the plant, a state-feedback gain matrix `K`, and an estimator gain matrix `L`. The gains `K` and `L` are typically designed using pole placement or LQG techniques. The function `reg` handles both continuous- and discrete-time cases.

This syntax assumes that all inputs of `sys` are controls, and all outputs are measured. The regulator `rsys` is obtained by connecting the state-feedback law $u = -Kx$ and the state estimator with gain matrix `L` (see `estim`). For a plant with equations

$$\dot{x} = Ax + Bu$$

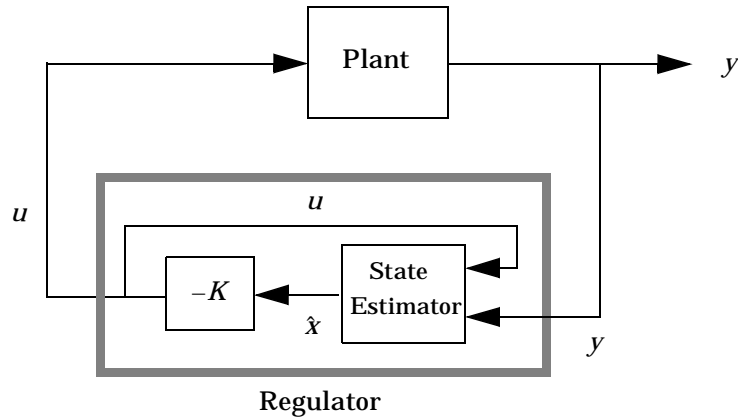
$$y = Cx + Du$$

this yields the regulator:

$$\dot{\hat{x}} = [A - LC - (B - LD)K] \hat{x} + Ly$$

$$u = -K\hat{x}$$

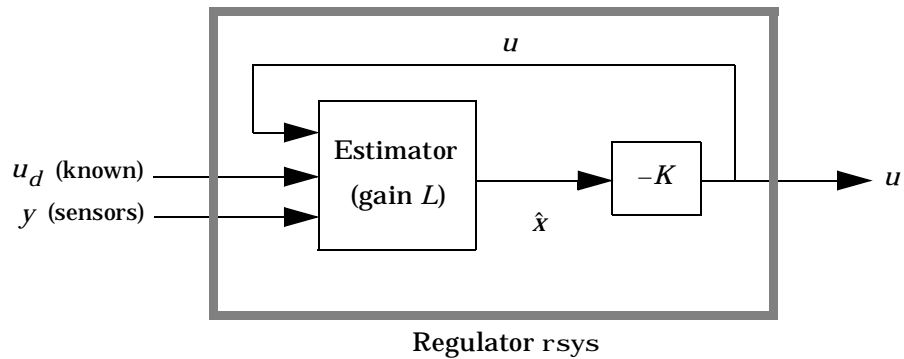
This regulator should be connected to the plant using *positive* feedback.



`rsys = reg(sys, K, L, sensors, known, control s)` handles more general regulation problems where

- The plant inputs consist of controls u , known inputs u_d , and stochastic inputs w .
- Only a subset y of the plant outputs is measured.

The index vectors `sensors`, `known`, and `control s` specify y , u_d , and u as subsets of the outputs and inputs of `sys`. The resulting regulator uses $[u_d; y]$ as inputs to generate the commands u (see figure below).



Example

Given a continuous-time state-space model

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

with seven outputs and four inputs, suppose you have designed:

- A state-feedback controller gain K using inputs 1, 2, and 4 of the plant as control inputs
- A state estimator with gain L using outputs 4, 7, and 1 of the plant as sensors, and input 3 of the plant as an additional known input

You can then connect the controller and estimator and form the complete regulation system by

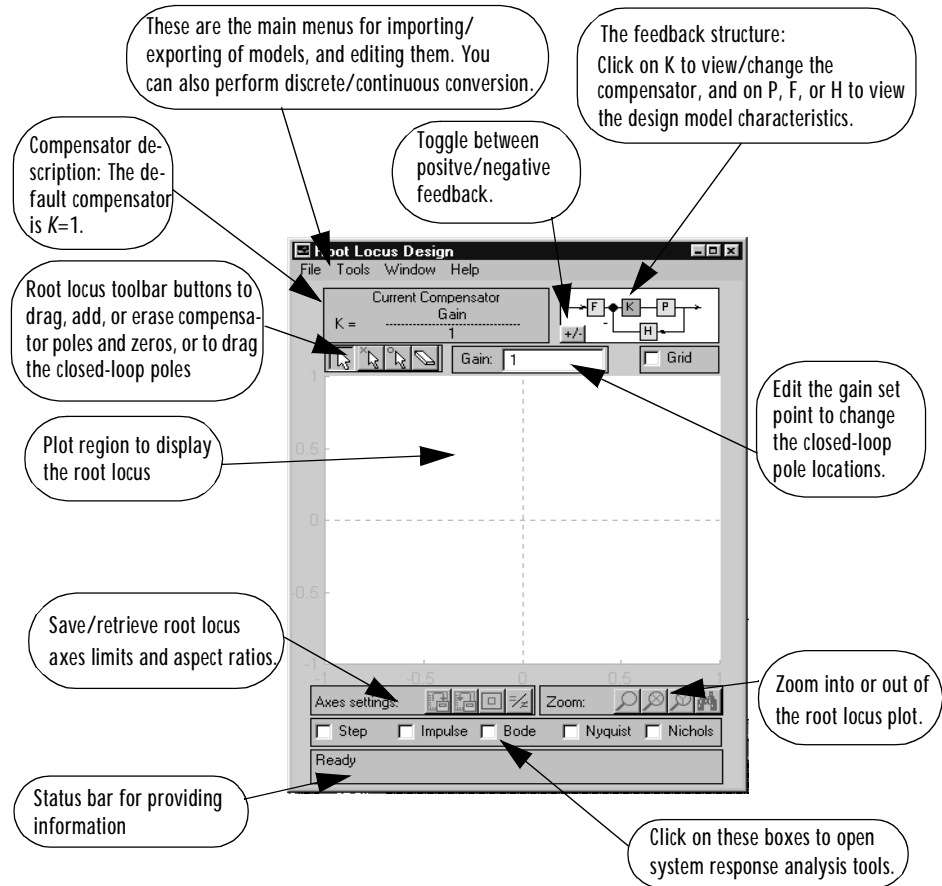
```
controls = [1, 2, 4];
sensors = [4, 7, 1];
known = [3];
regulator = reg(sys, K, L, sensors, known, controls)
```

See Also

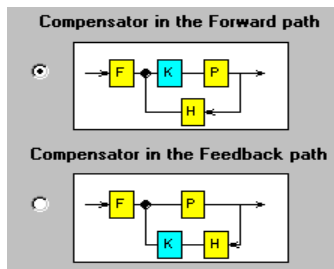
<code>lqgreg</code>	Form LQG regulator
<code>estim</code>	Form state estimator given estimator gain
<code>kalman</code>	Kalman estimator design
<code>place</code>	Pole placement
<code>lqr, dlqr</code>	State-feedback LQ regulator

Purpose	Initialize the Root Locus Design GUI
Syntax	<code>rl tool</code> <code>rl tool (sys)</code> <code>rl tool (sys, comp)</code> <code>rl tool (sys, comp, LocationFlag, FeedbackSign)</code>
Description	When invoked without input arguments, <code>rl tool</code> initializes a new Root Locus Design GUI for interactive compensator design. This GUI allows you to design a single-input/single-output (SISO) compensator using root locus techniques.

The Root Locus Design GUI looks like this:



This tool can be applied to SISO LTI systems whose feedback structure is in one of the following two configurations:



In either configuration, **F** is a pre-filter, **P** is the plant model, **H** is the sensor dynamics, and **K** is the compensator to be designed. In terms of the GUI design procedure, once you specify them, **F**, **P**, and **H** are *fixed* in the feedback structure. This triple, along with the feedback structure, is called the *design model*.

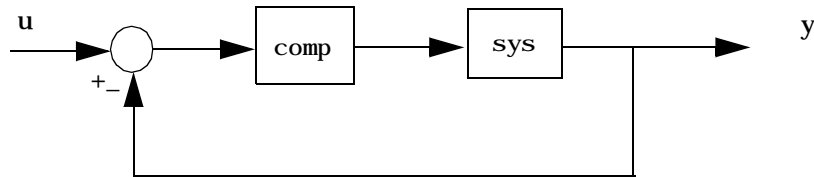
A design model can be constructed for the GUI by selecting the **Import Model** menu item from the **File** menu of the Root Locus Design GUI. Once you select the item, the **Import Design Model** window opens. You can then import SISO LTI models that have been created with `ss`, `tf`, or `zpk` in your workspace or on your disk (or SISO LTI blocks contained in open or saved Simulink models) into **F**, **P**, and **H**. Otherwise, you can specify your design model by defining **F**, **P**, and **H** manually with LTI models created using `ss`, `tf`, or `zpk` in the text boxes provided on the **Import Design Model** window.

`rltool(sys)` initializes a Root Locus Design GUI for the plant model **P** = `sys`. `sys` is any SISO LTI object (created with `ss`, `tf`, or `zpk`) that exists in the MATLAB workspace.

`rltool(sys, comp)` also initializes a Root Locus Design GUI for the design model `sys`. In addition, the root locus compensator is initialized to `comp`, where `comp` is any SISO LTI object that exists in the MATLAB workspace.

When either the plant, or both the plant and the compensator are provided as arguments to `rltool`, the root locus of the closed-loop poles and their locations for the current compensator gain are loaded and drawn on the Root Locus Design GUI. The closed-loop model is generated by placing the compensator

and design model in the forward loop of a negative unity feedback system, as shown in the diagram below:



In this case, **F** and **H** are taken to be 1, while **P** is sys. If you want to include **F** and **H** in the design model after loading `rltool(sys)` or `rltool(sys, comp)`, select the **Import Model** menu item from the **File** menu of the Root Locus Design GUI to load **F** and **H**.

`rltool(sys, comp, LocationFlag, FeedbackSign)` allows you to override the default compensator location and feedback sign. `LocationFlag` can be either the number 1 or the number 2:

- `LocationFlag = 1`: Places the compensator in the forward feedback configuration (this is the default)
- `LocationFlag = 2`: Places the compensator in the reverse feedback configuration

`FeedbackSign` can be either the number 1 or the number -1:

- `FeedbackSign = -1`: For negative feedback (this is the default)
- `FeedbackSign = 1`: For positive feedback

Example

See Chapter 6.

See Also

<code>rllocus</code>	Plot root locus
<code>rllocfind</code>	Select gain from the root locus plot

Purpose	Select feedback gain from root locus plot
Syntax	<pre>[k, poles] = rlocfind(sys) [k, poles] = rlocfind(sys, p)</pre>
Description	<p><code>rlocfind</code> returns the feedback gain associated with a particular set of poles on the root locus. <code>rlocfind</code> works with both continuous- and discrete-time SISO systems.</p> <p><code>[k, poles] = rlocfind(sys)</code> is used for interactive gain selection from the root locus plot of the SISO system <code>sys</code> generated by <code>rlocus</code>. The function <code>rlocfind</code> puts up a crosshair cursor on the root locus plot that you use to select a particular pole location. The root locus gain associated with this point is returned in <code>k</code> and the column vector <code>poles</code> contains the closed-loop poles for this gain. To use this command, the root locus of the SISO open-loop model <code>sys</code> must be present in the current figure window.</p> <p><code>[k, poles] = rlocfind(sys, p)</code> takes a vector <code>p</code> of desired root locations and computes a root locus gain for each of these locations (that is, a gain for which one of the closed-loop roots is near the desired location). The jth entry of the vector <code>k</code> gives the computed gain for the pole location <code>p(j)</code>, and the jth column of the matrix <code>poles</code> lists the resulting closed-loop poles.</p>
Example	<p>Determine a feedback gain such that the closed-loop poles of the system</p> $h(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$ <p>have damping ratio $\zeta = 0.707$:</p> <pre>h = tf([2 5 1], [1 2 3]); rlocus(h) % Plot the root locus k = rlocfind(h) % Select pole with ζ=.707 graphically</pre>
Algorithm	<p><code>rlocfind</code> uses the root locus magnitude rule to determine the gains for a particular root location. If a point off the root locus is chosen, <code>rlocfind</code> returns the gain computed from that point. If the point chosen is close to the root locus, the closed-loop poles returned will be near the desired point.</p>

rlocfind

Limitations	rlocfind assumes that a root locus is in the current figure window when this function is called without second input argument.	
See Also	rlocus	Plot root locus
References	[1] Ogata, K., <i>Modern Control Engineering</i> , Prentice Hall, 1970.	

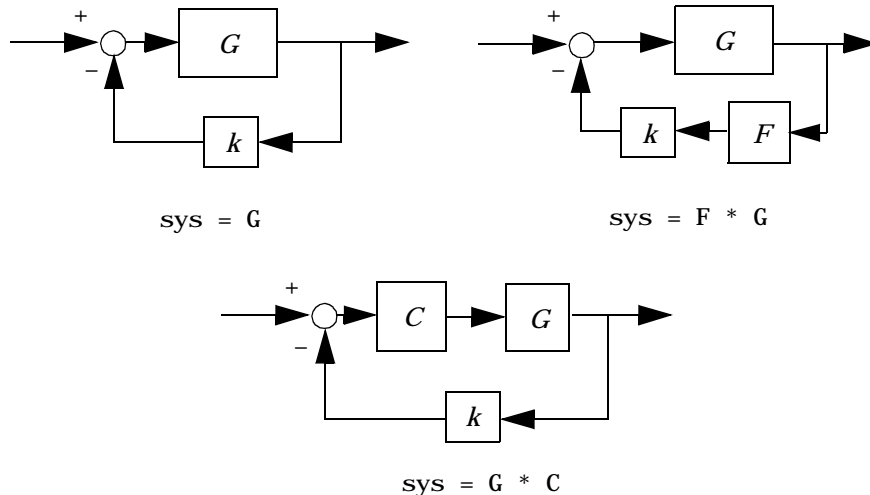
Purpose Evans root locus

Syntax `rlocus(sys)`
`rlocus(sys, k)`

`[r, k] = rlocus(sys)`
`r = rlocus(sys, k)`

Description `rlocus` computes the Evans root locus of a SISO open-loop model. The root locus gives the closed-loop pole trajectories as a function of the feedback gain k (assuming negative feedback). Root loci are used to study the effects of varying feedback gains on closed-loop pole locations. In turn, these locations provide indirect information on the time and frequency responses.

`rlocus(sys)` calculates and plots the root locus of the open-loop SISO model `sys`. This function can be applied to any of the following *negative* feedback loops by setting `sys` appropriately:



If `sys` has transfer function

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

the closed-loop poles are the roots of

$$d(s) + k n(s) = 0$$

`rlocus` adaptively selects a set of positive gains k to produce a smooth plot. Alternatively,

```
rlocus(sys, k)
```

uses the user-specified vector `k` of gains to plot the root locus.

When invoked with lefthand arguments,

```
[r, k] = rlocus(sys)  
r = rlocus(sys, k)
```

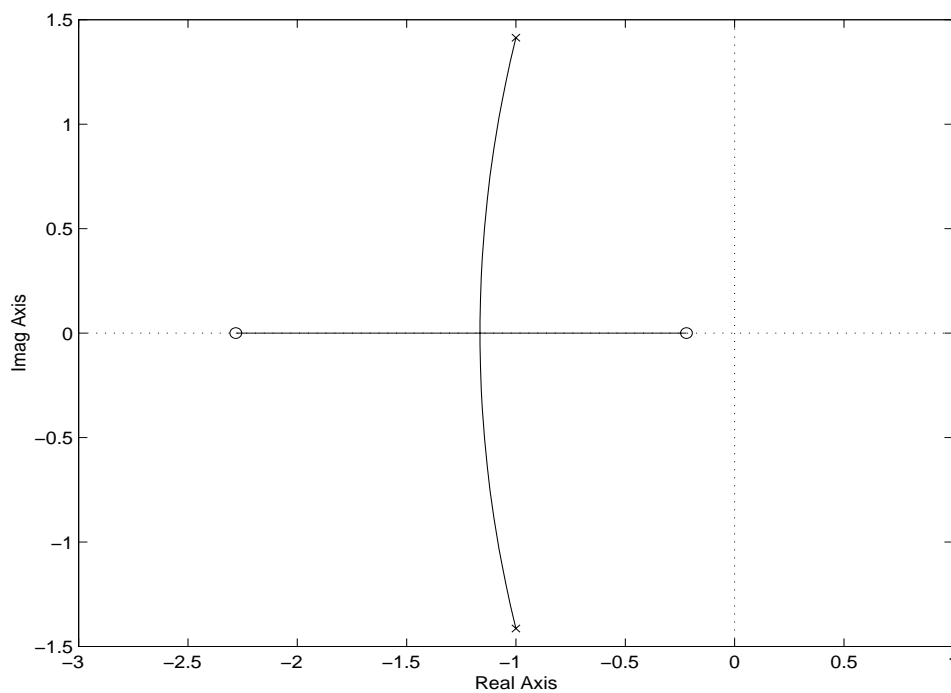
return the vector `k` of selected gains and the complex root locations `r` for these gains. The matrix `r` has `length(k)` columns and its j th column lists the closed-loop roots for the gain `k(j)`.

Example

Find and plot the root-locus of the system:

$$h(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
h = tf([2 5 1],[1 2 3]);
rlocus(h)
```



See also page 7-9 and page 7-19.

See Also

`rlocfind`
`pzmap`
`pole`

Select gain from root locus plot
 Pole-zero map
 System poles

rmodel, rss

Purpose Generate stable random continuous test models

Syntax

```
sys = rss(n)
sys = rss(n, p)
sys = rss(n, p, m)

[num, den] = rmodel(n)
[A, B, C, D] = rmodel(n)
[A, B, C, D] = rmodel(n, p, m)
```

Description `rss(n)` produces a stable random n th order model with one input and one output and returns the model in the state-space object `sys`.

`rss(n, p)` produces a random n th order stable model with one input and p outputs, and `rss(n, m, p)` produces a random n th order stable model with m inputs and p outputs. The output `sys` is always a state-space model.

Use `tf` or `zpk` to convert the state-space object `sys` to transfer function or zero-pole-gain form.

`rmodel(n)` produces a random n th order stable model and returns either the transfer function numerator `num` and denominator `den` or the state-space matrices `A`, `B`, `C`, and `D`, depending on the number of lefthand arguments. The resulting model always has one input and one output.

`[A, B, C, D] = rmodel(n, m, p)` produces a stable random n th order state-space model with m inputs and p outputs.

Example Obtain a stable random continuous LTI model with three states, two inputs, and two outputs by typing:

```
sys = rss(3, 2, 2)
```

MATLAB responds with:

```
a =
```

	x1	x2	x3
x1	-0.54175	0.09729	0.08304
x2	0.09729	-0.89491	0.58707
x3	0.08304	0.58707	-1.95271

```
b =
```

	u1	u2
x1	-0.88844	-2.41459
x2	0	-0.69435
x3	-0.07162	-1.39139

```
c =
```

	x1	x2	x3
y1	0.32965	0.14718	0
y2	0.59854	-0.10144	0.02805

```
d =
```

	u1	u2
y1	-0.87631	-0.32758
y2	0	0

Continuous-time system.

See Also

drmodel, drss	Generate stable random discrete test models
tf	Convert LTI systems to transfer functions form
zpk	Convert LTI systems to zero-pole-gain form

series

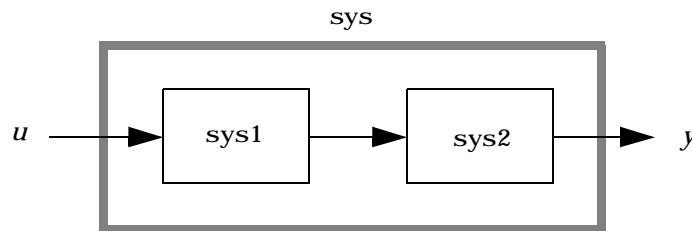
Purpose Series connection of two LTI models

Syntax

```
sys = series(sys1, sys2)
sys = series(sys1, sys2, outputs1, inputs2)
```

Description `series` connects two LTI models in series. This function accepts any type of LTI model. The two systems must be either both continuous or both discrete with identical sample time. Static gains are neutral and can be specified as regular matrices.

`sys = series(sys1, sys2)` forms the basic series connection shown below.

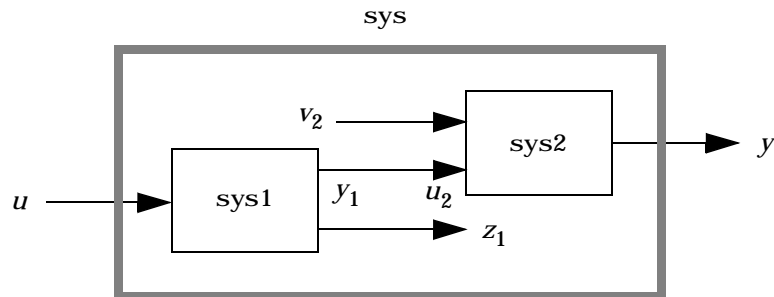


This command is equivalent to the direct multiplication

```
sys = sys2 * sys1
```

See page 2-30 for details on LTI system multiplication.

`sys = series(sys1, sys2, outputs1, inputs2)` forms the more general series connection:



The index vectors `outputs1` and `inputs2` indicate which outputs y_1 of `sys1` and which inputs u_2 of `sys2` should be connected. The resulting model `sys` has u as input and y as output.

Example

Consider a state-space system `sys1` with five inputs and four outputs and another system `sys2` with two inputs and three outputs. Connect the two systems in series by connecting outputs 2 and 4 of `sys1` with inputs 1 and 2 of `sys2`:

```
outputs1 = [2 4];  
inputs2 = [1 2];  
sys = series(sys1, sys2, outputs2, inputs1)
```

See Also

`append`
`parallel`
`feedback`

Append LTI systems
Parallel connection
Feedback connection

set

Purpose Set or modify LTI model properties

Syntax

```
set(sys, 'Property', Value)
set(sys, 'Property1', Value1, 'Property2', Value2, ...)
```



```
set(sys, 'Property')
set(sys)
```

Description `set` is used to set or modify the properties of an LTI model (see page 2-18 for background on LTI properties). Like its Handle Graphics counterpart, `set` uses property name/property value pairs to update property values.

`set(sys, 'Property', Value)` assigns the value `Value` to the property of the LTI model `sys` specified by the string `'Property'`. This string can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`). The specified property must be compatible with the model type. For example, if `sys` is a transfer function, `Variable` is a valid property but `StateName` is not (see page 2-19 for details). Type `set(sys)` for the list of assignable properties of a given LTI model `sys`.

`set(sys, 'Property1', Value1, 'Property2', Value2, ...)` sets multiple property values with a single statement. Each property name/property value pair updates one particular property.

`set(sys, 'Property')` displays admissible values for the property specified by `'Property'`. See “Property Values” below for an overview of legitimate LTI property values.

`set(sys)` displays all assignable properties of `sys` and their admissible values.

Example Consider the SISO state-space model created by

```
sys = ss(1, 2, 3, 4)
```

You can add an input delay of 0.1 second, label the input as “torque,” reset the *D* matrix to zero, and store its DC gain in the `'UserData'` property by

```
set(sys, 'td', 0.1, 'inputn', 'torque', 'd', 0, 'user', dcgain(sys))
```


Note that `set` does not require any output argument. Check the result with `get` by typing:

```
get(sys)
```

to which MATLAB returns:

```
a = 1
b = 2
c = 3
d = 0
e = []
StateName = {' '}
Ts = 0
Td = 0.1
InputName = {'torque'}
OutputName = {' '}
Notes = {}
UserData = -2
```

Property Values

The following table lists the admissible values for each LTI property. N_u and N_y denote the number of inputs and outputs of the underlying LTI model, and N_x denotes the number of states, when applicable.

Property Name	Admissible Property Values
Ts	<ul style="list-style-type: none">• 0 (zero) for continuous-time systems• Sample time in seconds for discrete-time systems• -1 or [] for discrete systems with unspecified sample time <p>Note: Resetting the sample time property does not alter the model data. Use c2d, d2c, or d2d for discrete/continuous and discrete/discrete conversions.</p>
Td	Continuous-time input delays (in seconds) <ul style="list-style-type: none">• Nonnegative scalar when Nu=1 or system has uniform input delay• Vector of length Nu to specify independent delay times for each input channel
Notes	String, array of strings, or cell array of strings
UserData	Arbitrary MATLAB variable
InputName	<ul style="list-style-type: none">• String for single-input systems, for example, 'thrust'• Cell vector of strings for multi-input systems (with as many cells as inputs), for example, {'u' ; 'w'} for a two-input system• Padded array of strings with as many rows as inputs, for example, ['rudder' ; 'aileron']
OutputName	Same as InputName (with Input replaced by Output)

Property Name	Admissible Property Values
StateName (SS only)	Same as InputName (with Input replaced by State)
Variabl e (TF and ZPK only)	Transfer function variable: <ul style="list-style-type: none"> • String ' s ' (default) or ' p ' for continuous-time systems • String ' z ' (default), ' q ' , or ' z⁻¹ ' for discrete-time systems
num, den (TF only)	<ul style="list-style-type: none"> • Row vectors of numerator or denominator coefficients in SISO case. List the coefficients in <i>descending</i> powers of <i>s</i> or <i>z</i> by default, and in <i>ascending</i> powers of $q = z^{-1}$ when the Variabl e property is set to ' q ' or ' z⁻¹ ' (see note below). • Ny-by-Nu cell arrays of row vectors in MIMO case, for example, $\{[1 \ 2] \ ; \ [1 \ 0 \ 3]\}$ for a two-output/one-input transfer function
z, p (ZPK only)	<ul style="list-style-type: none"> • Vectors of zeros and poles in SISO case • Ny-by-Nu cell arrays of vectors in MIMO case, for example, $z = \{[] \ , \ [-1 \ 0]\}$ for a model with two inputs and one output
k (ZPK only)	Matrix with as many rows as outputs and as many columns as inputs
a, b, c, d, e (SS only)	State-space matrices with dimensions compatible with the number of states, inputs, and outputs

Note: For discrete-time transfer functions, the convention used to represent the numerator and denominator depends on the choice of variable (see tf entry for details). Like tf, set switches conventions to remain consistent with the choice of variable. For example, if the Variabl e property is set to ' z ' (the default),

set(h, ' num' , [1 2] , ' den' , [1 3 4])

produces the transfer function

$$h(z) = \frac{z + 2}{z^2 + 3z + 4}$$

However, if you change the Variable to 'z⁻¹' (or 'q') by

```
set(h, 'Variable', 'z-1'),
```

the same command

```
set(h, 'num', [1 2], 'den', [1 3 4])
```

now interprets the row vectors [1 2] and [1 3 4] as the polynomials $1 + 2z^{-1}$ and $1 + 3z^{-1} + 4z^{-2}$ and produces:

$$\bar{h}(z^{-1}) = \frac{1 + 2z^{-1}}{1 + 3z^{-1} + 4z^{-2}} = zh(z)$$

Because the resulting transfer functions are different, make sure to use the convention consistent with your choice of variable.

See Also

get	Access/query LTI model properties
ss	Specify state-space model
tf	Specify transfer function
zpk	Specify zero-pole-gain model

Purpose Generate an s -plane grid of constant damping factors and natural frequencies

Syntax `sgrid`
`sgrid(z, wn)`

Description `sgrid` generates a grid of constant damping factors from zero to one in steps of 0.1 and natural frequencies from zero to 10 rad/sec in steps of one rad/sec, and plots the grid over the current axis. If the current axis contains a continuous s -plane root locus diagram or pole-zero map, `sgrid` draws the grid over the plot.

`sgrid(z, wn)` plots a grid of constant damping factor and natural frequency lines for the damping factors and natural frequencies in the vectors `z` and `wn`, respectively. If the current axis contains a continuous s -plane root locus diagram or pole-zero map, `sgrid(z, wn)` draws the grid over the plot.

Example Plot s -plane grid lines on the root locus for the system:

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

Type:

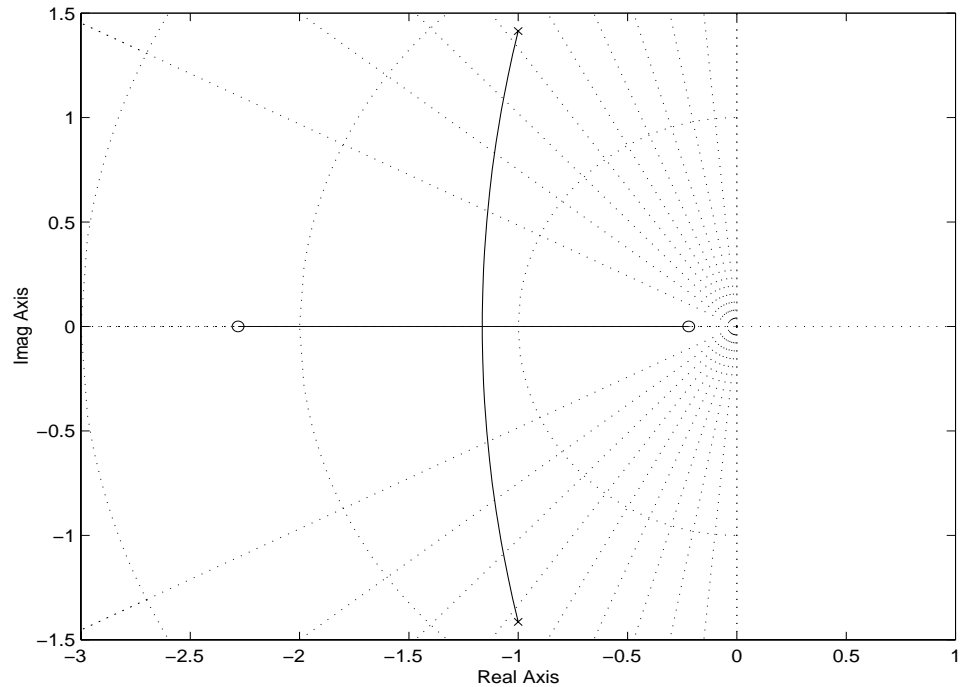
```
H = tf([2 5 1],[1 2 3])
```

MATLAB returns:

```
Transfer function:
 2 s^2 + 5 s + 1
-----
 s^2 + 2 s + 3
```

Type:

```
rlocus(H)
sgrid
```



Limitations

`sgrid` plots the grid over the current axis regardless of whether the axis contains a root locus diagram or pole-zero map. Therefore, if the current axes contains, for example, a step response, you may superimpose a meaningless s -plane grid over the plot. In addition, `sgrid` does not rescale the axis limits of the current axis. Therefore, the s -plane grid may not appear in the plot if the left half of the s -plane is not encompassed by the axis limits.

See Also

<code>pzmap</code>	Plot pole-zero map
<code>rl locus</code>	Plot root locus
<code>zgri d</code>	Generate z -plane grid lines

Purpose Singular value response of LTI systems

Syntax

```
si gma(sys)
si gma(sys, w)
si gma(sys, w, type)

si gma(sys1, sys2, . . . , sysN)
si gma(sys1, sys2, . . . , sysN, w)
si gma(sys1, sys2, . . . , sysN, w, type)
si gma(sys1, 'Pl otStyl e1', . . . , sysN, 'Pl otStyl eN' )

[sv, w] = si gma(sys)
```

Description `si gma` calculates the singular value response of an LTI system. For continuous-time systems with transfer function $H(s)$, `si gma` computes the singular values of $H(j\omega)$ as a function of the frequency ω . For discrete-time systems with transfer function $H(z)$ and sample time T_s , `si gma` computes the singular values of

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

for frequencies ω between 0 and the Nyquist frequency $\omega_N = \pi/T_s$.

The singular value response is an extension of the Bode magnitude response for MIMO systems and is useful in robustness analysis. The singular value response of a SISO system is identical to its Bode magnitude response. When invoked without lefthand arguments, `si gma` produces a singular value plot on the screen.

`si gma(sys)` plots the singular value response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. The frequency points are chosen automatically based on the system poles and zeros.

`si gma(sys, w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval $[w_{\min}, w_{\max}]$, set $w = \{w_{\min}, w_{\max}\}$. To use particular frequency points, set w to the corresponding vector of frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. Frequencies should be specified in rad/sec.

`sigma(sys, [], type)` or `sigma(sys, w, type)` plots the following modified singular value responses:

- `type = 1` Singular value response of H^{-1} where H is the transfer function of `sys`.
- `type = 2` Singular value response of $I + H$.
- `type = 3` Singular value response of $I + H^{-1}$.

These options are available only for square systems, that is, with the same number of inputs and outputs.

To superimpose the singular value plots of several LTI models on a single figure, use:

```
sigma(sys1, sys2, ..., sysN)
sigma(sys1, sys2, ..., sysN, [], type) % modified SV plot
sigma(sys1, sys2, ..., sysN, w)       % specify frequency range/grid
```

All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous and discrete systems. You can also specify a distinctive color, linestyle, and/or marker for each system plot with the syntax

```
sigma(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')
```

See bode for an example.

When invoked with lefthand arguments,

```
[sv, w] = sigma(sys)
sv = sigma(sys, w)
```

return the singular values `sv` of the frequency response at the frequencies `w`. For a system with `Nu` input and `Ny` outputs, the array `sv` has `min(Nu, Ny)` rows and as many columns as frequency points (length of `w`). The singular values at the frequency `w(k)` are given by `sv(:, k)`.

Example

Plot the singular value responses of

$$H(s) = \begin{bmatrix} 0 & \frac{3s}{s^2 + s + 10} \\ \frac{s+1}{s+5} & \frac{2}{s+6} \end{bmatrix}$$

and $I + H(s)$:

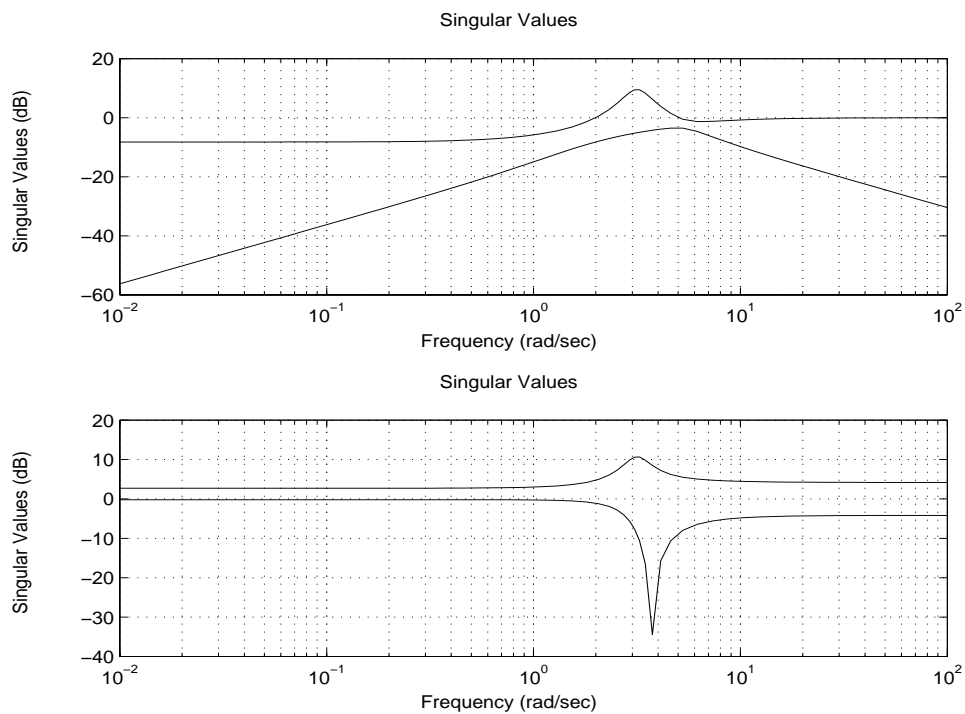
$$H = [0 \text{ tf}([3 \ 0], [1 \ 1 \ 10]) ; \text{tf}([1 \ 1], [1 \ 5]) \text{ tf}(2, [1 \ 6])]$$

subplot(211)

sigma(H)

subplot(212)

sigma(H, [], 2)



sigma

Algorithm sigma uses the svd function in MATLAB to compute the singular values of a complex matrix.

See Also lti view LTI system viewer
 bode Bode plot
 nichols Nichols plot
 nyquist Nyquist plot
 freqresp Frequency response computation
 evalfr Response at single complex frequency

Purpose Output/input/state dimensions of LTI models

Syntax

```
d = size(sys)
[p, m] = size(sys)
[p, m, n] = size(sys)    % state-space models only

p = size(sys, 1)
m = size(sys, 2)
n = size(sys, 3)         % state-space models only

size(sys)
```

Description `size` returns the number of outputs and inputs of a given LTI model, as well as the number of states for state-space models. `size` is the overloaded version of the MATLAB function `size` for LTI objects.

`[p, m] = size(sys)` returns the number `p` of outputs and the number `m` of inputs of the LTI model `sys`.

`[p, m, n] = size(sys)` also returns the number `n` of states (`sys` must then be a state-space model).

`d = size(sys)` returns the two-entry row vector `d = [p m]`.

To query just one of the three dimensions, use the syntax

```
p = size(sys, 1)
m = size(sys, 2)
n = size(sys, 3)
```

Finally, when invoked without output arguments,

```
size(sys)
```

displays the number of inputs, outputs, and states (when applicable) on the screen.

Example Consider the random state-space model

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

size

Its dimensions are obtained by typing `size(sys)`:

State-space model with 2 input(s), 3 output(s), and 5 state(s).

See Also

- `isempty` Test if LTI model is empty
- `issiso` Test if LTI model is SISO

Purpose	Specify state-space models or convert LTI model to state space
Syntax	<pre> sys = ss(a, b, c, d) sys = ss(a, b, c, d, Ts) sys = ss(d) sys = ss(a, b, c, d, lti sys) sys = ss(a, b, c, d, 'Property1', Value1, ..., 'PropertyN', ValueN) sys = ss(a, b, c, d, Ts, 'Property1', Value1, ..., 'PropertyN', ValueN) sys_ss = ss(sys) </pre>

Description `ss` is used to create state-space model (SS objects) or to convert transfer function or zero-pole-gain models to state space.

Creation of State-Space Models

`sys = ss(a, b, c, d)` creates the continuous-time state-space model

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

The output `sys` is an SS object storing the model data (see page 2-2). If $D = 0$, you need not dimension `d` and can simply set `d` to the scalar 0 (zero).

`sys = ss(a, b, c, d, Ts)` creates the discrete-time model

$$x[n+1] = Ax[n] + Bu[n]$$

$$y[n] = Cx[n] + Du[n]$$

with sample time `Ts` (in seconds). Set `Ts = -1` or `Ts = []` to leave the sample time unspecified.

`sys = ss(d)` specifies a static gain matrix D and is equivalent to:

$$\text{sys} = \text{ss}([], [], [], d)$$

`sys = ss(a, b, c, d, lti sys)` creates a state-space model with generic LTI properties inherited from the LTI model `lti sys` (including the sample time). See page 2-18 for an overview of generic LTI properties.

Any of the previous syntaxes can be followed by property name/property value pairs

`' Property' , Val ue`

Each pair specifies a particular LTI property of the model, for example, the input names or some notes on the model history. See the `set` entry and the example below for details. Note that

`sys = ss(a, b, c, d, ' Property1' , Val ue1, . . . , ' PropertyN' , Val ueN)`

is equivalent to the sequence of commands:

```
sys = ss(a, b, c, d)
set(sys, ' Property1' , Val ue1, . . . , ' PropertyN' , Val ueN)
```

Conversion to State Space

`sys_ss = ss(sys)` converts an arbitrary LTI model `sys` to state space. The output `sys_ss` is an equivalent state-space model (SS object). This operation is known as *state-space realization*.

Examples

Example 1

The command

```
sys = ss(A, B, C, D, 0.05, ' statename' , {' posi ti on' ' vel oci ty' }, . . .
      ' inputname' , ' force' , . . .
      ' notes' , ' Created 10/15/96' )
```

creates a discrete-time model with matrices A, B, C, D and sample time 0.05 second. This model has two states labeled `posi ti on` and `vel oci ty`, and one input labeled `force` (the dimensions of A, B, C, D should be consistent with these numbers of states and inputs). Finally, a note is attached with the date of creation of the model.

Example 2

Compute a state-space realization of the transfer function

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

by typing:

```
H = [tf([1 1], [1 3 3 2]) ; tf([1 0 3], [1 1 1])]
ss(H)
```

MATLAB returns:

a =

	x1	x2	x3
x1	-2.00000	0	0.50000
x2	0	-1.00000	-0.50000
x3	0	2.00000	0

b =

	u1
x1	0
x2	2.00000
x3	0

c =

	x1	x2	x3
y1	-0.50000	0	0.25000
y2	0	-0.50000	0.50000

d =

	u1
y1	0
y2	1.00000

Continuous-time system.

Note that this realization has minimal order three.

Limitations	ss is guaranteed to produce minimal state-space realizations only for single-input or single-output transfer functions.	
See Also	ssdata	Retrieve A, B, C, D matrices of state-space model
	dss	Specify descriptor state-space models
	set	Set properties of LTI models
	get	Get properties of LTI models
	tf	Specify transfer functions or convert to transfer function form
	zpk	Specify zero-pole-gain models or convert to ZPK

Purpose State coordinate transformation for state-space models

Syntax `sysT = ss2ss(sys, T)`

Description Given a state-space model `sys` with equations

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

(or their discrete-time counterpart), `ss2ss` performs the similarity transformation $\bar{x} = Tx$ on the state vector x and produces the equivalent state-space model `sysT` with equations:

$$\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu$$

$$y = CT^{-1}\bar{x} + Du$$

`sysT = ss2ss(sys, T)` returns the transformed state-space model `sysT` given `sys` and the state coordinate transformation `T`. The model `sys` must be in state-space form and the matrix `T` must be invertible. `ss2ss` is applicable to both continuous- and discrete-time models.

Example Perform a similarity transform to improve the conditioning of the A matrix:

```
T = balance(sys, a)
sysb = ss2ss(sys, inv(T))
```

See `ssbal` for a more direct approach.

See Also

<code>canon</code>	Canonical state-space realizations
<code>ssbal</code>	Balancing of state-space model using diagonal similarities
<code>bal real</code>	Gramian-based I/O balancing

ssbal

Purpose Balance state-space models using diagonal similarity

Syntax `[sysb, T] = ssbal (sys)`
`[sysb, T] = ssbal (sys, condT)`

Description Given a state-space model `sys` with matrices (A, B, C, D) ,
`[sysb, T] = ssbal (sys)`
computes a diagonal similarity transformation T and a scalar α such that

$$\begin{bmatrix} TAT^{-1} & TB/\alpha \\ \alpha CT^{-1} & 0 \end{bmatrix}$$

has approximately equal row and column norms. `ssbal` returns the balanced model `sysb` with matrices

$$(TAT^{-1}, TB/\alpha, \alpha CT^{-1}, D)$$

and the state transformation $\bar{x} = TX$ where \bar{x} is the new state.

`[sysb, T] = ssbal (sys, condT)` specifies an upper bound `condT` on the condition number of T . Since balancing with ill-conditioned T can inadvertently magnify rounding errors, `condT` gives control over the worst-case roundoff amplification factor. The default value is `condT=1/eps`.

Example Consider the continuous-time state-space model with data

$$A = \begin{bmatrix} 1 & 10^4 & 10^2 \\ 0 & 10^2 & 10^5 \\ 10 & 1 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad C = [0.1 \ 10 \ 100]$$

```
a = [1 1e4 1e2; 0 1e2 1e5; 10 1 0];  
b = [1; 1; 1];  
c = [0.1 10 1e2];  
sys = ss(a, b, c, 0)
```

Balance this model with `ssbal` by typing `ssbal (sys)`. MATLAB returns:

```
a =
```

	x1	x2	x3
x1	1.00000	2500.00000	0.39062
x2	0	100.00000	1562.50000
x3	2560.00000	64.00000	0

```
b =
```

	u1
x1	0.12500
x2	0.50000
x3	32.00000

```
c =
```

	x1	x2	x3
y1	0.80000	20.00000	3.12500

```
d =
```

	u1
y1	0

Continuous-time system.

Direct inspection shows that the range of numerical values has been compressed by a factor 100 and that the B and C matrices now have nearly equal norms.

Algorithm `ssbal` uses the MATLAB function `balance` to compute T and α .

See Also `balreal` Gramian-based I/O balancing
`ss2ss` State coordinate transformation

ssdata

Purpose Quick access to state-space model data

Syntax `[a, b, c, d] = ssdata(sys)`
`[a, b, c, d, Ts, Td] = ssdata(sys)`

Description `[a, b, c, d] = ssdata(sys)` extracts the matrix data (*A, B, C, D*) from the state-space model `sys`. If `sys` is a transfer function or zero-pole-gain model, it is first converted to state space.

`[a, b, c, d, Ts, Td] = ssdata(sys)` also returns the sample time `Ts` and the input delay data `Td`. For continuous-time models, `Td` is a row vector with one entry per input channel (`Td(j)` indicates by how many seconds the *j* th input is delayed). For discrete-time models, `Td` is the empty matrix `[]` (see `d2d` for delays in discrete systems).

You can access the remaining LTI properties of `sys` with `get` or by direct referencing, for example,

`sys.statename`

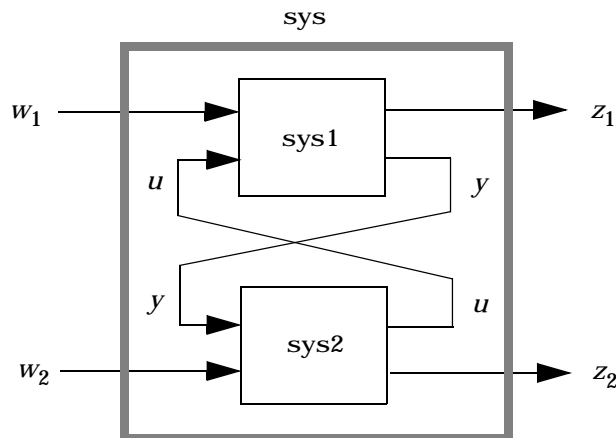
See Also	<code>ss</code>	Specify state-space models
	<code>get</code>	Get properties of LTI models
	<code>dssdata</code>	Quick access to descriptor state-space data
	<code>tfdata</code>	Quick access to transfer function data
	<code>zpkdata</code>	Quick access to zero-pole-gain data

Purpose Redheffer star product of two LTI models

Syntax
`sys = star(sys1, sys2)`
`sys = star(sys1, sys2, nu, ny)`

Description `star` forms the star product or linear fractional transformation (LFT) of two LTI models. Such interconnections are widely used in robust control techniques.

`sys = star(sys1, sys2, nu, ny)` forms the star product `sys` of the two LTI models `sys1` and `sys2`. The star product amounts to the following feedback connection:



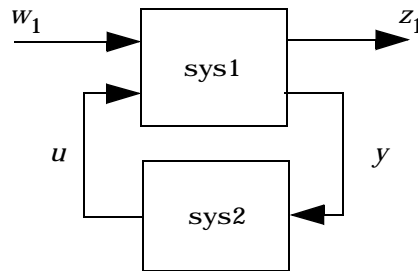
This feedback loop connects the first `nu` outputs of `sys2` to the last `nu` inputs of `sys1` (signals `u`), and the last `ny` outputs of `sys1` to the first `ny` inputs of `sys2` (signals `y`). The resulting system `sys` maps the input vector $[w_1; w_2]$ to the output vector $[z_1; z_2]$.

The abbreviated syntax

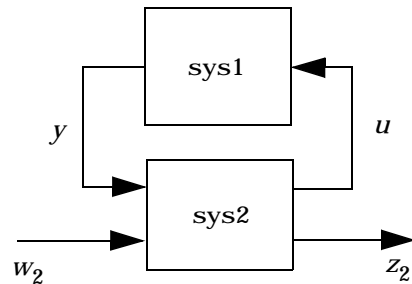
```
sys = star(sys1, sys2)
```

produces:

- The lower LFT of sys1 and sys2 if sys2 has fewer inputs and outputs than sys1. This amounts to deleting w_2 and z_2 in the above diagram.
- The upper LFT of sys1 and sys2 if sys1 has fewer inputs and outputs than sys2. This amounts to deleting w_1 and z_1 in the above diagram.



Lower LFT connection



Upper LFT connection

Algorithm

The closed-loop model is derived by elementary state-space manipulations.

Limitations

There should be no algebraic loop in the feedback connection.

See Also

feedback
connect

Feedback connection
Derive state-space model for block diagram
interconnection

Purpose Step response of LTI systems

Syntax

```
step(sys)
step(sys, t)
```

```
step(sys1, sys2, ..., sysN)
step(sys1, sys2, ..., sysN, t)
step(sys1, 'PlotStyle1', ..., sysN, 'PlotStyleN')
```

```
[y, t, x] = step(sys)
```

Description

`step` calculates the unit step response of a linear system. Zero initial state is assumed in the state-space case. When invoked without lefthand arguments, this function plots the step response on the screen.

`step(sys)` plots the step response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. The step response of multi-input systems is the collection of step responses for each input channel. The duration of simulation is determined automatically based on the system poles and zeros.

`step(sys, t)` sets the simulation horizon explicitly. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form

$$t = 0: dt: T_{final}$$

For discrete systems, the spacing `dt` should match the sample period. For continuous systems, `dt` becomes the sample time of the discretized simulation model (see “Algorithm”), so make sure to choose `dt` small enough to capture transient phenomena.

To plot the step responses of several LTI models `sys1,..., sysN` on a single figure, use

```
step(sys1, sys2, ..., sysN)
step(sys1, sys2, ..., sysN, t)
```

All systems must have the same number of inputs and outputs but may otherwise be a mix of continuous- and discrete-time systems. This syntax is useful to compare the step responses of multiple systems.

You can also specify a distinctive color, linestyle, and/or marker for each system. For example,

```
step(sys1, 'y:', sys2, 'g- -')
```

plots the step response of `sys1` with a dotted yellow line and the step response of `sys2` with a green dashed line.

When invoked with lefthand arguments,

```
[y, t] = step(sys)
[y, t, x] = step(sys)      % for state-space models only
y = step(sys, t)
```

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x` (for state-space models only). No plot is drawn on the screen. For single-input systems, `y` has as many rows as time samples (length of `t`), and as many columns as outputs. In the multi-input case, the step responses of each input channel are stacked up along the third dimension of `y`. The dimensions of `y` are then

$(\text{length of } t) \times (\text{number of outputs}) \times (\text{number of inputs})$

and `y(:, :, j)` gives the response to a unit step command injected in the j th input channel. Similarly, the dimensions of `x` are

$(\text{length of } t) \times (\text{number of states}) \times (\text{number of inputs})$

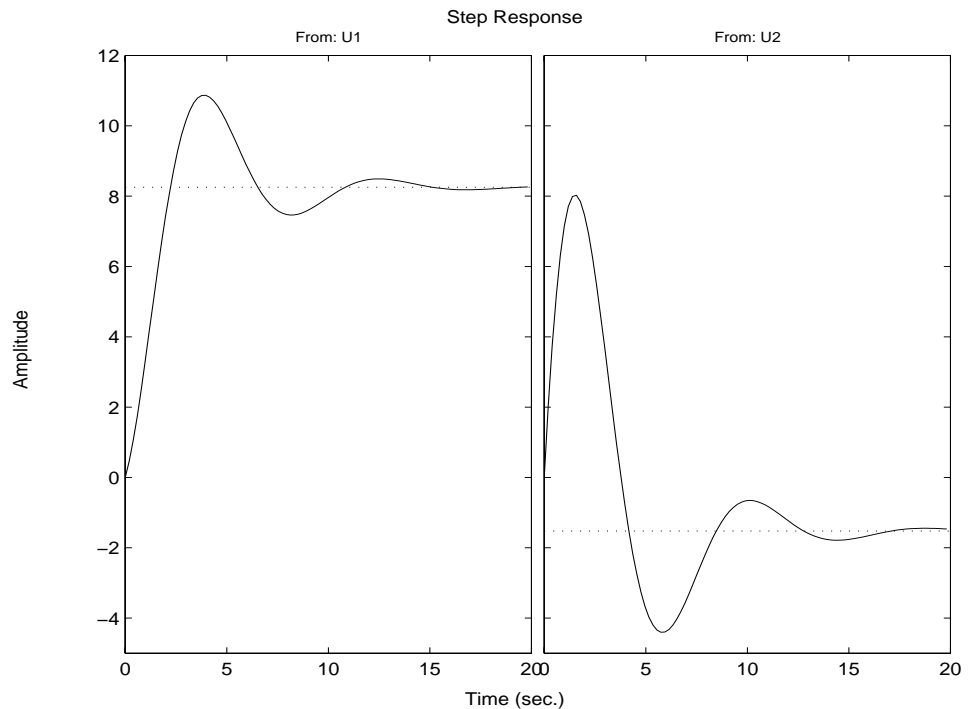
Example

Plot the step response of the second-order state-space model:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

```
a = [-0.5572  -0.7814; 0.7814  0];
b = [1  -1; 0  2];
c = [1.9691  6.4493];
sys = ss(a, b, c, 0);
step(sys)
```



The left plot shows the step response of the first input channel, and the right plot shows the step response of the second input channel.

Algorithm Continuous-time models are converted to state space and discretized using zero-order hold on the inputs. The sampling period is chosen automatically based on the system dynamics, except when a time vector $t = 0:dt:T_f$ is supplied (dt is then used as sampling period).

See Also `lti view` LTI system viewer
 `impul se` Impulse response
 `ini ti al` Free response to initial condition
 `l si m` Simulate response to arbitrary inputs

Purpose Specify transfer functions or convert LTI model to transfer function form

Syntax

```

sys = tf(num, den)
sys = tf(num, den, Ts)
sys = tf(M)
sys = tf(num, den, lti sys)

sys = tf(num, den, 'Property1', Value1, ..., 'PropertyN', ValueN)
sys = tf(num, den, Ts, 'Property1', Value1, ..., 'PropertyN', ValueN)

tf sys = tf(sys)
tf sys = tf(sys, 'inv')    % for state-space sys only

```

Description `tf` is used to create transfer function models (TF objects) or to convert state-space or zero-pole-gain models to transfer function form.

Creation of Transfer Functions

`sys = tf(num, den)` creates a continuous-time transfer function with numerator(s) and denominator(s) specified by `num` and `den`. The output `sys` is a TF object storing the transfer function data (see page 2-2).

In the SISO case, `num` and `den` are the row vectors of numerator and denominator coefficients ordered in *descending* powers of s . These two vectors need not have equal length and the transfer function need not be proper. For example, `h = tf([1 0], 1)` specifies the pure derivative $h(s) = s$.

To create MIMO transfer functions, specify the numerator and denominator of each SISO entry. In this case,

- `num` and `den` are cell arrays of row vectors with as many rows as outputs and as many columns as inputs.
- The row vectors `num{i, j}` and `den{i, j}` specify the numerator and denominator of the transfer function from input j to output i (with the SISO convention).

If all SISO entries of a MIMO transfer function have the same denominator, you can set `den` to the row vector representation of this common denominator. See “Examples” for more details.

`sys = tf(num, den, Ts)` creates a discrete-time transfer function with sample time `Ts` (in seconds). Set `Ts = -1` or `Ts = []` to leave the sample time unspecified. The input arguments `num` and `den` are as in the continuous-time case and must list the numerator and denominator coefficients in *descending* powers of z .

`sys = tf(M)` creates a static gain `M` (scalar or matrix).

`sys = tf(num, den, lti sys)` creates a transfer function with generic LTI properties inherited from the LTI model `lti sys` (including the sample time). See page 2-18 for an overview of generic LTI properties.

Any of the previous syntaxes can be followed by property name/property value pairs

`'Property', Value`

Each pair specifies a particular LTI property of the model, for example, the input names or the transfer function variable. See `set` entry and the example below for details. Note that

`sys = tf(num, den, 'Property1', Value1, ..., 'PropertyN', ValueN)`

is a shortcut for:

```
sys = tf(num, den)
set(sys, 'Property1', Value1, ..., 'PropertyN', ValueN)
```

Conversion to Transfer Function

`tfsys = tf(sys)` converts an arbitrary LTI model `sys` to transfer function form. The output `tfsys` (TF object) is the transfer function of `sys`. By default, `tf` uses `tzero` to compute the numerators when converting a state-space model to transfer function form. Alternatively,

`tfsys = tf(sys, 'inv')`

uses inversion formulas for state-space models to derive the numerators. This algorithm is faster but less accurate for high-order models with low gain at $s = 0$.

Examples

Example 1

Create the two-output/one-input transfer function

$$H(p) = \begin{bmatrix} \frac{p+1}{p^2+2p+2} \\ \frac{1}{p} \end{bmatrix}$$

with input current and outputs torque and ang velocity. Type:

```
num = {[1 1] ; 1}
den = {[1 2 2] ; [1 0]}
H = tf(num, den, 'inputn', 'current', ...
        'outputn', {'torque' 'ang. velocity'}, ...
        'variable', 'p')
```

MATLAB returns:

Transfer function from input "current" to output...

```
      p + 1
torque:  -----
      p^2 + 2 p + 2
```

```
      1
ang. velocity:  -
                p
```

Note how setting the 'variable' property to 'p' causes the result to be displayed as a transfer function of the variable p .

Example 2

Specify the discrete MIMO transfer function

$$H(z) = \begin{bmatrix} \frac{1}{z+0.3} & \frac{z}{z+0.3} \\ \frac{-z+2}{z+0.3} & \frac{3}{z+0.3} \end{bmatrix}$$

with common denominator $d(z) = z + 0.3$ and sample time 0.2 second:

```
nums = {1 [1 0]; [-1 2] 3}
Ts = 0.2
H = tf(nums, [1 0.3], Ts)    % Note: row vector for common den. d(z)
```

Example 3

Compute the transfer function of the state-space model with data

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

Type:

```
sys = ss([-2 -1; 1 -2], [1 1; 2 -1], [1 0], [0 1])
tf(sys)
```

MATLAB returns:

Transfer function from input 1 to output:

$$\frac{s^2 + 4s + 5}{s^2 + 4s + 5}$$

Transfer function from input 2 to output:

$$\frac{s^2 + 5s + 8}{s^2 + 4s + 5}$$

Discrete-Time Conventions

The control and digital signal processing (DSP) communities tend to use different conventions to specify discrete transfer functions. Most control engineers use the z variable and order the numerator and denominator terms in descending powers of z , for example,

$$h(z) = \frac{z^2}{z^2 + 2z + 3}$$

The polynomials z^2 and $z^2 + 2z + 3$ are then specified by the row vectors $[1 \ 0 \ 0]$ and $[1 \ 2 \ 3]$, respectively. By contrast, DSP engineers prefer to write this transfer function as

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

and specify its numerator as 1 (instead of $[1 \ 0 \ 0]$) and its denominator as $[1 \ 2 \ 3]$.

These two conventions clash when the numerator and denominator have different lengths. To be consistent with both sets of expectations, `tf` switches convention based on your choice of variable (value of the 'Variable' property):

Variable	Convention
'z' (default)	Use the row vector $[a_k \ \dots \ a_1 \ a_0]$ to specify the polynomial $a_k z^k + \dots + a_1 z + a_0$ (coefficients ordered in <i>descending</i> powers of z).
'z ⁻¹ ', 'q'	Use the row vector $[b_0 \ b_1 \ \dots \ b_k]$ to specify the polynomial $b_0 + b_1 z^{-1} + \dots + b_k z^{-k}$ (coefficients in <i>ascending</i> powers of z^{-1} or q).

For example,

$$g = \text{tf}([1 \ 1], [1 \ 2 \ 3], 0.1)$$

specifies the discrete transfer function

$$g(z) = \frac{z + 1}{z^2 + 2z + 3}$$

because z is the default variable. In contrast,

$$h = \text{tf}([1 \ 1], [1 \ 2 \ 3], 0.1, 'variable', 'z^{-1}')$$

uses the DSP convention and creates

$$h(z^{-1}) = \frac{1 + z^{-1}}{1 + 2z^{-1} + 3z^{-2}} = zg(z)$$

See also `filt` for direct specification of discrete transfer functions using the DSP convention.

Note that `tf` stores a variable-independent representation where the numerator and denominator lengths are made equal. Specifically, `tf` stores the values

```
num = [0 1 1]; den = [1 2 3]
```

for `g` (the numerator is padded with zeros on the left) and the values

```
num = [1 1 0]; den = [1 2 3]
```

for `h` (the numerator is padded with zeros on the right).

Algorithm

`tf` uses the MATLAB function `poly` to convert zero-pole-gain models, and the functions `tzero` and `pole` to convert state-space models.

See Also

<code>filt</code>	Specify discrete transfer functions in DSP format
<code>tfdata</code>	Retrieve transfer function data
<code>set</code>	Set properties of LTI models
<code>get</code>	Get properties of LTI models
<code>zpk</code>	Specify zero-pole-gain models or convert to ZPK
<code>ss</code>	Specify state-space models or convert to state space

Purpose	Quick access to transfer function data
Syntax	<pre>[num, den] = tfdata(sys) [num, den] = tfdata(sys, 'v') [num, den, Ts, Td] = tfdata(sys)</pre>
Description	<p><code>[num, den] = tfdata(sys)</code> returns the numerator(s) and denominator(s) of the transfer function <code>sys</code>. The outputs <code>num</code> and <code>den</code> are cell arrays with the following characteristics:</p> <ul style="list-style-type: none"> • <code>num</code> and <code>den</code> have as many rows as outputs and as many columns as inputs. • The (i, j) entries <code>num{i, j}</code> and <code>den{i, j}</code> are row vectors specifying the numerator and denominator coefficients of the transfer function from input j to output i. These coefficients are ordered in <i>descending</i> powers of s or z. <p>If <code>sys</code> is a state-space or zero-pole-gain model, it is first converted to transfer function form using <code>tf</code>.</p> <p>For SISO transfer functions, the convenience syntax</p> <pre>[num, den] = tfdata(sys, 'v')</pre> <p>forces <code>tfdata</code> to return the numerator and denominator directly as row vectors rather than as cell arrays (see example below).</p> <p><code>[num, den, Ts, Td] = tfdata(sys)</code> also returns the sample time <code>Ts</code> and the input delay data <code>Td</code>. For continuous-time transfer functions, <code>Td</code> is a row vector with one entry per input channel (<code>Td(j)</code> indicates by how many seconds the jth input is delayed). For discrete-time transfer functions, <code>Td</code> is the empty matrix <code>[]</code> (see <code>d2d</code> for delays in discrete systems).</p> <p>You can access the remaining LTI properties of <code>sys</code> with <code>get</code> or by direct referencing, for example,</p> <pre>sys.Ts sys.variable</pre>
Example	<p>Given the SISO transfer function</p> <pre>h = tf([1 1], [1 2 5])</pre>

you can extract the numerator and denominator coefficients by typing:

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

MATLAB returns:

```
num =  
    0    1    1
```

```
den =  
    1    2    5
```

This syntax returns two row vectors.

If you turn `h` into a MIMO transfer function by

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

the command

```
[num, den] = tfdata(H)
```

now returns two cell arrays with the numerator/denominator data for each SISO entry. Use `cell disp` to visualize this data. Type:

```
cell disp(num)
```

and MATLAB returns the numerators' vectors of the entries of `H`:

```
num{1} =  
    0    1    1
```

```
num{2} =  
    0    1
```

Type:

```
cell disp(den)
```

and MATLAB returns:

```
den{1} =  
    1    2    5
```

```
den{2} =  
    1    1
```

See Also

tf	Specify transfer functions
get	Get properties of LTI models
ssdata	Quick access to state-space data
zpkdata	Quick access to zero-pole-gain data

tzero

Purpose Transmission zeros of LTI models

Syntax

```
z = tzero(sys)
[z, gain] = tzero(sys)
z = tzero(a, b, c, d)
```

Description `tzero` computes the zeros of SISO systems and the transmission zeros of MIMO systems. For a MIMO system with matrices (A, B, C, D) , the transmission zeros are the complex values λ for which the normal rank of

$$\begin{bmatrix} A - \lambda I & B \\ C & D \end{bmatrix}$$

drops.

`z = tzero(sys)` returns the (transmission) zeros of the LTI model `sys` as a column vector.

`[z, gain] = tzero(sys)` also returns the gain (in the zero-pole-gain sense) if `sys` is a SISO system.

`z = tzero(a, b, c, d)` works directly with the state-space matrices and is equivalent to

$$z = \text{tzero}(\text{ss}(a, b, c, d))$$

Algorithm The transmission zeros are computed using the algorithm in [1].

See Also

<code>pzmap</code>	Pole-zero map
<code>pol e</code>	LTI system poles

References [1] Emami-Naeini, A., and P. Van Dooren, “Computation of Zeros of Linear Multivariable Systems,” *Automatica*, 18 (1982), pp. 415–430.

Purpose Generate a z -plane grid of constant damping factors and natural frequencies

Syntax `zgrid`
`zgrid(z, wn)`

Description `zgrid` generates a grid of constant damping factors from zero to one in steps of 0.1 and natural frequencies from zero to π in steps of $\pi/10$, and plots the grid over the current axis. If the current axis contains a discrete z -plane root locus diagram or pole-zero map, `zgrid` draws the grid over the plot without altering the current axis limits.

`zgrid(z, wn)` plots a grid of constant damping factor and natural frequency lines for the damping factors and normalized natural frequencies in the vectors `z` and `wn`, respectively. If the current axis contains a discrete z -plane root locus diagram or pole-zero map, `zgrid(z, wn)` draws the grid over the plot. The frequency lines for unnormalized (true) frequencies can be plotted using

`zgrid(z, wn/Ts)`

where T_s is the sample time.

`zgrid([], [])` draws the unit circle.

Example Plot z -plane grid lines on the root locus for the system:

$$H(z) = \frac{2z^2 - 3.4z + 1.5}{z^2 - 1.6z + 0.8}$$

by typing:

`H = tf([2 -3.4 1.5], [1 -1.6 0.8], -1)`

MATLAB returns:

Transfer function:

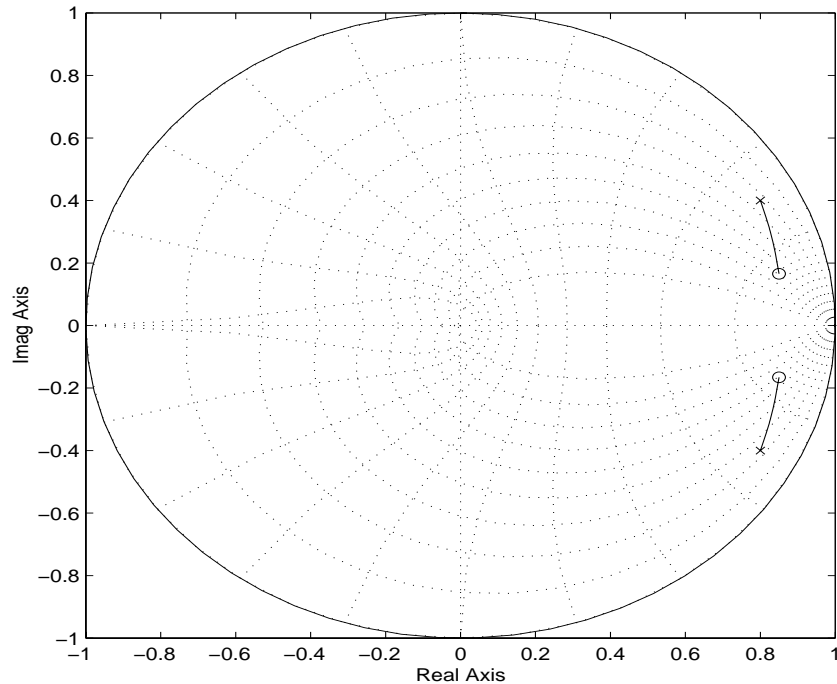
$$\frac{2z^2 - 3.4z + 1.5}{z^2 - 1.6z + 0.8}$$

Sampling time: unspecified

zgrid

Type:

```
rlocus(H)
zgrid
axis('square')
```



Limitations

`zgrid` plots the grid over the current axis regardless of whether the axis contains a root locus diagram or pole-zero map. Therefore, if the current axis contains, for example, a step response, you may superimpose a meaningless z -plane grid over the plot.

See Also

`pzmap`
`rlocus`
`sgrid`

Plot pole-zero map of LTI systems
Plot root locus
Generate s -plane grid lines

Purpose Specify zero-pole-gain models or convert LTI model to zero-pole-gain form

Syntax

```

sys = zpk(z, p, k)
sys = zpk(z, p, k, Ts)
sys = zpk(M)
sys = zpk(z, p, k, lti sys)

sys = zpk(z, p, k, 'Property1', Value1, ..., 'PropertyN', ValueN)
sys = zpk(z, p, k, Ts, 'Property1', Value1, ..., 'PropertyN', ValueN)

zsys = zpk(sys)
zsys = zpk(sys, 'inv')    % for state-space sys only

```

Description zpk is used to create zero-pole-gain models (ZPK objects) or to convert transfer functions or state-space models to zero-pole-gain form.

Creation of Zero-Pole-Gain Models

`sys = zpk(z, p, k)` creates a continuous-time zero-pole-gain model with zeros `z`, poles `p`, and gain(s) `k`. The output `sys` is a ZPK object storing the model data (see page 2-2).

In the SISO case, `z` and `p` are the vectors of zeros and poles and `k` is the scalar gain:

$$h(s) = k \frac{(s - z(1))(s - z(2)) \dots (s - z(m))}{(s - p(1))(s - p(2)) \dots (s - p(n))}$$

Set `z` or `p` to `[]` for systems without zeros or poles. These two vectors need not have equal length and the model need not be proper (that is, have an excess of poles).

To create a MIMO zero-pole-gain model, specify the zeros, poles, and gain of each SISO entry of this model. In this case,

- z and p are cell arrays of vectors with as many rows as outputs and as many columns as inputs, and k is a matrix with as many rows as outputs and as many columns as inputs.
- The vectors $z\{i, j\}$ and $p\{i, j\}$ specify the zeros and poles of the transfer function from input j to output i .
- $k(i, j)$ specifies the (scalar) gain of the transfer function from input j to output i .

See below for a MIMO example.

`sys = zpk(z, p, k, Ts)` creates a discrete-time zero-pole-gain model with sample time T_s (in seconds). Set $T_s = -1$ or $T_s = []$ to leave the sample time unspecified. The input arguments z , p , k are as in the continuous-time case.

`sys = zpk(M)` specifies a static gain M .

`sys = zpk(z, p, k, lti_sys)` creates a zero-pole-gain model with generic LTI properties inherited from the LTI model `lti_sys` (including the sample time). See page 2-18 for an overview of generic LTI properties.

Any of the previous syntaxes can be followed by property name/property value pairs

`'Property', Value`

Each pair specifies a particular LTI property of the model, for example, the input names or the input delay time. See `set` entry and the example below for details. Note that

`sys = zpk(z, p, k, 'Property1', Value1, ..., 'PropertyN', ValueN)`

is a shortcut for the sequence of commands:

```
sys = zpk(z, p, k)
set(sys, 'Property1', Value1, ..., 'PropertyN', ValueN)
```


Conversion to Zero-Pole-Gain Form

`zsys = zpk(sys)` converts an arbitrary LTI model `sys` to zero-pole-gain form. The output `zsys` is a ZPK object. By default, `zpk` uses `tzero` to compute the zeros when converting from state-space to zero-pole-gain. Alternatively,

```
zsys = zpk(sys, 'inv')
```

uses inversion formulas for state-space models to compute the zeros. This algorithm is faster but less accurate for high-order models with low gain at $s = 0$.

Variable Selection

As for transfer functions, you can specify which variable to use in the display of zero-pole-gain models. Available choices include s (default) and p for continuous-time models, and z (default), z^{-1} , or $q = z^{-1}$ for discrete-time models. Reassign the 'Variable' property to override the defaults. Changing the variable affects only the display of zero-pole-gain models.

Example

Example 1

Specify the zero-pole-gain model:

$$H(z) = \left[\frac{\frac{1}{z-0.3}}{\frac{2(z+0.5)}{(z-0.1+j)(z-0.1-j)}} \right]$$

```
z = {[ ] ; -0.5}
```

```
p = {0.3 ; [0.1+i 0.1-i]}
```

```
k = [1 ; 2]
```

```
H = zpk(z, p, k, -1) % unspecified sample time
```

Example 2

Convert the transfer function. Type:

```
h
```

MATLAB returns:

Transfer function:

$$\frac{-10 s^2 + 20 s}{s^5 + 7 s^4 + 20 s^3 + 28 s^2 + 19 s + 5}$$

to zero-pole-gain form. Type:

zpk(h)

and MATLAB returns:

Zero/pole/gain:

$$\frac{-10 s (s-2)}{(s+1)^3 (s^2 + 4s + 5)}$$

Algorithm

zpk uses the MATLAB function roots to convert transfer functions and the functions tzero and pole to convert state-space models.

See Also

zpkdata	Retrieve zero-pole-gain data
set	Set properties of LTI models
get	Get properties of LTI models
tf	Specify transfer functions or convert to TF form
ss	Specify state-space models or convert to state space

Purpose Quick access to zero-pole-gain data

Syntax

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

Description `[z, p, k] = zpkdata(sys)` returns the zeros z , poles p , and gain(s) k of the zero-pole-gain model `sys`. The outputs z and p are cell arrays with the following characteristics:

- z and p have as many rows as outputs and as many columns as inputs.
- The (i, j) entries $z\{i, j\}$ and $p\{i, j\}$ are the (column) vectors of zeros and poles of the transfer function from input j to output i .

The output k is a matrix with as many rows as outputs and as many columns as inputs such that $k(i, j)$ is the gain of the transfer function from input j to output i . If `sys` is a transfer function or state-space model, it is first converted to zero-pole-gain form using `zpk`.

For SISO zero-pole-gain models, the convenience syntax

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

forces `zpkdata` to return the zeros and poles directly as column vectors rather than as cell arrays (see example below).

`[z, p, k, Ts, Td] = zpkdata(sys)` also returns the sample time T_s and the input delay data T_d . For continuous-time models, T_d is a row vector with one entry per input channel ($T_d(j)$ indicates by how many seconds the j th input is delayed). For discrete-time models, T_d is the empty matrix `[]` (see `d2d` for delays in discrete systems).

You can access the remaining LTI properties of `sys` with `get` or by direct referencing, for example,

```
sys.Ts
sys.inputname
```

Example Given the zero-pole-gain model with two outputs and one input, type:

```
H
```

MATLAB returns:

Zero/pole/gain from input to output...

1
#1: $\frac{1}{(z-0.3)}$

2 (z+0.5)
#2: $\frac{(z+0.5)}{(z^2 - 0.2z + 1.01)}$

Sampling time: unspecified

and extracts the zero/pole/gain data embedded in H:

```
[z, p, k] = zpkdata(H)
```

The outputs z and p are 2-by-1 cell arrays as confirmed by typing:

```
class(z)
```

to which MATLAB returns

```
ans =  
cell
```

and typing:

```
size(z)
```

to which MATLAB returns:

```
ans =  
2 1
```

To access the zeros and poles of the second output channel of H, get the content of the second cell in z and p by typing:

```
z{2, 1}
```

to which MATLAB responds:

```
ans =  
-0.5000
```

and typing:

```
p{2, 1}
```

to which MATLAB responds:

```
ans =  
    0.1000+ 1.0000i  
    0.1000- 1.0000i
```

See Also

zpk	Specify zero-pole-gain models
get	Get properties of LTI models
ssdata	Quick access to state-space data
tfdata	Quick access to transfer function data

A

acker 9-10
 addition 2-29
 scalar 2-30
 algebraic loop 9-76
 aliasing 3-11
 append 9-11
 augstate 9-14

B

balancing 3-6, 9-15, 9-200
 bal real 9-15
 bandwidth 9-18
 bilinear approximation. *See* Tustin
 approximation
 block diagram. *See* model building
 bode 9-18
 Bode response (bode) 9-18

C

c2d 9-23
 cancellation 9-131
 canon 9-26
 canonical. *See* realization
 care 9-28
 cell array 2-7, 2-9, 2-15, 9-85, 9-215
 classical control 7-3, 7-19
 closed loop. *See* feedback
 companion realization 9-26
 comparing models 3-11, 9-18, 9-88, 9-119, 9-125
 compensator 5-7
 concatenation 2-7, 2-31, 2-36
 conditioning 8-4
 connect 9-32
 connection

 feedback 7-11, 9-73
 parallel 2-29, 7-53, 9-158
 series 2-30, 7-14, 9-180
 star product (LFT) 9-203
 continuous-time 3-2, 9-98
 conversion to. *See* conversion
 random model 9-178
 control design. *See* design
 controllability
 gramian 9-86
 matrix (ctrb) 9-41
 staircase form 9-43
 conversion
 automatic 2-27
 between model types 2-26, 2-28, 9-196, 9-210,
 9-223
 continuous to discrete (c2d) 2-40, 9-23
 discrete to continuous (d2c) 2-23, 2-40, 9-46
 with negative real poles 2-41, 9-47
 explicit 2-26
 resampling a discrete model 2-45, 9-49
 to state space 2-27, 9-196
 covar 9-38
 covariance 5-10, 9-38
 error. *See* error covariance
 noise. *See* noise
 output. *See* output
 state. *See* state
 ctrb 9-41
 ctrbf 9-43

D

d2c 9-46
 d2d 9-49
 damp 9-51

damping 9-51, 9-187, 9-219
dare 9-53
DC gain 9-56
dcgain 9-56
delay 2-37, 2-43
 d2d 2-37
 for discrete systems 2-37
 Pade approximation 2-39, 9-156
 supported functionalities 2-38
 time 2-18, 9-184
denominator
 common denominator 9-78, 9-209
 property 2-19, 9-185
 specification 2-6, 2-7, 2-14, 9-77
 value 2-16
descriptor. *See* state-space model
design 1-19
 classical 7-3, 7-19
 compensator 5-7
 Kalman estimator 5-10, 7-35, 7-56, 9-102, 9-107
 LQ regulator 9-58
 LQG 1-19, 5-9, 5-11, 7-30, 9-109
 pole placement 5-6, 9-10, 9-160
 regulator 5-7, 5-11, 7-30, 9-109, 9-166
 robustness 7-28
 root locus 5-4, 7-9, 7-23
 state estimator 5-6, 5-10, 9-70, 9-102, 9-107
design model 6-11, 9-171
digital filter
 filt 2-14
 specification 2-14, 9-77
Dirac input 9-88
discrete-time 3-2, 9-98
 control design 7-19
 equivalent continuous poles 9-51
 frequency 3-11, 9-22, 9-79

Kalman estimator 7-49, 9-102
 models. *See* LTI models
 random model 9-61
discretization 2-23, 2-40, 7-20, 9-23
 available methods 2-40, 9-23
 intersample behavior 9-120
 of delay systems 2-43
dlqr 9-58
dlyap 9-60
drmode 9-61
drss 9-61
dsort 9-63
DSP convention 2-14, 9-77, 9-185, 9-212
dss 9-64
dssdata 9-66
dynamics 3-4

E

eig 9-67
error covariance 7-55, 7-60, 9-105
esort 9-68
estim 9-70
estimator 5-6, 5-10, 9-70, 9-102, 9-107
 current 9-104
 discrete 9-102
 discrete for continuous plant 9-107
 gain 5-7
evalfr 9-72

F

feedback 7-11, 9-73
 algebraic loop 9-76
 negative 9-73
 positive 9-73
feedback 9-73

feedthrough gain 2-20
 filter 2-14, 9-77
 filtering. *See* Kalman estimator
 final time. *See* time response
 first-order hold (FOH) 2-42, 9-23
 with delays 2-43
 freqresp 9-79
 frequency
 crossover 9-128
 for discrete systems 3-11, 9-22
 grid 3-10
 linearly spaced frequencies 3-11
 logarithmically spaced frequencies 3-11, 9-18
 natural 9-51, 9-187, 9-219
 Nyquist. *See* Nyquist
 range 3-10, 9-18
 frequency response 3-9
 at single frequency (eval fr) 9-72
 Bode plot 9-18
 customized plots 3-13
 discrete-time frequency 3-11, 9-22, 9-79
 magnitude 9-18
 MIMO 3-10, 9-18, 9-139, 9-146
 Nichols chart (ngrid) 9-137
 Nichols plot 9-139
 Nyquist plot 9-146
 over grid of frequencies (freqresp) 9-79
 phase 9-18
 plotting/comparing multiple systems 3-11, 9-18, 9-125
 singular value plot 9-189
 viewing the gain and phase margins 9-128

G

gain 2-8
 estimator gain 5-7, 5-10

feedthrough 2-20
 low frequency (DC) 9-56
 margin 3-9, 7-28, 9-18, 9-128
 property 2-19
 selection 5-4, 5-6, 5-7, 5-10
 state-feedback gain 5-6, 5-10, 9-58
 gensig 9-82
 get 2-23, 9-84
 gram 9-86
 gramian (gram) 9-15, 9-86
 GUI 6-2

H

Hamiltonian matrix and pencil 9-29
 hidden oscillations 9-120

I

I/O
 concatenation (append) 5-3
 dimensions 3-2, 9-193
 names 2-18, 2-21
 relation 2-33
 impulse 9-88
 impulse response 9-88
 inheritance 2-28, 9-64, 9-195
 initial 9-92
 initial condition 9-92
 innovation 9-104
 input 2-2
 delay. *See* delay
 Dirac input 9-88
 generate test input signals 9-82
 names 2-18, 2-28, 9-184
 number of inputs 3-2, 9-193
 pulse 9-82, 9-88

- resampling 9-120
- sine wave 9-82
- square wave 9-82
- input point 4-41, 4-43
- interconnection. *See* model building
- intersample behavior 9-120
- inv 9-95
- inversion 2-31, 9-95
 - limitations 9-96
- isct 9-98
- isdtd 9-98
- isempty 9-99
- isproper 9-100
- isssiso 9-101

K

- Kalman
 - filter. *See* Kalman estimator
 - filtering 5-12, 7-49
 - gain 5-10
- kalman 9-102
- Kalman estimator
 - continuous 5-10, 7-35
 - current 9-104
 - discrete 7-49, 9-102
 - discrete for continuous plant 9-107
 - innovation 9-104
 - steady-state 5-10, 9-102
 - time-varying 7-56
- kalmd 9-107

L

- LFT. *See* linear-fractional transformation
- linear-fractional transformation (LFT) 9-203
- linear-quadratic (LQ)

- continuous LQ regulator 5-10, 7-35, 9-113
- cost function 7-35, 9-58, 9-113
- discrete LQ regulator 9-58, 9-115
- optimal state-feedback gain 5-10, 9-113, 9-115, 9-117
- output weighting 9-117
- weighting matrices 5-10

LQG

- current regulator 9-110
- design 1-19, 5-9, 5-11, 7-30, 7-46
- Kalman state estimator 5-10, 9-102, 9-107
- LQ-optimal gain 5-10, 7-35, 9-113, 9-115, 9-117
- regulator 1-20, 5-11, 7-30, 9-109
- weighting matrices 5-10

- lqgreg 9-109

- lqr 9-113

- lqrd 9-115

- lqry 9-117

- lsim 9-118

- LTi (linear time-invariant) 2-2

LTi models

- accessing the model data 2-15, 9-202
- characteristics 3-2
- comparing multiple models 3-11, 9-18, 9-88, 9-119, 9-125
- continuous 3-2
- conversion. *See* conversion
- discrete 2-13, 3-2, 9-98
- dynamics 3-4
- empty 2-8, 3-2, 9-99
- frequency response. *See* frequency response
- model order reduction 3-16, 9-15, 9-133
- norms 9-142
- operations on LTi models. *See* operations
- proper 3-2, 9-100
- random 9-61, 9-178
- resizing 2-35

- second-order 9-154
- SISO 9-101
- time response. *See* time response
- type 3-2
- LTI objects 2-2, 2-18, 2-24
 - methods 2-3
 - properties. *See* LTI properties
- LTI properties 2-3, 2-18
 - accessing property values (get) 2-23, 2-24, 9-84
 - admissible values 9-183
 - displaying properties 2-24, 9-84
 - generic properties 2-18
 - inheritance 2-28, 9-64, 9-195
 - model-specific properties 2-19
 - online help (l ti props) 2-18
 - property name 2-18, 2-21, 9-84, 9-182
 - property value 2-18, 2-23, 9-84, 9-182
 - setting property values 2-21, 9-182, 9-196, 9-210, 9-222
- LTI Viewer 9-124
 - deselection of models 4-10
 - multiselection of models 4-10
 - selected browser 4-47
 - selection or deselection of listbox items 4-10
- l ti view 9-124
- lyap 9-126
- Lyapunov equation 9-39, 9-87
 - continuous 9-126
 - discrete 9-60

M

- magnitude 9-18
- margi n 9-128
- margins 3-9, 7-28, 9-18, 9-128
- matched pole-zero 2-43, 9-23

- methods 2-3
- MIMO 2-2, 2-32, 3-8, 3-10, 9-18, 9-88
- mi nreal 9-131
- model building 5-3
 - appending LTI models 9-11
 - feedback connection 7-11, 9-73
 - LFT connection 9-203
 - modeling block diagrams (connect) 5-3, 9-32
 - parallel connection 2-29, 7-53, 9-158
 - series connection 2-30, 7-14, 9-180
- model dynamics 3-4
- model order reduction 3-16, 9-15, 9-133
- modeling. *See* model building
- modred 9-133
- multiplication 2-30
 - scalar 2-30

N

- natural frequency 9-51
- ngri d 9-137
- Nichols
 - chart 9-137
 - plot (ni chol s) 9-139
- ni chol s 9-139
- noise
 - covariance 5-10
 - measurement 5-9, 9-70
 - process 5-9, 9-70
 - white 5-9, 9-38
- norm 9-142
- norms of LTI systems (norm) 9-142
- numerator
 - property 2-19, 9-185
 - specification 2-6, 2-7, 2-14, 9-77
 - value 2-16, 9-85
- numerical stability 8-6

Nyquist

- frequency 3-11, 9-22
- plot (nyquist) 9-146

nyquist 9-146

O

object-oriented programming 2-3

objects. *See* LTI objects

observability

- gramian 9-86
- matrix (ctrb) 9-149
- staircase form 9-151

obsv 9-149

obsvf 9-151

operations 2-28

- addition 2-29
- appending LTI models 9-11
- arithmetic 2-29
- augmenting state with outputs 9-14
- concatenation 2-7, 2-31, 2-36
- extracting a subsystem 2-4, 2-33
- inversion 2-31, 9-95
- modifying a subsystem 2-33, 2-36
- multiplication 2-30
- pertransposition 2-33
- resizing 2-35
- sorting the poles 9-63, 9-68
- subtraction 2-30
- transposition 2-33

ord2 9-154

output 2-2

- covariance 9-38
- names 2-18, 2-28, 9-184
- number of outputs 3-2, 9-193

output point 4-41, 4-43

overloaded operations 2-3, 9-67

overshoot 3-7

P

pade 9-156

Pade approximation (pade) 2-39, 9-156

parallel 9-158

parallel connection 2-29, 7-53, 9-158

pertransposition 2-33

phase 9-18

- margin 3-9, 7-28, 9-18, 9-128

place 9-160

plotting

- customized plots 3-13
- frequency response. *See* frequency response
- multiple systems 3-11, 9-18, 9-88, 9-119, 9-125
- Nichols chart (ngri d) 9-137
- s-plane grid (sgri d) 9-187
- time response. *See* time response
- z-plane grid (zgri d) 9-219

pole 9-162

pole placement 5-6, 9-10, 9-160

- conditioning 5-8

poles 2-8

- computing the poles 9-67, 9-162
- damping 9-51, 9-187, 9-219
- equivalent continuous poles 9-51
- multiple 9-162
- natural frequency 9-51, 9-187, 9-219
- pole-zero map 9-163
- property 2-19
- sorting by magnitude (dsort) 9-63
- sorting by real part (esort) 9-68
- s-plane grid (sgri d) 9-187
- z-plane grid (zgri d) 9-219

pole-zero

- cancellation 9-131

- map (pzmap) 9-163
- precedence rules 2-3, 2-28
- proper 3-2, 9-100
- properties. *See* LTI properties
- pulse 9-82, 9-88
- pzmap 9-163

R

- random models 9-61, 9-178
- realization 3-6, 9-196
 - balanced 3-6, 9-15, 9-200
 - canonical 3-6, 9-26
 - companion form 9-26
 - minimal 9-131
 - modal form 9-26
 - state coordinate transformation 3-6, 9-27, 9-199
- Redheffer star product 9-203
- reduced-order model 3-16, 9-15, 9-133
- regulation 1-19, 5-9, 7-30, 9-166
 - performance 5-10
- resampling (d2d) 2-45, 9-49
- resizing an LTI model 2-35
- Riccati equation 5-10
 - continuous (care) 9-28
 - discrete (dare) 9-53
 - for LQG design 9-104, 9-113
 - H_∞ -like 9-30
 - stabilizing solution 9-28, 9-53
- rise time 3-7
- rlcfind 9-173
- rl locus 9-175
- rl tool 9-169
- rmodel 9-178
- robustness 7-28
- root locus

- design 5-4, 7-9, 7-23
- plot (rl locus) 9-175
- select gain from 9-173
- Root Locus Design GUI 5-4, 6-1, 6-6, 6-7, 6-13, 9-171, 9-172
 - add grid/boundary 6-24
 - axes settings 6-19
 - clearing model and compensator data 6-45
 - continuous to discrete model conversion 6-3, 6-38, 6-44
 - current compensator text 6-28
 - design region boundaries 6-24
 - design specifications 6-35
 - discrete to continuous model conversion 6-3, 6-38, 6-44
 - drawing a Simulink diagram from 6-43
 - edit compensator 6-27
 - erasing compensator poles and zeros 6-3, 6-38, 6-41
 - feedback configurations 6-10, 9-172
 - feedback structure 6-10, 9-171
 - gain set point 6-13
 - import model 6-9
 - listing poles and zeros 6-41
 - LTI Viewer response plots 6-35
 - model source 6-38, 6-39
 - printing 6-27
 - toolbar 6-27
 - zoom tools 6-15, 6-16
- Root Locus Design GUI tool 6-6
- Root Locus Design model
 - compensator 6-11, 9-171
- rss 9-178

S

- sample time 2-13, 2-18, 2-28

- accessing the sample time value 2-15, 9-202
 - resampling 2-45, 9-49
 - setting the sample time value 2-23, 9-184, 9-195, 9-210, 9-222
 - unspecified 2-18, 9-22
 - scaling 8-15
 - second-order model 9-154
 - series 9-180
 - series connection 2-30, 7-14, 9-180
 - set 2-21, 9-182
 - settling time 3-7
 - signal generator 9-82
 - simulation of linear systems. *See* time response
 - Simulink LTI Viewer 4-37, 4-39
 - adding input points 4-41
 - adding output points 4-41
 - analysis model 4-39
 - get linearized model 4-48
 - model_inputs_and_outputs block set 4-39
 - opening the viewer 4-39
 - saving analysis models 4-50
 - setting operating conditions 4-45
 - sine wave 9-82
 - singular value plot (bode) 9-189
 - SISO 2-2, 3-2, 6-2, 9-101
 - square wave 9-82
 - ss 2-10, 9-195
 - SS object. *See* state-space model
 - stability
 - numerical 8-6
 - stabilizable 9-31
 - stabilizing 9-28, 9-53
 - star product 9-203
 - state 2-10
 - augment with outputs 9-14
 - covariance 9-38
 - estimator 5-6, 5-10, 9-70, 9-102, 9-107
 - feedback 5-6, 9-58
 - matrix 2-20
 - names 2-20, 9-185
 - number of states 9-193
 - transformation 3-6, 9-27, 9-199
 - uncontrollable 9-131
 - unobservable 9-131, 9-151
 - vector 2-2
 - state-space model 2-2, 8-8
 - balancing 3-6, 9-15, 9-200
 - conversion. *See* conversion
 - descriptor 2-12, 2-15, 9-64, 9-66
 - dss 9-64
 - initial condition response 9-92
 - matrices 2-10
 - quick data retrieval 2-15, 9-66, 9-202
 - random 9-61, 9-178
 - scaling 8-15
 - specification 2-10, 9-195
 - ss 2-10, 9-195
 - SS object 2-2
 - transfer function of 2-26
 - state-space realization. *See* realization
 - steady state
 - error 3-7
 - step response 9-205
 - subsystem 2-4, 2-33, 2-36
 - subtraction 2-30
 - Sylvester equation 9-126
 - symplectic pencil 9-54
- T**
- tf 2-6, 9-209
 - TF object. *See* transfer function
 - time delay. *See* delay
 - time response 3-7

- customized plots 3-13
- final time 3-8, 9-88
- impulse response (i mpul se) 9-88
- initial condition response (i ni ti al) 9-92
- MIMO 3-8, 9-88, 9-118
- plotting/comparing multiple systems 3-11, 9-88, 9-119, 9-125
- response to arbitrary inputs (l si m) 9-82, 9-118
- step response (step) 9-205
- time range
- to white noise 9-38
- vector of time samples 3-8
- time-varying Kalman filter 7-56
- transfer function 2-2, 8-8
 - common denominator 9-78, 9-209
 - conversion. *See* conversion
 - denominator. *See* denominator
 - discrete 2-13, 2-14, 9-77, 9-185, 9-212
 - DSP convention 2-14, 9-77, 9-185, 9-212
 - fi l t 2-14, 9-77
 - limitations 2-27
 - MIMO 2-6, 2-32, 9-209
 - numerator. *See* numerator
 - quick data retrieval 2-15, 9-215
 - random 9-61, 9-178
 - specification 2-6, 9-209
 - static gain 2-8, 9-210
 - t f 2-6, 9-209
 - TF object 2-2, 2-6
 - variable 2-19, 2-29, 9-185, 9-212
- transmission zeros. *See* zeros
- transposition 2-33
- triangle approximation 2-42, 9-23
- Tustin approximation 2-42, 9-23
 - with frequency prewarping 2-43, 9-23

U

- undersampling 9-120

W

- white noise. *See* noise

Z

- zero-order hold (ZOH) 2-40, 7-20, 9-23
 - with delays 2-43
- zero-pole-gain model 2-2, 8-14
 - conversion. *See* conversion
 - MIMO 2-9, 2-32, 9-222
 - quick data retrieval 2-15, 9-225
 - specification 2-8, 9-221
 - static gain 9-222
 - zpk 2-8, 9-221
 - ZPK object 2-2, 2-8
- zeros 2-8
 - computing the zeros 9-218
 - pole-zero map 9-163
 - property 2-20
 - transmission 9-218
- zpk 2-8, 9-221
- ZPK object. *See* zero-pole-gain model