Control System Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

Reference

Version 5

The MathWorks

How to Contact The MathWorks:

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information
508-647-7000	Phone
508-647-7001	Fax
The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098	Mail

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

Control System Toolbox Reference

Printing History:			
	June 2001	Online only	New for Version 5.1 (Release 12.1)
	July 2002	Online only	Version 5.2 (Release 13)

Contents

1 SISO Design Tool

2 LTI Viewer

3 Right-Click Menus for Response Plots

4 Function Reference

5 Block Reference

SISO Design Tool

The SISO Design Tool is a graphical-user interface (GUI) that allows you to use root-locus, Bode diagram, and Nichols plot techniques to design compensators. The SISO Design Tool by default displays the root locus and Bode diagrams for your imported systems. The two are dynamically linked; for example, if you change the gain in the root locus, it immediately affects the Bode diagrams as well.

This tool is used extensively in *Getting Started with the Control System Toolbox*. In particular, you should read Chapter 4, “Designing Compensators,” of that book to see how to do typical design tasks with the SISO Design Tool. This document, on the other hand, is a reference that describes all available options for the SISO Design Tool.

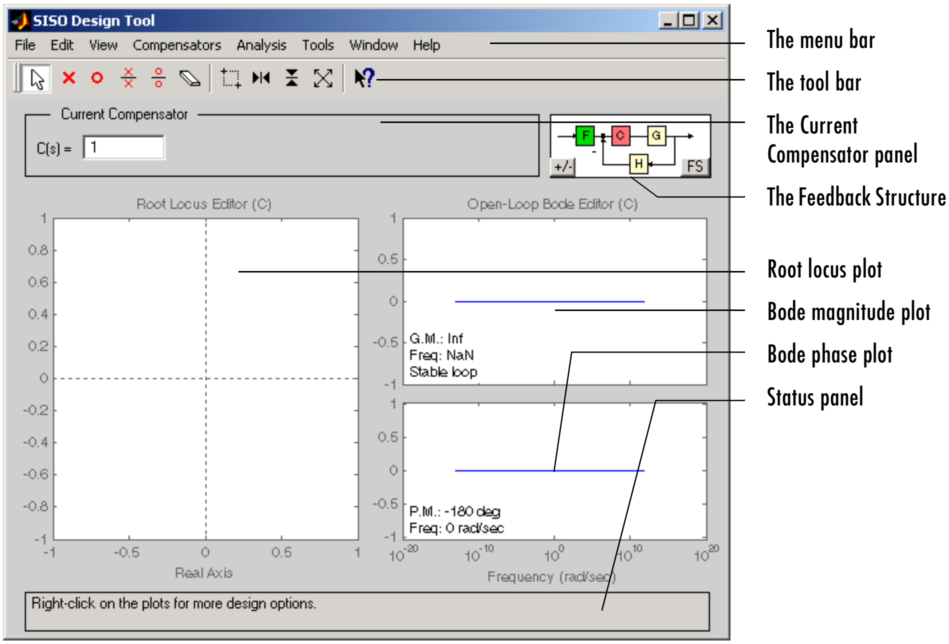
Opening the SISO Design Tool

Type

```
sisotool
```

to open the SISO Design Tool.

This picture shows the GUI and introduces some terminology.



The SISO Design Tool

This document describes the SISO Design Tool features left-to-right and top-to-bottom, starting with the menu bar and ending with the status panel at the bottom of the window.

If you want to match the SISO Design Tool pictures shown below, type

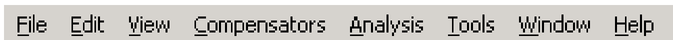
```
load ltiexamples
```

at the MATLAB prompt. This loads the same set of linear models that this document uses as examples in the GUI. The examples all use the `Gservo` system for plot displays.

Menu Bar

Note Click on items on the menu bar pictured below to get help contents.

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



File

Note Click on items in the **File** menu pictured below to get help contents.

.

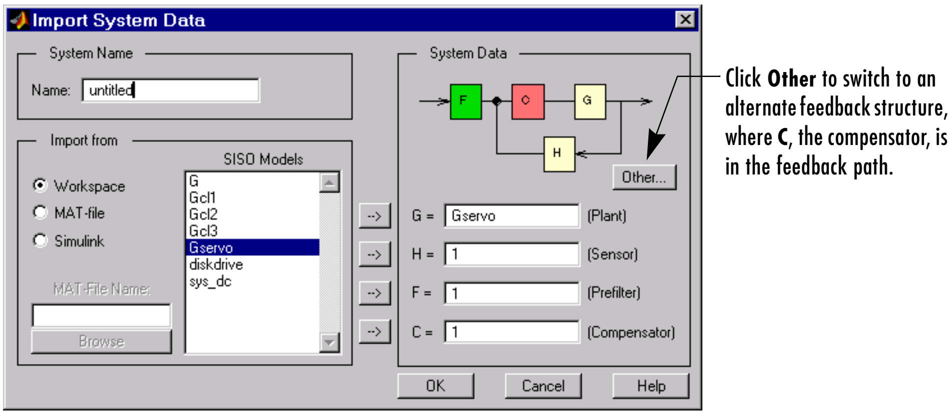
Using the **File** menu, you can:

- Import and export models
- Save and reload sessions
- Set toolbox preferences
- Print and print to figure
- Close the SISO Design Tool

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

Import

To import models into the SISO Design Tool, select **Import** from the **File** menu. This opens the **Import System Data** window, which is shown below.



The Import System Data Window

The following sections discuss the System Name, Import from, and System Data panels of the **Import System Data** window.

System Name. Use the **Name** field to assign a name to the imported system. The default name is `untitled`.

Import From. To import models, select them from the SISO Models list and use the right arrow buttons to place the models in **G** (plant), **H** (sensor), **F** (prefilter), or **C** (compensator). You can import models from:

- The MATLAB Workspace
- A MAT-file
- Simulink (`.mdl` files)

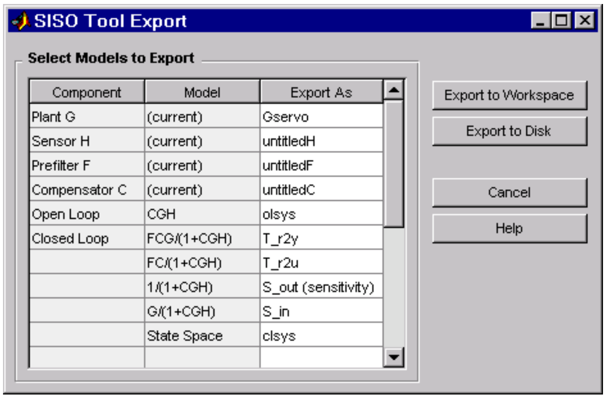
System Data. The System Data panel performs two functions:

- Feedback structure specification — Click **Other** to toggle between placing the compensator in the forward and feedback paths
- Model import specification — You can import models for the plant (**G**), compensator (**C**), prefilter (**F**), and/or sensor (**H**). To import a model, select it

from the SISO model list and click the right-arrow button next to the desired model field.

Export

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



The SISO Tool Export Window

With this window, you can:

- Export models to the MATLAB Workspace or to a disk
- Rename models when exporting
- Save variations on models, including open and closed loop models, sensitivity transfer functions, and state-space representations

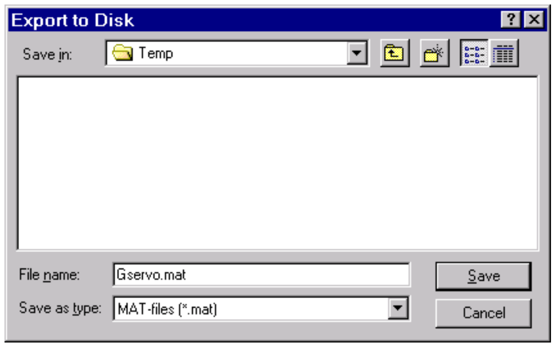
Exporting to the Workspace. To export models to the MATLAB workspace, follow these steps:

1. Select the model you want to export from the Component list by left-clicking the model name. To select more than one model, hold down the **Shift** key if they are adjacent on the list. If you want to save nonadjacent models, hold down the **Ctrl** key while selecting the models.

2. For each model you want to save, specify a name in the model's cell in the Export As list. A default name exists if you do not want to assign a new name.

3 Click **Export to Workspace**.

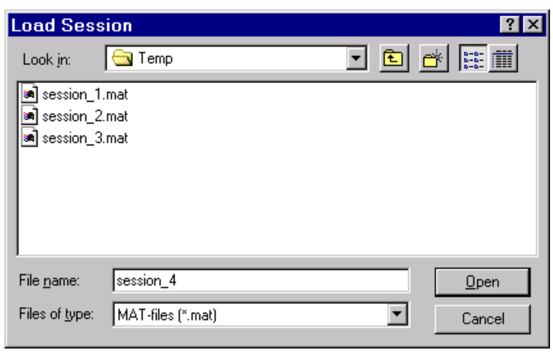
Exporting to a MAT-file. If you want to save your models in a MAT-file, follow steps 1 and 2 and click **Export to Disk**, which opens this window.



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

Save Session

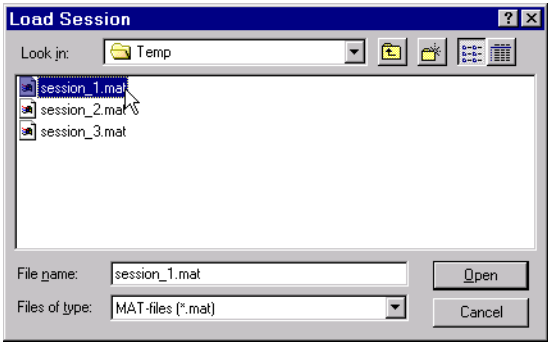
You can quit MATLAB and later restore the SISO Design Tool to the state you left it in by saving the session. Select **Save Session** from the **File** menu. This opens the **Save Session** window.



To save a session, specify a file name and click **Save**. The current state and configuration of your SISO Design Tool are saved as a MAT-file. To load a saved session, see the “Load Session” on page 1-8 section.

Load Session

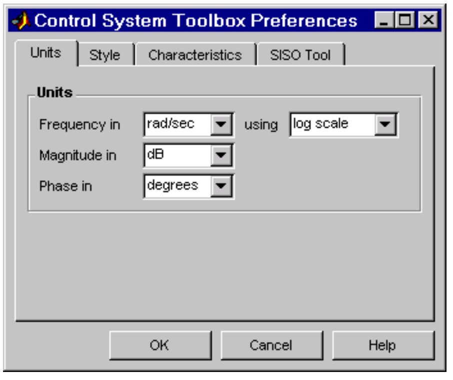
To load a saved **SISO Design Tool** session, select **Load Session** from the **File** menu. This opens the **Load Session** menu.



Sessions are saved as MAT-files. Select the session you want to load from the list, and click **Open**. See "Save Session" on page 1-7 for information on saving **SISO Design Tool** sessions.

Toolbox Preferences

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



The Control System Toolbox Preferences Window

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

Print

Use **Print** to send a picture of the SISO Design Tool to your printer.

Print to Figure

Print to Figure opens a separate figure window containing the design views in your current SISO Design Tool.

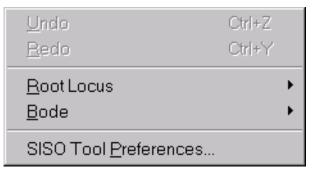
Close

Use **Close** to close the SISO Design Tool.

Edit

Note Click on items in the **Edit** menu pictured below to get help contents.

.



Undo and Redo

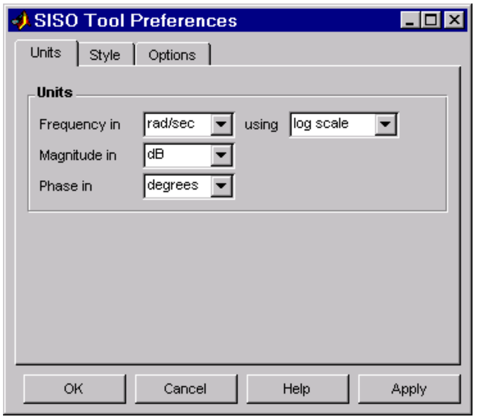
Use **Undo** and **Redo** to go back and forward in the design steps. Note that both **Undo** and **Redo** menus change when the task you have just performed changes. For example, if you change the compensator gain, the menu items become **Undo Gain** and **Redo Gain**.

Root Locus and Bode Diagrams

The **Root Locus** and **Bode Diagrams** menu options replicate the functionality of the right-click menus. If you open a Nichols plot or a Prefilter Bode diagram, the **Edit** menu replicates the right-click menus for these features as well. See “Right-Click Menus” on page 1-28 for information about the features available from the right-click menus.

SISO Tool Preferences

SISO Tool Preferences opens the **SISO Tool Preferences** editor. This picture shows the open window.



The SISO Tool Preferences Editor

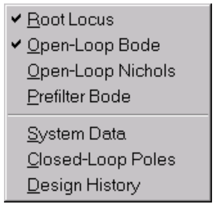
You can use this window to do the following:

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

For a complete description of properties and preferences, see "SISO Design Preferences" online in the Control System Toolbox documentation.

View

Note Click on items in the **View** menu pictured below to get help contents.



Root Locus and Bode Diagrams

By default, the SISO Design Tool displays the root locus and Bode magnitude and phase diagrams. You can deselect either to show only the root locus or the Bode diagram.

Open-Loop Nichols

Select **Open-Loop Nichols** from the **View** to add an interactive open-loop Nichols plot to the SISO Design Tool. All the options available from the root locus and Bode diagrams for compensator design are also available from the Nichols plot.

For a worked example, see "Nichols Plot Design" in *Getting Started with the Control System Toolbox*.

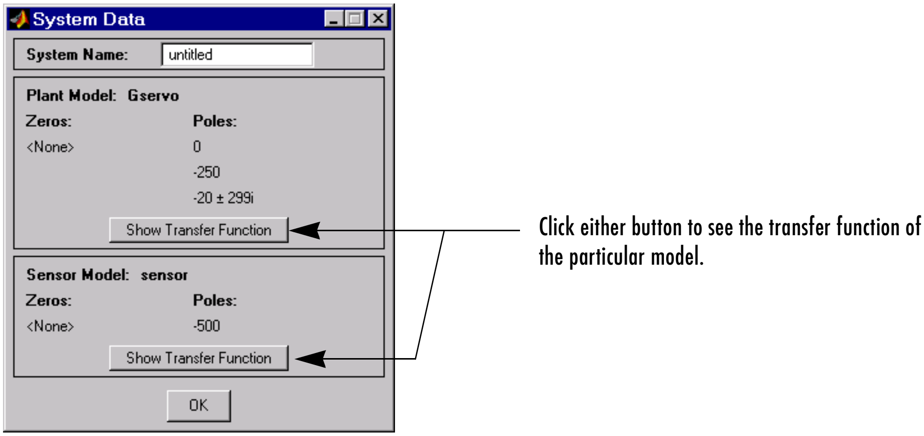
Prefilter Bode

Select **Prefilter Bode** to open a Bode diagram for the prefilter (**F**). You can either edit a prefilter that you imported into your design or create a new prefilter. The SISO Design Tool provides right-click menus and interactive graphics that facilitate prefilter design; the features are the same as those available from the Bode diagrams for the compensator (**C**).

For an example of prefilter design, see "Adding a Prefilter" in *Getting Started with the Control System Toolbox*.

System Data

To see information about your plant and sensor models, select **System Data** under **View**. This opens the window shown below.

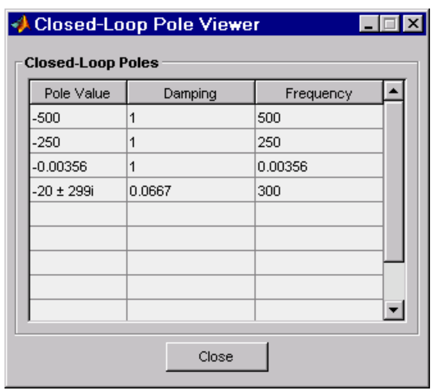


The System Data Window

The **System Data** window displays basic information about the models you've imported.

Closed-Loop Poles

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



This window displays all the closed-loop pole values of the current system, and their damping and frequency.

Design History

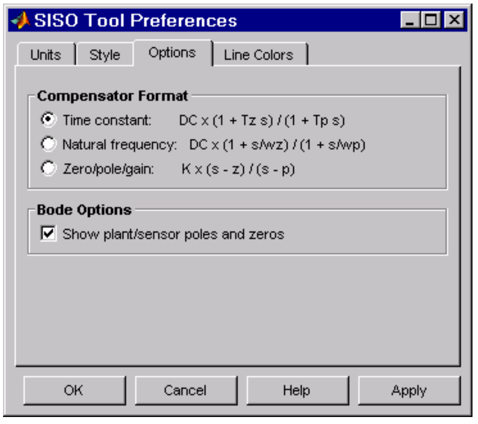
Selecting **Design History** from the **View** opens the **Design History** window, which displays all the actions you've performed during a design session. You can save the history to an ASCII flat text file.

Compensators

Note Click on items in the **Compensators** menu pictured below to get help contents.

Format

Selecting **Format** under **Compensators** activates the **SISO Tool Preferences** editor with the **Options** page open. This figure shows the **Options** page.



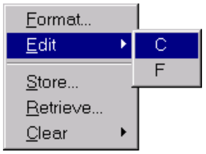
Use the radio buttons to toggle between time constant, natural frequency, and zero/pole/gain compensator formats.

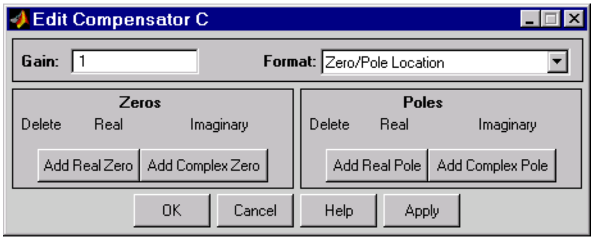
By default, the **SISO Design Tool** shows the plant poles and zeros on Bode diagrams as red x's and o's, respectively. Uncheck the Show plant/sensor poles and zeros box to hide the plant and sensor poles and zeros.

For a general description of the SISO Tool Preferences editor, see "SISO Design Tool Preferences" online in the Control System Toolbox documentation.

Edit

Choose **C** or **F** from **Edit** under the **Compensators** menu to open the **Edit Compensator** window for the compensator (**C**) or the prefilter (**F**), respectively. For example, this figure shows the selection of the compensator.





The Edit Compensator C Window

If you had chosen **F**, the **Edit Compensator F** window would have opened. Both windows have the same functionality.

You can use this window to inspect pole, zero, and gain data, and to edit this data using your keyboard (as opposed to graphically editing the compensator data). You have the following choices available from this window:

- "Adjusting the Gain"
- "Changing the Format" for specifying pole and zero locations
- "Adding Poles and Zeros"
- "Editing Poles and Zeros"
- "Deleting Poles and Zeros"

In the following sections, the descriptions of these tasks apply equally to the prefilter (**F**) and the compensator (**C**).

Adjusting the Gain. To change the compensator gain, enter the new value in the **Gain** field.

Changing the Format. You can see the poles and zeros either as complex numbers (Zero/Pole Location) or as damping ratio and natural frequency pairs (Damping/Natural Frequency). The default is Zero/Pole Location, which means that the window shows the numerical values. Use the **Format** menu to toggle between the two formats.

Adding Poles and Zeros. To add real poles to your compensators, click **Add Real Pole**. This action opens an empty field in the Poles panel. Specify the pole value in the field. To add a pair of complex poles, click **Add Complex Pole**. In this case, two fields appear: one for the real and another for the imaginary part of the poles. Note that you must specify the a negative sign for the real part of the pole if you want to specify a pair left-plane poles, but that the imaginary part is defined as +/-, so you do not have to specify the sign for that part.

If you specify the damping/natural frequency format, there is no distinction between the real and complex pole specifications. Clicking either button opens two fields: one for specifying the damping and another for the natural frequency. If you clicked **Add Real Pole**, you only need to specify the natural frequency since the **Edit Compensator** window automatically places a 1 in the damping field in this case.

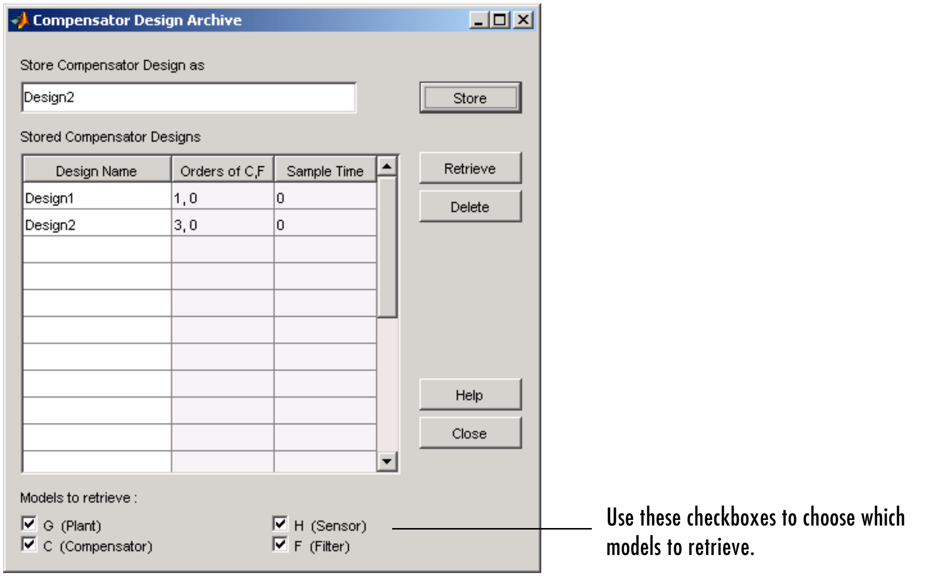
Adding zeros is exactly the same; click **Add Real Zero** or **Add Complex Zero** and proceed as above.

Editing Poles and Zeros. You can change the pole locations or damping ratios/natural frequencies for existing poles and zeros by specifying new values in the appropriate fields. The SISO Design Tool automatically updates to reflect the changes.

Deleting Poles and Zeros. Whenever you add poles or zeros using the **Edit Compensator** window, a delete box appears to the left of the fields used to specify the pole/zero values. Check this box anytime you want to delete the pole or zero specified next to it.

Store/Retrieve

Use **Store/Retrieve** to open the **Compensator Design Archive** window.



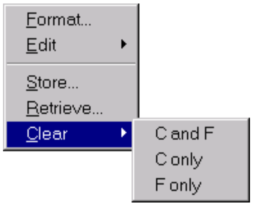
To store a design, type a design name in the **Store Compensator Design as** field and click **Store**.

This window lists all the compensator designs you have stored during a SISO Design Tool session. It also lists the orders of your compensator (**C**) and prefilter (**F**) pairs, and their sample times (0 means that they're continuous).

To retrieve a stored design, left-click on the design name to select it and click **Retrieve**. To delete a design, select it and click the **Delete** button.

Clear

Select **Clear** to eliminate prefilter and compensator dynamics and set the gain to 1.

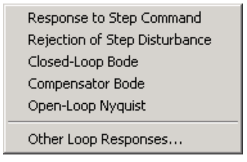


You can clear:

- **C** and **F** (the compensator and prefilter both)
- **C** only
- **F** only

Analysis

Note Click on items in the **Tools** menu pictured below to get help contents.



Each of the top group of items opens an LTI Viewer that is dynamically linked to your SISO Design Tool. You have the following response plot choices:

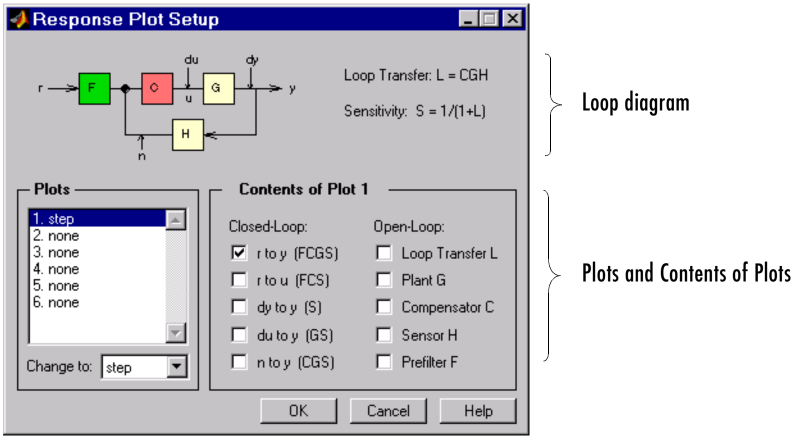
- **Response to Step Command** — The closed-loop step response of your system
- **Rejection of Step Disturbance** — The open-loop step response of your system
- **Closed-Loop Bode** — The closed-loop Bode diagram for your system
- **Compensator Bode** — The open-loop Bode diagram for your compensator

- **Open-Loop Nyquist** — The open-loop Nyquist plot for your system

When you make changes to the design in the SISO Design Tool, the response plots in the LTI Viewer automatically change to reflect the new design's responses.

Customizing Loop Responses

If you choose **Other Loop Responses**, the **Response Plot Setup** window opens.



Response Plot Setup Window

The following sections describe the main components of the **Response Plot Setup** window.

Loop diagram. At the top of the Response Plot Setup window is a loop diagram. This block diagram shows the feedback structure of your system. The diagram in "Response Plot Setup Window" on page 1-18 shows the default configuration; the compensator is in the forward path. If your system has the compensator in the feedback path, this window correctly displays the alternate feedback structure.

Note that window lists two transfer functions next to the loop diagram:

- Loop transfer — This is defined as the compensator (**C**), the plant (**G**), and the sensor (**H**) multiplied together (**CGH**). If you haven't defined a sensor, its default value is 1.

- Sensitivity function — This is defined as $\frac{1}{1+L}$, where L is the loop transfer function.

Some of the open- and closed-loop responses use these definitions. See "Contents of plots" on page 1-19 for more information.

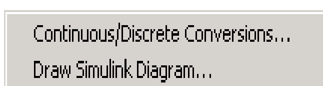
Plots. You can have up to six plots in one LTI Viewer. By default, the Response Plot Setup window specifies one step response plot. To add a plot, start by selecting "2. None" from the list of plots and then specify a new plot type in the **Change to** field. You can choose any of the plots available in the LTI Viewer. Select "None" to remove a plot.

Contents of plots. Once you have selected a plot type, you can include several open- and closed-loop transfer functions to be displayed in that plot. You can plot open-loop responses for each of the components of your system, including your compensator (**C**), plant (**G**), prefilter (**F**), or sensor (**H**). In addition, loop transfer and sensitivity transfer functions are available. Their definitions are listed in the Response Plot Setup window.

See the block diagram in "Response Plot Setup Window" on page 1-18 for definitions of the input/output points for closed-loop responses.

Tools

Note Click on items in the **Tools** menu pictured below to get help contents.



Loop Responses

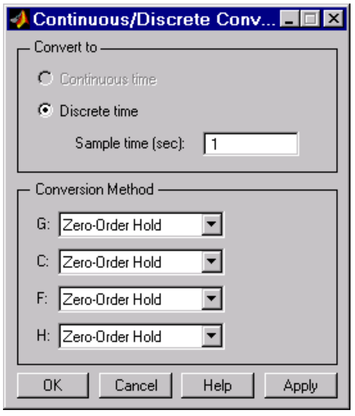
For examples that use LTI Viewers linked with the SISO Design Tool, see "Designing Compensators" in *Getting Started with the Control System Toolbox*. See the "LTI Viewer" on page 2-1 for a complete description of all the features of the LTI Viewer.

Continuous/Discrete Conversions

Selecting **Continuous/Discrete Conversions** opens the **Continuous/Discrete Conversions** window, which you can use to convert between continuous to discrete designs.You can select the following:

- Conversion method
- Sample time
- Critical frequency (where applicable)

This picture shows the window.



The Continuous/Discrete Conversion Window

Conversion domain. If your current model is continuous-time, the upper panel of the Continuous/Discrete Conversion window automatically selects the **Discrete time** radio button. If your model is in discrete-time, see “Discrete-time domain” on page 1-21.

To convert to discrete time, you must specify a positive number for the sample time in the **Sample time (sec)** field.

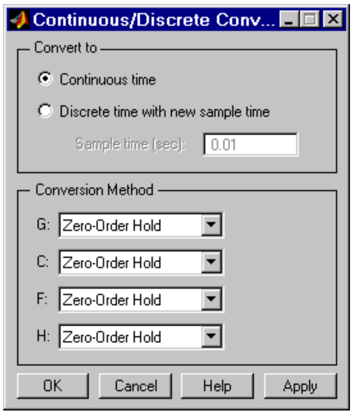
You can perform continuous to discrete conversions on any of the components of your model: the plant (**G**), the compensator (**C**), the prefilter (**F**), or the sensor (**H**). Select the method you want to use from the menus next to the model elements.

Conversion method. The following are the available continuous-to-discrete conversion methods:

- Zero-order hold
- First-order hold
- Tustin
- Tustin with prewarping
- Matched pole/zero

If you choose Tustin with prewarping, you must specify the critical frequency in rad/sec.

Discrete-time domain. If you currently have a discrete-time system, the Continuous/Discrete Conversion window looks like this figure.



You can either change the sample time of the discrete system (resampling) or do a discrete-to-continuous conversion.

To resample your system, select **Discrete time with new sample time** and specify the new sample time in the **Sample time (sec)** field. The sample time must be a positive number.

To convert from discrete-time to continuous-time, you have the following options for the conversion method:

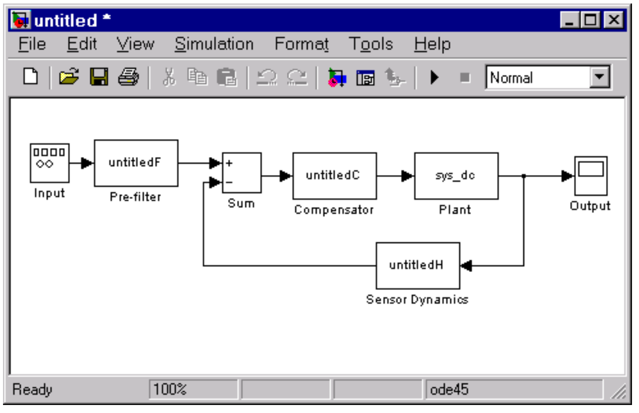
- Zero-order hold
- Tustin
- Tustin with prewarping
- Matched pole/zero

Again, if you choose Tustin with prewarping, you must specify the critical frequency.

Draw Simulink Diagram

Note You must have a license for Simulink to use this feature. If you do not have Simulink, you will not see this option under the **Tools** menu.

Select **Draw Simulink Diagram** to draw a block diagram of your system (plant, compensator, prefilter, and sensor). For the DC motor example described in Getting Started with the Control System Toolbox, this picture is the result.

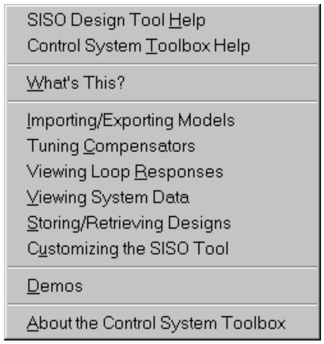


Window

The **Window** menu item lists all window open in MATLAB. The first item is always the MATLAB Command Window. After that, windows you have opened are listed in the order in which you invoked them. Any window you select from the list become the active window.

Help

Help brings you to various places in the Control System Toolbox help system. This figure shows the menu.



Each topics takes you to brief discussions of basic information about the SISO Design Tool and the Control System Toolbox:

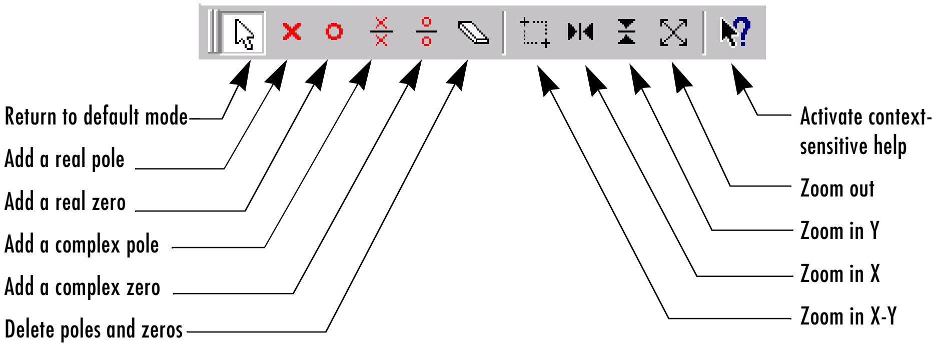
- **SISO Design Tool Help** — An overview of the SISO Design Tool
- **Control System Toolbox Help** — A roadmap for the Control System Toolbox help
- **What's This?** — Activates the "What's This?" cursor, which appears as a question mark. Click in various regions of the SISO Design Tool to see brief descriptions of the tool's features.
- **Importing/Exporting Models** — How to import models into the SISO Design Tool and how to export completed designs
- **Tuning Compensators** — Basic information about adjusting gains and adding dynamics to your prefilter (**F**) and compensator (**C**)
- **Viewing Loop Responses** — How to open an LTI Viewer containing loop responses for your system. Many response types are available.
- **Viewing System Data** — How to see information about your model
- **Storing/Retrieving Designs** — How to store and retrieve designed systems
- **Customizing the SISO Tool** — How to open the SISO Tool Preferences editor, which allows you to customize plot displays in the tool
- **Demos** — A link to the Control System Toolbox demos
- **About the Control System Toolbox** — The version number of your Control System Toolbox

Toolbar

The toolbar performs the following operations:

- Add and delete real and complex poles and zeros
- Zoom in and out
- Invoke the SISO Design Tool's context-sensitive help

This picture shows the toolbar.



Options Available from the Toolbar

You can use the tool tips feature to find out what a particular icon does. Just place your mouse over the icon in question, and you will see a brief description of what it does.

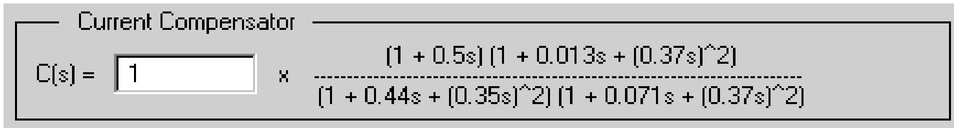
Once you've selected an icon, your mouse stays in that mode until you click the icon again.

You can reach all of these options from two other places:

- Right-click menus
- From **Root Locus**, **Open-Loop Bode**, **Open-Loop Nichols**, or **Prefilter Bode** under **Edit** in the menu bar (these replicate the right-click menus for each of these views). Note that the **Edit** menu adjusts the options to match the views that you have open. For example, if you have the root locus open alone, you will only see the **Root Locus** option.

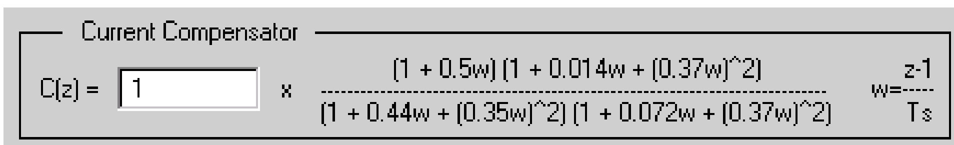
Current Compensator

The **Current Compensator** panel shows the structure of the compensator you are designing. The default compensator structure is a unity gain with no dynamics. Once you add poles and/or zeros, the Current Compensator panel displays the compensator in zero/pole/gain format. This picture shows a Current Compensator panel with `Gcl1` entered as the compensator.



You can change the gain of the compensator by changing the number in the text field. If you want to change the poles and zeros of the compensator, click on the window to open the Edit Compensator window.

If you have a discrete time system, the Current Compensator panel display changes. This figure shows the Current Compensator panel with `Gcl1` discretized with a time step of 0.001 second.



Here, *w* is the *z*-transform shifted by -1 and scaled by the sample time; see the definition to the right of the transfer function. This is done to simplify the representation; note that the coefficients are a close match to those shown for the continuous time representation.

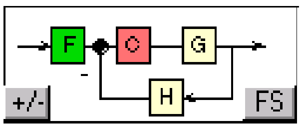
If you see either `NumC` or `DenC` in place of a polynomial, it means that the numerator or denominator of the transfer function is too large to fit in the panel. Try stretching the SISO Design Tool horizontally to see the complete transfer function.

Feedback Structure

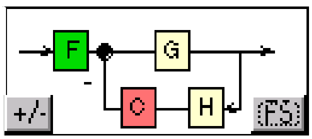
The **Feedback Structure** panel displays the current configuration of these components:

- Compensator (**C**)
- Prefilter (**F**)
- Plant (**G**)
- Sensor (**H**)

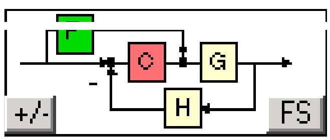
The default configuration is shown below.



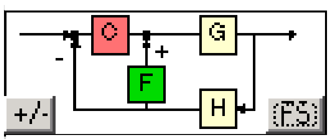
To cycle through the feedback structures, click the **FS** button. This figure shows the alternate feedback structures.



Compensator in the feedback path



Feedforward prefilter



Prefilter in the feedback path with positive feedback

Clicking the **+/-** button toggles between positive and negative feedback signs. Negative feedback is the default.

Additional Features

Left-click on the **G** or **H** boxes to open the System Data window. Click on **F** or **C** to open the **Edit Compensator** window for the prefilter or compensator, respectively.

Right-Click Menus

The SISO Design Tool provides right-click menus for all the views available in the tool. These views include the root-locus, open-loop Bode diagrams, Nichols plot, and the prefilter Bode diagrams. The menu items in each of these views are identical. The design constraints, however, differ, depending on which view you are accessing the menus from.

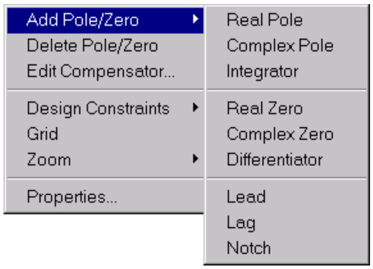
You can use the right-click menu to design a compensator by adding poles, zeros, lead, lag, and notch filters. In addition, you can use this menu to add grids and zoom in on selected regions. Also, you can open each view's **Property Editor** to customize units and other elements of the display.

Note Click on items in the right-click menu pictured below to get help contents.

.

Add

The **Add** menu options give you the ability to add dynamics to your compensator design, including poles, zeros, lead and lag networks, and notch filters. This figure shows the **Add** submenu.



The following pole/zero configurations are available:

- **Real Pole**
- **Complex Pole**
- **Integrator**
- **Real Zero**
- **Complex Zero**
- **Differentiator**
- **Lead**
- **Lag**
- **Notch**

In all but the integrator and differentiator, once you select the configuration, your cursor changes to an 'x'. To add the item to your compensator design, place the x at the desired location on the plot and left-click your mouse. You will see the root locus design automatically update to include the new compensator dynamics.

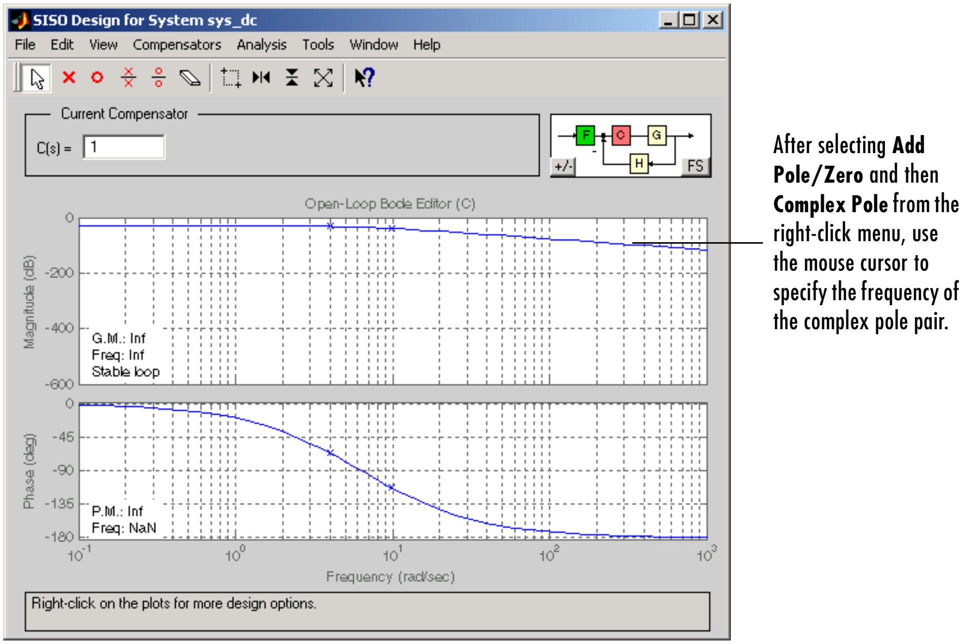
The notch filter has three adjustable parameters. For a discussion about how to add and adjust notch filters, see "Adding a Notch Filter" in *Getting Started with the Control System Toolbox*.

Example: Adding a Complex Pair of Poles

This example shows you how to add a complex pair of poles to the open-loop Bode diagram. First, type

```
load ltiexamples
sisotool('bode',sys_dc)
```

at the MATLAB prompt. This opens the SISO Design Tool with the DC motor example loaded and the open-loop Bode diagram displayed.



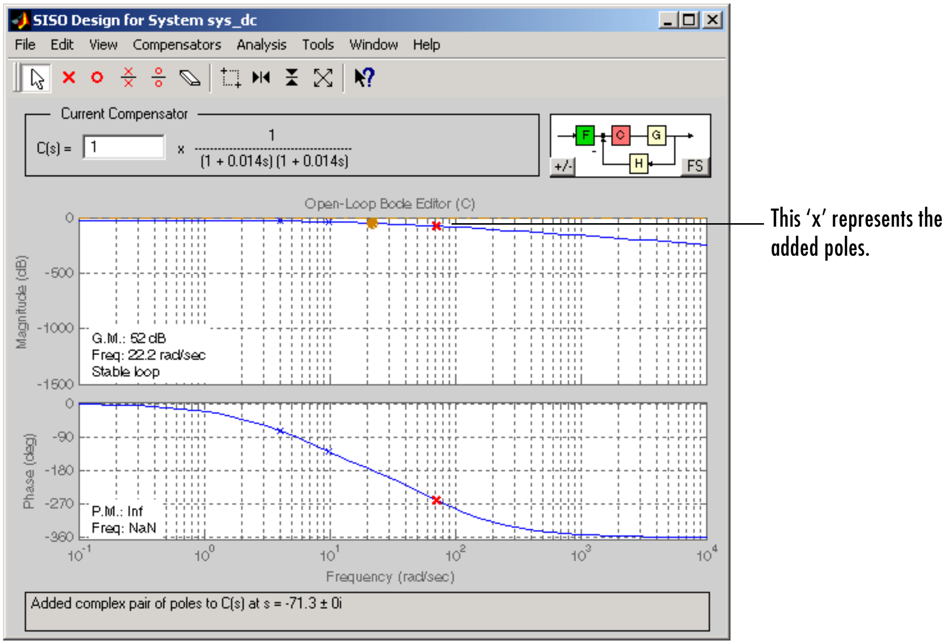
To add a complex pair of poles:

1 Select **Add Pole/Zero** and then **Complex Pole** from the right-click menu

2 Place the mouse cursor where you want the pole to be located

3 Left-click to add the pole

Your SISO Design Tool should look similar to this.



In the case of Bode diagrams, when you place a complex pole, the default damping value is 1, which means you have a double real pole. To change the damping, grab the red 'x' by left-clicking on it and drag it upward with your mouse. You will see damping ratio change in the Status Panel at the bottom of the SISO Design Tool.

Delete Pole/Zero

Select **Delete Pole/Zero** to delete poles and zeros from your compensator design. When you make this selection, your cursor changes to an eraser. Place the eraser over the pole or zero you want to delete and left-click your mouse.

Note the following:

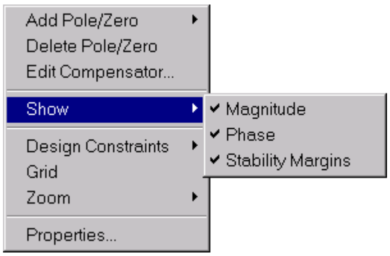
- You can only delete compensator poles and zeros. Plant (**G** in the feedback structure panel) poles and zeros cannot be altered.
- If you delete one of a pair of poles or zeros, the other member of the pair is also removed.

Edit Compensator

Edit Compensator opens the **Edit Compensator C** or **F** window, depending on which compensator you're working with. You can use this window to adjust the compensator gain and add or remove compensator poles and zeros from your compensator (**C**) or prefilter (**F**) design. See "Edit" on page 1-14 for a discussion of this window.

Show

Use **Show** to select/deselect the display of characteristics relevant to which view you are working with. This figure displays the Show submenu for the open-loop Bode diagram.

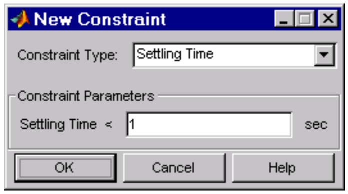


For this particular view, the options available are magnitude, phase, and stability margins. Selecting any of these toggles between showing and hiding the feature. A check next to the feature means that it is currently displayed on the Bode diagram plots. Although the characteristics are different for each view in the SISO Design Tool, they all toggle on and off in the same manner.

Design Constraints

When designing compensators, it is common to have design specifications that call for specific settling times, damping ratios, and other characteristics. The SISO Design Tool provides design constraints that can help make the task of meeting design specifications easier. The **New Constraint** window, which allows you to create design constraints, automatically changes to reflect which constraints are available for the view in which you are working. Select **Design**

Constraints and then **New** to open the **New Constraint** window, which is shown below.



Since each view has a different set of constraint types, click on the following links to go to the appropriate descriptions:

- Root locus
- Open-loop Bode diagram and prefilter Bode diagram (same)
- Nichols plot

Design Constraints for the Root Locus

For the root locus, you have the following constraint types:

- "Settling Time"
- "Percent Overshoot"
- "Damping Ratio"
- "Natural Frequency"

Use the Constraint Type menu to select a design constraint. In each case, to specify the constraint, enter the value in the Constraint Parameters panel. You can select any or all of them, or have more than one of each.

Settling Time. If you specify a settling time in the continuous-time root locus, a vertical line appears on the root locus plot at the pole locations associated with the value provided (using a first-order approximation). In the discrete-time case, the constraint is a curved line.

Percent Overshoot. Specifying percent overshoot in the continuous-time root locus causes two rays, starting at the root locus origin, to appear. These rays are the locus of poles associated with the percent value (using a second-order approximation). In the discrete-time case, In the discrete-time case, the constraint appears as two curves originating at (1,0) and meeting on the real axis in the left-hand plane.

Note that the percent overshoot (p.o.) constraint can be expressed in terms of the damping ratio, as in this equation.

$$p.o. = 100e^{-\pi\zeta/\sqrt{1-\zeta^2}}$$

where ζ is the damping ratio.

Damping Ratio. Specifying a damping ratio in the continuous-time root locus causes two rays, starting at the root locus origin, to appear. These rays are the locus of poles associated with the damping ratio. In the discrete-time case, the constraint appears as curved lines originating at (1,0) and meeting on the real axis in the left-hand plane.

Natural Frequency. If you specify a natural frequency, a semicircle centered around the root locus origin appears. The radius equals the natural frequency.

Example: Adding Damping Ratio Constraints

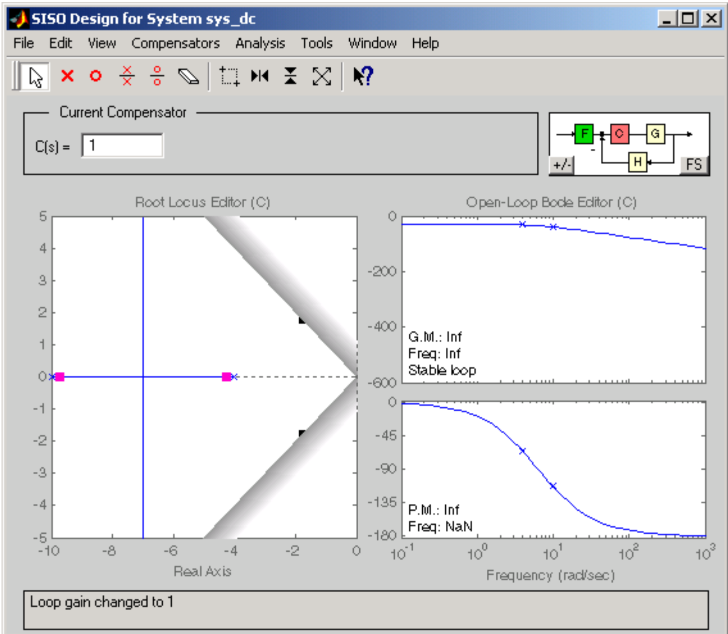
This example add a damping ratio of 0.707 inequality constraint. First, type

```
load ltiexamples
sisotool(sys_dc)
```

at the MATLAB prompt. This opens the SISO Design Tool with the DC motor example imported.

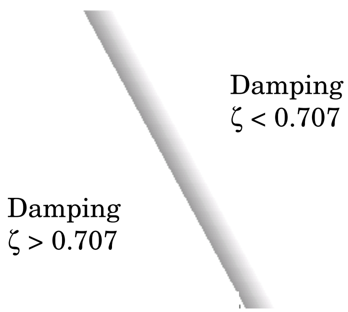
From the root locus right-click menu, select **Design Constraints** and then **New** to open the **New Constraint** window. To add the constraint, select **Damping**

Ratio as the constraint type. The default damping ratio is 0.707. The SISO Design Tool should now look similar to this figure.



Damping Ratio Constraints in the Root Locus

The two rays centered at (0,0) represent the damping ratio constraint. The dark edge is the region boundary, and the shaded area outlines the exclusion region. This figure explains what this means for this constraint.



You can, for example, use this design constraint to ensure that the closed-loop poles, represented by the red squares, have some minimum damping. Try adjusting the gain until the damping ratio of the closed-loop poles is 0.7.

Design Constraints for Open-Loop and Prefilter Bode Diagrams

For both the open-loop and prefilter Bode diagrams, you have the following options:

- “Upper Gain Limit”
- “Lower Gain Limit”

Specifying any of these constraint types causes lines to appear in the Bode magnitude curve. To specify an upper or lower gain limit, enter the frequency range, the magnitude limit, and/or the slope in decibels per decade, in the appropriate fields of the Constraint Parameters panel. You can have as many gain limit constraints as you like in your Bode magnitude plots.

Upper Gain Limit. You can specify an upper gain limit, which appears as a straight line on the Bode magnitude curve. You must select frequency limits, the upper gain limit in decibels, and the slope in dB/decade.

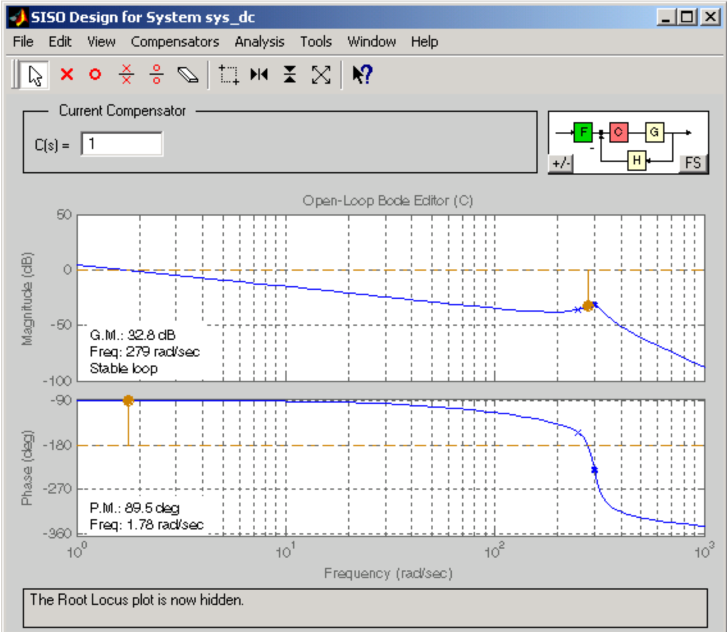
Lower Gain Limit. Specify the lower gain limit in the same fashion as the upper gain limit.

Example: Adding Upper Gain Limits

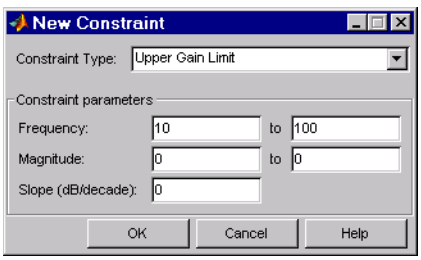
This example shows you how to add two upper gain limit constraints to the open-loop Bode diagram. First, type

```
load ltiexamples
sisotool('bode',Gservo)
```

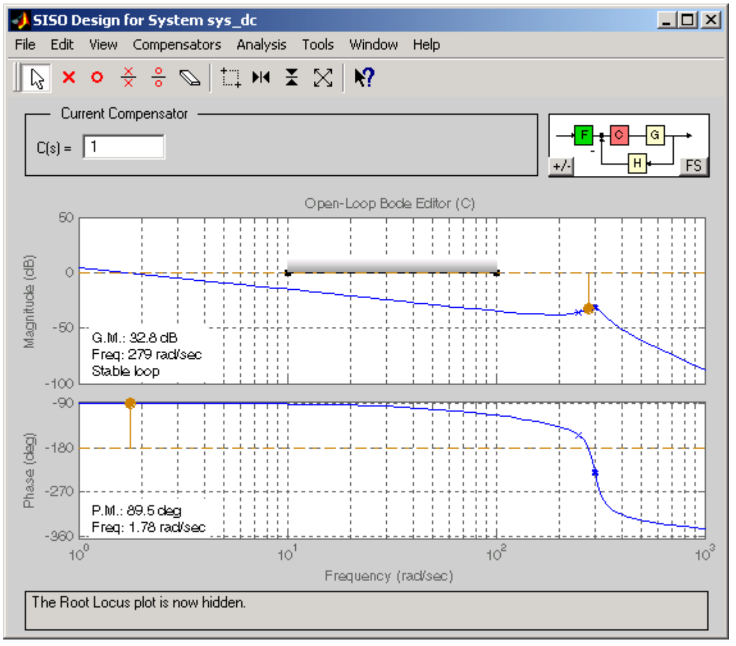
at the MATLAB prompt. This opens the SISO Design Tool with the servomechanism model loaded. Use the right-click menu to add a grid.



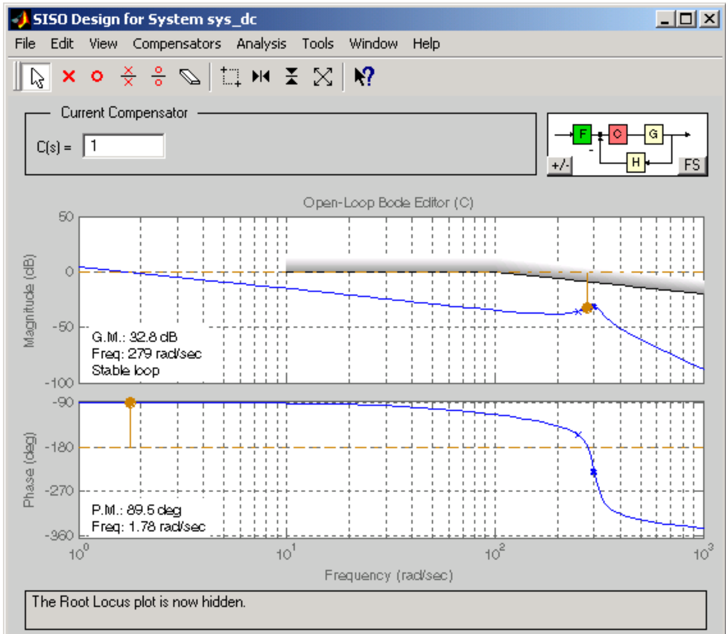
First, add an upper gain limit constraint of 0 dB from 10 rad/sec to 100 rad/sec. This figure shows the **New Constraint** editor with the correct parameters.



Your SISO Design Tool should now look like this.



Now, to constraint the roll off, open the **New Constraint** editor and add an upper gain limit from 100 rad/sec to 1000 rad/sec with a slope of -20 db/decade. This figure shows the result.



With these constraints in place, you can see how much you can increase the compensator gain and still meet design specifications.

Note that you can change the constraints by moving them with your mouse. See “Editing Constraints” on page 1-42 for more information.

Design Constraints for Open-Loop Nichols Plots

For open-loop Nichols plots, you have the following design constraint options:

- “Phase Margin”
- “Gain Margin”
- “Closed-Loop Peak Gain”

Specifying any of these constraint types causes lines or curves to appear in the Nichols plot. In each case, to specify the constraint, enter the value in the

Constraint Parameters panel. You can select any or all of them, or have more than one of each.

Phase Margin. Specify a minimum phase amount at a given location. For example, you can require a minimum of 30 degrees at the -180 degree crossover. The phase margin specified should be a number greater than 0. The location must be a -180 plus a multiple of 360 degrees. If you enter an invalid location point, the closed valid location is selected.

Gain Margin. Specify a gain margin at a given location. For example, you can require a minimum of 20 dB at the -180 degree crossover. The location must be -180 plus a multiple of 360 degrees. If you enter an invalid location point, the closed valid location is selected.

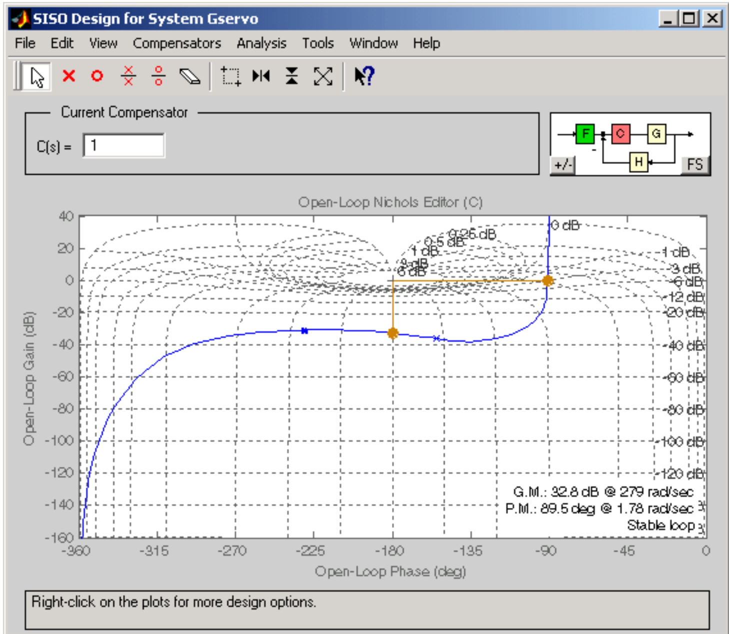
Closed-Loop Peak Gain. Specify a peak closed-loop gain at a given location. The specified value can be positive or negative in dB. The constraint follows the curves of the Nichols plot grid, so it is recommended that you have the grid on when using this feature.

Example: Adding a Closed-Loop Peak Gain Constraint

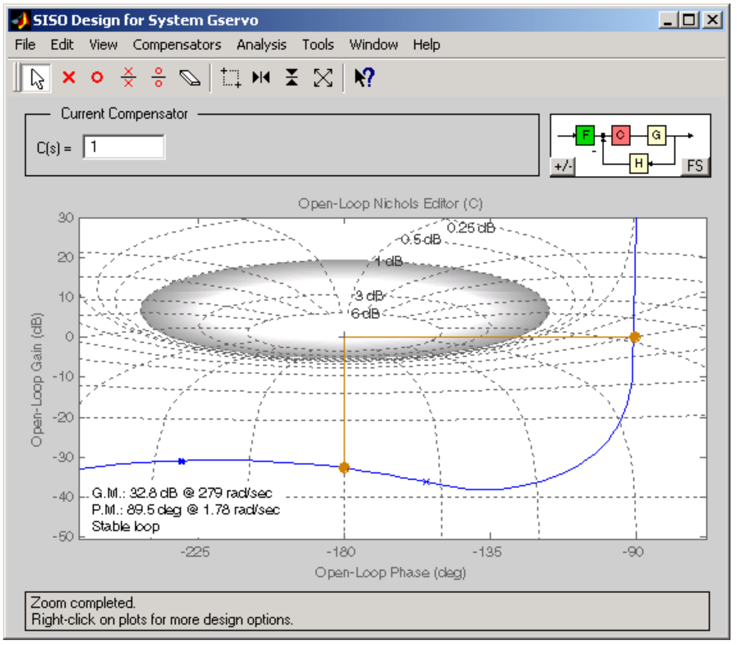
This example shows how to add a closed-loop peak gain constraint to the Nichols plot. First, type

```
load ltiexamples
sisotool('nichols',Gservo)
```

This opens the SISO Design Tool with `Gservo` imported as the plant. Use the right-click menu to add a grid, as this figure shows.



To add closed-loop peak gain of 1 dB at -180 degrees, open the **New Constraint** editor and select **Closed-Loop Peak Gain** from the pull-down menu. Set the peak gain field to 1 dB. The figure shows the resulting design constraint; use Zoom X-Y to zoom in on the plot for clarity.



As long as the curve is outside of the grey region, the closed-loop gain is guaranteed to be less than 1 dB. Note that this is equivalent, up to second order, to specifying the peak overshoot in the time domain. In this case, a 1 dB closed-loop peak gain corresponds to an overshoot of 15%.

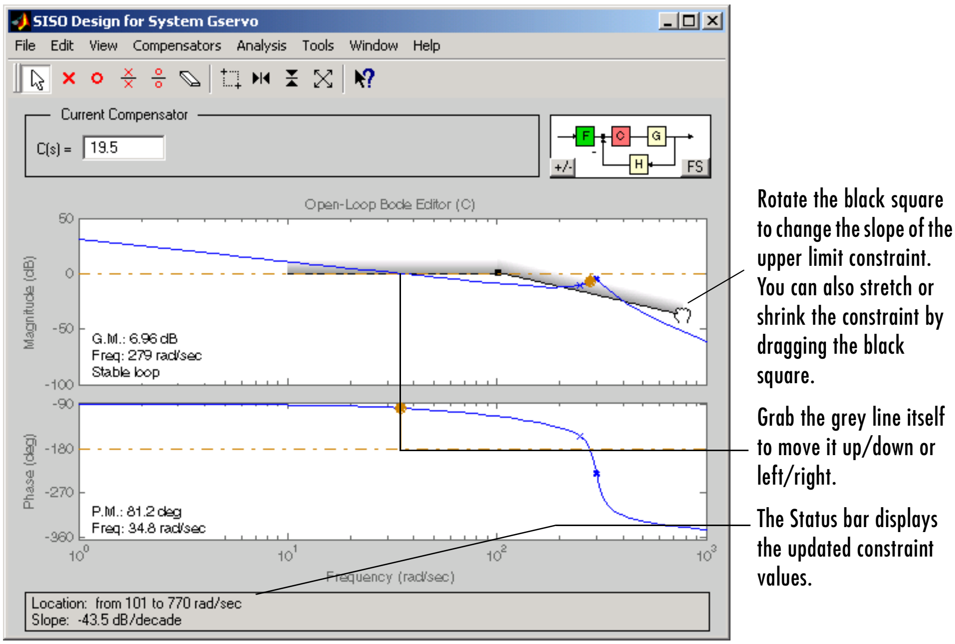
Editing Constraints

To edit an existing constraint, left-click on the constraint itself to select it. Two black squares appear on the constraint when it is selected, and your mouse cursor turns into a large black cross (+). In general, there are two ways to adjust a constraint:

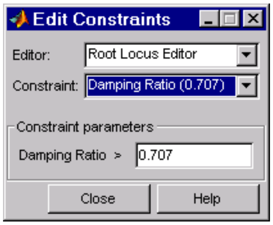
- Click on the constraint and drag it. This does not change the shape of the constraint. That is, the adjustment is strictly a translation of the constraint.
- Grab a black square and drag it. In this case, you can rotate, expand, and/or contract the constraint.

For example, in Bode diagrams you can move an upper gain limit by clicking on it and moving it anywhere in the plot region. As long as you haven't grabbed a black square, the length and slope of the gain limit will not change as you move the line. On the other hand, you can change the slope of the upper gain limit by grabbing one of the black squares and rotating the line. In all cases, the Status panel at the bottom of the SISO Design Tool displays the constraint values as they change.

This figure shows the process of editing an upper gain limit in the open-loop Bode diagram.



An alternative way to adjust a constraint is to select **Design Constraints** and then **Edit** from the right-click menu. The **Edit Constraints** window opens.



To adjust a constraint, select the constraint by clicking on it and change the values in the fields of the Constraint parameters panel. If you have additional constraints in, for example, the Bode diagram, you can edit them directly from this window by selecting **Open-Loop Bode** from the **Editor** menu.

Deleting Constraints

To delete a constraint, place your cursor directly over the constraint itself. You cursor changes into a large 'x'. Right-click to open a menu containing **Edit** and **Delete**. Select **Delete** from the menu list; this eliminates the constraints. You can also delete constraints by left-clicking on the constraint and then pressing the **BackSpace** or **Delete** key on your keyboard.

Finally, you can delete constraints by selecting **Undo Add Constraint** from the **Edit** menu, or pressing **Ctrl+Z** if adding constraints was the last action you took.

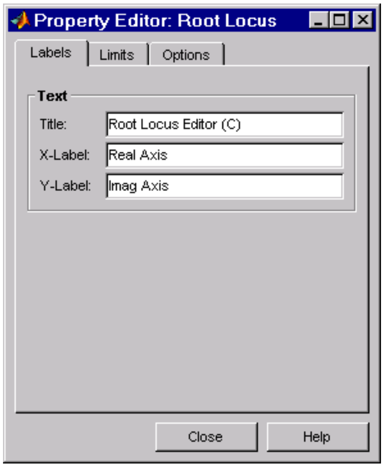
Grid

Grid adds a grid to the selected plot.

Properties

Properties opens the **Property Editor**, which is a GUI for customizing root locus, Bode diagrams, and Nichols plots inside the SISO Design Tool. The Property Editor automatically reconfigures as you select among the different plots open.

This picture shows the open window for the root locus.



You can use this window to change titles and axis labels, reset axes limits, add grid lines, and change the aspect ratio of the plot. For a complete discussion of the **Property Editor**, see "Customizing Plots Inside the SISO Design Tool" online in the Control System Toolbox documentation.

Note that you can also activate this menu by double-clicking anywhere in the root locus away from the curve.

The are only three pages in the Property Editor: Labels, Limits, and Options. The configuration of each page differs, depending on whether you're working with the root-locus, Bode diagrams, or the open-loop Nichols plot. Click the **Help** button on the Property Editor you have open to view information specific to that editor, or click on the links below:

- Root locus
- Bode diagram
- Nichols plot.

Status Panel

The Status panel is located at the bottom of the SISO Design Tool. It displays the most recent action you have performed, occasionally provides advice on how to use the SISO Design Tool, and tracks key parameters when moving objects in the design views.

LTI Viewer

The LTI Viewer is a graphical user interface (GUI) that supports ten plot responses, including step, impulse, Bode, Nyquist, Nichols, zero/pole, sigma (singular values), `lsim`, and `initial` plots. The latter two are only available at the initialization of the LTI Viewer; see `ltiview` for more information.

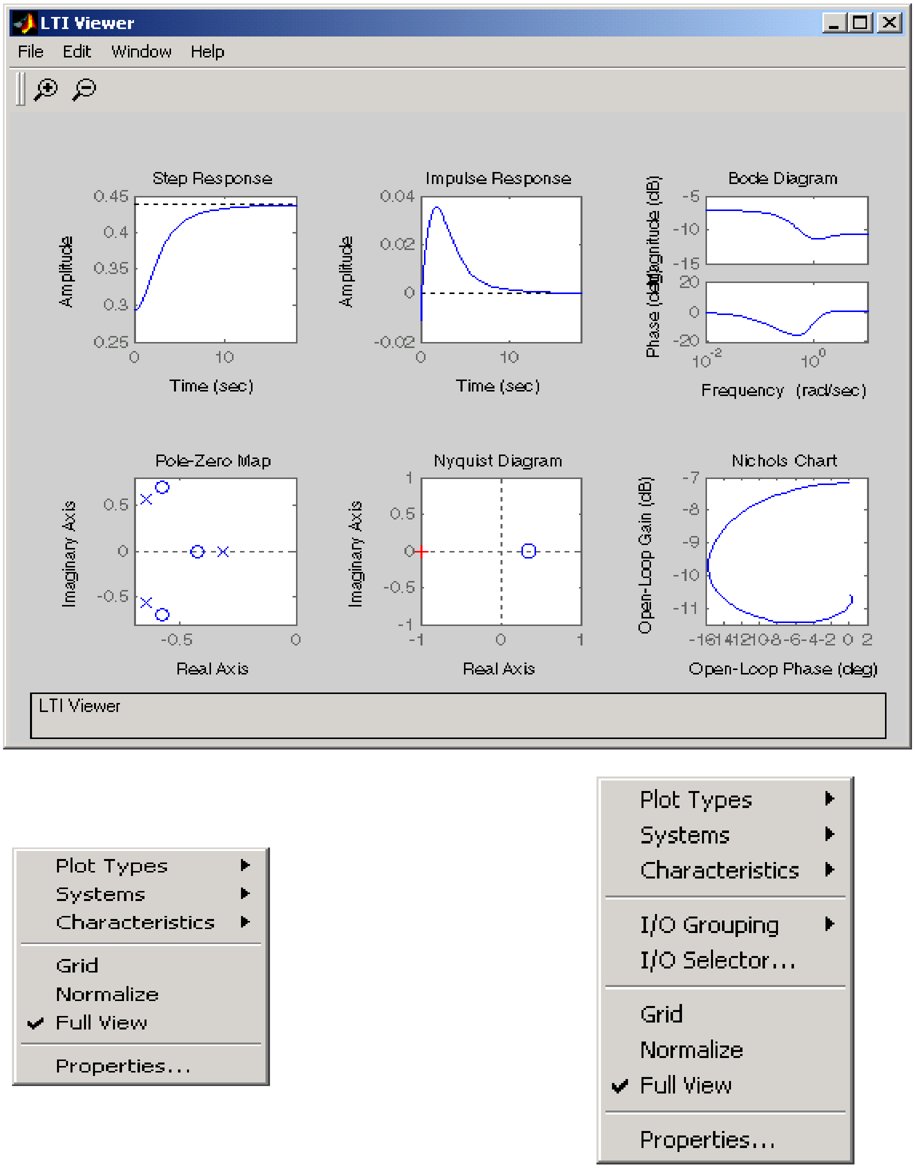
The LTI Viewer is configurable and can display up to six plot type and any number of models in a single viewer. In addition, you can display information specific to the response plots, such as peak response, gain and phase margins, and so on.

You can open the LTI Viewer by typing

```
ltiview
```

at the MATLAB prompt. You can also open an LTI Viewer from the SISO Design Tool; see “SISO Design Tool” on page 1-1 for more information.

Note Click on any of the plots of the LTI Viewer, shown below, to get help on selecting characteristics for the plot. Click on the menu bar to get help on its contents. Click on the right-click menus, also shown below, to get help on right-click menu features.



The LTI Viewer and Right-Click Menus for SISO and MIMO/LTI Array Models.

LTI Viewer Menu Bar

Note Click on **File**, **Edit**, **Window**, or **Help** on the menu bar pictured below to get help on the menu items.

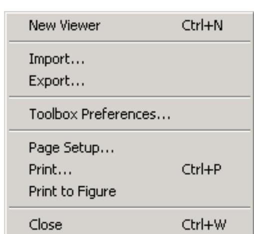
This picture shows the LTI Viewer menu bar.

Tasks that you can perform using the LTI Viewer menu bar include:

- Importing and exporting models
- Printing plot responses
- Reconfiguring the Viewer (add or remove plot responses)
- Displaying critical values (peak responses, etc.) and markers on each plot

File

Note Click on any of the items listed in the **File** menu pictured below to get help contents.



You can use the **File** menu to do the following:

- Open a new LTI Viewer

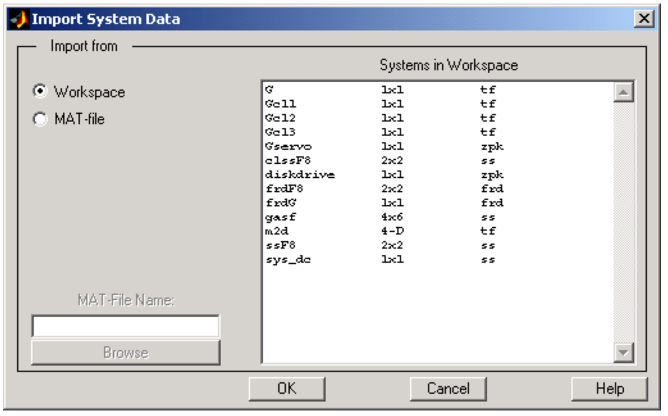
- Import and export models
- Set plot preferences for all the plots generated by the Control System Toolbox
- Print response plots
- Close the LTI Viewer

New Viewer

Select this option to open a new LTI Viewer.

Import Using the Import System Data Window

Import in the **File** menu opens the **Import System Data** window.



You can use the **LTI Browser** to import LTI models into the LTI Viewer.

To import a model

- Click on the desired model in the LTI Browser List. To perform multiple selections:

  a Hold the **Control** key and click on the names of nonadjacent models.

  b Hold the **Shift** key while clicking, to select a set of adjacent models.
- Click the **OK** or **Apply** Button

Note that models must have identical numbers of inputs and outputs to be imported into a single LTI Viewer.

For importing, the LTI Browser lists only the LTI models in the main MATLAB workspace.

Export Using the LTI Viewer Export Window

Export in the **File** menu opens the **LTI Viewer Export** window.

The LTI Viewer Export window lists all the models with responses currently displayed in your LTI Viewer. You can export models back to the MATLAB workspace or to disk. In the latter case, the Control System Toolbox saves the files as MAT-files.

If you select **Export to Disk**, this window appears.

Choose a name for your model(s) and click **Save**. Your models are stored in a MAT-file.

Toolbox Preferences

Select **Toolbox Preferences** to open the Toolbox Preferences editor, which sets preferences for all response objects in the Control System Toolbox, including the viewer.

Page Setup and Print

Page Setup opens a GUI with selections for page layout, etc. **Print** sends the entire LTI Viewer window to your printer.

Print to Figure

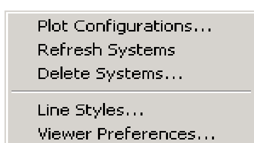
Print to Figure sends a picture of the selected system to a new figure window. Note that this new figure is a MATLAB figure window and not an LTI Viewer.

Close

Close closes the LTI Viewer.

Edit

Note Click on any of the items listed in the **Edit** menu pictured below to get help contents.



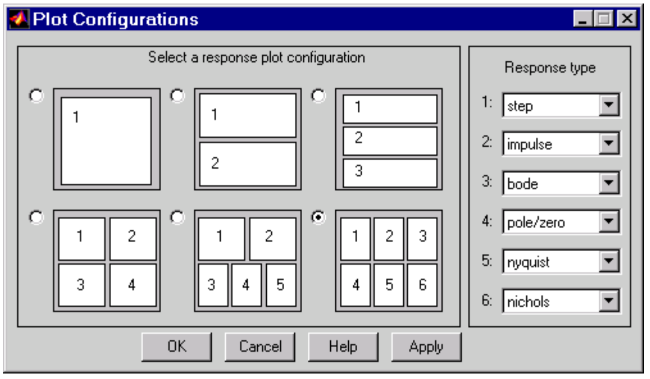
The **Edit** menu contains the following options:

- Plot Configurations — Opens the Plot Configurations window
- Refresh Systems — Updates imported systems
- **Delete** opens the LTI Viewer Delete window

- Line Styles — Opens the Line Styles editor
- Viewer Preferences — Opens the Viewer Preferences editor

Plot Configurations Window — Selecting Response Types

Plot Configuration under the **Edit** menu opens the **Plot Configurations** window.



Use this window to select the number and kind of response plots you want in a single instance of the LTI Viewer. You can plot up to six response plots in a single viewer. Click the radio button to the upper left of the configuration you want the viewer to use.

You can select among eight response types for each plot in the viewer. These are the available response types:

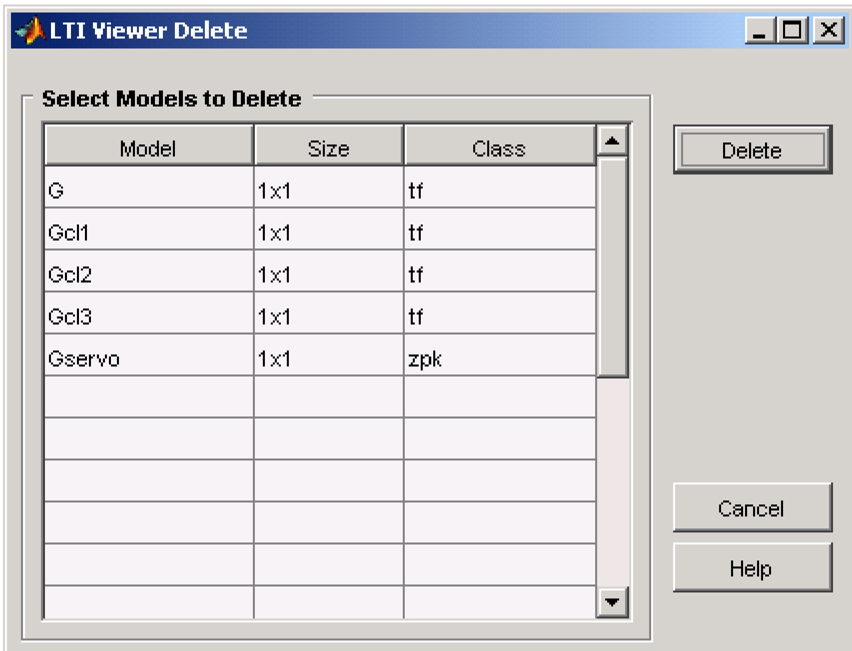
- Step
- Impulse
- Bode — Plots the Bode magnitude and phase
- Bode mag. — Plots the Bode magnitude only
- Nyquist
- Nichols
- Singular Values
- Pole/Zero map
- I/O pole/zero map

Refresh Systems

Refresh updates imported models to reflect any changes made in the MATLAB workspace since you imported them.

Delete Systems

Delete under **Systems** in the **Edit** menu opens the **LTI Viewer Delete** window

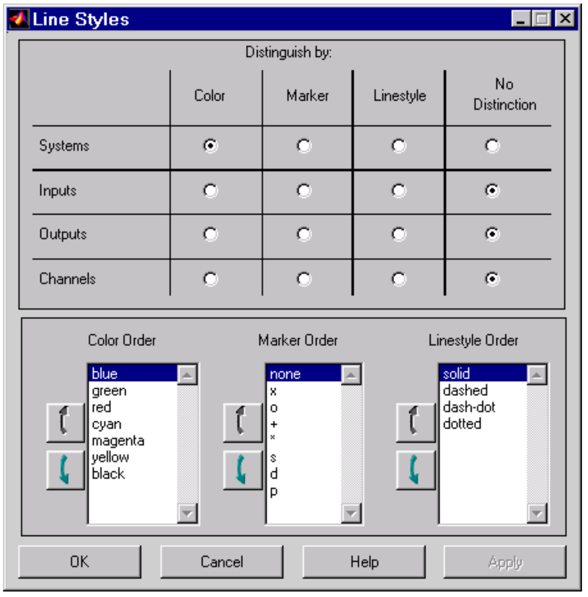


To delete a model

- Click on the desired model in the Model ist. To perform multiple selections:
  - **a** Click and drag over several variables in the list.
  - **b** Hold the Control key and click on individual variables.
  - **c** Hold the Shift key while clicking, to select a range.
- Click the **OK** or **Apply** Button

Line Styles Editor

Select **Line Styles** under the **Edit** menu to open the **Line Styles** editor.



The **Line Styles** editor is particularly useful when you have multiple systems imported. You can use it change line colors, add and rearrange markers, and alter line styes (solid, dashed, and so on).

The **Linestyle Preferences** window allows you to customize the appearance of the response plots by specifying:

- The line property used to distinguish different systems, inputs, or outputs
- The order in which these line properties are applied

Each LTI Viewer has its own **Linestyle Preferences** window.

Setting Preferences. You can use the "Distinguish by" matrix to specify the line property that will vary throughout the response plots. You can group multiple plot curves by systems, inputs, outputs, or channels (individual input/output relationships). Note that the Line Styles editor uses radio buttons, which means that you can only assign one property setting for each grouping (system, input, etc.).

Ordering Properties. The Order field allows you to change the default property order used when applying the different line properties. You can reorder the colors, markers, and linestyles (e.g., solid or dashed).

To change any of the property orders, click the up or down arrow button to the left of the associated property list to move the selected property up or down in the list

Viewer Preferences

Viewer Preferences opens the LTI Viewer Preferences editor, which you can use to set response plot defaults for the LTI Viewer that is currently open.

For a complete description of the LTI Viewer Preference editor, as well as all the property and preference editors available in the Control System Toolbox, see “Customization” in the online Control System Toolbox documentation. To go directly to the LTI Viewer Preferences editor documentation, see “LTI Viewer Preferences” in the same document.

Window

Use the **Window** menu to select which of your MATLAB windows is active. This menu lists any window associated with MATLAB and the Control System Toolbox. The MATLAB Command Window is always listed first.

Help

The **Help** menu links to this help file.

LTI Viewer Toolbar

This figure shows the LTI Viewer Toolbar.

From left to right:

- Click the paper icon to open a new LTI Viewer
- Click the printer icon to print the contents of the LTI Viewer
- Click the magnifying glass icons and then click anywhere in a plot region to zoom in and out

Right-Click Menu for SISO Systems

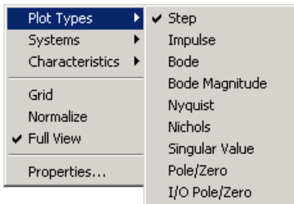
Note Click on items in the right-click menu pictured below for help contents.

This right-click menu appears when you have a SISO system imported into your LTI Viewer. If you have a MIMO system, or an LTI array containing multiple models, there are additional menu options. See "Right-Click Menus for MIMO Systems and LTI Arrays" on page 2-20 for more information.

You can use the right-click menus to perform the following tasks:

- Change the plot type in the viewer
- Select and deselect imported models for display
- Add or remove grid lines
- Normalize a view
- Go to a full view
- Open the Property Editor

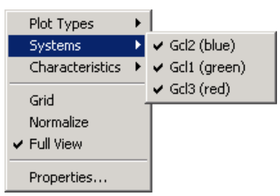
Plot Type



Select which plot type you want to display. The LTI Viewer shows a check to mark which plot is currently displayed. These are the available options:

- **Step** — Step response
- **Impulse** — Impulse response
- **Bode** — Magnitude and phase plots
- **Bode Mag.** — Magnitude only
- **Nyquist** — Nyquist diagram
- **Nichols** — Nichols chart
- **Singular Values** — Singular values plot
- **Pole/Zero** — Pole/Zero map
- **I/O Pole/Zero** — Pole/Zero map for I/O pairs

Systems



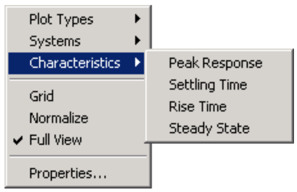
Use **Systems** to select which of the imported systems to display. Selecting a system causes a check mark to appear beside the system. To deselect a system, select it again; the menu toggles between selected and deselected.

Characteristics

The **Characteristics** menu changes for each plot response type. The next sections describe the menu for each of the eight plot types.

Step Response

Step plots the model's response to a step input.

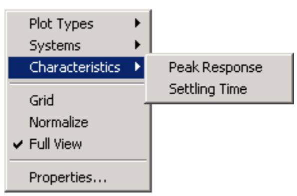


You can display the following information in the step response:

- **Peak Response** — The largest deviation from the steady-state value of the step response
- **Settling Time** — The time required for the step response to decline and stay at 5% of its final value
- **Rise Time** — The time require for the step response to rise from 10% to 90% of its final value
- **Steady-State** — The final value for the step response

Impulse Response

Impulse Response plots the model's response to an impulse.

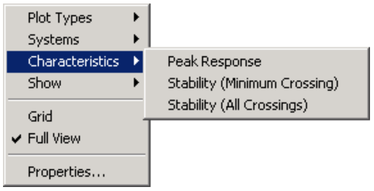


The LTI Viewer can display the following information in the impulse response:

- **Peak Response** — The maximum positive deviation from the steady-state value of the impulse response
- **Settling Time** — The time required for the step response to decline and stay at 5% of its final value

Bode Diagram

Bode plots the open-loop Bode phase and magnitude diagrams for the model.

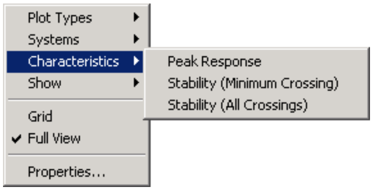


The LTI Viewer can display the following information in the Bode diagram:

- **Peak Response** — The maximum value of the Bode magnitude plot over the specified region
- **Stability Margins (Minimum Crossing)** — The minimum phase and gain margins. The gain margin is defined to the gain (in dB) when the phase first crosses -180°. The phase margin is the distance, in degrees, of the phase from -180° when the gain magnitude is 0 dB.
- **Stability Margins (All Crossings)** — Display all stability margins

Bode Magnitude

Bode Magnitude plots the Bode magnitude diagram for the model.

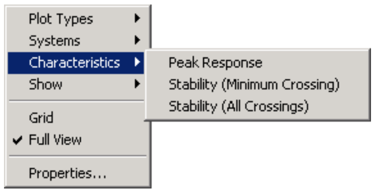


The LTI Viewer can display the following information in the Bode magnitude diagram:

- **Peak Response**, which is the maximum value of the Bode magnitude in decibels (dB), over the specified range of the diagram.
- **Stability (Minimum Crossing)** — The minimum gain margins. The gain margin is defined to the gain (in dB) when the phase first crosses -180°.
- **Stability (All Crossings)** — Display all gain stability margins

Nyquist Diagrams

Nyquist plots the Nyquist diagram for the model.

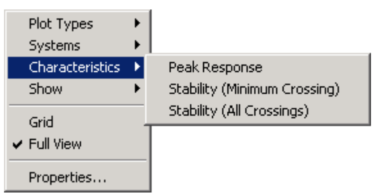


The LTI Viewer can display the following types of information in the Nyquist diagram:

- **Peak Response** — The maximum value of the Nyquist diagram over the specified region
- **Stability (Minimum Crossing)** — The minimum gain and phase margins for the Nyquist diagram. The gain margin is the distance from the origin to the phase crossover of the Nyquist curve. The phase crossover is where the curve meets the real axis. The phase margin is the angle subtended by the real axis and the gain crossover on the circle of radius 1.
- **Stability (All Crossings)** — Display all gain stability margins

Nichols Charts

Nichols plots the Nichols Chart for the model.



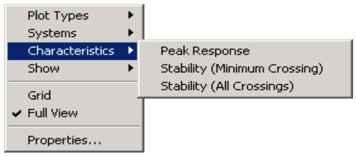
The LTI Viewer can display the following types of information in the Nichols chart:

- **Peak Response** — The maximum value of the Nichols chart in the plotted region.
- **Stability (Minimum Crossing)** — The minimum gain and phase margins for the Nichols chart.

- **Stability (All Crossings)** — Display all gain stability margins

Singular Values

Singular Values plots the singular values for the model.



The LTI Viewer can display the **Peak Response**, which is the largest magnitude of the Singular Values curve over the plotted region.

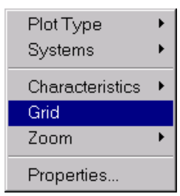
Pole/Zero and I/O Pole/Zero

Pole/Zero plots the poles and zeros of the model with ‘x’ for poles and ‘o’ for zeros. I/O Pole/Zero plots the poles and zeros of I/O pairs.

There are no **Characteristics** available for pole-zero plots.

Grid

The **Grid** command activates a grid appropriate to the plot in the region you select.



Normalize

Select **Normalize** to scale responses to fit the view (only available for time-domain plot types).

Full View

Selecting **Full View** causes the LTI Viewer to scale limits so that the entire curve is visible.

Properties

Use **Properties** to open the Property Editor. This GUI allows you to customize labels, axes limits and units, grids and font styles, and response characteristics (e.g., rise time) for your plot.

For a full description of the Property Editor, see "Customizing Response Plot Properties" online in the Control System Toolbox documentation.

Right-Click Menus for MIMO Systems and LTI Arrays

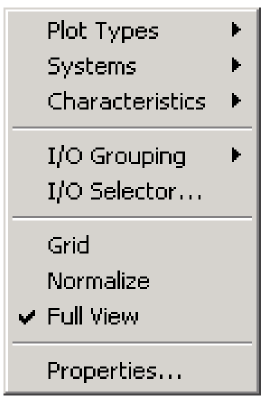
All of the menu options described in Right-Click Menu for SISO Systems hold when you have imported a MIMO model or LTI Array containing multiple models.

Note, however, that when you have a MIMO model or LTI array displayed, the right-click menus contain additional options: **I/O Grouping** and **I/O selector**. These features allow you to quickly reshuffle multiple plots in a single LTI Viewer

---

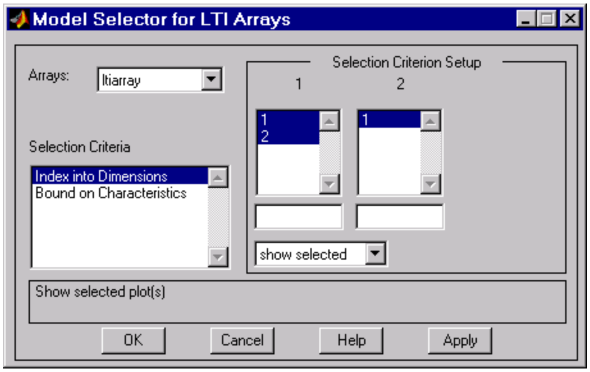
Note Click on items in the right-click menu pictured below to get help contents.

.



Array Selector

If you import an LTI array into your LTI Viewer, **Array Selector** appears as an option in the right-click menu. Selecting this option opens the **Model Selector for LTI Arrays**, shown below.



You can use this window to include or exclude models within the LTI array using various criteria. The following subsections discuss the features in turn.

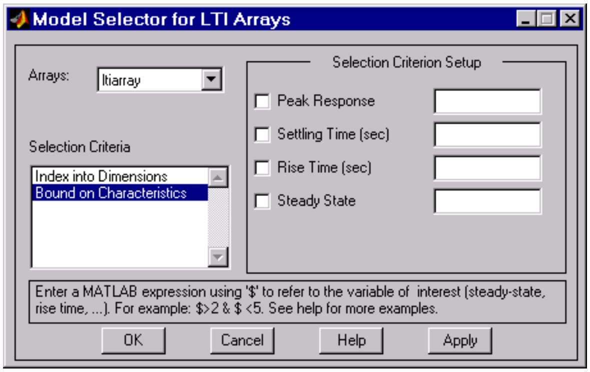
Arrays

Select which LTI array for applying model selection options by using the Arrays pull-down list.

Selection Criteria

There are two selection criteria. The default, **Index into Dimensions**, allows you to include or exclude specified indices of the LTI Array. Select systems from the **Selection Criteria Setup** and specify whether to show or hide the systems using the pull-down menu below the Setup lists.

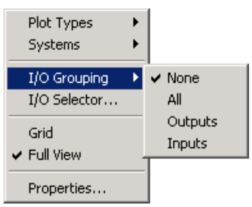
The second criterion is **Bound on Characteristics**. Selecting this options causes the Model Selector to reconfigure. The reconfigured window is shown below.



Use this option to select systems for inclusion or exclusion in your LTI Viewer based on their time response characteristics. The panel directly above the buttons describes how to set the inclusion or exclusion criteria based on which selection criteria you select from the reconfigured **Selection Criteria Setup** panel.

I/O Grouping

You can use **I/O Grouping** to change the grouping of MIMO system plots in your LTI Viewer. This picture shows the menu options.



There are four options:

- **None** — By default, there is no I/O grouping. For example, if you display the step responses for a 3-input, 2- output system, there will be six plots in your LTI Viewer.
- **All** — Groups all the responses into a single plot

- **Inputs** — Groups all the responses by inputs. For example, for a 3-input, 2-output system, selecting Inputs reconfigures the viewer so that there are 3 plots. Each plot contains two curves.
- **Outputs** — Groups all the responses by outputs. For example, for a 3-input, 2-output system, selecting Inputs reconfigures the viewer so that there are 2 plots. Each plot contains three curves.

I/O Selector

I/O Selector opens the **I/O Selector** window, shown below.

The **I/O Selector** window contains buttons corresponding to each I/O pair. In this example, there are 2 inputs and 3 outputs, so there are six buttons. By default, all the I/O pairs are selected. If you click on a button, that I/O pair alone is displayed in the LTI Viewer. The other buttons automatically deselect.

To select a column of inputs, click on the input name above the column. The names are **U(1)**, **U(2)**, and so on. The LTI Viewer displays the responses from the specified input to all the outputs.

To select a row of output, click on the output name to the left of the row. The names are **Y(1)**, **Y(2)**, and so on. The LTI Viewer displays the responses from all the inputs to the specified output.

To reestablish the default setting, click **[all]**. The LTI Viewer displays all the I/O pairs.

Status Panel

The Status Panel is located at the bottom of the LTI Viewer. It contains useful information about changes you have made to the LTI Viewer.

3

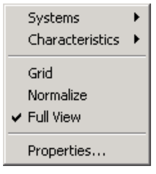
Right-Click Menus for Response Plots

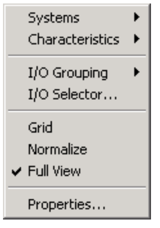
Introduction

All the response plots that the Control System Toolbox creates have right-click menus available. The plots include the following:

- `bode`
- `bodemag`
- `impulse`
- `initial`
- `nichols`
- `nyquist`
- `pzmap`
- `sigma`
- `step`

Note Click on any of the items in the right-click menus, shown below, to get help on the feature.





Right-Click Menus for SISO and MIMO/LTI Array Models.

You can do the following using the right-click menus for response plots:

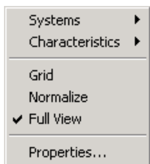
- Select and deselect imported systems
- Change plot characteristics
- Add and remove grid lines
- Zoom in and out of selected plot regions

- Open the Property Editor for the selected plot
- In the MIMO/LTI array case:
  - regroup the plots
  - Select subsets of I/O pairs

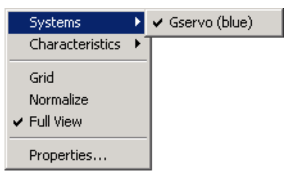
Right-Click Menus for SISO Systems

When you create a response plot for a SISO system, you have available a set of right-click menu options, which are described in the following sections.

Note Click on any of the items in the right-click menus, shown below, to get help on the feature.



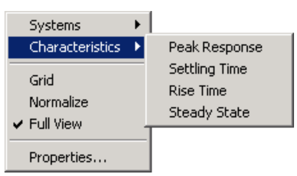
Systems



Use **Systems** to select which of the imported systems to display. Selecting a system causes a check mark to appear beside the system. To deselect a system, select it again; the menu toggles between selected and deselected.

Characteristics

The **Characteristics** menu changes for each plot response type. This picture shows the options for a step response.



The following table lists the characteristics available for each response plot type.

Table 3-1: Options Available from the Characteristics Menu

Function	Characteristics
`bode`	Peak Response
`bodemag`	Peak Response
`impulse`	Peak Response Settling Time
`initial`	Peak Response
`nichols`	Peak Response
`nyquist`	Peak Response
`pzmap`	None
`sigma`	Peak Response
`step`	Peak Response Settling Time Rise Time Steady State

You can find definitions for these characteristics in "Characteristics".

Grid

The **Grid** command activates a grid appropriate to the plot in the region you select.

Normalize

Select **Normalize** to scale responses to fit the view (only available for time-domain plot types).

Full View

Selecting **Full View** causes the response plot to scale limits so that the entire curve is visible.

Properties

Use **Properties** to open the Property Editor. This GUI allows you to customize labels, axes limits and units, grids and font styles, and response characteristics (e.g., rise time) for your plot.

For a full description of the Property Editor, see "Customizing Response Plot Properties" online in the Control System Toolbox documentation.

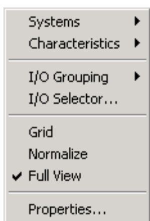
Right-Click Menus for MIMO and LTI Arrays

All of the menu options described in "Right-Click Menus for SISO Systems" on page 3-4 hold when you have generated a response plot for a MIMO model or an LTI Array.

Note, however, that when you have a MIMO model or LTI array displayed, the right-click menus contain additional options: **Axis Grouping** and **I/O selector**. These features allow you to quickly reshuffle multiple plots in a single window.

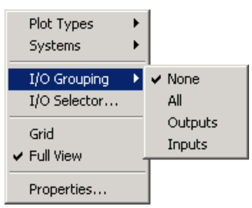
Note Click on items in the right-click menu pictured below to get help contents.

.



I/O Grouping

You can uses **I/O Grouping** to change the grouping of plots in a single plot window. This picture shows the menu options.

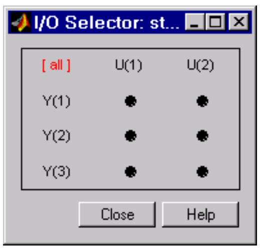


There are four options:

- **None** — By default, there is no axis grouping. For example, if you display the step responses for a 3-input, 2- output system, there will be six plots in your window.
- **All** — Groups all the responses into a single plot
- **Inputs** — Groups all the responses by inputs. For example, for a 3-input, 2-output system, selecting **Inputs** reconfigures the viewer so that there are 3 plots. Each plot contains two curves.
- **Outputs** — Groups all the responses by outputs. For example, for a 3-input, 2-output system, selecting **Outputs** reconfigures the viewer so that there are 2 plots. Each plot contains three curves.

I/O Selector

I/O Selector opens the **I/O Selector** window, shown below.



The **I/O Selector** window contains buttons corresponding to each I/O pair. In this example, there are 2 inputs and 3 outputs, so there are six buttons. By default, all the I/O pairs are selected. If you click on a button, that I/O pair alone is displayed in the plot window. The other buttons automatically deselect.

To select a column of inputs, click on the input name above the column. The names are **U(1)**, **U(2)**, and so on. The plot window displays the responses from the specified input to all the outputs.

To select a row of output, click on the output name to the left of the row. The names are **Y(1)**, **Y(2)**, and so on. The plot window displays the responses from all the inputs to the specified output.

To reestablish the default setting, click **[all]**. The plot window displays all the I/O pairs.

4

Function Reference

Functions -- By Category

LTI Models

Function Name	Description
`drss`	Generate random discrete state-space model
`dss`	Create descriptor state-space model
`filt`	Create discrete filter with DSP convention
`frd`	Create a frequency response data (FRD) model
`frdata`	Retrieve data from an FRD model
`get`	Query LTI model properties
`rss`	Generate random continuous state-space model
`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
`totaldelay`	Provide the aggregate delay for an LTI model
`zpk`	Create zero-pole-gain model
`zpkdata`	Retrieve zero-pole-gain data

Model Characteristics

Function Name	Description
`class`	Display model type (`'tf'`, `'zpk'`, `'ss'`, or `'frd'`)
`hasdelay`	Test true if LTI model has any type of delay
`isa`	Test true if LTI model is of specified type
`isct`	Test true for continuous-time models
`isdt`	Test true for discrete-time models

Function Name	Description
`isempty`	Test true for empty LTI models
`isproper`	Test true for proper LTI models
`issiso`	Test true for SISO models
`ndims`	Display the number of model/array dimensions
`size`	Display output/input/array dimensions

Model Conversions

Function Name	Description
`c2d`	Convert from continuous- to discrete-time models
`chgunits`	Convert the `units` property for FRD models
`d2c`	Convert from discrete- to continuous-time models
`d2d`	Resample discrete-time models
`delay2z`	Convert delays in discrete-time models or FRD models
`frd`	Convert to a frequency response data model
`pade`	Compute the Padé approximation of delays
`reshape`	Change the shape of an LTI array
`residue`	Provide partial fraction expansion
`ss`	Convert to a state space model
`tf`	Convert to a transfer function model
`zpk`	Convert to a zero-pole-gain model

Model Order Reduction

Function Name	Description
`balreal`	Calculate an I/O balanced realization
`minreal`	Calculate minimal realization or eliminate pole/zero pairs
`modred`	Delete states in I/O balanced realization
`sminreal`	Calculate structured model reduction

State-Space Realizations

Function Name	Description
`canon`	Canonical state-space realizations
`ctrb`	Controllability matrix
`ctrbf`	Controllability staircase form
`gram`	Controllability and observability grammians
`obsv`	Observability matrix
`obsvf`	Observability staircase form
`ss2ss`	State coordinate transformation
`ssbal`	Diagonal balancing of state-space realizations

Model Dynamics

Function Name	Description
`bandwidth`	Calculate the bandwidth of SISO models
`damp`	Calculate natural frequency and damping
`dcgain`	Calculate low-frequency (DC) gain
`covar`	Calculate covariance of response to white noise
`dsort`	Sort discrete-time poles by magnitude
`esort`	Sort continuous-time poles by real part

Function Name	Description
`iopzmap`	Plot the pole/zero map for I/O pairs of an LTI model
`norm`	Calculate norms of LTI models (H_2 and L_∞)
`pole`, `eig`	Calculate the poles of an LTI model
`pzmap`	Plot the pole/zero map of an LTI model
`rlocus`	Calculate and plot root locus
`roots`	Calculate roots of polynomial
`sgrid,zgrid`	Superimpose s- and z-plane grids for root locus or pole/zero maps
`zero`	Calculate zeros of an LTI model

Model Interconnections

Function Name	Description
`append`	Append models in a block diagonal configuration
`augstate`	Augment output by appending states
`connect`	Connect the subsystems of a block-diagonal model according to an interconnection scheme of your choice
`feedback`	Calculate the feedback connection of models
`lft`	Form the LFT interconnection (star product)
`ord2`	Generate second-order model
`parallel`	Create a generalized parallel connection
`series`	Create a generalized series connection
`stack`	Stack LTI models into a model array

Time Responses

Function Name	Description
gensig	Generate an input signal
impulse	Calculate and plot impulse response
initial	Calculate and plot initial condition response
lsim	Simulate response of LTI model to arbitrary inputs
ltiview	Open the LTI Viewer for linear response analysis
step	Calculate step response

Time Delays

Function Name	Description
delay2z	Convert delays in discrete-time models or FRD models
pade	Compute the Padé approximation of delays
totaldelay	Provide the aggregate delay for an LTI model

Frequency Response

Function Name	Description
allmargin	Calculate all crossover frequencies and associated gain, phase, and delay margins
bode	Calculate and plot Bode response
bodemag	Calculate and plot Bode magnitude only
evalfr	Evaluate response at single complex frequency
freqresp	Evaluate frequency response for selected frequencies
interp	Interpolate FRD model between frequency points
linspace	Create a vector of evenly spaced frequencies
logspace	Create a vector of logarithmically spaced frequencies

Function Name	Description
ltiview	Open the LTI Viewer for linear response analysis
margin	Calculate gain and phase margins
ngrid	Superimpose grid lines on a Nichols plot
nichols	Calculate Nichols plot
nyquist	Calculate Nyquist plot
sigma	Calculate singular value plot

Pole Placement

Function Name	Description
acker	Calculate SISO pole placement design
place	Calculate MIMO pole placement design
estim	Form state estimator given estimator gain
reg	Form output-feedback compensator given state-feedback and estimator gains

LQG Design

Function Name	Description
lqr	Calculate the LQ-optimal gain for continuous models
dlqr	Calculate the LQ-optimal gain for discrete models
lqry	Calculate the LQ-optimal gain with output weighting
lqrd	Calculate the discrete LQ gain for continuous models
kalman	Calculate the Kalman estimator
kalmd	Calculate the discrete Kalman estimator for continuous models
lqgreg	Form LQG regulator given LQ gain and Kalman filter

Equation Solvers

Function Name	Description
`care`	Solve continuous-time algebraic Riccati equations
`dare`	Solve discrete-time algebraic Riccati equations
`lyap`	Solve continuous-time Lyapunov equations
`dlyap`	Solve discrete-time Lyapunov equations

Graphical User Interfaces for Control System Analysis and Design

Function Name	Description
`ltiview`	Open the LTI Viewer for linear response analysis
`sisotool`	Open the SISO Design GUI

Alphabetical List of Functions

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 (A,b,p)`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). Here `A` is the state transmitter matrix and `b` is the input to state transmission vector.

You can also use `acker` for estimator gain selection by transposing the matrix `A` and substituting `c'` for `b` when $y = cx$ is a single output.

```
l = 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

`lqr`	Optimal LQ regulator
`place`	Pole placement design
`rlocus`	Root locus design

References

[1] Kailath, T., *Linear Systems*, Prentice-Hall, 1980, p. 201.

Purpose Compute all crossover frequencies and corresponding stability margins

Syntax

```
S = allmargin(sys)
```

Description `allmargin` computes the gain, phase, and delay margins and the corresponding crossover frequencies of the SISO open-loop model `sys`. `allmargin` is applicable to any SISO model, including models with delays.

The output `S` is a structure with the following fields:

- GMFrequency — All -180 degree crossover frequencies (in rad/sec)
- GainMargin — Corresponding gain margins, defined as 1/G where G is the gain at crossover
- PMFrequency — All 0 dB crossover frequencies in rad/sec
- PhaseMargin — Corresponding phase margins in degrees
- DMFrequency and DelayMargin — Critical frequencies and the corresponding delay margins. Delay margins are given in seconds for continuous-time systems and multiples of the sample time for discrete-time systems.
- Stable — 1 if the nominal closed-loop system is stable, 0 otherwise.

See Also

`ltimodels`	Help on LTI models
`ltiview`	LTI system viewer
`margin`	Gain and phase margins for SISO open-loop systems

append

Purpose

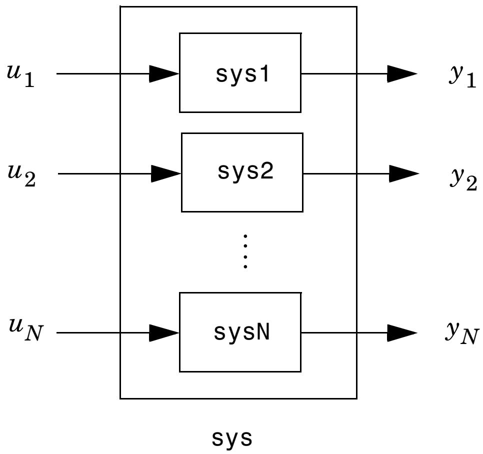
Group LTI models by appending their inputs and outputs

Syntax

```
sys = append(sys1,sys2,...,sysN)
```

Description

`append` appends the inputs and outputs of the LTI models `sys1`,...,`sysN` to form the augmented model sys depicted below.



For systems with transfer functions $H_1(s)$,...,$H_N(s)$, the resulting system `sys` has the block-diagonal transfer function

$$\begin{bmatrix} H_1(s) & 0 & .. & 0 \\ 0 & H_2(s) & . & : \\ : & . & . & 0 \\ 0 & .. & 0 & H_N(s) \end{bmatrix}$$

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 following 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 Precedence Rules 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

`connect`	Modeling of block diagram interconnections
`feedback`	Feedback connection
`parallel`	Parallel connection
`series`	Series connection

Purpose Append the state vector to the output vector

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 TF, ZPK or FRD models.

See Also

`feedback`	Feedback connection
`parallel`	Parallel connection
`series`	Series connection

balreal

Purpose

Input/output balancing of state-space realizations

Syntax

```
sysb = balreal(sys)
[sysb,g,T,Ti] = balreal(sys)
```

Description

`sysb = balreal(sys)` produces a balanced realization `sysb` of the LTI model `sys` with equal and diagonal controllability and observability grammians (see `gram` for a definition of grammian). `balreal` handles both continuous and discrete systems. If `sys` is not a state-space model, it is first and automatically converted to state space using `ss`.

`[sysb,g,T,Ti] = balreal(sys)` also returns the vector `g` containing the diagonal of the balanced grammian, the state similarity transformation $x_b = Tx$ used to convert `sys` to `sysb`, and the inverse transformation $T\mathrm{i} = T^{-1}$.

If the system is normalized properly, the diagonal `g` of the joint grammian can be used to reduce the model order. Because `g` reflects the combined controllability and observability of individual states of the balanced model, you can delete those states with a small `g(i)` while retaining the most important input-output characteristics of the original system. Use `modred` to perform the state elimination.

Example

Consider the zero-pole-gain model

```
sys = zpk([-10 -20.01],[-5 -9.9 -20.1],1)

Zero/pole/gain:
   (s+10) (s+20.01)
----------------------
(s+5) (s+9.9) (s+20.1)
```

A state-space realization with balanced grammians is obtained by

```
[sysb,g] = balreal(sys)
```

The diagonal entries of the joint grammian are

```
g'

ans =
```

```
    1.0062e-01   6.8039e-05   1.0055e-05
```

which indicates that the last two states of `sysb` are weakly coupled to the input and output. You can then delete these states by

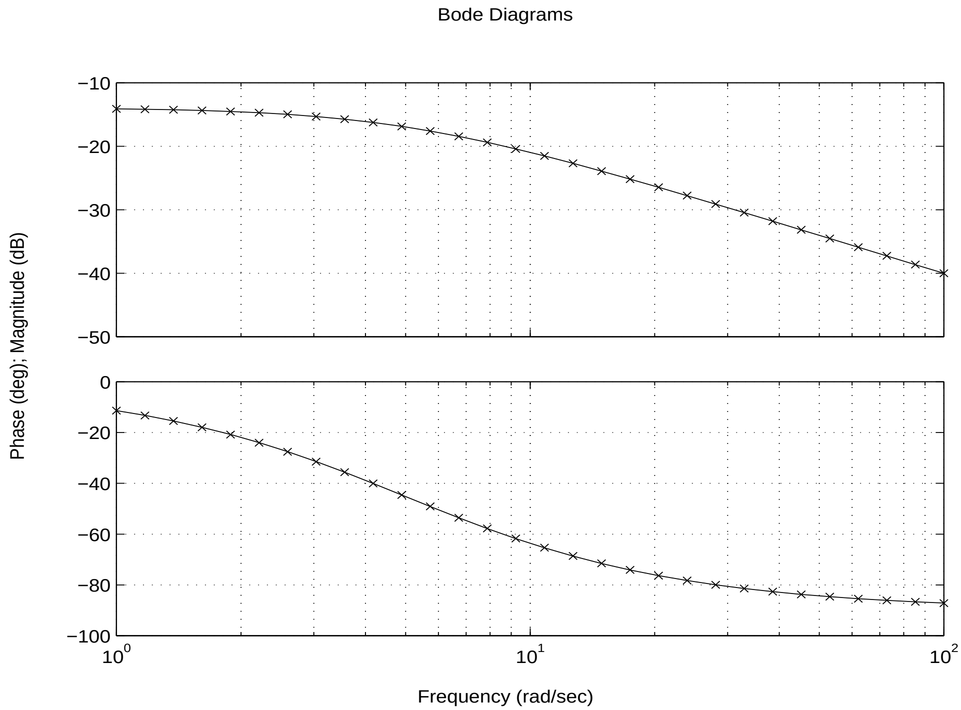
```
sysr = modred(sysb,[2 3],'del')
```

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 grammians 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 grammians to

$$\overline{W}_c = TW_cT^T, \qquad \overline{W}_o = T^{-T}W_oT^{-1}$$

The function `balreal` computes a particular similarity transformation T such that

$$\overline{W}_c = \overline{W}_o = diag(g)$$

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

`gram`	Controllability and observability grammians
`minreal`	Minimal realizations
`modred`	Model order reduction

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.

bandwidth

Purpose

Compute the frequency response bandwidth

Syntax

```
fb = bandwidth(sys)
fb = bandwidth(sys,dbdrop)
```

Description

`fb = bandwidth(sys)` computes the bandwidth `fb` of the SISO model `sys`, defined as the first frequency where the gain drops below 70.79 percent (-3 dB) of its DC value. The frequency `fb` is expressed in radians per second. You can create `sys` using `tf`, ss, or `zpk`, see `ltimodels` for details.

`fb = bandwidth(sys,dbdrop)` further specifies the critical gain drop in dB. The default value is -3 dB, or a 70.79 percent drop.

If `sys` is an S1-by...-by-Sp array of LTI models, `bandwidth` returns an array of the same size such that

```
fb(j1,...,jp) = bandwidth(sys(:,:,j1,...,jp))
```

See Also

`dcgain`	Compute the steady-state gain of LTI models
`issiso`	Returns 1 if the system is SISO
`ltimodels`	General information about LTI models

Purpose Compute the Bode frequency response of LTI models

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 models. When invoked without left-side arguments, `bode` produces a Bode plot on the screen. The magnitude is plotted in decibels (dB), and the phase in degrees. The decibel calculation for `mag` is computed as $20\log_{10}(|H(j\omega)|)$, where $|H(j\omega)|$ is the system's frequency response. 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')` 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-side 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)
```

Remark

If `sys` is an FRD model, `bode(sys,w)`, `w` can only include frequencies in `sys.frequency`.

Arguments

The output arguments `mag` and `phase` are 3-D arrays with dimensions

$$(\text{number of outputs}) \times (\text{number of inputs}) \times (\text{length of } w)$$

For SISO systems, `mag(1,1,k)` and `phase(1,1,k)` give the magnitude and phase of the response at the frequency $\omega_k = \texttt{w(k)}$.

$$\text{mag(1,1,k)} = |h(j\omega_k)|$$
$$\text{phase(1,1,k)} = \angle h(j\omega_k)$$

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)`.

$$\text{mag(i,j,k)} = |h_{ij}(j\omega_k)|$$
$$\text{phase(i,j,k)} = \angle h_{ij}(j\omega_k)$$

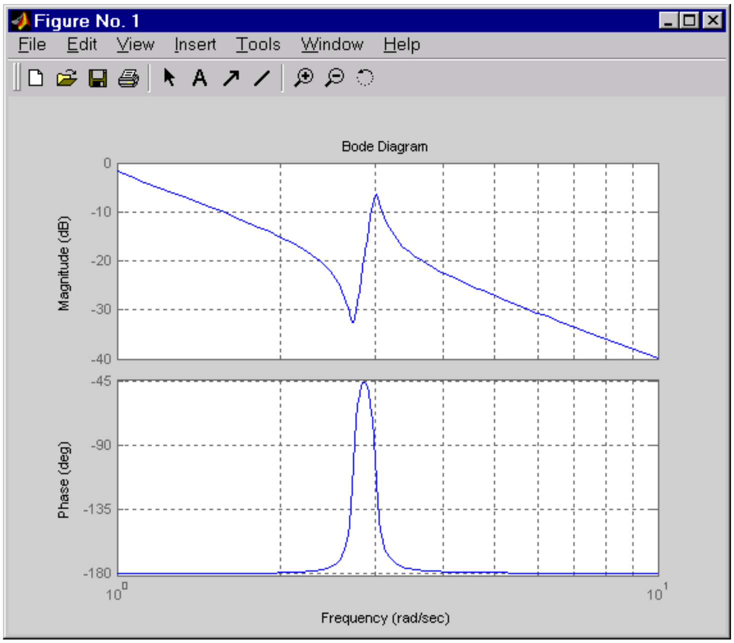
Example

You can plot the Bode response of the continuous SISO system

$$H(s) = \frac{s^2 + 0.1s + 7.5}{s^4 + 0.12s^3 + 9s^2}$$

by typing

```
g = tf([1 0.1 7.5],[1 0.12 9 0 0]);
bode(g)
```



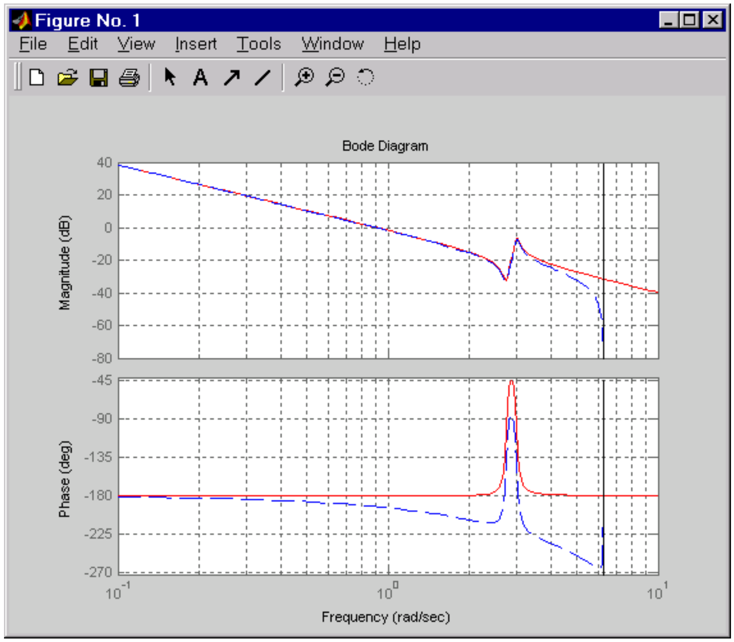
To plot the response on a wider frequency range, for example, from 0.1 to 100 rad/sec, type

```
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 $D + C(j\omega - A)^{-1}B \;, \qquad \omega \geq 0$

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

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

where T_s is the sample time. 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

`evalfr`	Response at single complex frequency
`freqresp`	Frequency response computation
`ltiview`	LTI system viewer
`nichols`	Nichols plot
`nyquist`	Nyquist plot
`sigma`	Singular value plot

References

[1] Laub, A.J., "Efficient Multivariable Frequency Response Computations," *IEEE Transactions on Automatic Control*, AC-26 (1981), pp. 407–408.

bodemag

Purpose

Compute the Bode magnitude response of LTI models

Syntax

```
bodemag(sys)
bodemag(sys,{wmin,wmax})
bodemag(sys,w)

bodemag(sys1,sys2,...,sysN,w)
bodemag(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
```

Description

`bodemag(sys)` plots the magnitude of the frequency response of the LTI model SYS (Bode plot without the phase diagram). The frequency range and number of points are chosen automatically.

`bodemag(sys,{wmin,wmax})` draws the magnitude plot for frequencies between `wmin` and `wmax` (in radians/second).

`bodemag(sys,w)` uses the user-supplied vector W of frequencies, in radians/second, at which the frequency response is to be evaluated.

`bodemag(sys1,sys2,...,sysN,w)` shows the frequency response magnitude of several LTI models `sys1,sys2,...,sysN` on a single plot. The frequency vector `w` is optional. You can also specify a color, line style, and marker for each model, as in

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

See Also

`bode`	Compute the Bode frequency response of LTI models
`ltiview`	Open an LTI Viewer
`ltimodels`	Help on LTI models

Purpose Discretize continuous-time systems

Syntax

```
sysd = c2d(sys,Ts)
sysd = c2d(sys,Ts,method)
[sysd,G] = 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 `Ts`.
`'foh'`	Triangle approximation (modified first-order hold, see [1], p. 151). The control inputs are assumed piecewise linear over the sampling period `Ts`.
`'tustin'`	Bilinear (Tustin) approximation.
`'prewarp'`	Tustin approximation with frequency prewarping.
`'matched'`	Matched pole-zero method. See [1], p. 147.

Refer to "Continuous/Discrete Conversions of LTI Models" for more detail on these discretization methods.

`c2d` supports MIMO systems (except for the `'matched'` method) as well as LTI models with delays with some restrictions for `'matched'` and `'tustin'` methods.

`[sysd,G] = c2d(sys,Ts,`*`method`*`)` returns a matrix G that maps the continuous initial conditions x_0 and u_0 to their discrete counterparts $x[0]$ and $u[0]$ according to

$$x[0] = G \cdot \begin{bmatrix} x_0 \\ u_0 \end{bmatrix}$$

$$u[0] = u_0$$

Example

Consider the system

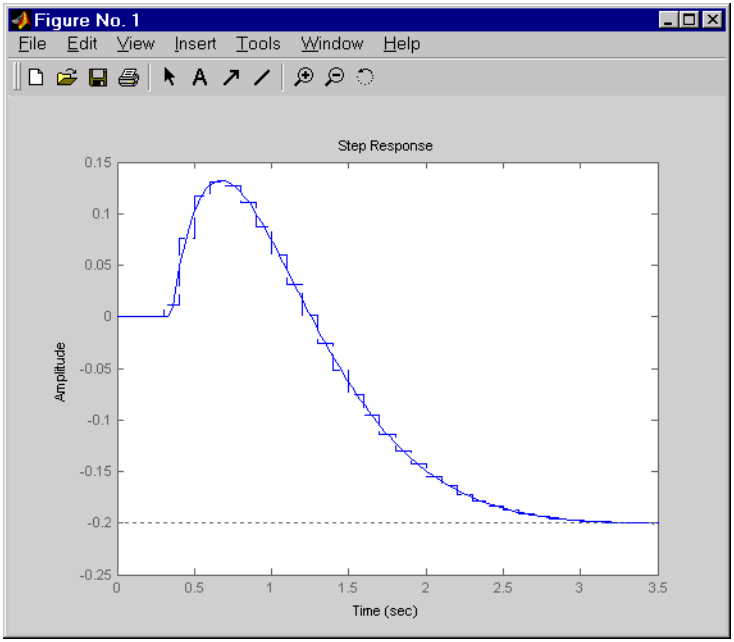
$$H(s) = \frac{s-1}{s^2+4s+5}$$

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],'inputdelay',0.35)

Transfer function:
                       s - 1
exp(-0.35*s) * -------------
                 s^2 + 4 s + 5

Hd = c2d(H,0.1,'foh')

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`	Discrete to continuous conversion
`d2d`	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.

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,'type')` 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,'type')` 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

$$p(s) = s^n + a_1 s^{n-1} + \ldots + a_{n-1} s + a_n$$

the corresponding companion A matrix is

$$A = \begin{bmatrix} 0 & 0 & .. & .. & 0 & -a_n \\ 1 & 0 & 0 & .. & 0 & -a_{n-1} \\ 0 & 1 & 0 & . & : & : \\ : & 0 & . & . & : & : \\ 0 & . & . & 1 & 0 & -a_2 \\ 0 & .. & .. & 0 & 1 & -a_1 \end{bmatrix}$$

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

ctrb	Controllability matrix
ctrbf	Controllability canonical form
ss2ss	State similarity transformation

References

[1] Kailath, T. *Linear Systems*, Prentice-Hall, 1980.

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

$$Ric(X) = A^TX + XA - XBB^TX + Q = 0$$

such that $A - BB^TX$ has all its eigenvalues in the open left-half plane. The matrix X is symmetric and called the *stabilizing* solution of $Ric(X) = 0$. `[X,L,G,rr] = care(A,B,Q)` also returns:

- The eigenvalues `L` of $A - BB^TX$
- The gain matrix $G = B^TX$
- 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 Riccati equation

$$Ric(X) = A^TXE + E^TXA - (E^TXB + S)R^{-1}(B^TXE + S^T) + Q = 0$$

Here the gain matrix is $G = R^{-1}(B^TXE + S^T)$ and the "closed-loop" eigenvalues are `L = 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., $X = X_2X_1^{-1}$ 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.

$$X = X_2 X_1^{-1}$$

Note that this syntax returns `report = 0` when successful.

Examples

Example 1

Given

$$A = \begin{bmatrix} -3 & 2 \\ 1 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 & -1 \end{bmatrix} \qquad R = 3$$

you can solve the Riccati equation

$$A^T X + XA - XBR^{-1}B^T X + C^T C = 0$$

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 =
    0.5895    1.8216
    1.8216    8.8188
```

You can verify that this solution is indeed stabilizing by comparing the eigenvalues of `a` and `a-b*g`.

```
[eig(a)  eig(a-b*g)]

ans =
```

```
    -3.4495   -3.5026
     1.4495   -1.4370
```

Finally, note that the variable `l` contains the closed-loop eigenvalues `eig(a-b*g)`.

```
l

l =
    -3.5026
    -1.4370
```

Example 2

To solve the H_∞-like Riccati equation

$$A^TX + XA + X(\gamma^{-2}B_1B_1^T - B_2B_2^T)X + C^TC = 0$$

rewrite it in the `care` format as

$$A^TX + XA - X\underbrace{[B_1, B_2]}_{B}\underbrace{\begin{bmatrix} -\gamma^{-2}I & 0 \\ 0 & I \end{bmatrix}}_{R}^{-1}\begin{bmatrix} B_1^T \\ B_2^T \end{bmatrix}X + C^TC = 0$$

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 $E = I$; otherwise it uses the extended Hamiltonian pencil and QZ algorithm.

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] Arnold, W.F., III and A.J. Laub, “Generalized Eigenproblem Algorithms and Software for Algebraic Riccati Equations,” *Proc. IEEE*, 72 (1984), pp. 1746–1754.

chgunits

Purpose

Convert the frequency units of an FRD model

Syntax

```
sys = chgunits(sys,units)
```

Description

`sys = chgunits(sys,units)` converts the units of the frequency points stored in an FRD model, `sys` to `units`, where `units` is either of the strings `'Hz'` or `'rad/s'`. This operation changes the assigned frequencies by applying the appropriate (`2*pi`) scaling factor, and the `'Units'` property is updated.

If the `'Units'` field already matches `units`, no conversion is made.

Example

```
w = logspace(1,2,2);
sys = rss(3,1,1);
sys = frd(sys,w)

From input 'input 1' to:

  Frequency(rad/s)       output 1
  ----------------       --------

          10        0.293773+0.001033i

         100        0.294404+0.000109i

Continuous-time frequency response data.

sys = chgunits(sys,'Hz')
sys.freq

ans =
    1.5915
   15.9155
```

See Also

`frd`	Create or convert to an FRD model
`get`	Get the properties of an LTI model
`set`	Set the properties of an LTI model

Purpose Form a model with complex conjugate coefficients

Syntax

```
sysc = conj(sys)
```

Description

`sysc = conj(sys)` is an constructs a complex conjugate model `sysc` by applying complex conjugation to all coefficients of the LTI model `sys`. This function accepts LTI models in transfer function (TF), zero/pole/gain (ZPK), and state space (SS) formats.

Example

If `sys` is the transfer function

```
(2+i)/(s+i)
```

then `conj(sys)` produces the transfer function

```
(2-i)/(s-i)
```

This operation is useful for manipulating partial fraction expansions.

See Also

`append`	Append LTI systems
`ss`	Specify or convert to state-space form
`tf`	Specify or convert to transfer function form
`zpk`	Specify or convert to zero-pole-gain form

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.

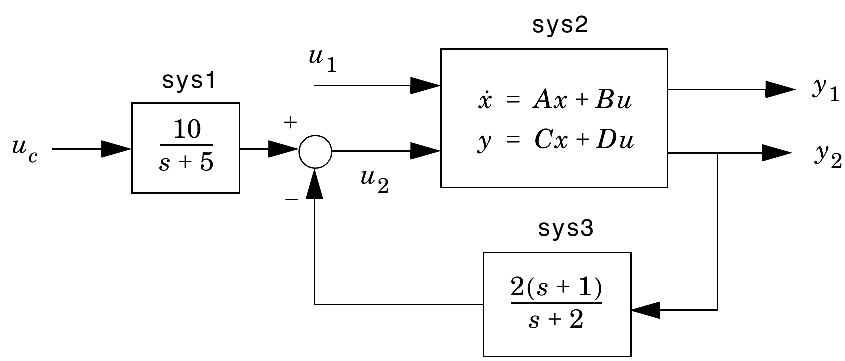
Since it is easy to make a mistake entering all the data required for a large model, be sure to verify your model in as many ways as you can. Here are some suggestions:

- 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.

The connect function does support delays in a reliable way. If you need to work extensively with block diagrams or you need to interconnect models with time delays, 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 = [ -.5112  .5362
```

```
     -.002  -1.8470];
C = [ -3.2897  2.4544
     -13.5009  18.0745];
D = [-.5476  -.1410
    -.6459  .2958 ];
```

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)
```

Next 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 b.

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 as follows.

```
inputs = [1 2];
outputs = [2 3];
```

You can obtain a state-space model for the overall interconnection by typing

```
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
`feedback`	Feedback connection
`minreal`	Minimal state-space realization
`parallel`	Parallel connection
`series`	Series connection

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.

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

$$E(w(t)w(\tau)^T) = W\ \delta(t-\tau) \qquad \text{(continuous time)}$$

$$E(w[k]w[l]^T) = W\ \delta_{kl} \qquad \text{(discrete time)}$$

`[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 `[]`).

When applied to an `N`-dimensional LTI array `sys`, `covar` returns multi-dimensional arrays P, Q such that

`P(:,:,i1,...iN)` and `Q(:,:,i1,...iN)` are the covariance matrices for the model `sys(:,:,i1,...iN)`.

Example

Compute the output response covariance of the discrete SISO system

$$H(z) = \frac{2z+1}{z^2+0.2z+0.5}, \qquad T_s = 0.1$$

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 this output of covar to 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

$$\dot{x} = Ax + Bw$$
$$y = Cx + Dw$$

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 *stable* systems only. For continuous systems, the output response covariance P is finite only when the D matrix is zero (strictly proper system).

See Also

`dlyap`	Solver for discrete-time Lyapunov equations
`lyap`	Solver for continuous-time Lyapunov equations

References

[1] Bryson, A.E. and Y.C. Ho, *Applied Optimal Control,* 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

$$Co = \begin{bmatrix} B & AB & A^2B & \dots & A^{n-1}B \end{bmatrix} \tag{4-1}$$

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 the following data

```
A =
      1     1
      4    -2

B =
      1    -1
      1    -1
```

is controllable. Type

```
Co=ctrb(A,B);

% Number of uncontrollable states
unco=length(A)-rank(Co)
```

and MATLAB returns

```
unco =
      1
```

Limitations

Estimating the rank of the controllability matrix is ill-conditioned; that is, it is very sensitive to round-off errors and errors in the data. An indication of this can be seen from this simple example.

$$A = \begin{bmatrix} 1 & \delta \\ 0 & 1 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 \\ \delta \end{bmatrix}$$

This pair is controllable if $\delta \neq 0$ but if $\delta < \sqrt{eps}$, where *eps* is the relative machine precision. `ctrb(A,B)` returns

$$\begin{bmatrix} B & AB \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ \delta & \delta \end{bmatrix}$$

which is not full rank. For cases like these, it is better to determine the controllability of a system using `ctrbf`.

See Also

`ctrbf`	Compute the controllability staircase form
`obsv`	Compute the observability matrix

ctrbf

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

If the controllability matrix of (A, B) has rank $r \le n$, where n is the size of A, then there exists a similarity transformation such that

$$\bar{A} = TAT^T, \qquad \bar{B} = TB, \qquad \bar{C} = CT^T$$

where T is unitary, and the transformed system has a *staircase* form, in which the uncontrollable modes, if there are any, are in the upper left corner.

$$\bar{A} = \begin{bmatrix} A_{uc} & 0 \\ A_{21} & A_c \end{bmatrix}, \qquad \bar{B} = \begin{bmatrix} 0 \\ B_c \end{bmatrix}, \qquad \bar{C} = \begin{bmatrix} C_{nc} & C_c \end{bmatrix}$$

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 `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*norm(A,1)*eps`.

Example

Compute the controllability staircase form for

```
A =
      1     1
      4    -2
```

```
B =
     1    -1
     1    -1

C =
     1     0
     0     1
```

and locate the uncontrollable mode.

```
[Abar,Bbar,Cbar,T,k]=ctrbf(A,B,C)

Abar =
   -3.0000         0
   -3.0000    2.0000

Bbar =
    0.0000    0.0000
    1.4142   -1.4142

Cbar =
   -0.7071    0.7071
    0.7071    0.7071

T =
   -0.7071    0.7071
    0.7071    0.7071
k =
     1     0
```

The decomposed system `Abar` shows an uncontrollable mode located at –3 and a controllable mode located at 2.

Algorithm

`ctrbf` is an M-file that implements the Staircase Algorithm of [1].

See Also

`ctrb`	Form the controllability matrix
`minreal`	Minimum realization and pole-zero cancellation

References

[1] Rosenbrock, M.M., *State-Space and Multivariable Theory*, John Wiley, 1970.

d2c

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 of LTI Models" 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 the MATLAB documentation).

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 = zpk(-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)
```

Convert Hc 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

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

Description

`sys1 = d2d(sys,Ts)` resamples the discrete-time LTI model `sys` to produce an equivalent discrete-time model `sys1` with the new sample time `Ts` (in seconds). The resampling assumes zero-order hold on the inputs and is equivalent to consecutive `d2c` and `c2d` conversions.

```
sys1 = c2d(d2c(sys),Ts)
```

Example

Consider the zero-pole-gain model

$$H(z) = \frac{z-0.7}{z-0.5}$$

with sample time 0.1 second. You can resample this model at 0.05 second by typing

```
H = zpk(0.7,0.5,1,0.1)
H2 = d2d(H,0.05)

Zero/pole/gain:
(z-0.8243)
----------
(z-0.7071)

Sampling time: 0.05
```

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
```

See Also

`c2d`	Continuous- to discrete-time conversion
`d2c`	Discrete- to continuous-time conversion

damp

Purpose

Compute damping factors and natural frequencies

Syntax

```
[Wn,Z] = damp(sys)
[Wn,Z,P] = damp(sys)
```

Description

`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.

`[Wn,Z] = damp(sys)` returns column vectors `Wn` 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

$$z = e^{sT_s}$$

The values `Wn` and `Z` are then relative to the continuous-time poles s. Both `Wn` and `Z` are empty if the sample time is unspecified.

`[Wn,Z,P] = damp(sys)` returns an additional vector `P` containing the (true) poles of `sys`. Note that `P` returns the same values as `pole(sys)` (up to reordering).

Example

Compute and display the eigenvalues, natural frequencies, and damping factors of the continuous transfer function

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

Type

```
H = tf([2 5 1],[1 2 3])

Transfer function:
2 s^2 + 5 s + 1
---------------
 s^2 + 2 s + 3
```

Type

```
damp(H)
```

and MATLAB returns

```
Eigenvalue                Damping     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
`pole`	Compute system poles
`pzmap`	Pole-zero map
`zero`	Compute (transmission) zeros

Purpose Solve discrete-time algebraic Riccati equations (DARE)

Syntax

```
[X,L,G,rr] = dare(A,B,Q,R)
[X,L,G,rr] = dare(A,B,Q,R,S,E)

[X,L,G,report] = dare(A,B,Q,...,'report')
[X1,X2,L,report] = dare(A,B,Q,...,'implicit')
```

Description `[X,L,G,rr] = dare(A,B,Q,R)` computes the unique solution X of the discrete-time algebraic Riccati equation

$$Ric(X) = A^TXA - X - A^TXB(B^TXB + R)^{-1}B^TXA + Q = 0$$

such that the "closed-loop" matrix

$$A_{cl} = A - B(B^TXB + R)^{-1}B^TXA$$

has all its eigenvalues inside the unit disk. The matrix X is symmetric and called the *stabilizing* solution of $Ric(X) = 0$. `[X,L,G,rr] = dare(A,B,Q,R)` also returns:

- The eigenvalues `L` of A_{cl}
- The gain matrix

$$G = (B^TXB + R)^{-1}B^TXA$$

- The relative residual `rr` defined by

$$rr = \frac{\|Ric(X)\|_F}{\|X\|_F}$$

`[X,L,G,rr] = dare(A,B,Q,R,S,E)` solves the more general DARE:

$$A^TXA - E^TXE - (A^TXB + S)(B^TXB + R)^{-1}(B^TXA + S^T) + Q = 0$$

The corresponding gain matrix and closed-loop eigenvalues are

$$G = (B^TXB + R)^{-1}(B^TXA + S^T)$$

and `L = 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, $X = X_2X_1^{-1}$ 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

$$X = X_2X_1^{-1}$$

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`	Solve continuous-time Riccati equations
`dlyap`	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.

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

$$K = D + C(I - A)^{-1}B$$

Remark The DC gain is infinite for systems with integrators.

Example To compute the DC gain of the MIMO transfer function

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

type

```
H = [1 tf([1 -1],[1 1 3]) ; tf(1,[1 1]) tf([1 2],[1 -3])]
dcgain(H)

ans =
    1.0000   -0.3333
    1.0000   -0.6667
```

See Also

`evalfr`	Evaluates frequency response at single frequency
`norm`	LTI system norms

delay2z

Purpose

Replace delays of discrete-time TF, SS, or ZPK models by poles at z=0, or replace delays of FRD models by a phase shift

Syntax

```
sys = delay2z(sys)
```

Description

`sys = delay2z(sys)` maps all time delays to poles at z=0 for discrete-time TF, ZPK, or SS models `sys`. Specifically, a delay of `k` sampling periods is replaced by `(1/z)^k` in the transfer function corresponding to the model.

For FRD models, `delay2z` absorbs all time delays into the frequency response data, and is applicable to both continuous- and discrete-time FRDs.

Example

```
z=tf('z',-1);
sys=(-.4*z -.1)/(z^2 + 1.05*z + .08)

Transfer function:

-0.4 z - 0.1
-------------------
z^2 + 1.05 z + 0.08

Sampling time: unspecified

sys.InputDelay = 1;
sys = delay2z(sys)

Transfer function:

     -0.4 z - 0.1
-----------------------
z^3 + 1.05 z^2 + 0.08 z

Sampling time: unspecified
```

See Also

`hasdelay`	True for LTI models with delays
`pade`	Pade approximation of time delays
`totaldelay`	Combine delays for an LTI model

Purpose Design linear-quadratic (LQ) state-feedback regulator for discrete-time plant

Syntax

```
[K,S,e] = dlqr(a,b,Q,R)
[K,S,e] = dlqr(a,b,Q,R,N)
```

Description `[K,S,e] = 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 mode

$$1x[n+1] = Ax[n] + Bu[n]$$

The default value `N=0` is assumed when `N` is omitted.

In addition to the state-feedback gain `K`, `dlqr` returns the infinite horizon solution S of the associated discrete-time Riccati equation

$$A^T SA - S - (A^T SB + N)(B^T SB + R)^{-1}(B^T SA + N^T) + Q = 0$$

and the closed-loop eigenvalues `e = eig(a-b*K)`. Note that K is derived from S by

$$K = (B^T SB + R)^{-1}(B^T SA + N^T)$$

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

`dare`	Solve discrete Riccati equations
`lqgreg`	LQG regulator

`lqr`	State-feedback LQ regulator for continuous plant
`lqrd`	Discrete LQ regulator for continuous plant
`lqry`	State-feedback LQ regulator with output weighting

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 $\alpha_1, \alpha_2, ..., \alpha_n$ of A satisfy $\alpha_i \alpha_j \neq 1$ for all (i, j).

If this condition is violated, `dlyap` produces the error message

```
Solution does not exist or is not unique.
```

See Also

`covar`	Covariance of system response to white noise
`lyap`	Solve continuous Lyapunov equations

drss

Purpose

Generate stable random discrete test models

Syntax

```
sys = drss(n)
sys = drss(n,p)
sys = drss(n,p,m)
sys = drss(n,p,m,s1,...sn)
```

Description

`sys = drss(n)` produces a random *n*-th order stable model with one input and one output, and returns the model in the state-space object `sys`.

`drss(n,p)` produces a random *n*-th order stable model with one input and `p` outputs.

`drss(n,m,p)` generates a random *n*-th order stable model with `m` inputs and `p` outputs.

`drss(n,p,m,s1,...sn)` generates a *s1*–by–*sn* array of random *n*-th order stable model with `m` inputs and `p` outputs.

In all cases, the discrete-time state-space model or array returned by `drss` has an unspecified sampling time. To generate transfer function or zero-pole-gain systems, convert `sys` using `tf` or `zpk`.

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

`rss`	Generate stable random continuous test models
`tf`	Convert LTI systems to transfer functions form
`zpk`	Convert LTI systems to zero-pole-gain form

dsort

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

`eig`	Calculate eigenvalues and eigenvectors
`esort`, `sort`	Sort system poles
`pole`	Compute system poles
`pzmap`	Pole-zero map
`zero`	Compute (transmission) zeros

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,ltisys)

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

$$E\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

The E matrix must be nonsingular. The output `sys` is an SS model storing the model data (see "LTI Objects" on page 2-3). Note that `ss` produces the same type of object. If the matrix $D = 0$, do 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,ltisys)` creates a descriptor model with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See "LTI Properties" on page 2-26 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

$$5\dot{x} = x + 2u$$
$$y = 3x + 4u$$

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
`get`	Get properties of LTI models
`set`	Set properties of LTI models
`ss`	Specify (regular) state-space models

Purpose

Quick access to descriptor state-space data

Syntax

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

Description

`[a,b,c,d,e] = dssdata(sys)` extracts the descriptor matrix data (A, B, C, D, E) from the state-space model `sys`. If `sys` is a transfer function or zero-pole-gain model, it is first converted to state space. Note that `dssdata` is then equivalent to `ssdata` because it always returns $E = I$.

`[a,b,c,d,e,Ts] = dssdata(sys)` also returns the sample time `Ts`.

You can access the remaining LTI properties of `sys` with `get` or by direct referencing, for example,

```
sys.notes
```

See Also

`dss`	Specify descriptor state-space models
`get`	Get properties of LTI models
`ssdata`	Quick access to state-space data
`tfdata`	Quick access to transfer function data
`zpkdata`	Quick access to zero-pole-gain data

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 left-hand argument, s = esort(p) returns the sorted eigenvalues in s.

[s,ndx] = esort(p) returns the additional argument ndx, a vector 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

dsort, sort	Sort system poles
eig	Calculate eigenvalues and eigenvectors
pole	Compute system poles
pzmap	Pole-zero map

`zero`	Compute (transmission) zeros

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

$$\dot{x} = Ax + Bw$$
$$y = Cx + Dw$$

`estim` generates plant output and state estimates $\hat{y}$ and $\hat{x}$ as given by the following model.

$$\dot{\hat{x}} = A\hat{x} + L(y - C\hat{x})$$
$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} \hat{x}$$

The discrete-time estimator has similar equations.

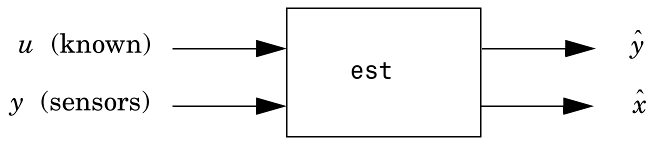
`est = estim(sys,L,sensors,known)` handles more general plants `sys` with both known inputs u and stochastic inputs w, and both measured outputs y and nonmeasured outputs z.

$$\dot{x} = Ax + B_1 w + B_2 u$$
$$\begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} x + \begin{bmatrix} D_{11} \\ D_{21} \end{bmatrix} w + \begin{bmatrix} D_{12} \\ D_{22} \end{bmatrix} u$$

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.

$$\dot{\hat{x}} = A\hat{x} + B_2 u + L(y - C_2\hat{x} - D_{22}u)$$

$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C_2 \\ I \end{bmatrix} \hat{x} + \begin{bmatrix} D_{22} \\ 0 \end{bmatrix} u$$



`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

`kalman`	Design Kalman estimator
`place`	Pole placement
`reg`	Form regulator given state-feedback and estimator gains

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 TF, SS, or ZPK 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)

ans =
  2.3077e-01 +  1.5385e-01i
```

Limitations The response is not finite when `f` is a pole of `sys`.

See Also

`bode`	Bode frequency response
`freqresp`	Frequency response over a set of frequencies
`sigma`	Singular value response

Purpose

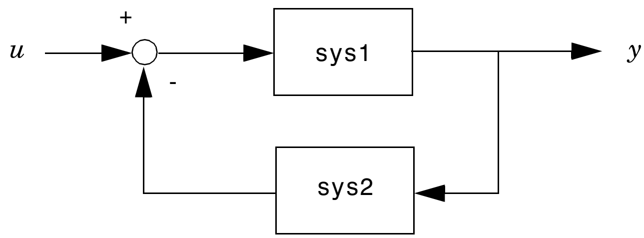
Feedback connection of two LTI models

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 Precedence Rules).

To apply positive feedback, use the syntax

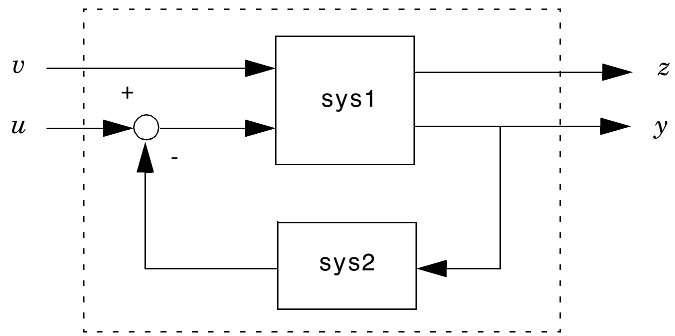
```
sys = feedback(sys1,sys2,+1)
```

By default, `feedback(sys1,sys2)` assumes negative feedback and is equivalent to `feedback(sys1,sys2,-1)`.

Finally,

```
sys = feedback(sys1,sys2,feedin,feedout)
```

computes a closed-loop model sys for the more general feedback loop.



sys

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,

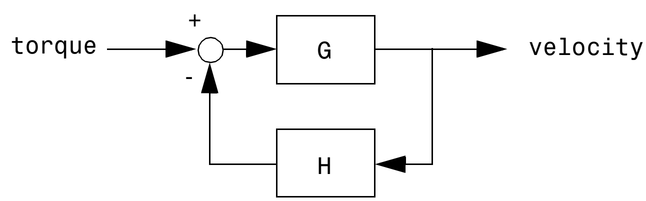
```
sys = feedback(sys1,2)
```

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 = zpk(-2,-10,5)
Cloop = 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 `Cloop` inherited the input and output names from `G`.

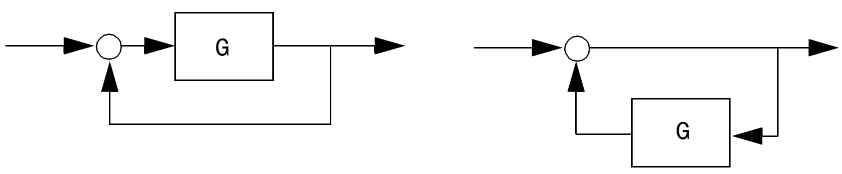
Example 2

Consider a state-space plant P with five inputs and four outputs and a state-space feedback controller K with three inputs and two 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];
Cloop = feedback(P,K,feedin,feedout)
```

Example 3

You can form the following negative-feedback loops



by

```
Cloop = feedback(G,1)    % left diagram
Cloop = feedback(1,G)    % right diagram
```

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

series	Series connection
parallel	Parallel connection
connect	Derive state-space model for block diagram interconnection

Purpose Specify discrete transfer functions in DSP format

Syntax

```
sys = filt(num,den)
sys = filt(num,den,Ts)
sys = filt(M)

sys = filt(num,den,'Property1',Value1,...,'PropertyN',ValueN)
sys = filt(num,den,Ts,'Property1',Value1,...,'PropertyN',ValueN)
```

Description 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 *ascending* powers of z^{-1}, for example,

$$H(z^{-1}) = \frac{2 + z^{-1}}{1 + 0.4z^{-1} + 2z^{-2}}$$

The function `filt` is provided to facilitate the specification of transfer functions in DSP format.

`sys = filt(num,den)` creates a discrete-time transfer function `sys` with numerator(s) `num` and denominator(s) `den`. The sample time is left unspecified (`sys.Ts = -1`) and the output `sys` is a TF object.

`sys = filt(num,den,Ts)` further specifies the sample time `Ts` (in seconds).

`sys = filt(M)` specifies a static filter with gain matrix `M`.

Any of the previous syntaxes can be followed by property name/property value pairs of the form

```
'Property',Value
```

Each pair specifies a particular LTI property of the model, for example, the input names or the transfer function variable. See LTI Properties and the `set` entry for additional information on LTI properties and admissible property values.

Arguments For SISO transfer functions, `num` and `den` are row vectors containing the numerator and denominator coefficients ordered in ascending powers of z^{-1}. For example, `den = [1 0.4 2]` represents the polynomial $1 + 0.4z^{-1} + 2z^{-2}$.

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 MIMO Transfer Function Models 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',{'channel1' 'channel2'})
```

creates the two-input digital filter

$$H(z^{-1}) = \begin{bmatrix} \dfrac{1}{1+z^{-1}+2z^{-2}} & \dfrac{1+0.3z^{-1}}{5+2z^{-1}} \end{bmatrix}$$

with unspecified sample time and input names 'channel1' and 'channel2'.

See Also

tf	Create transfer functions
zpk	Create zero-pole-gain models
ss	Create state-space models

Purpose Create a frequency response data (FRD) object or convert another model type to an FRD model

Syntax

```
sys = frd(response,frequency)
sys = frd(response,frequency,Ts)
sys = frd
sys = frd(response,frequency,ltisys)

sysfrd = frd(sys,frequency)
sysfrd = frd(sys,frequency,'Units',units)
```

Description

`sys = frd(response,frequency)` creates an FRD model `sys` from the frequency response data stored in the multidimensional array `response`. The vector `frequency` represents the underlying frequencies for the frequency response data. See Table 4-1, Data Format for the Argument response in FRD Models.

`sys = frd(response,frequency,Ts)` creates a discrete-time FRD model `sys` with scalar sample time `Ts`. Set `Ts` = -1 to create a discrete-time FRD model without specifying the sample time.

`sys = frd` creates an empty FRD model.

The input argument list for any of these syntaxes can be followed by property name/property value pairs of the form

```
'PropertyName',PropertyValue
```

You can use these extra arguments to set the various properties of FRD models (see the `set` command, or LTI Properties and Model-Specific Properties). These properties include `'Units'`. The default units for FRD models are in `'rad/s'`.

To force an FRD model `sys` to inherit all of its generic LTI properties from any existing LTI model `refsys`, use the syntax

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

`sysfrd = frd(sys,frequency)` converts a TF, SS, or ZPK model to an FRD model. The frequency response is computed at the frequencies provided by the vector `frequency`.

`sysfrd = frd(sys,frequency,'Units',units)` converts an FRD model from a TF, SS, or ZPK model while specifying the units for `frequency` to be `units` (`'rad/s'` or `'Hz'`).

Arguments

When you specify a SISO or MIMO FRD model, or an array of FRD models, the input argument `frequency` is always a vector of length `Nf`, where `Nf` is the number of frequency data points in the FRD. The specification of the input argument `response` is summarized in the following table.

Table 4-1: Data Format for the Argument response in FRD Models

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

Remarks

See Frequency Response Data (FRD) Models for more information on single FRD models, and Creating LTI Models for information on building arrays of FRD models.

Example

Type the commands

```
freq = logspace(1,2);
resp = .05*(freq).*exp(i*2*freq);
sys = frd(resp,freq)
```

to create a SISO FRD model.

See Also

`chgunits`	Change units for an FRD model
`frdata`	Quick access to data for an FRD model
`set`	Set the properties for an LTI model
`ss`	Create state-space models

`tf`	Create transfer functions
`zpk`	Create zero-pole-gain models

frdata

Purpose

Quick access to data for a frequency response data object

Syntax

```
[response,freq] = frdata(sys)
[response,freq,Ts] = frdata(sys)
[response,freq] = frdata(sys,'v')
```

Description

`[response,freq] = frdata(sys)` returns the response data and frequency samples of the FRD model `sys`. For an FRD model with `Ny` outputs and `Nu` inputs at `Nf` frequencies:

- `response` is an `Ny`-by-`Nu`-by-`Nf` multidimensional array where the `(i,j)` entry specifies the response from input `j` to output `i`.
- `freq` is a column vector of length `Nf` that contains the frequency samples of the FRD model.

See Table 11-14, "Data Format for the Argument response in FRD Models," on page 80 for more information on the data format for FRD response data.

For SISO FRD models, the syntax

```
[response,freq] = frdata(sys,'v')
```

forces `frdata` to return the response data and frequencies directly as column vectors rather than as cell arrays (see example below).

`[response,freq,Ts] = frdata(sys)` also returns the sample time `Ts`.

Other properties of `sys` can be accessed with `get` or by direct structure-like referencing (e.g., `sys.Units`).

Arguments

The input argument `sys` to `frdata` must be an FRD model.

Example

Typing the commands

```
freq = logspace(1,2,2);
resp = .05*(freq).*exp(i*2*freq);
sys = frd(resp,freq);
[resp,freq] = frdata(sys,'v')
```

returns the FRD model data

```
resp =
   0.2040 + 0.4565i
```

```
    2.4359 - 4.3665i

freq =
      10
     100
```

See Also

`frd`	Create or convert to FRD models
`get`	Get the properties for an LTI model
`set`	Set model properties

freqresp

Purpose Compute frequency response over grid of frequencies

Syntax

```
H = freqresp(sys,w)
```

Description `H = freqresp(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. For single LTI Models, `freqresp(sys,w)` returns a 3-D array `H` with the frequency as the last dimension (see "Arguments" below). For LTI arrays of size `[Ny Nu S1 ... Sn]`, `freqresp(sys,w)` returns a [`Ny`-by-`Nu`-by-`S1`-by-...-by-`Sn`] length (w) array.

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

$$H(j\omega) = D + C(j\omega I - A)^{-1}B$$

In discrete time, the real frequencies `w(1)`,..., `w(N)` are mapped to points on the unit circle using the transformation $z = e^{j\omega T_s}$

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.

Remark If `sys` is an FRD model, `freqresp(sys,w)`, `w` can only include frequencies in `sys.frequency`. Interpolation and extrapolation are not supported. To interpolate an FRD model, use `interp`.

Arguments The output argument `H` is a 3-D array with dimensions

(number of outputs) $\times$ (number of inputs) $\times$ (length of `w`)

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.

Example Compute the frequency response of

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

at the frequencies $\omega = 1, 10, 100$. Type

```
w = [1 10 100]
H = freqresp(P,w)

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

$$\omega = 1, \qquad \omega = 10, \qquad \omega = 100$$

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

$$D + C(j\omega - A)^{-1}B\,, \qquad \omega = \omega_1, \ldots, \omega_N$$

For efficiency, 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 `w` happens to contain this frequency point, the gain is infinite, $j\omega I - A$ is singular, and `freqresp` produces the following 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
`interp`	Interpolate FRD model between frequency points

References

[1] Laub, A.J., “Efficient Multivariable Frequency Response Computations,” *IEEE Transactions on Automatic Control*, AC-26 (1981), pp. 407-408.

Purpose Generate test input signals for `lsim`

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.

'sin'	Sine wave.
'square'	Square wave.
'pulse'	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 `lsim` 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`.

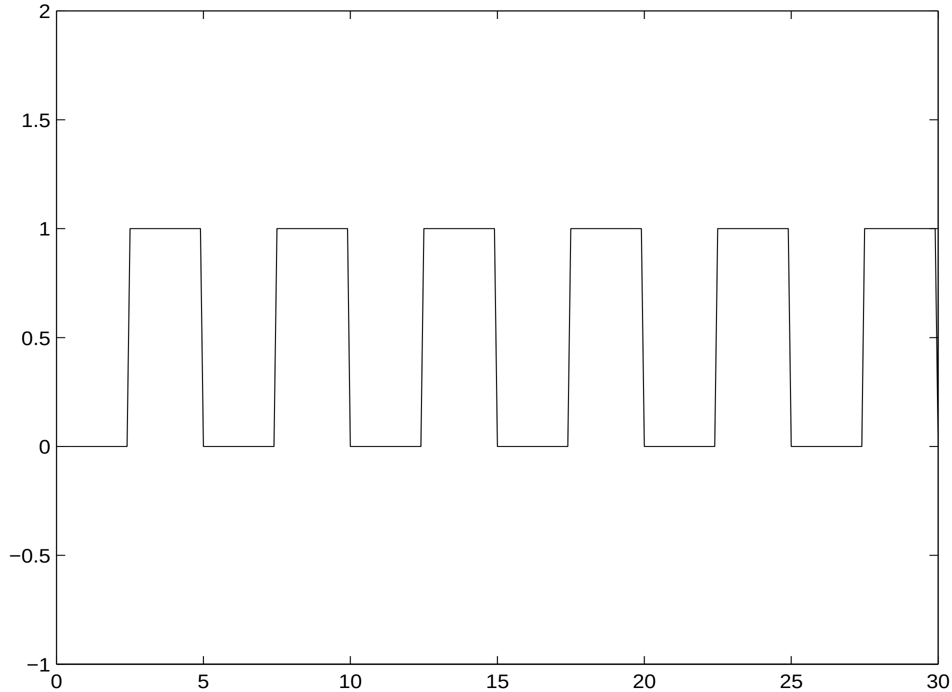
Example Generate a square wave with period 5 seconds, duration 30 seconds, and sampling every 0.1 second.

```
[u,t] = gensig('square',5,30,0.1)
```

Plot the resulting signal.

```
plot(t,u)
```

```
axis([0 30 -1 2])
```



See Also

`lsim`	Simulate response to arbitrary inputs

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 "LTI Properties" 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-side 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` with

```
get(h)
        num = {[0 1]}
        den = {[1 2]}
         Variable = 'z'
         Ts = 0.1
         InputDelay = 0
         OutputDelay = 0
         ioDelay = 0
         InputName = {'voltage'}
         OutputName = {''}
         InputGroup = {0x2 cell}
         OutputGroup = {0x2 cell}
```

```
          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

`frdata`	Quick access to frequency response data
`set`	Set/modify LTI properties
`ssdata`	Quick access to state-space data
`tfdata`	Quick access to transfer function data
`zpkdata`	Quick access to zero-pole-gain data

Purpose Compute controllability and observability grammians

Syntax

```
Wc = gram(sys,'c')
Wo = gram(sys,'o')
```

Description `gram` calculates controllability and observability grammians. You can use grammians 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 grammian is defined by

$$W_c = \int_0^\infty e^{A\tau} B B^T e^{A^T \tau} d\tau$$

and the observability grammian by

$$W_o = \int_0^\infty e^{A^T \tau} C^T C e^{A\tau} d\tau$$

The discrete-time counterparts are

$$W_c = \sum_{k=0}^{\infty} A^k B B^T (A^T)^k , \qquad W_o = \sum_{k=0}^{\infty} (A^T)^k C^T C A^k$$

The controllability grammian is positive definite if and only if (A, B) is controllable. Similarly, the observability grammian is positive definite if and only if (C, A) is observable.

Use the commands

```
Wc = gram(sys,'c')   % controllability grammian
Wo = gram(sys,'o')   % observability grammian
```

to compute the grammians of a continuous or discrete system. The LTI model `sys` must be in state-space form.

Algorithm

The controllability grammian W_c is obtained by solving the continuous-time Lyapunov equation

$$AW_c + W_cA^T + BB^T = 0$$

or its discrete-time counterpart

$$AW_cA^T - W_c + BB^T = 0$$

Similarly, the observability grammian W_o solves the Lyapunov equation

$$A^TW_o + W_oA + C^TC = 0$$

in continuous time, and the Lyapunov equation

$$A^TW_oA - W_o + C^TC = 0$$

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

`balreal`	Grammian-based balancing of state-space realizations
`ctrb`	Controllability matrix
`lyap`, `dlyap`	Lyapunov equation solvers
`obsv`	Observability matrix

References

[1] Kailath, T., *Linear Systems*, Prentice-Hall, 1980.

Purpose Test if an LTI model has time delays

Syntax

```
hasdelay(sys)
```

Description `hasdelay(sys)` returns 1 (true) if the LTI model `sys` has input delays, output delays, or I/O delays, and 0 (false) otherwise.

See Also

`delay2z`	Changes transfer functions of discrete-time LTI models with delays to rational functions or absorbs FRD delays into the frequency response phase information
`totaldelay`	Combines delays for an LTI model

impulse

Purpose

Compute the impulse response of LTI models

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,

```
impulse(sys1,'y:',sys2,'g--')
```

See “Plotting and Comparing Multiple Systems” and the bode entry in this section for more details.

When invoked with left-side arguments,

```
[y,t] = impulse(sys)
[y,t,x] = impulse(sys)       % for state-space models only
y = impulse(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 jth 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

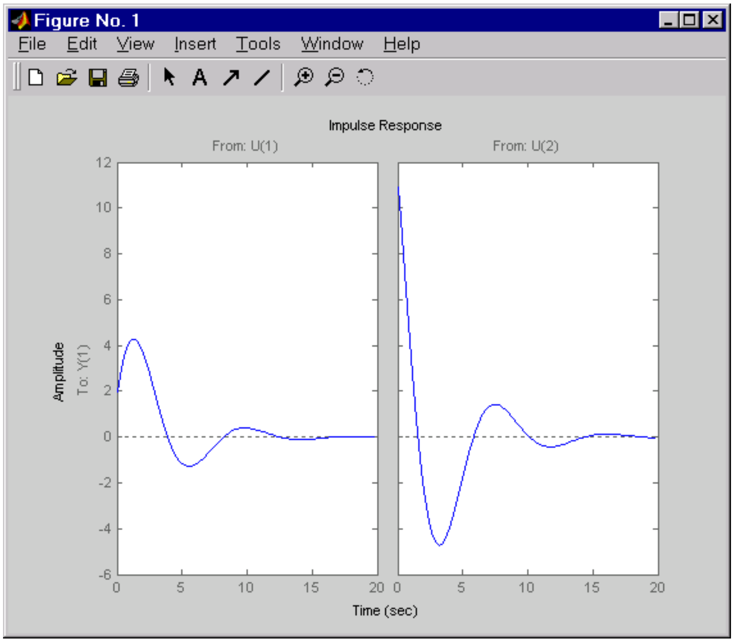
$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

use the following 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);
impulse(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

$$\dot{x} = Ax + bu$$
$$y = Cx$$

is equivalent to the following unforced response with initial state b.

$$\dot{x} = Ax\,, \qquad x(0) = b$$
$$y = Cx$$

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 $t = 0$. `impulse` ignores this discontinuity and returns the lower continuity value Cb at $t = 0$.

See Also

`ltiview`	LTI system viewer
`step`	Step response
`initial`	Free response to initial condition
`lsim`	Simulate response to arbitrary inputs

initial

Purpose Compute the initial condition response of state-space models

Syntax

```
initial(sys,x0)
initial(sys,x0,t)

initial(sys1,sys2,...,sysN,x0)
initial(sys1,sys2,...,sysN,x0,t)
initial(sys1,'PlotStyle1',...,sysN,'PlotStyleN',x0)

[y,t,x] = initial(sys,x0)
```

Description

`initial` calculates the unforced response of a state-space model with an initial condition on the states.

$$\dot{x} = Ax\ , \qquad x(0) = x_0$$
$$y = Cx$$

This function is applicable to either continuous- or discrete-time models. When invoked without left-side arguments, `initial` plots the initial condition response on the screen.

`initial(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.

`initial(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 `impulse`), 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 `impulse` for details).

When invoked with left-side 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

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

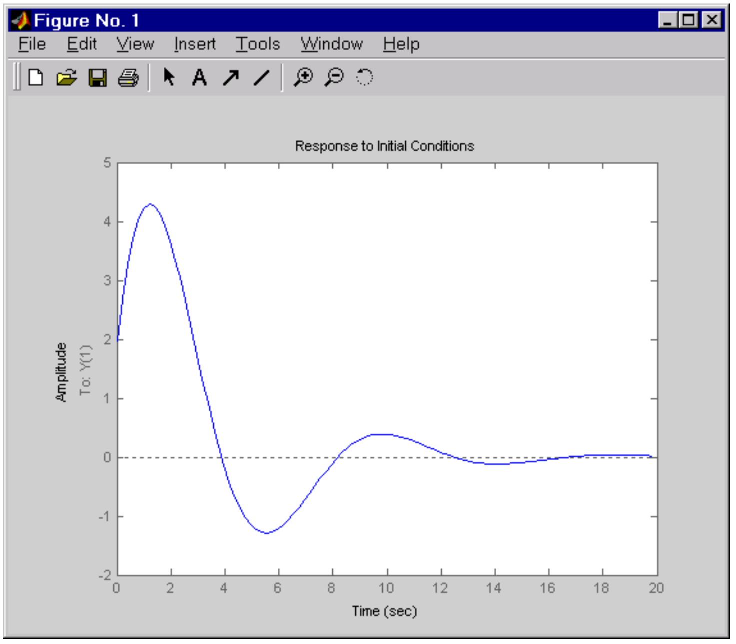
$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

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

`impulse`	Impulse response
`lsim`	Simulate response to arbitrary inputs
`ltiview`	LTI system viewer
`step`	Step response

Purpose Interpolate an FRD model between frequency points

Syntax `isys = interp(sys,freqs)` interpolates the frequency response data contained in the FRD model `sys` at the frequencies `freqs`. `interp`, which is an overloaded version of the MATLAB function `interp`, uses linear interpolation and returns an FRD model `isys` containing the interpolated data at the new frequencies `freqs`.

You should express the frequency values `freqs` in the same units as `sys.frequency`. The frequency values must lie between the smallest and largest frequency points in `sys` (extrapolation is not supported).

freqresp	Frequency response of LTI models
ltimodels	Help on LTI models

inv

Purpose Invert LTI systems

Syntax

```
isys = inv(sys)
```

Description

`inv` inverts the input/output relation

$$y = G(s)u$$

to produce the LTI system with the 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. `inv` 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 = inv(H)
```

to invert it. 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
```

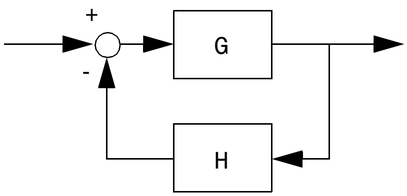
You can verify that

```
H * Hi
```

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

```
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])
cloop = inv(1+g*h) * g
```

yields a third-order closed-loop model with an unstable pole-zero cancellation at s = 1.

```
cloop

Zero/pole/gain:
      s (s-1)
-------------------
(s-1) (s^2 + s + 1)
```

Use feedback to avoid such pitfalls.

```
cloop = feedback(g,h)

Zero/pole/gain:
      s
-------------
(s^2 + s + 1)
```

iopzmap

Purpose

Plot pole-zero maps for I/O pairs of LTI models

Syntax

```
iopzmap(sys)
iopzmap(sys1,sys2,...)
```

Description

`iopzmap(sys)` computes and plots the poles and zeros of each input/output pair of the LTI model `sys`. The poles are plotted as x's and the zeros are plotted as o's.

`iopzmap(sys1,sys2,...)` shows the poles and zeros of multiple LTI models `sys1`,`sys2`,... on a single plot. You can specify distinctive colors for each model, as in `iopzmap(sys1,'r',sys2,'y',sys3,'g')`.

The functions `sgrid` or `zgrid` can be used to plot lines of constant damping ratio and natural frequency in the *s* or *z* plane.

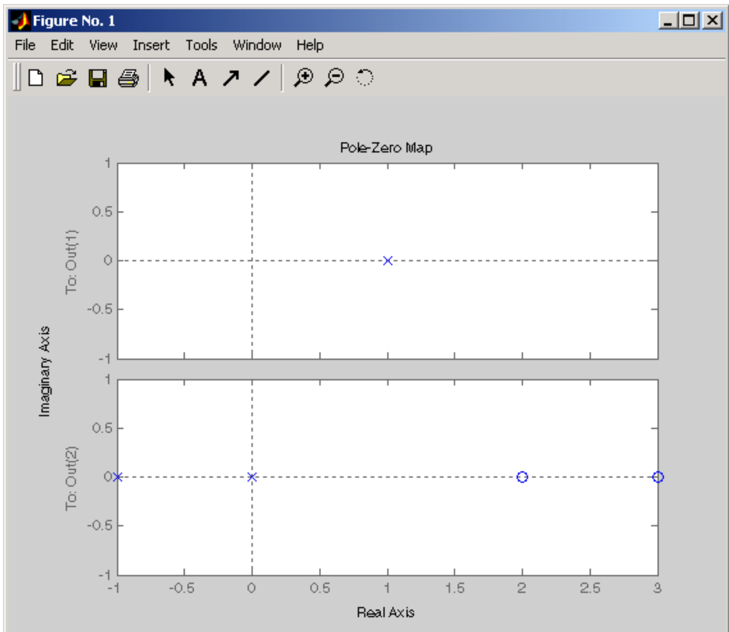
For arrays sys of LTI models, `iopzmap` plots the poles and zeros o each model in the array on the same diagram.

Example

Create a one-input, two-output system and plot pole-zero maps for I/O pairs.

```
H = [tf(-5 ,[1 -1]); tf([1 -5 6],[1 1 0])];
```

```
iopzmap(H)
```



See Also

`pzmap`	Pole-zero map
`pole`	Compute system poles
`zero`	Compute system zeros
`sgrid`	Grid for *s*-plane plots
`zgrid`	Grid for *z*-plane plots
`ltimodels`	Information about LTI models

Purpose Determine whether an LTI model is continuous or discrete

Syntax

```
boo = isct(sys)
boo = isdt(sys)
```

Description `boo = isct(sys)` returns `1` (true) if the LTI model `sys` is continuous and `0` (false) otherwise. `sys` is continuous if its sample time is zero, that is, `sys.Ts=0`.

`boo = isdt(sys)` returns `1` (true) if `sys` 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

```
isct(tf(10))
isdt(tf(10))
```

are true. However, if you explicitly label a gain as discrete, for example, by typing

```
g = tf(10,'ts',0.01)
```

`isct(g)` now returns false and only `isdt(g)` is true.

See Also

`isa`	Determine LTI model type
`isempty`	True for empty LTI models
`isproper`	True for proper LTI models

Purpose

Test if an LTI model is empty

Syntax

```
boo = isempty(sys)
```

Description

`isempty(sys)` returns `1` (true) if the LTI model `sys` has no input or no output, and `0` (false) otherwise.

Example

Both commands

```
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).

See Also

`issiso`	True for SISO systems
`size`	I/O dimensions and array dimensions of LTI models

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

```
boo = issiso(sys)
```

Description `issiso(sys)` returns `1` (true) if the LTI model `sys` is SISO and `0` (false) otherwise.

See Also

`isempty`	True for empty LTI models
`size`	I/O dimensions and array dimensions of LTI models

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

$$\dot{x} = Ax + Bu + Gw \qquad \text{(state equation)}$$
$$y_v = Cx + Du + Hw + v \qquad \text{(measurement equation)}$$

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

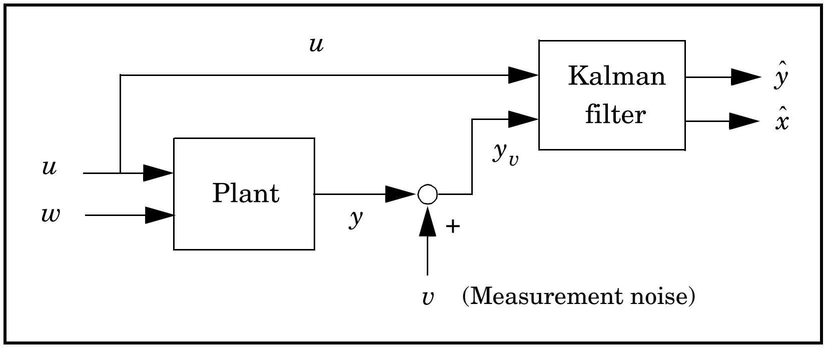
$$P = \lim_{t \to \infty} E(\{x - \hat{x}\}\{x - \hat{x}\}^T)$$

The optimal solution is the Kalman filter with equations

$$\dot{\hat{x}} = A\hat{x} + Bu + L(y_v - C\hat{x} - Du)$$
$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} \hat{x} + \begin{bmatrix} D \\ 0 \end{bmatrix} u$$

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

$$y = Cx + Du + Hw$$



Kalman estimator

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 $y[n|n]$ and $x[n|n]$ 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

```
[kest,L,P,M,Z] = kalman(sys,Qn,Rn,Nn)
```

returns the filter gain L and innovations gain M, as well as the steady-state error covariances

$$P = \lim_{n \to \infty} E(e[n|n-1]e[n|n-1]^T), \qquad e[n|n-1] = x[n] - x[n|n-1]$$

$$Z = \lim_{n \to \infty} E(e[n|n]e[n|n]^T), \qquad e[n|n] = x[n] - x[n|n]$$

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 “LQG Design for the x-Axis” and “Kalman Filtering” for examples that use the kalman function.

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

care	Solve continuous-time Riccati equations
dare	Solve discrete-time Riccati equations
estim	Form estimator given estimator gain
kalmd	Discrete Kalman estimator for continuous plant
lqgreg	Assemble LQG regulator
lqr	Design state-feedback LQ regulator

References

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

kalmd

Purpose

Design discrete Kalman estimator for continuous plant

Syntax

```
[kest,L,P,M,Z] = kalmd(sys,Qn,Rn,Ts)
```

Description

`kalmd` designs a discrete-time Kalman estimator that has response characteristics similar to a continuous-time estimator designed with `kalman`. This command is useful to derive a discrete estimator for digital implementation after a satisfactory continuous estimator has been designed.

`[kest,L,P,M,Z] = kalmd(sys,Qn,Rn,Ts)` produces a discrete Kalman estimator `kest` with sample time `Ts` for the continuous-time plant

$$\begin{aligned} \dot{x} &= Ax + Bu + Gw \qquad &\text{(state equation)} \\ y_v &= Cx + Du + v \qquad &\text{(measurement equation)} \end{aligned}$$

with process noise w and measurement noise v satisfying

$$E(w) = E(v) = 0, \quad E(ww^T) = Q_n, \quad E(vv^T) = R_n, \quad E(wv^T) = 0$$

The estimator `kest` is derived as follows. The continuous plant `sys` is first discretized using zero-order hold with sample time `Ts` (see `c2d` entry), and the continuous noise covariance matrices Q_n and R_n are replaced by their discrete equivalents

$$\begin{aligned} Q_d &= \int_0^{T_s} e^{A\tau} G Q G^T e^{A^T\tau} d\tau \\ R_d &= R / T_s \end{aligned}$$

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 `kalman` for details on discrete-time Kalman estimation.

`kalmd` also returns the estimator gains `L` and `M`, and the discrete error covariance matrices `P` and `Z` (see `kalman` for details).

Limitations

The discretized problem data should satisfy the requirements for `kalman`.

See Also

`kalman`	Design Kalman estimator

`lqgreg`	Assemble LQG regulator
`lqrd`	Discrete LQ-optimal gain for continuous plant

References

[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

Redheffer star product (linear fractional transformation) of two LTI models

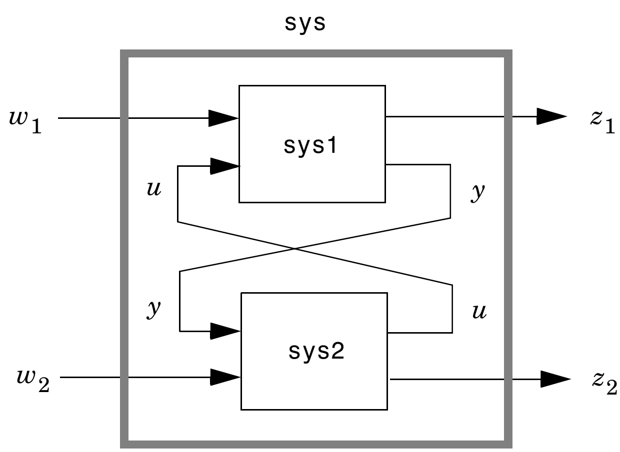
Syntax

```
sys = lft(sys1,sys2)
sys = lft(sys1,sys2,nu,ny)
```

Description

`lft` forms the star product or linear fractional transformation (LFT) of two LTI models or LTI arrays. Such interconnections are widely used in robust control techniques.

`sys = lft(sys1,sys2,nu,ny)` forms the star product `sys` of the two LTI models (or LTI arrays) `sys1` and `sys2`. The star product amounts to the following feedback connection for single LTI models (or for each model in an LTI array).



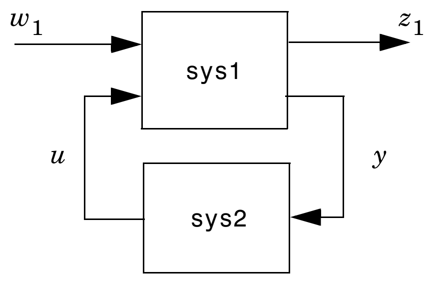
This feedback loop connects the first `nu` outputs of `sys2` to the last `nu` inputs of `sys1` (signals u), and the last `ny` outputs of `sys1` to the first `ny` inputs of `sys2` (signals y). The resulting system `sys` maps the input vector $[w_1 ; w_2]$ to the output vector $[z_1 ; z_2]$.

The abbreviated syntax

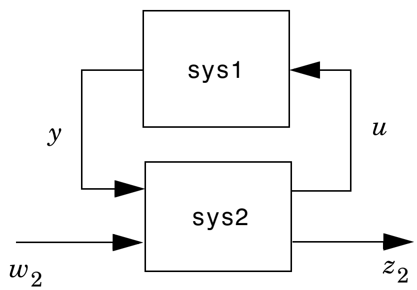
```
sys = lft(sys1,sys2)
```

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

`connect`	Derive state-space model for block diagram interconnection
`feedback`	Feedback connection

lqgreg

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.

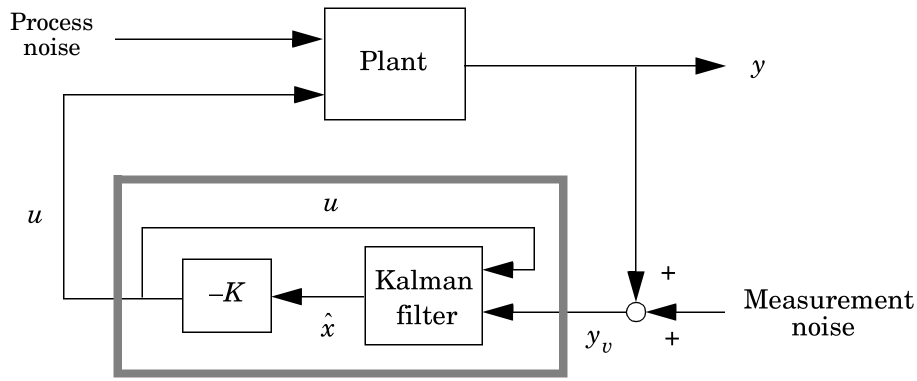
In continuous time, the LQG regulator generates the commands

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

where $\hat{x}$ is the Kalman state estimate. The regulator state-space equations are

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

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.



LQG regulator

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 `lqry` 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

$$u[n] = -K\hat{x}[n|n]$$

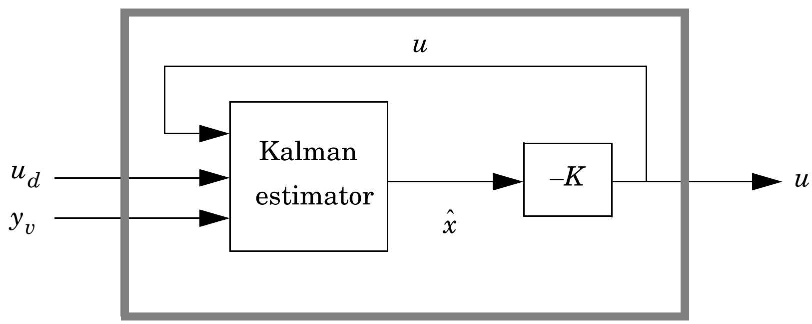
the syntax

```
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.



LQG regulator

Example

See the example LQG Regulation.

See Also

`kalman`	Kalman estimator design
`kalmd`	Discrete Kalman estimator for continuous plant
`lqr`, `dlqr`	State-feedback LQ regulator
`lqrd`	Discrete LQ regulator for continuous plant
`lqry`	LQ regulator with output weighting
`reg`	Form regulator given state-feedback and estimator gains

Purpose

Design linear-quadratic (LQ) state-feedback regulator for continuous plant

Syntax

```
[K,S,e] = lqr(A,B,Q,R)
[K,S,e] = lqr(A,B,Q,R,N)
```

Description

`[K,S,e] = lqr(A,B,Q,R,N)` calculates the optimal gain matrix `K` such that the state-feedback law $u = -Kx$

minimizes the quadratic cost function

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

for the continuous-time state-space model $\dot{x} = Ax + Bu$

The default value `N=0` is assumed when `N` is omitted.

In addition to the state-feedback gain `K`, `lqr` returns the solution `S` of the associated Riccati equation

$$A^T S + SA - (SB + N)R^{-1}(B^T S + N^T) + Q = 0$$

and the closed-loop eigenvalues `e = eig(A-B*K)`. Note that K is derived from S by

$$K = R^{-1}(B^T S + N^T)$$

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 imaginary axis.

See Also

`care`	Solve continuous Riccati equations
`dlqr`	State-feedback LQ regulator for discrete plant
`lqgreg`	Form LQG regulator
`lqrd`	Discrete LQ regulator for continuous plant
`lqry`	State-feedback LQ regulator with output weighting

Purpose

Design discrete LQ regulator for continuous plant

Syntax

```
[Kd,S,e] = lqrd(A,B,Q,R,Ts)
[Kd,S,e] = lqrd(A,B,Q,R,N,Ts)
```

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.

`[Kd,S,e] = lqrd(A,B,Q,R,Ts)` 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 `Ts` 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 = eig(Ad-Bd*Kd)`.

`[Kd,S,e] = lqrd(A,B,Q,R,N,Ts)` 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 `Kd` is determined by discretizing the continuous plant and weighting matrices using the sample time `Ts` and the zero-order hold approximation.

With the notation

$$\Phi(\tau) = e^{A\tau}, \qquad A_d = \Phi(T_s)$$

$$\Gamma(\tau) = \int_0^{\tau} e^{A\eta} B d\eta, \qquad B_d = \Gamma(T_s)$$

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 `dlqr`.

Limitations

The discretized problem data should meet the requirements for `dlqr`.

See Also

`c2d`	Discretization of LTI model
`dlqr`	State-feedback LQ regulator for discrete plant
`kalmd`	Discrete Kalman estimator for continuous plant
`lqr`	State-feedback LQ regulator for continuous plant

References

[1] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems*, Second Edition, Addison-Wesley, 1980, pp. 439–440

[2] Van Loan, C.F., “Computing Integrals Involving the Matrix Exponential,” *IEEE Trans. Automatic Control*, 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

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

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 LQG Design for the x-Axis for an example.

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
`lqgreg`	Form LQG regulator

lsim

Purpose Simulate LTI model response to arbitrary inputs

Syntax

```
lsim(sys,u,t)
lsim(sys,u,t,x0)
lsim(sys,u,t,x0,'zoh')
lsim(sys,u,t,x0,'foh')

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 left-hand 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:Tfinal
```

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. If `dt` is too large (undersampling), `lsim` issues a warning suggesting that you use a more appropriate sample time, but will use the specified sample time. See Algorithm on page 132 for a discussion of sample times.

`lsim(sys,u,t,x0)` further specifies an initial condition `x0` for the system states. This syntax applies only to state-space models.

`lsim(sys,u,t,x0,'zoh')` or `lsim(sys,u,t,x0,'foh')` explicitly specifies how the input values should be interpolated between samples (zero-order hold or linear interpolation). By default, `lsim` selects the interpolation method automatically based on the smoothness of the signal U.

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 left-hand 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 on page 132).

Example

Simulate and plot the response of the system

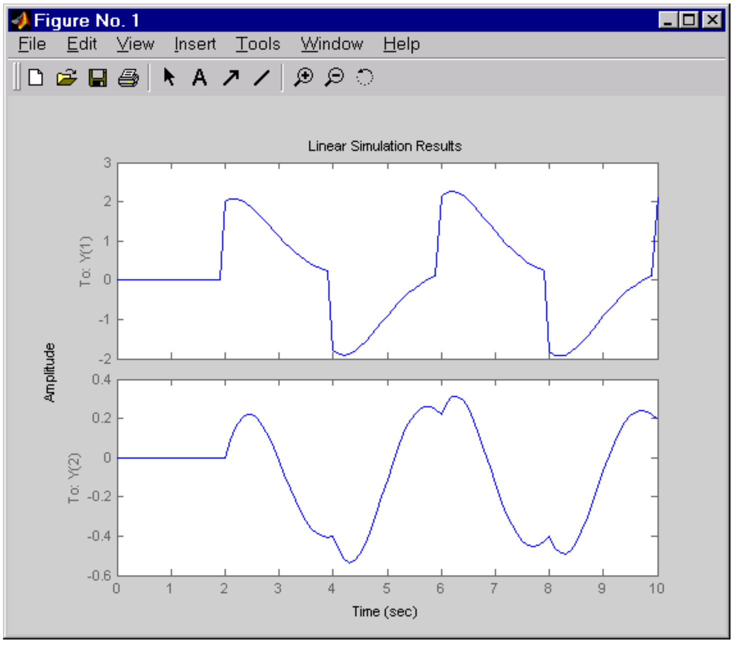
$$H(s) = \begin{bmatrix} \dfrac{2s^2 + 5s + 1}{s^2 + 2s + 3} \\ \dfrac{s - 1}{s^2 + s + 5} \end{bmatrix}$$

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). The sampling period is set to the spacing `dt` between the user-supplied time samples `t`.

The choice of sampling period can drastically affect simulation results. To illustrate why, consider the second-order model

$$H(s) = \frac{\omega^2}{s^2 + 2s + \omega^2}, \qquad \omega = 62.83$$

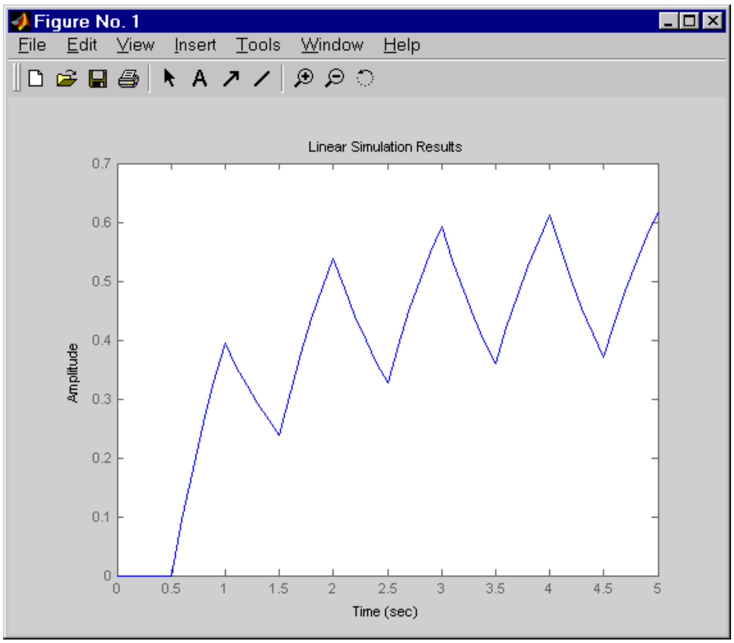
To simulate its response to a square wave with period 1 second, you can proceed as follows:

```
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)
```

`lsim` evaluates the specified sample time, gives this warning

```
Warning: Input signal is undersampled. Sample every 0.016 sec or
faster.
```

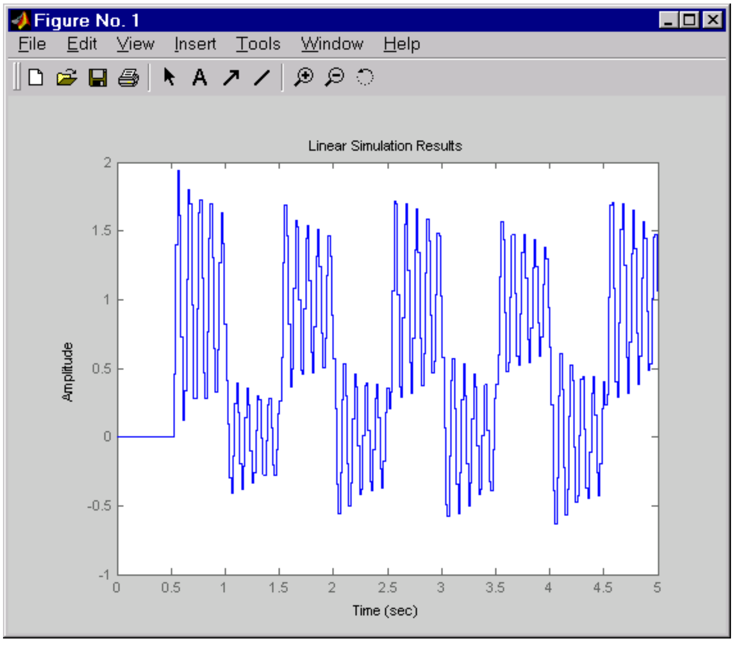
and produces this plot.



To improve on this response, discretize $H(s)$ using the recommended sampling period:

```
dt=0.016;
ts=0:dt:5;
us = (rem(ts,1)>=0.5)
hd = c2d(h,dt)
```

```
lsim(hd,us,ts)
```



This response exhibits strong oscillatory behavior hidden from the undersampled version.

See Also

`gensig`	Generate test input signals for `lsim`
`impulse`	Impulse response
`initial`	Free response to initial condition
`ltiview`	LTI system viewer
`step`	Step response

Purpose Help on LTI models

Syntax

```
ltimodels
ltimodels(modeltype)
```

Description

`ltimodels` displays general information on the various types of LTI models supported in the Control System Toolbox.

`ltimodels(`*`modeltype`*`)` gives additional details and examples for each type of LTI model. The string *`modeltype`* selects the model type among the following:

- `tf` — Transfer functions (TF objects)
- `zpk` — Zero-pole-gain models (ZPK objects)
- `ss` — State-space models (SS objects)
- `frd` — Frequency response data models (FRD objects).

Note that you can type

```
ltimodels zpk
```

as a shorthand for

```
ltimodels('zpk')
```

See Also

`frd`	Create or convert to FRD models
`ltiprops`	Help on LTI model properties
`ss`	Create or convert to a state-space model
`tf`	Create or convert to a transfer function model
`zpk`	Create or convert to a zero/pole/gain model

ltiprops

Purpose Help on LTI model properties

Syntax

```
ltimodels
ltimodels(modeltype)
```

Description `ltiprops` displays details on the generic properties of LTI models.

`ltiprops(`*`modeltype`*`)` gives details on the properties specific to the various types of LTI models. The string *`modeltype`* selects the model type among the following:

- `tf` — transfer functions (TF objects)
- `zpk` — zero-pole-gain models (ZPK objects)
- `ss` — state-space models (SS objects)
- `frd` — frequency response data (FRD objects).

Note that you can type

```
ltiprops tf
```

as a shorthand for

```
ltiprops('tf')
```

See also

`get`	Get the properties for an LTI model
`ltimodels`	Help on LTI models
`set`	Set or modify LTI model properties

Purpose

Initialize an LTI Viewer for LTI system response analysis

Syntax

```
ltiview
ltiview(sys1,sys2,...,sysn)
ltiview('plottype',sys1,sys2,...,sysn)
ltiview('plottype',sys,extras)
ltiview('clear',viewers)
ltiview('current',sys1,sys2,...,sysn,viewers)
```

Description

`ltiview` when invoked without input arguments, initializes a new LTI Viewer for LTI system response analysis.

`ltiview(sys1,sys2,...,sysn)` opens an LTI Viewer containing the step response of the LTI models `sys1,sys2,...,sysn`. You can specify a distinctive color, line style, and marker for each system, as in

```
sys1 = rss(3,2,2);
sys2 = rss(4,2,2);
ltiview(sys1,'r-*',sys2,'m--');
```

`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 one of the following:

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

or,

`plottype` can be a cell vector containing up to six of these plot types. For example,

```
ltiview({'step';'nyquist'},sys)
```

displays the plots of both of these response types for a given system `sys`.

`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`.

`ltiview('clear',viewers)` clears the plots and data from the LTI Viewers with handles `viewers`.

`ltiview('current',sys1,sys2,...,sysn,viewers)` adds the responses of the systems `sys1,sys2,...,sysn` to the LTI Viewers with handles `viewers`. If these new systems do not have the same I/O dimensions as those currently in the LTI Viewer, the LTI Viewer is first cleared and only the new responses are shown.

Finally,

```
ltiview(plottype,sys1,sys2,...sysN)
ltiview(plottype,sys1,PlotStyle1,sys2,PlotStyle2,...)
ltiview(plottype,sys1,sys2,...sysN,extras)
```

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.

See Also

`bode`	Bode response
`impulse`	Impulse response
`initial`	Response to initial condition
`lsim`	Simulate LTI model response to arbitrary inputs

`nichols`	Nichols response
`nyquist`	Nyquist response
`pzmap`	Pole/zero map
`sigma`	Singular value response
`step`	Step response

lyap

Purpose

Solve continuous-time Lyapunov equations

Syntax

```
X = lyap(A,Q)
X = lyap(A,B,C)
```

Description

`lyap` 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.

`X = lyap(A,Q)` solves the Lyapunov equation

$$AX + XA^T + Q = 0$$

where A and Q are square matrices of identical sizes. The solution `X` is a symmetric matrix if Q is.

`X = lyap(A,B,C)` solves the generalized Lyapunov equation (also called Sylvester equation).

$$AX + XB + C = 0$$

The matrices A, B, C must have compatible dimensions but need not be square.

Algorithm

`lyap` 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

The continuous Lyapunov equation has a (unique) solution if the eigenvalues $\alpha_1, \alpha_2, ..., \alpha_n$ of A and $\beta_1, \beta_2, ..., \beta_n$ of B satisfy

$$\alpha_i + \beta_j \neq 0 \qquad \text{for all pairs } (i, j)$$

If this condition is violated, `lyap` produces the error message

```
Solution does not exist or is not unique.
```

See Also

`covar`	Covariance of system response to white noise
`dlyap`	Solve discrete Lyapunov equations

References

[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

```
[Gm,Pm,Wcg,Wcp] = margin(sys)
[Gm,Pm,Wcg,Wcp] = margin(mag,phase,w)
margin(sys)
```

Description

`margin` calculates the minimum 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 left-hand arguments, `margin` produces a Bode plot and displays the margins on this plot.

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.

`[Gm,Pm,Wcg,Wcp] = margin(sys)` computes the gain margin `Gm`, the phase margin `Pm`, and the corresponding crossover frequencies `Wcg` and `Wcp`, given the SISO open-loop model `sys`. This function handles both continuous- and discrete-time cases. When faced with several crossover frequencies, `margin` returns the smallest gain and phase margins.

`[Gm,Pm,Wcg,Wcp] = margin(mag,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.

When invoked without left-hand argument,

```
margin(sys)
```

plots the open-loop Bode response with the gain and phase margins marked by vertical lines.

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

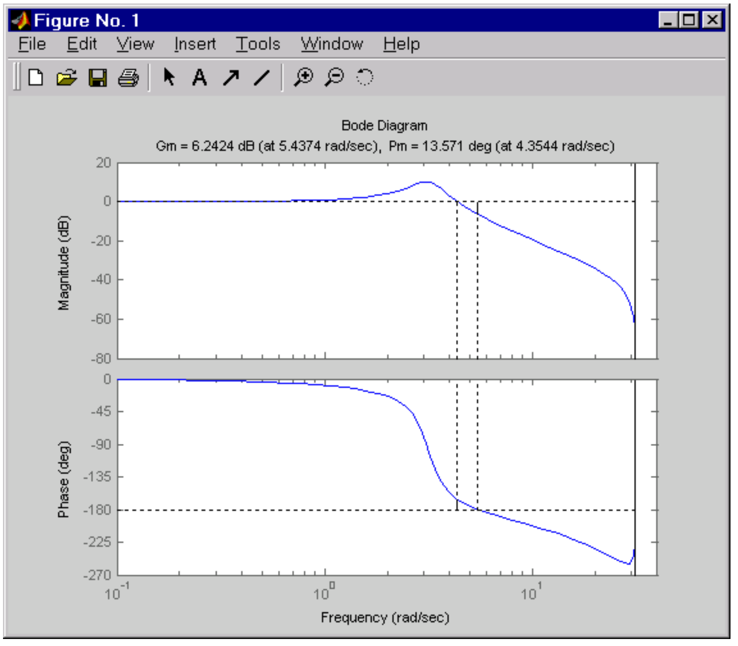
```
[Gm,Pm,Wcg,Wcp] = margin(hd);
[Gm,Pm,Wcg,Wcp]
```

and MATLAB returns

```
ans =
    2.0517   13.5711    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

`bode`	Bode frequency response
`ltiview`	LTI system viewer

Purpose Minimal realization or pole-zero cancellation

Syntax

```
sysr = minreal(sys)
sysr = minreal(sys,tol)
[sysr,u] = minreal(sys,tol)
```

Description `sysr = minreal(sys)` eliminates uncontrollable or unobservable state in state-space models, or cancels pole-zero pairs in transfer functions or zero-pole-gain models. The output `sysr` has minimal order and the same response characteristics as the original model `sys`.

`sysr = minreal(sys,tol)` specifies the tolerance used for state elimination or pole-zero cancellation. The default value is `tol = sqrt(eps)` and increasing this tolerance forces additional cancellations.

`[sysr,u] = minreal(sys,tol)` returns, for state-space model `sys`, an orthogonal matrix U such that `(U*A*U',U*B,C*U')` is a Kalman decomposition of `(A,B,C)`

Example The commands

```
g = zpk([],1,1)
h = tf([2 1],[1 0])
cloop = inv(1+g*h) * g
```

produce the nonminimal zero-pole-gain model by typing `cloop`.

```
Zero/pole/gain:
      s (s-1)
-------------------
(s-1) (s^2 + s + 1)
```

To cancel the pole-zero pair at $s = 1$, type

```
cloop = minreal(cloop)
```

and MATLAB returns

```
Zero/pole/gain:
      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.

See Also

`balreal`	Grammian-based input/output balancing
`modred`	Model order reduction
`sminreal`	Structured model reduction

Purpose

Model order reduction

Syntax

```
rsys = modred(sys,elim)
rsys = modred(sys,elim,'mdc')
rsys = modred(sys,elim,'del')
```

Description

`modred` reduces the order of a continuous or discrete state-space model `sys`. This function is usually used in conjunction with `balreal`. Two order reduction techniques are available:

- `rsys = modred(sys,elim)` or `rsys = modred(sys,elim,'mdc')` produces a reduced-order model `rsys` with matching DC gain (or equivalently, matching steady state in the step response). The index vector `elim` specifies the states to be eliminated. The resulting model `rsys` has `length(elim)` fewer states. This technique consists of setting the derivative of the eliminated states to zero and solving for the remaining states.
- `rsys = modred(sys,elim,'del')` simply deletes the states specified by `elim`. While this method does not guarantee matching DC gains, it tends to produce better approximations in the frequency domain (see example below).

If the state-space model sys has been balanced with `balreal` and the grammians have m small diagonal entries, you can reduce the model order by eliminating the last m states with `modred`.

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 `balreal` by typing

```
h = tf([1 11 36 26],[1 14.6 74.96 153.7 99.65])
[hb,g] = balreal(h)
g'
```

MATLAB returns

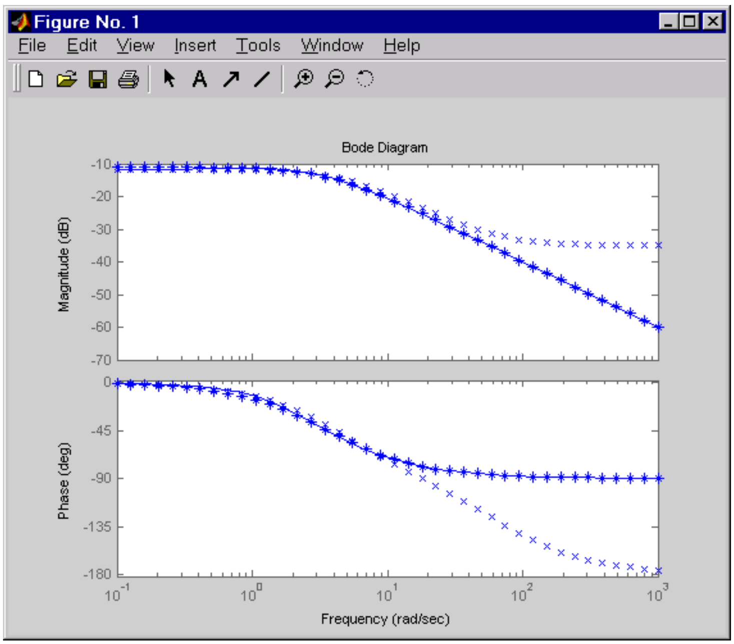
```
ans =
   1.3938e-01   9.5482e-03   6.2712e-04   7.3245e-06
```

The last three diagonal entries of the balanced grammians 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')
```

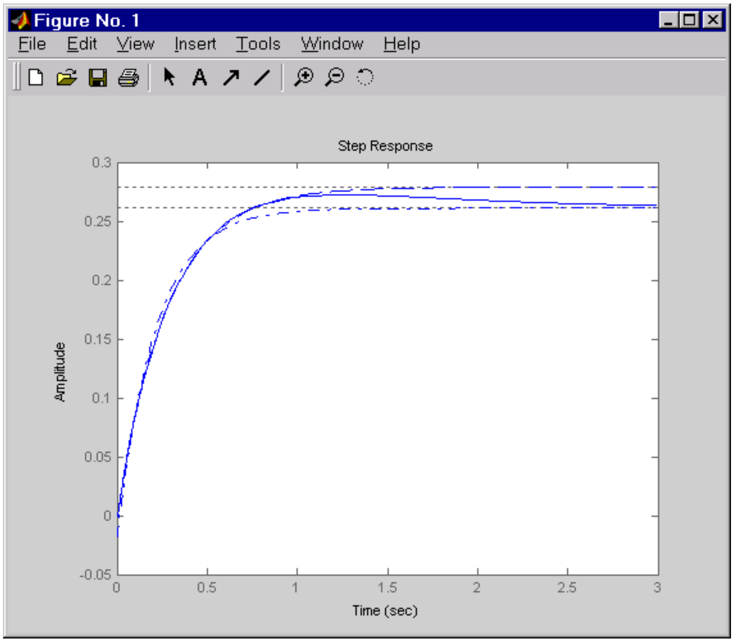
Both `hmdc` and `hdel` are first-order models. Compare their Bode responses against that of the original model $h(s)$.

```
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

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

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_1. The reduced-order model is given by

$$\dot{x}_1 = [A_{11} - A_{12}A_{22}^{-1}A_{21}]x_1 + [B_1 - A_{12}A_{22}^{-1}B_2]u$$
$$y = [C_1 - C_2A_{22}^{-1}A_{21}]x + [D - C_2A_{22}^{-1}B_2]u$$

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

`balreal`	Input/output balancing of state-space models
`minreal`	Minimal state-space realizations

Purpose

Provide the number of the dimensions of an LTI model or LTI array

Syntax

```
n = ndims(sys)
```

Description

`n = ndims(sys)` is the number of dimensions of an LTI model or an array of LTI models `sys`. A single LTI model has two dimensions (one for outputs, and one for inputs). An LTI array has 2+p dimensions, where $p \geq 2$ is the number of array dimensions. For example, a 2-by-3-by-4 array of models has 2+3=5 dimensions.

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

Example

```
sys = rss(3,1,1,3);
ndims(sys)

ans =
     4
```

`ndims` returns 4 for this 3-by-1 array of SISO models.

See Also

`size`	Returns a vector containing the lengths of the dimensions of an LTI array or model

ngrid

Purpose Superimpose a Nichols chart on a Nichols plot

Syntax

```
ngrid
```

Description `ngrid` 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.

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

$$\frac{H}{1+H}$$

is the corresponding value of the closed-loop frequency response assuming unit negative feedback.

If the current axis is empty, `ngrid` 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, `ngrid` returns a warning.

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])
```

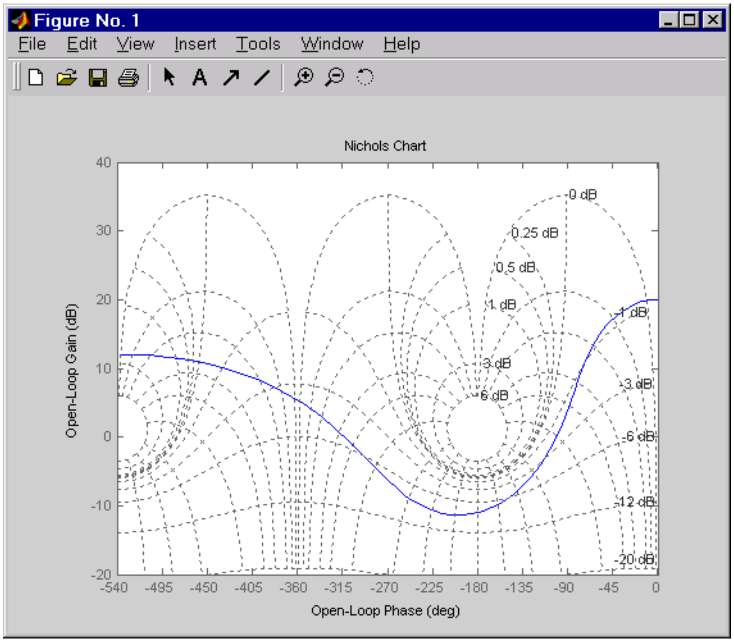
MATLAB returns

```
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

nichols

Purpose

Compute Nichols frequency response of LTI models

Syntax

```
nichols(sys)
nichols(sys,w)

nichols(sys1,sys2,...,sysN)
nichols(sys1,sys2,...,sysN,w)
nichols(sys1,'PlotStyle1',...,sysN,'PlotStyleN')

[mag,phase,w] = nichols(sys)
[mag,phase] = nichols(sys,w)
```

Description

`nichols` computes the frequency response of an LTI model 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 `ngrid` to superimpose a Nichols chart on an existing SISO Nichols plot.

`nichols(sys)` produces a Nichols plot of the LTI model `sys`. This model can be continuous or discrete, SISO or MIMO. In the MIMO case, `nichols` 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.

`nichols(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. Frequencies should be specified in radians/sec.

`nichols(sys1,sys2,...,sysN)` or `nichols(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

```
nichols(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
```

See `bode` for an example.

When invoked with left-hand 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 the `bode` reference page). They have dimensions

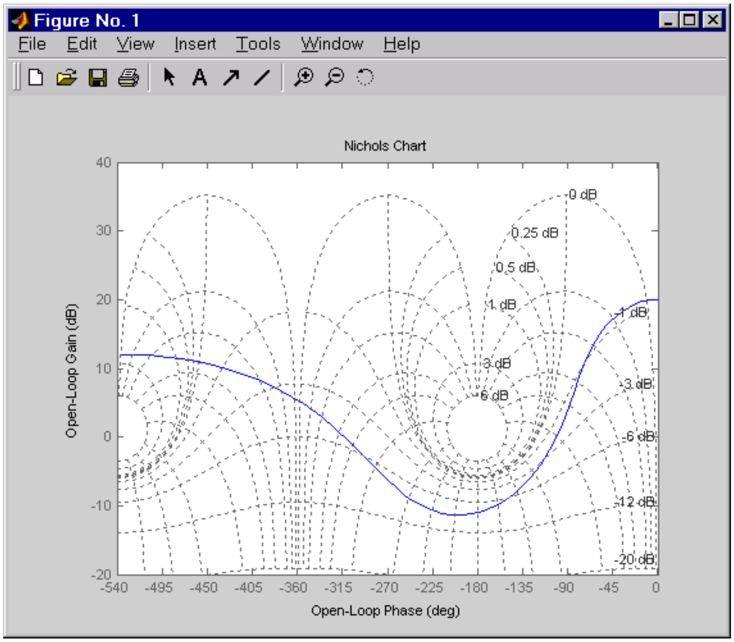
$$(\text{number of outputs}) \times (\text{number of inputs}) \times (\text{length of } \mathtt{w})$$

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); ngrid
```



The right-click menu for Nichols plots includes the **Tight** option under **Zoom**. You can use this to clip unbounded branches of the Nichols plot.

Algorithm

See `bode`.

See Also

`bode`	Bode plot
`evalfr`	Response at single complex frequency
`freqresp`	Frequency response computation
`ltiview`	LTI system viewer
`ngrid`	Grid on Nichols plot
`nyquist`	Nyquist plot
`sigma`	Singular value plot

Purpose Compute LTI model norms

Syntax

```
norm(sys)
norm(sys,2)

norm(sys,inf)
norm(sys,inf,tol)
[ninf,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

$$\|H\|_2 = \sqrt{\frac{1}{2\pi}\int_{-\infty}^{\infty} \mathrm{Trace}(H(j\omega)^H H(j\omega))\, d\omega}$$

This norm measures the steady-state covariance (or power) of the output response $y = Hw$ to unit white noise inputs w.

$$\|H\|_2^2 = \lim_{t\to\infty} E\{y(t)^T y(t)\}\ , \qquad E(w(t)w(\tau)^T) = \delta(t-\tau)I$$

Infinity Norm

The infinity norm is the peak gain of the frequency response, that is,

$$\|H(s)\|_\infty = \max_{\omega} |H(j\omega)| \qquad \text{(SISO case)}$$

$$\|H(s)\|_\infty = \max_{\omega} \sigma_{max}(H(j\omega)) \qquad \text{(MIMO case)}$$

where $\sigma_{max}(.)$ 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)` both return the H_2 norm of the TF, SS, or ZPK 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 any type of LTI model `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`).

`[ninf,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)

ans =
    1.2438
```

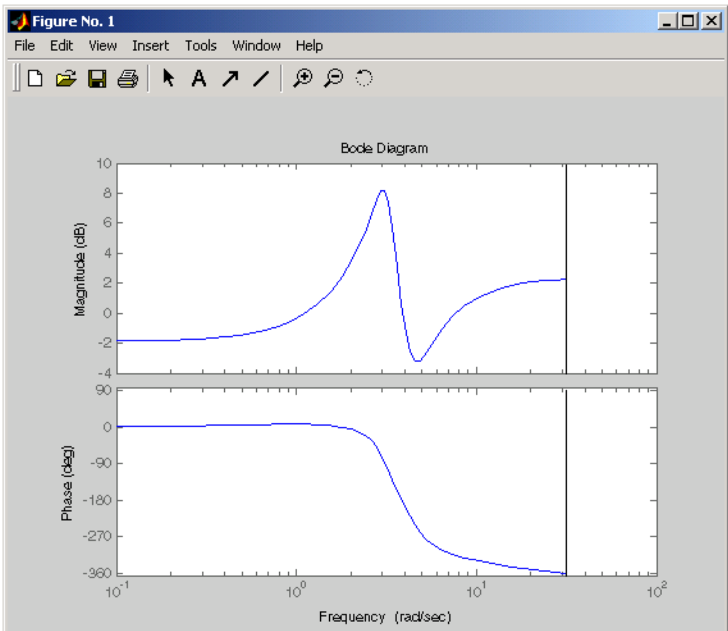
Compute its infinity norm by typing

```
[ninf,fpeak] = norm(H,inf)
```

```
ninf =
    2.5488

fpeak =
    3.0844
```

These values are confirmed by the Bode plot of $H(z)$.

```
bode(H)
```



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
freqresp	Frequency response computation
sigma	Singular value plot

References

[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.

Purpose

Compute Nyquist frequency response of LTI models

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)
[re,im] = nyquist(sys,w)
```

Description

`nyquist` calculates the Nyquist frequency response of LTI models. When invoked without left-hand 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, 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 left-hand 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 (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 $\omega_k = \text{w(k)}$.

$$\text{re(1,1,k)} = \text{Re}(h(j\omega_k))$$
$$\text{im(1,1,k)} = \text{Im}(h(j\omega_k))$$

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,

$$\text{re(i,j,k)} = \text{Re}(h_{ij}(j\omega_k))$$
$$\text{im(i,j,k)} = \text{Im}(h_{ij}(j\omega_k))$$

where h_{ij} is the transfer function from input j to output i.

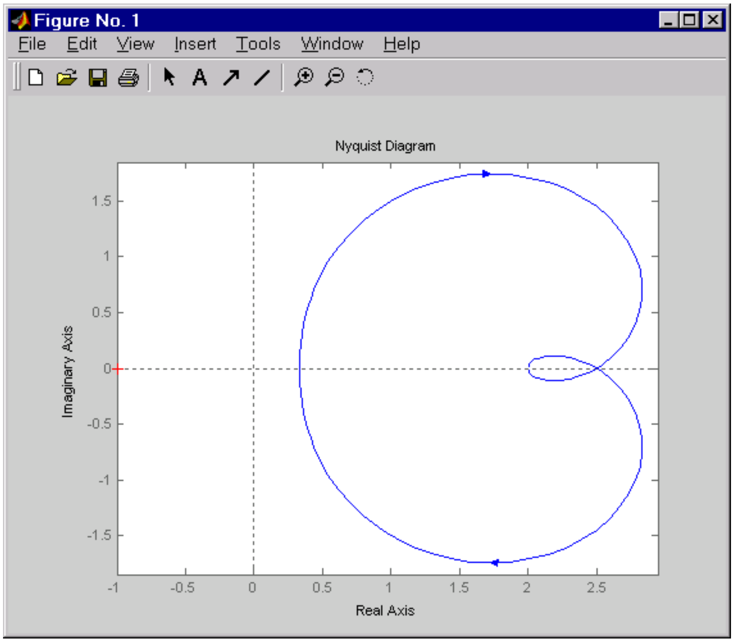
Example

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)
```



The nyquist function has support for M-circles, which are the contours of the constant closed-loop magnitude. M-circles are defined as the locus of complex numbers where

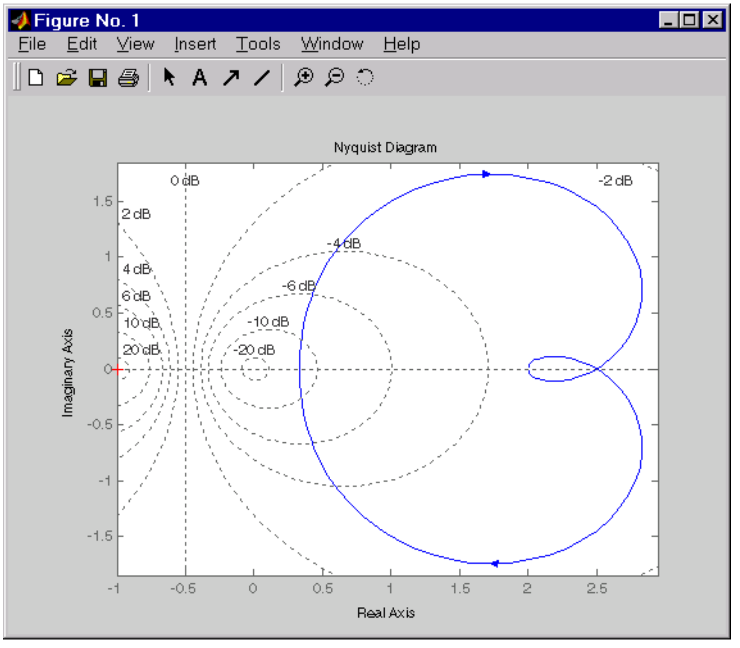
$$T(j\omega) = \left|\frac{G(j\omega)}{1 + G(j\omega)}\right|$$

is a constant value. In this equation, ω is the frequency in radians/second, and G is the collection of complex numbers that satisfy the constant magnitude requirement.

To activate the grid, select **Grid** from the right-click menu or type

```
grid
```

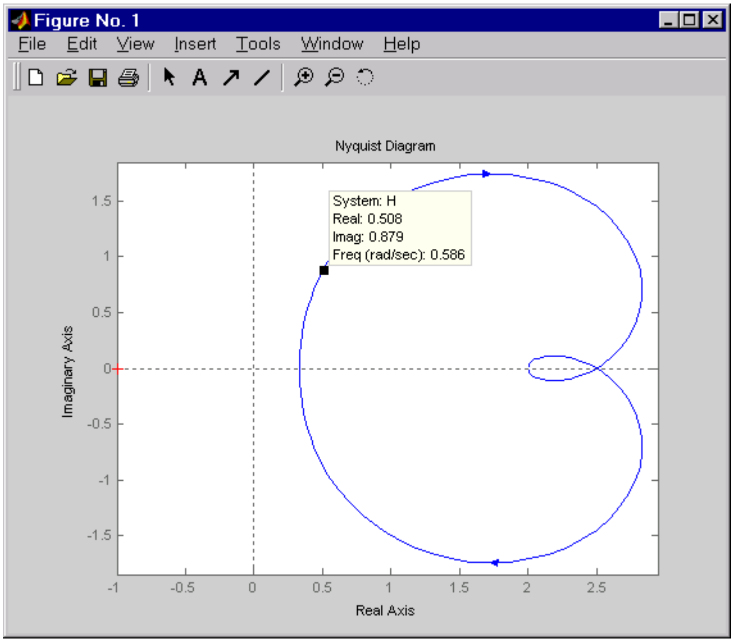
at the MATLAB prompt. This figure shows the M circles for transfer function *H*.



You have two zoom options available from the right-click menu that apply specifically to Nyquist plots:

- **Tight** —Clips unbounded branches of the Nyquist plot, but still includes the critical point (-1, 0)
- **On (-1,0)** — Zooms around the critical point (-1,0)

Also, click anywhere on the curve to activate data markers that display the real and imaginary values at a given frequency. This figure shows the nyquist plot with a data marker.



See Also

`bode`	Bode plot
`evalfr`	Response at single complex frequency
`freqresp`	Frequency response computation
`ltiview`	LTI system viewer
`nichols`	Nichols plot
`sigma`	Singular value plot

obsv

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

$$Ob = \begin{bmatrix} C \\ CA \\ CA^2 \\ : \\ CA^{n-1} \end{bmatrix}$$

with n columns and np rows.

`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 =
     1     1
     4    -2

C =
     1     0
     0     1
```

is observable. Type

```
Ob = obsv(A,C);

% Number of unobservable states
unob = length(A)-rank(Ob)
```

MATLAB responds with

```
unob =
     0
```

See Also

`obsvf`	Compute the observability staircase form

obsvf

Purpose

Compute the observability staircase form

Syntax

```
[Abar,Bbar,Cbar,T,k] = obsvf(A,B,C)
[Abar,Bbar,Cbar,T,k] = obsvf(A,B,C,tol)
```

Description

If the observability matrix of (A,C) has rank $r \le n$, where n is the size of A, then there exists a similarity transformation such that

$$\bar{A} = TAT^T, \qquad \bar{B} = TB, \qquad \bar{C} = CT^T$$

where T is unitary and the transformed system has a *staircase* form with the unobservable modes, if any, in the upper left corner.

$$\bar{A} = \begin{bmatrix} A_{no} & A_{12} \\ 0 & A_o \end{bmatrix}, \qquad \bar{B} = \begin{bmatrix} B_{no} \\ B_o \end{bmatrix}, \qquad \bar{C} = \begin{bmatrix} 0 & C_o \end{bmatrix}$$

where (C_o, A_o) is observable, and the eigenvalues of A_{no} are the unobservable modes.

`[Abar,Bbar,Cbar,T,k] = 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 sum(k) is the number of states in A_o, the observable portion of Abar.

`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*norm(a,1)*eps`.

Example

Form the observability staircase form of

```
A =
     1     1
     4    -2

B =
```

```
        1    -1
        1    -1

C =
        1     0
        0     1
```

by typing

```
[Abar,Bbar,Cbar,T,k] = obsvf(A,B,C)

Abar =
        1     1
        4    -2
Bbar =
        1     1
        1    -1
Cbar =
        1     0
        0     1
T =
        1     0
        0     1
k =
        2     0
```

Algorithm

`obsvf` is an M-file that implements the Staircase Algorithm of [1] by calling `ctrbf` and using duality.

See Also

`ctrbf`	Compute the controllability staircase form
`obsv`	Calculate the observability matrix

References

[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 `wn` (ω_n) and damping factor `z` (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 To 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)

num =
     1
den =
    1.0000    1.9200    5.7600

sys = tf(num,den)

Transfer function:
         1
------------------
s^2 + 1.92 s + 5.76
```

See Also

`rss`	Generate random stable continuous models
`ss`	Create a state-space LTI model
`tf`	Create a transfer function LTI model

Purpose

Compute the Padé approximation of models with time delays

Syntax

```
[num,den] = pade(T,N)
pade(T,N)

sysx = pade(sys,N)
sysx = pade(sys,NI,NO,Nio)
```

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 an time delay of T seconds is $\exp(-sT)$. This exponential transfer function is approximated by a rational transfer function using the Padé approximation formulas [1].

[num,den] = pade(T,N) returns the Nth-order (diagonal) Padé approximation of the continuous-time I/O delay $\exp(-sT)$ in transfer function form. The row vectors num and den contain the numerator and denominator coefficients in descending powers of s. Both are Nth-order polynomials.

When invoked without output arguments,

```
pade(T,N)
```

plots the step and phase responses of the Nth-order Padé approximation and compares them with the exact responses of the model with I/O delay T. Note that the Padé approximation has unit gain at all frequencies.

sysx = pade(sys,N) produces a delay-free approximation sysx of the continuous delay system sys. All delays are replaced by their Nth-order Padé approximation. See Time Delays for details on LTI models with delays.

sysx = pade(sys,NI,NO,Nio) specifies independent approximation orders for each input, output, and I/O delay. These approximation orders are given by the arrays of integers NI, NO, and Nio, such that:

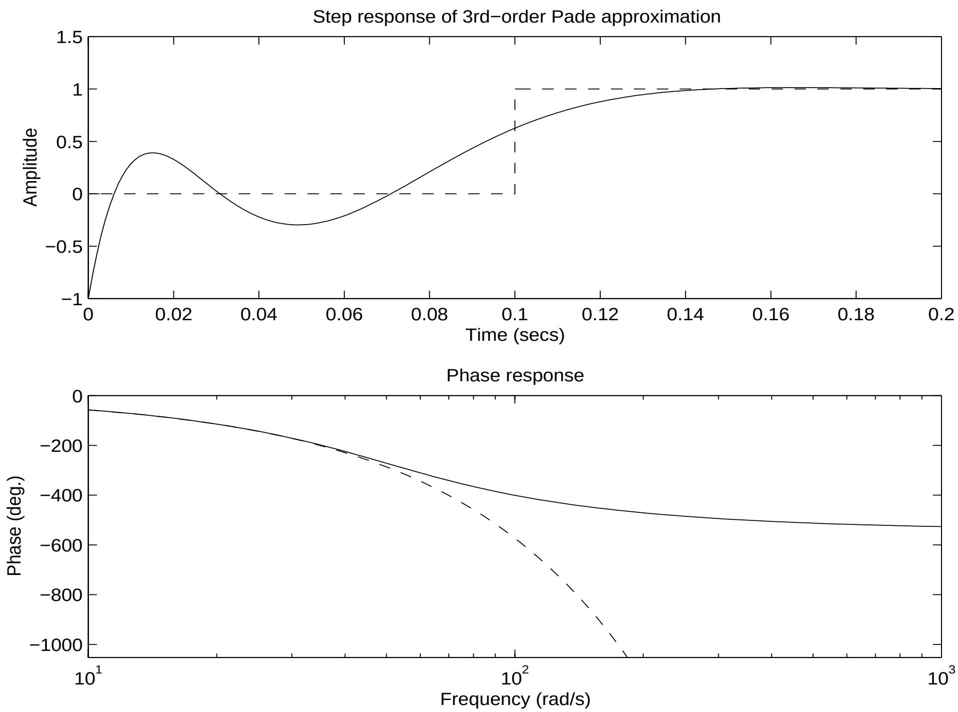
- NI(j) is the approximation order for the j-th input channel.
- NO(i) is the approximation order for the i-th output channel.
- Nio(i,j) is the approximation order for the I/O delay from input j to output i.

You can use scalar values to specify uniform approximation orders, and [] if there are no input, output, or I/O delays.

Example

Compute a third-order Padé approximation of a 0.1 second I/O delay and compare the time and frequency responses of the true delay and its approximation. To do this, type

```
pade(0.1,3)
```



Limitations

High-order Padé approximations produce transfer functions with clustered poles. Because such pole configurations tend to be very sensitive to perturbations, Padé approximations with order N>10 should be avoided.

See Also

c2d	Discretization of continuous system

`delay2z`	Changes transfer functions of discrete-time LTI models with delays to rational functions or absorbs FRD delays into the frequency response phase information

References

[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

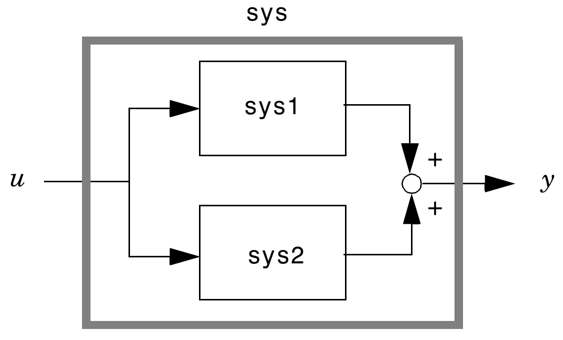
Syntax

```
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.

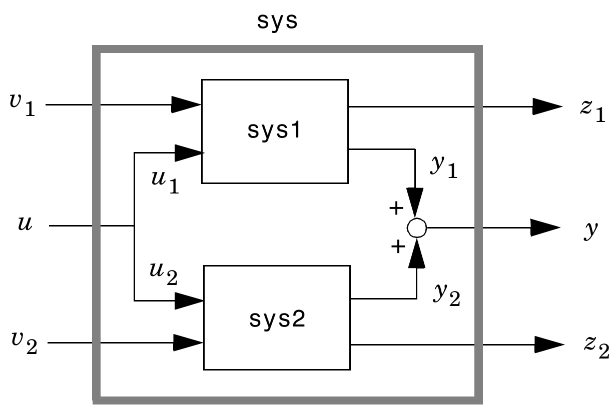


This command is equivalent to the direct addition

```
sys = sys1 + sys2
```

(See Addition and Subtraction for details on LTI system addition.)

`sys = parallel(sys1,sys2,inp1,inp2,out1,out2)` forms the more general parallel connection.



The index vectors `inp1` and `inp2` specify which inputs u_1 of `sys1` and which inputs u_2 of `sys2` are connected. Similarly, the index vectors `out1` and `out2` 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 Kalman Filtering for an example.

See Also

`append`	Append LTI systems
`feedback`	Feedback connection
`series`	Series connection

place

Purpose Pole placement design

Syntax

```
K = place(A,B,p)
[K,prec,message] = 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 = 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,prec,message] = 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 = 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 23 5.0]` by

```
p = [1 1.23 5.0];
K = place(a,b,p)
```

Algorithm

`place` uses the algorithm of [1] which, for multi-input systems, optimizes the choice of eigenvectors for a robust solution. We recommend `place` 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

`acker`	Pole placement using Ackermann's formula
`lqr`	State-feedback LQ regulator design
`rlocus`	Root locus design

References

[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

```
p = pole(sys)
```

Description `pole` computes the poles `p` of the SISO or MIMO LTI model `sys`.

Algorithm 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.

For SISO transfer functions or zero-pole-gain models, the poles are simply the denominator roots (see `roots`).

For MIMO transfer functions (or zero-pole-gain models), the poles are computed as the union of the poles for each SISO entry. If some columns or rows have a common denominator, the roots of this denominator are counted only once.

Limitations 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

$$\rho \approx \varepsilon^{1/m}$$

where ε is the relative machine precision (`eps`).

See Also

`damp`	Damping and natural frequency of system poles
`esort`, `dsort`	Sort system poles
`pzmap`	Pole-zero map
`zero`	Compute (transmission) zeros

Purpose Compute the pole-zero map of an LTI model

Syntax

```
pzmap(sys)
pzmap(sys1,sys2,...,sysN)
[p,z] = pzmap(sys)
```

Description `pzmap(sys)` plots the pole-zero map of the continuous- or discrete-time LTI model `sys`. For SISO systems, `pzmap` 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.

`pzmap(sys1,sys2,...,sysN)` plots the pole-zero map of several LTI models on a single figure. The LTI models can have different numbers of inputs and outputs and can be a mix of continuous and discrete systems.

When invoked with left-hand arguments,

```
[p,z] = pzmap(sys)
```

returns the system poles and (transmission) zeros in the column vectors `p` and `z`. No plot is drawn on the screen.

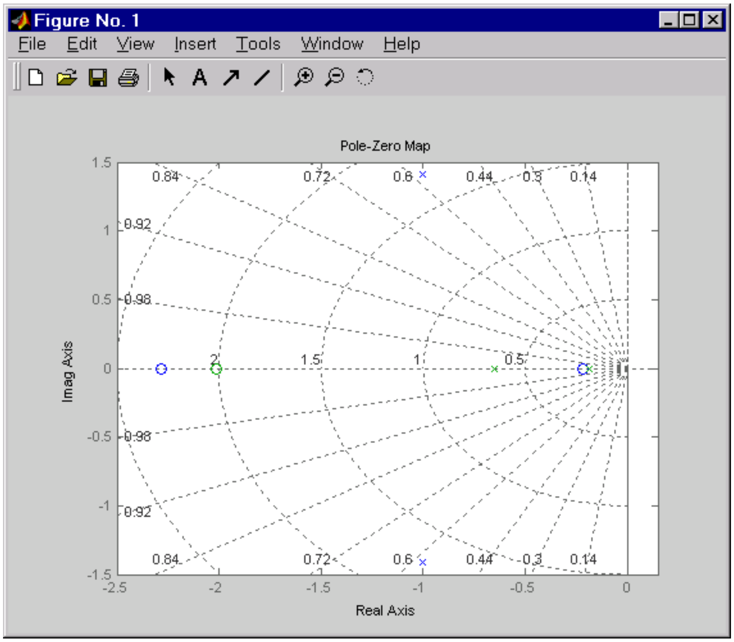
You can use the functions `sgrid` or `zgrid` to plot lines of constant damping ratio and natural frequency in the s - or z -plane.

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]); sgrid
```

```
pzmap(H)
```



Algorithm

`pzmap` uses a combination of `pole` and `zero`.

See Also

`damp`	Damping and natural frequency of system poles
`esort`, `dsort`	Sort system poles
`pole`	Compute system poles
`rlocus`	Root locus
`sgrid`, `zgrid`	Plot lines of constant damping and natural frequency
`zero`	Compute system (transmission) zeros

Purpose Form regulator given state-feedback and estimator gains

Syntax

```
rsys = reg(sys,K,L)
rsys = reg(sys,K,L,sensors,known,controls)
```

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.

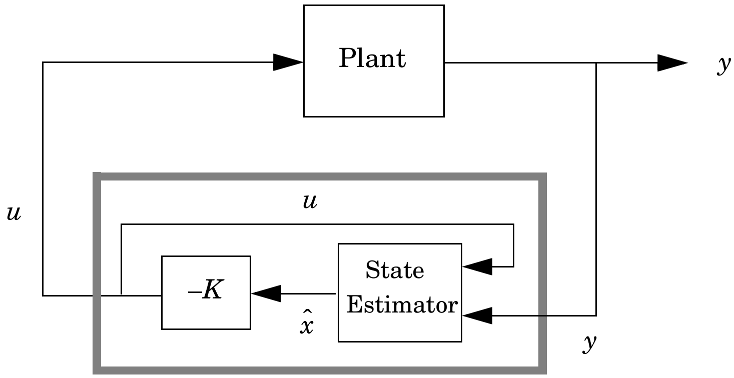
This syntax assumes that all inputs of `sys` are controls, and all outputs are measured. The regulator `rsys` is obtained by connecting the state-feedback law $u = -Kx$ and the state estimator with gain matrix `L` (see `estim`). For a plant with equations

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

this yields the regulator

$$\dot{\hat{x}} = \left[A - LC - (B - LD)K\right]\hat{x} + Ly$$
$$u = -K\hat{x}$$

This regulator should be connected to the plant using *positive* feedback.

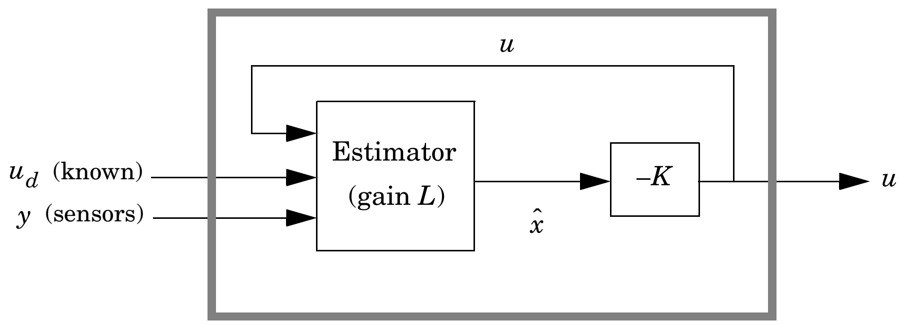


Regulator

`rsys = reg(sys,K,L,sensors,known,controls)` 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 `controls` 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).



Regulator `rsys`

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

`estim`	Form state estimator given estimator gain
`kalman`	Kalman estimator design
`lqgreg`	Form LQG regulator
`lqr`, `dlqr`	State-feedback LQ regulator
`place`	Pole placement

reshape

Purpose Change the shape of an LTI array

Syntax

```
sys = reshape(sys,s1,s2,...,sk)
sys = reshape(sys,[s1 s2 ... sk])
```

Description

`sys = reshape(sys,s1,s2,...,sk)` (or, equivalently, `sys = reshape(sys,[s1 s2 ... sk])`) reshapes the LTI array sys into an `s1`-by-`s2`-by...-`sk` array of LTI models. Equivalently, `sys = reshape(sys,[s1 s2 ... sk])` reshapes the LTI array sys into an `s1`-by-`s2`-by...-`sk` array of LTI models. With either syntax, there must be `s1*s2*...*sk` models in sys to begin with.

Example

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

2x3 array of state-space models
Each model has 1 output, 1 input, and 4 states.

sys1 = reshape(sys,6);
size(sys1)

6x1 array of state-space models
Each model has 1 output, 1 input, and 4 states.
```

See Also

`ndims`	Provide the number of dimensions of an LTI array
`size`	Provide the lengths of each dimension of an LTI array

Purpose

Evans root locus

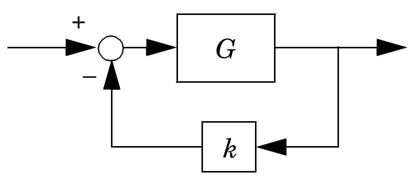
Syntax

```
rlocus(sys)
rlocus(sys,k)
rlocus(sys1,sys2,...)

[r,k] = rlocus(sys)
r = rlocus(sys,k)
```

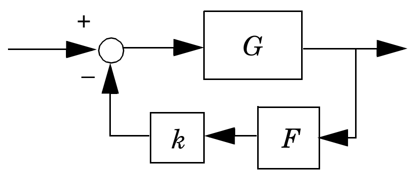
Description

`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.

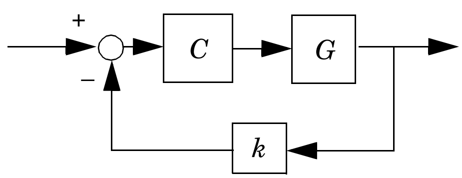
`rlocus(sys)` calculates and plots the root locus of the open-loop SISO model `sys`. This function can be applied to any of the following *negative* feedback loops by setting `sys` appropriately.



`sys = G`



`sys = F * G`



`sys = G * C`

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.

`rlocus(sys1,sys2,...)` draws the root loci of multiple LTI models `sys1`, `sys2,...` on a single plot. You can specify a color, line style, and marker for each model, as in

```
rlocus(sys1,'r',sys2,'y:',sys3,'gx').
```

When invoked with output 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)`.

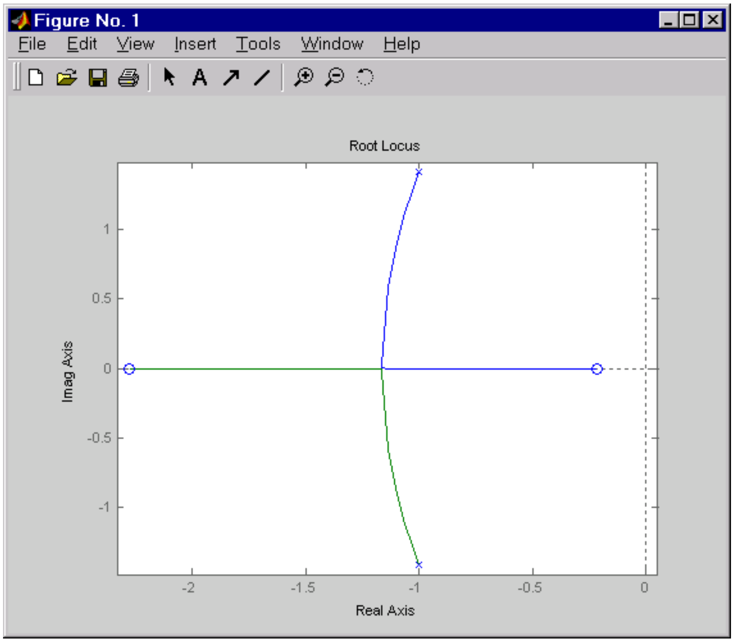
Example

Find and plot the root-locus of the following system.

$$h(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

```
h = tf([2 5 1],[1 2 3]);
```

```
rlocus(h)
```



You can use the right-click menu for rlocus to add grid lines, zoom in or out, and invoke the Property Editor to customize the plot. Also, click anywhere on the curve to activate a data marker that displays the gain value, pole, damping, overshoot, and frequency at the selected point.

See Also

`pole`	System poles
`pzmap`	Pole-zero map

Purpose Generate stable random continuous test models

Syntax

```
sys = rss(n)
sys = rss(n,p)
sys = rss(n,p,m)
sys = rss(n,p,m,s1,...,sn)
```

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 `nth` 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.

`rss(n,p,m,s1,...,sn)`produces an `s1`-by-`...`-by-`sn` array of random n-th order stable state-space models with `m` inputs and `p` outputs.

Use `tf`, `frd`, or `zpk` to convert the state-space object `sys` to transfer function, frequency response, or zero-pole-gain form.

Example Obtain a stable random continuous LTI model with three states, two inputs, and two outputs by typing

```
sys = rss(3,2,2)

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

`drss`	Generate stable random discrete test models
`frd`	Convert LTI systems to frequency response form
`tf`	Convert LTI systems to transfer function form
`zpk`	Convert LTI systems to zero-pole-gain form

series

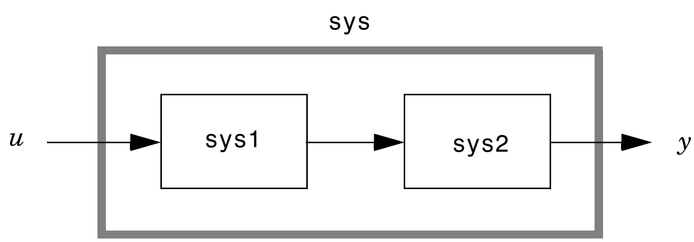
Purpose Series connection of two LTI models

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.

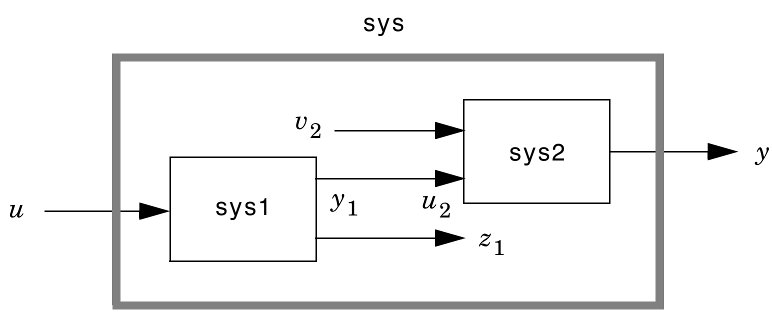


This command is equivalent to the direct multiplication

```
sys = sys2 * sys1
```

See Multiplication for details on multiplication of LTI models.

`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,outputs1,inputs2)
```

See Also

append	Append LTI systems
feedback	Feedback connection
parallel	Parallel 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 "LTI Properties" 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 "Model-Specific Properties" for details).

`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,'inputd',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)

        a = 1
          b = 2
          c = 3
          d = 0
          e = []
          Nx = 1
          StateName = {''}
          Ts = 0
          InputDelay = 0.1
          OutputDelay = 0
          ioDelay = 0
          InputName = {'torque'}
          OutputName = {''}
          InputGroup = {0x2 cell}
          OutputGroup = {0x2 cell}
          Notes = {}
          UserData = -6
```

Property Values

The following table lists the admissible values for each LTI property. N_u and N_y denotes the number of inputs and outputs of the underlying LTI model. For K-dimensional LTI arrays, let $S_1, S_2, \ldots, S_K$ denote the array dimensions.

Table 4-2: LTI Properties

Property Name	Admissible Property Values
Ts	• 0 (zero) for continuous-time systems • Sample time in seconds for discrete-time systems • -1 or [] for discrete systems with unspecified sample time **Note:** Resetting the sample time property does not alter the model data. Use c2d, d2c, or d2d for discrete/continuous and discrete/discrete conversions.
ioDelay	Input/Output delays specified with • Nonnegative real numbers for continuous-time models (seconds) • Integers for discrete-time models (number of sample periods) • Scalar when all I/O pairs have the same delay • N_y-by-N_u matrix to specify independent delay times for each I/O pair • Array of size N_y-by-N_u-by-S_1-by-...-by-S_n to specify different I/O delays for each model in an LTI array.
InputDelay	Input delays specified with • Nonnegative real numbers for continuous-time models (seconds) • Integers for discrete-time models (number of sample periods) • Scalar when $N_u = 1$ or system has uniform input delay • Vector of length N_u to specify independent delay times for each input channel • Array of size N_y-by-N_u-by-S_1-by-...-by-S_n to specify different input delays for each model in an LTI array.

Table 4-2: LTI Properties (Continued)

Property Name	Admissible Property Values
`OutputDelay`	Output delays specified with • Nonnegative real numbers for continuous-time models (seconds) • Integers for discrete-time models (number of sample periods) • Scalar when $N_y = 1$ or system has uniform output delay • Vector of length N_y to specify independent delay times for each output channel • Array of size N_y-by-N_u-by-S_1-by-. . .-by-S_n to specify different output delays for each model in an LTI array.
`Notes`	String, array of strings, or cell array of strings
`UserData`	Arbitrary MATLAB variable
`InputName`	• 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")
`InputGroup`	Cell array. See "Input Groups and Output Groups."
`OutputGroup`	Same as `InputGroup`

Table 4-3: State-Space Model Properties

Property Name	Admissible Property Values
`StateName`	Same as `InputName` (with `Input` replaced by `State`)
`a`, `b`, `c`, `d`, `e`	Real- or complex-valued state-space matrices (multidimensional arrays, in the case of LTI arrays) with compatible dimensions for the number of states, inputs, and outputs. See "The Size of LTI Array Data for SS Models."
`Nx`	• Scalar integer representing the number of states for single LTI models or LTI arrays with the same number of states in each model • S_1-by-...-by-S_K-dimensional array of integers when all of the models of an LTI array do not have the same number of states

Table 4-4: TF Model Properties

Property Name	Admissible Property Values
`num`, `den`	• Real- or complex-valued row vectors for the coefficients of the numerator or denominator polynomials in the SISO case. List the coefficients in *descending* powers of the variable s or z by default, and in *ascending* powers of $q = z^{-1}$ when the `Variable` property is set to `'q'` or `'z^-1'` (see note below). • N_y-by-N_u cell arrays of real- or complex-valued row vectors in the MIMO case, for example, `{[1 2];[1 0 3]}` for a two-output/one-input transfer function • N_y-by-N_u-by-S_1-by-...-by-S_K-dimensional real- or complex-valued cell arrays for MIMO LTI arrays
`Variable`	• String `'s'` (default) or `'p'` for continuous-time systems • String `'z'` (default), `'q'`, or `'z^-1'` for discrete-time systems

Table 4-5: ZPK Model Properties

Property Name	Admissible Property Values
`z, p`	• Vectors of zeros and poles (either real- or complex-valued) in SISO case • N_y-by-N_u cell arrays of vectors (entries are real- or complex valued) in MIMO case, for example, `z = {[],[-1 0]}` for a model with two inputs and one output • N_y-by-N_u-by-S_1-by-…-by-S_K-dimensional cell arrays for MIMO LTI arrays
`Variable`	• String `'s'` (default) or `'p'` for continuous-time systems • String `'z'` (default), `'q'`, or `'z^-1'` for discrete-time systems

Table 4-6: FRD Model Properties

Property Name	Admissible Property Values
`Frequency`	Real-valued vector of length N_f-by-1, where N_f is the number of frequencies
`Response`	• N_y-by-N_u-by-N_f-dimensional array of complex data for single LTI models • N_y-by-N_u-by-N_f-by-S_1-by-…-by-S_K-dimensional array for LTI arrays
`Units`	String `'rad/s'` (default), or `'Hz'`

Remark

For discrete-time transfer functions, the convention used to represent the numerator and denominator depends on the choice of variable (see the `tf` entry for details). Like `tf`, the syntax for `set` changes to remain consistent with the choice of variable. For example, if the `Variable` 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:

$$\bar{h}(z^{-1}) = \frac{1+2z^{-1}}{1+3z^{-1}+4z^{-2}} = zh(z)$$

Note 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
`frd`	Specify a frequency response data model
`ss`	Specify a state-space model
`tf`	Specify a transfer function
`zpk`	Specify a 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, for pole-zero and root locus plots, 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.

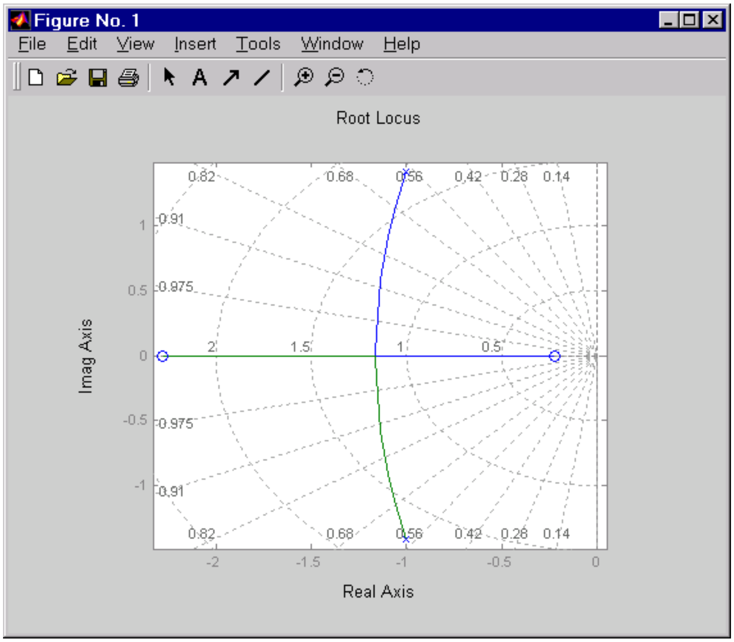
Alternatively, you can select **Grid** from the right-click menu to generate the same s-plane grid.

Example Plot s-plane grid lines on the root locus for the following system.

$$H(s) = \frac{2s^2 + 5s + 1}{s^2 + 2s + 3}$$

You can do this by typing

```
H = tf([2 5 1],[1 2 3])

Transfer function:
2 s^2 + 5 s + 1
---------------
 s^2 + 2 s + 3

rlocus(H)
sgrid
```



See Also

`pzmap`	Plot pole-zero map
`rlocus`	Plot root locus
`zgrid`	Generate *z*-plane grid lines

Purpose

Singular values of the frequency response of LTI models

Syntax

```
sigma(sys)
sigma(sys,w)
sigma(sys,w,type)

sigma(sys1,sys2,...,sysN)
sigma(sys1,sys2,...,sysN,w)
sigma(sys1,sys2,...,sysN,w,type)
sigma(sys1,'PlotStyle1',...,sysN,'PlotStyleN')

[sv,w] = sigma(sys)
sv = sigma(sys,w)
```

Description

`sigma` calculates the singular values of the frequency response of an LTI model. For an FRD model, `sys`, `sigma` computes the singular values of `sys.Response` at the frequencies, `sys.frequency`. For continuous-time TF, SS, or ZPK models with transfer function $H(s)$, `sigma` computes the singular values of $H(j\omega)$ as a function of the frequency ω. For discrete-time TF, SS, or ZPK models with transfer function $H(z)$ and sample time T_s, `sigma` 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 values of the frequency response extend the Bode magnitude response for MIMO systems and are useful in robustness analysis. The singular value response of a SISO system is identical to its Bode magnitude response. When invoked without output arguments, `sigma` produces a singular value plot on the screen.

`sigma(sys)` plots the singular values of the frequency 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, or from `sys.frequency` if `sys` is an FRD.

`sigma(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 corresponding vector of frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. The frequencies must be specified in rad/sec.

`sigma(sys,[],type)` or `sigma(sys,w,type)` plots the following modified singular value responses:

`type = 1`	Singular values of the frequency response H^{-1}, where H is the frequency response of `sys`.
`type = 2`	Singular values of the frequency response $I + H$.
`type = 3`	Singular values of the frequency response $I + H^{-1}$.

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
```

The models `sys1,sys2,...,sysN` need not have the same number of inputs and outputs. Each model can be either continuous- or discrete-time. 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 output 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

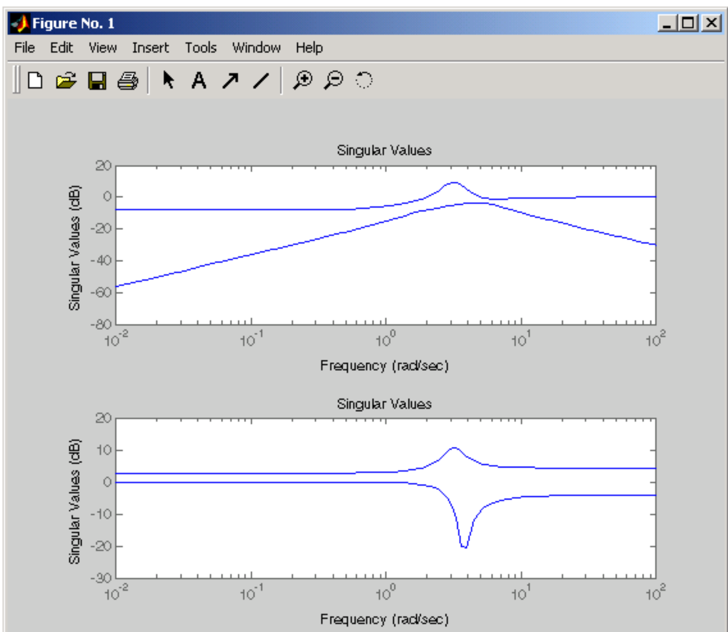
Plot the singular value responses of

$$H(s) = \begin{bmatrix} 0 & \dfrac{3s}{s^2+s+10} \\ \dfrac{s+1}{s+5} & \dfrac{2}{s+6} \end{bmatrix}$$

and $I + H(s)$.

You can do this by typing

```
H = [0 tf([3 0],[1 1 10]) ; tf([1 1],[1 5]) tf(2,[1 6])]

subplot(211)
sigma(H)
subplot(212)
sigma(H,[],2)
```



Algorithm

`sigma` uses the `svd` function in MATLAB to compute the singular values of a complex matrix.

See Also

`bode`	Bode plot
`evalfr`	Response at single complex frequency
`freqresp`	Frequency response computation
`ltiview`	LTI system viewer
`nichols`	Nichols plot
`nyquist`	Nyquist plot

Purpose Initialize the SISO Design Tool

Syntax

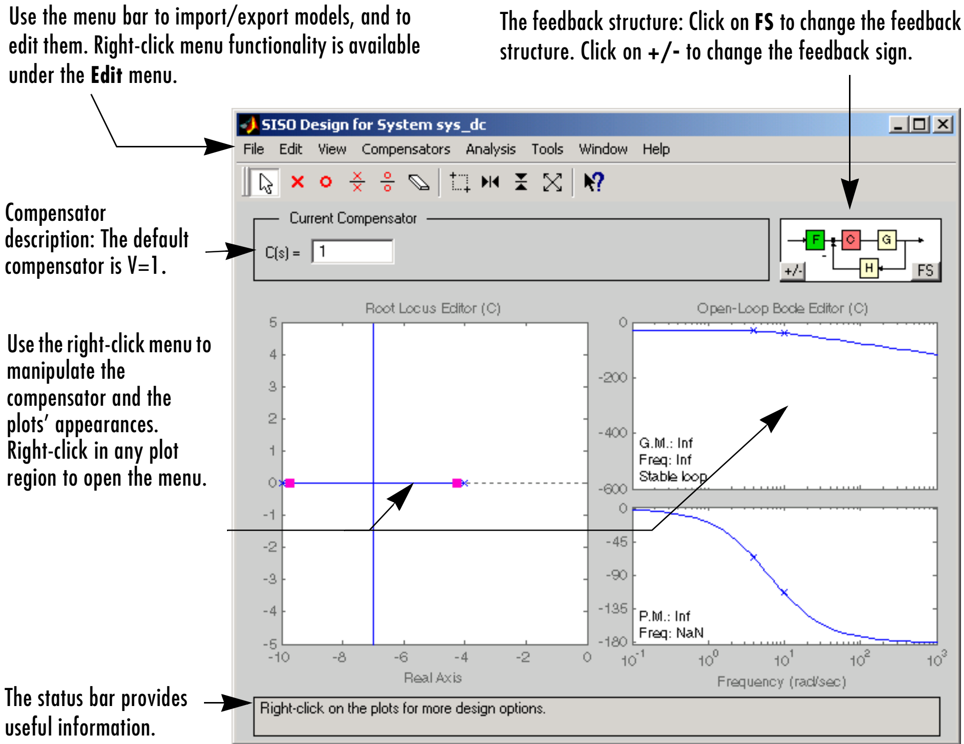
```
sisotool
sisotool(plant)
sisotool(plant,comp)
sisotool(views)
sisotool(views,plant,comp,sensor,prefilt)
sisotool(views,plant,comp,options)
```

Description When invoked without input arguments, `sisotool` opens a SISO Design GUI for interactive compensator design. This GUI allows you to design a single-input/single-output (SISO) compensator using root locus and Bode diagram techniques.

By default, the SISO Design Tool:

- Opens root locus and open-loop Bode diagrams.
- Places the compensator, **C**, in the forward path in series with the plant, **G**.
- Assumes the prefilter, **F**, and the sensor, **H**, are unity gains. Once you specify **G** and **H**, they are *fixed* in the feedback structure.

This picture shows the SISO Design Tool.



`sisotool(plant)` opens the SISO Design Tool, imports `plant`, and initializes the plant model **G** to `plant`. The workspace variable `plant` can be any SISO LTI model created with either `ss`, `tf`, or `zpk`.

`sisotool(plant,comp)` initializes the plant model **G** to `plant`, the compensator **C** to `comp`.

`sisotool(plant,comp,sensor,prefilt)` initializes the plant **G** to `plant`, compensator C to `comp`, sensor H to `sensor`, and the prefilter F to `prefilt`. All arguments must be SISO LTI objects.

`sisotool(views)` or `sisotool(`*`views`*`,plant,comp)` specifies the initial configuration of the SISO Design Tool. The argument *`views`* can be any of the following strings (or combination thereof):

- 'rlocus' — Root Locus plot
- 'bode' — Bode diagrams of the open-loop response
- 'nichols — Nichols plot
- 'filter — Bode diagrams of the prefilter **F** and the closed-loop response from the command into **F** to the output of the compensator **G** (see the feedback structure figure below)

For example

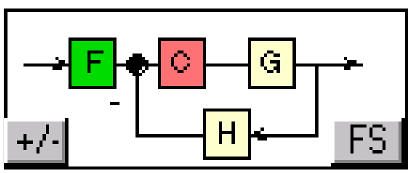
```
sisotool('bode')
```

opens a SISO Design Tool with only the Bode Diagrams on.

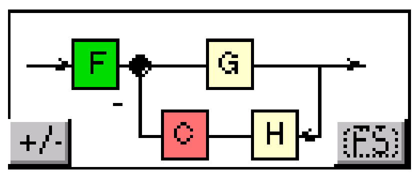
sisotool(plant,comp,options) allows you to override the default compensator location and feedback sign by using an extra input argument options with the following fields:

- options.Location = 'forward' — Compensator in the forward loop
- options.Location = 'feedback' — Compensator in the feedback loop
- options.Sign = –1 — Negative feedback
- options.Sign = 1 — Positive feedback

You can design compensators for one of the following two feedback loop configurations.



Compensator in the Forward Path



Compensator in the Feedback Path

The SISO Design Tool Supports Two Feedback Structures.

For more details on the SISO Design Tool, see “Designing Compensators” in the Getting Started documentation for the Control System Toolbox.

See Also

bode	Bode response
ltiview	Open an LTI Viewer

`rlocus`	Root locus
`nichols`	Nichols response

Purpose

Provide the output/input/array dimensions of LTI models, the model order of TF, SS, and ZPK models, and the number of frequencies of FRD models

Syntax

```
size(sys)
d = size(sys)
Ny = size(sys,1)
Nu = size(sys,2)
Sk = size(sys,2+k)
Ns = size(sys,'order')
Nf = size(sys,'frequency')
```

Description

When invoked without output arguments, `size(sys)` returns a vector of the number of outputs and inputs for a single LTI model. The lengths of the array dimensions are also included in the response to `size` when `sys` is an LTI array. `size` is the overloaded version of the MATLAB function `size` for LTI objects.

`d = size(sys)` returns:

- The row vector `d = [Ny Nu]` for a single LTI model `sys` with `Ny` outputs and `Nu` inputs
- The row vector `d = [Ny Nu S1 S2 ... Sp]` for an `S1`-by-`S2`-by-...-by-`Sp` array of LTI models with `Ny` outputs and `Nu` inputs

`Ny = size(sys,1)` returns the number of outputs of `sys`.

`Nu = size(sys,2)` returns the number of inputs of `sys`.

`Sk = size(sys,2+k)` returns the length of the `k`-th array dimension when `sys` is an LTI array.

Ns = size(sys,'order') returns the model order of a TF, SS, or ZPK model. This is the same as the number of states for state-space models. When `sys` is an LTI array, ns is the maximum order of all of the models in the LTI array.

`Nf = size(sys,'frequency')` returns the number of frequencies when `sys` is an FRD. This is the same as the length of `sys.frequency`.

Example

Consider the random LTI array of state-space models

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

Its dimensions are obtained by typing

```
size(sys)

3x1 array of state-space models
Each model has 3 outputs, 2 inputs, and 5 states.
```

See Also

`isempty`	Test if LTI model is empty
`issiso`	Test if LTI model is SISO
`ndims`	Number of dimensions of an LTI array

Purpose

Perform model reduction based on structure

Syntax

```
msys = sminreal(sys)
```

Description

`msys = sminreal(sys)` eliminates the states of the state-space model `sys` that don't affect the input/output response. All of the states of the resulting state-space model `msys` are also states of `sys` and the input/output response of `msys` is equivalent to that of `sys`.

`sminreal` eliminates only structurally non minimal states, i.e., states that can be discarded by looking only at hard zero entries in the A, B, and C matrices. Such structurally nonminimal states arise, for example, when linearizing a Simulink model that includes some unconnected state-space or transfer function blocks.

Remark

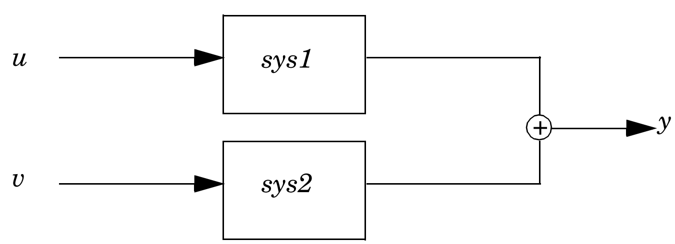
The model resulting from `sminreal(sys)` is not necessarily minimal, and may have a higher order than one resulting from `minreal(sys)`. However, `sminreal(sys)` retains the state structure of `sys`, while, in general, `minreal(sys)` does not.

Example

Suppose you concatenate two SS models, `sys1` and `sys2`.

```
sys = [sys1,sys2];
```

This operation is depicted in the diagram below.



If you extract the subsystem `sys1` from `sys`, with

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

all of the states of `sys`, including those of `sys2` are retained. To eliminate the unobservable states from `sys2`, while retaining the states of `sys1`, type

```
sminreal(sys(1,1))
```

See Also

`minreal`	Model reduction by removing unobservable/uncontrollable states or cancelling pole/zero pairs

Purpose Specify state-space models or convert an LTI model to state space

Syntax

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

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)
sys_ss = ss(sys,'minimal')
```

Description ss is used to create real- or complex-valued state-space models (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

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

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

- a is an Nx-by-Nx real- or complex-valued matrix.
- b is an Nx-by-Nu real- or complex-valued matrix.
- c is an Ny-by-Nx real- or complex-valued matrix.
- d is an Ny-by-Nu real- or complex-valued matrix.

The output sys is an SS model that stores the model data (see "State-Space Models" on page 2-14). If $D = 0$, you can simply set d to the scalar 0 (zero), regardless of the dimension.

sys = ss(a,b,c,d,Ts) creates the discrete-time model

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

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

```
sys = ss([],[],[],d)
```

`sys = ss(a,b,c,d,ltisys)` creates a state-space model with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See “Generic Properties” on page 2-26 for an overview of generic LTI properties.

See “Building LTI Arrays” on page 4-12 for information on how to build arrays of state-space models.

Any of the previous syntaxes can be followed by property name/property value pairs.

```
'PropertyName',PropertyValue
```

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',Value1,...,'PropertyN',ValueN)
```

is equivalent to the sequence of commands.

```
sys = ss(a,b,c,d)
set(sys,'Property1',Value1,...,'PropertyN',ValueN)
```

Conversion to State Space

`sys_ss = ss(sys)` converts an arbitrary TF or ZPK 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*.

`sys_ss = ss(sys,'minimal')` produces a state-space realization with no uncontrollable or unobservable states. This is equivalent to `sys_ss = minreal(ss(sys))`.

Examples

Example 1

The command

```
sys = ss(A,B,C,D,0.05,'statename',{'position' 'velocity'},...
                      '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 `position` and `velocity`, 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

$$H(s) = \begin{bmatrix} \dfrac{s+1}{s^3+3s^2+3s+2} \\ \dfrac{s^2+3}{s^2+s+1} \end{bmatrix}$$

by typing

```
H = [tf([1 1],[1 3 3 2]) ; tf([1 0 3],[1 1 1])];
sys = ss(H);
size(sys)

State-space model with 2 outputs, 1 input, and 5 states.
```

Note that the number of states is equal to the cumulative order of the SISO entries of $H(s)$.

To obtain a minimal realization of $H(s)$, type

```
sys = ss(H,'min');
size(sys)

State-space model with 2 outputs, 1 input, and 3 states.
```

The resulting state-space model order has order three, the minimum number of states needed to represent $H(s)$. This can be seen directly by factoring $H(s)$ as the product of a first order system with a second order one.

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

See Also

`dss`	Specify descriptor state-space models.
`frd`	Specify FRD models or convert to an FRD.
`get`	Get properties of LTI models.
`set`	Set properties of LTI models.
`ssdata`	Retrieve the A, B, C, D matrices of state-space model.
`tf`	Specify transfer functions or convert to TF.
`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

balreal	Grammian-based I/O balancing
canon	Canonical state-space realizations
ssbal	Balancing of state-space models using diagonal similarity transformations

ssbal

Purpose

Balance state-space models using a diagonal similarity transformation

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=Inf`.

`ssbal` returns an error if the state-space model `sys` has varying state dimensions.

Example

Consider the continuous-time state-space model with the following data.

$$A = \begin{bmatrix} 1 & 10^4 & 10^2 \\ 0 & 10^2 & 10^5 \\ 10 & 1 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \qquad C = \begin{bmatrix} 0.1 & 10 & 100 \end{bmatrix}$$

```
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)

a =
                        x1           x2           x3
           x1            1         2500      0.39063
           x2            0          100       1562.5
           x3         2560           64            0


b =
                        u1
           x1        0.125
           x2          0.5
           x3           32


c =
                        x1           x2           x3
           y1          0.8           20        3.125


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`	Grammian-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] = ssdata(sys)
```

Description `[a,b,c,d] = ssdata(sys)` extracts the matrix (or multidimensional array) data (A, B, C, D) from the state-space model (LTI array) `sys`. If `sys` is a transfer function or zero-pole-gain model (LTI array), it is first converted to state space. See Table 11-16, "State-Space Model Properties," on page 11-195 for more information on the format of state-space model data.

`[a,b,c,d,Ts] = ssdata(sys)` also returns the sample time `Ts`.

You can access the remaining LTI properties of `sys` with `get` or by direct referencing, for example,

```
sys.statename
```

For arrays of state-space models with variable numbers of states, use the syntax

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

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

See Also

`dssdata`	Quick access to descriptor state-space data
`get`	Get properties of LTI models
`set`	Set model properties
`ss`	Specify state-space models
`tfdata`	Quick access to transfer function data
`zpkdata`	Quick access to zero-pole-gain data

Purpose

Build an LTI array by stacking LTI models or LTI arrays along array dimensions of an LTI array

Syntax

```
sys = stack(arraydim,sys1,sys2,...)
```

Description

`sys = stack(arraydim,sys1,sys2,...)` produces an array of LTI models `sys` by stacking (concatenating) the LTI models (or LTI arrays) `sys1,sys2,...` along the array dimension `arraydim`. All models must have the same number of inputs and outputs (the same I/O dimensions), but the number of states can vary. The I/O dimensions are not counted in the array dimensions. See "Dimensions, Size, and Shape of an LTI Array" and "Building LTI Arrays Using the stack Function" for more information.

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

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

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

Example

If `sys1` and `sys2` are two LTI models:

- `stack(1,sys1,sys2)` produces a 2-by-1 LTI array.
- `stack(2,sys1,sys2)` produces a 1-by-2 LTI array.
- `stack(3,sys1,sys2)` produces a 1-by-1-by-2 LTI array.

step

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 with no output 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: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 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 output 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

$$(\text{length of t}) \times (\text{number of states}) \times (\text{number of inputs})$$

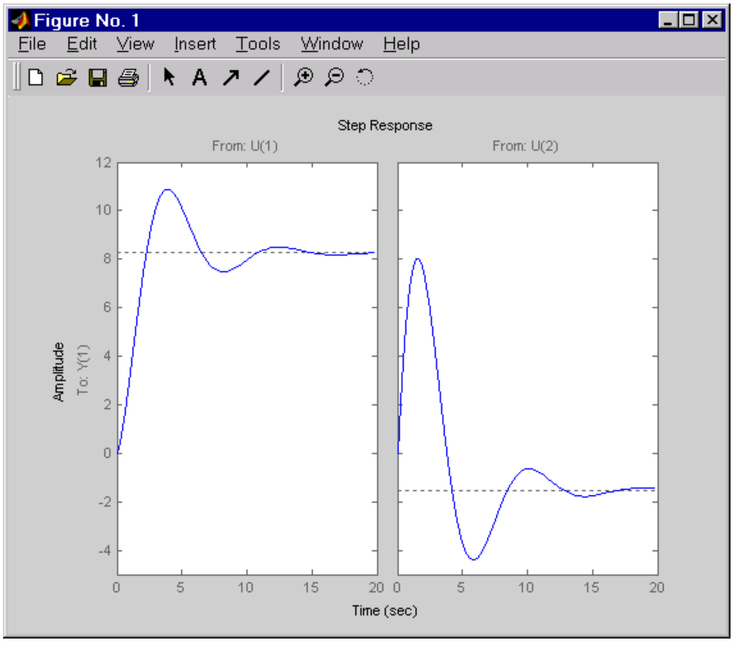
Example

Plot the step response of the following 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:Tf` is supplied (`dt` is then used as sampling period).

See Also

`impulse`	Impulse response
`initial`	Free response to initial condition
`lsim`	Simulate response to arbitrary inputs
`ltiview`	LTI system viewer

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,ltisys)

sys = tf(num,den,'Property1',Value1,...,'PropertyN',ValueN)
sys = tf(num,den,Ts,'Property1',Value1,...,'PropertyN',ValueN)

sys = tf('s')
sys = tf('z')

tfsys = tf(sys)
tfsys = tf(sys,'inv')   % for state-space sys only
```

Description `tf` is used to create real- or complex-valued 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 “Transfer Function Models” on page 2-8).

In the SISO case, `num` and `den` are the real- or complex-valued 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,ltisys)` creates a transfer function with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See "Generic Properties" on page 2-26 for an overview of generic LTI properties.

There are several ways to create LTI arrays of transfer functions. To create arrays of SISO or MIMO TF models, either specify the numerator and denominator of each SISO entry using multidimensional cell arrays, or use a `for` loop to successively assign each TF model in the array. See "Building LTI Arrays" on page 4-12 for more information.

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)
```

Transfer Functions as Rational Expressions in s or z

You can also use real- or complex-valued rational expressions to create a TF model. To do so, first type either:

- `s = tf('s')` to specify a TF model using a rational function in the Laplace variable, `s`.
- `z = tf('z',Ts)` to specify a TF model with sample time `Ts` using a rational function in the discrete-time variable, `z`.

Once you specify either of these variables, you can specify TF models directly as rational expressions in the variable `s` or `z` by entering your transfer function as a rational expression in either `s` or `z`.

Conversion to Transfer Function

`tfsys = tf(sys)` converts an arbitrary SS or ZPK LTI model `sys` to transfer function form. The output `tfsys` (TF object) is the transfer function of `sys`. By default, `tf` uses `zero` 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`.

To do this, type

```
num = {[1 1] ; 1}
den = {[1 2 2] ; [1 0]}
```

```
H = tf(num,den,'inputn','current',...
                'outputn',{'torque' 'ang. velocity'},...
                'variable','p')

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

To use a rational expression to create a SISO TF model, type

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

This produces the same transfer function as

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

Example 3

Specify the discrete MIMO transfer function

$$H(z) = \begin{bmatrix} \dfrac{1}{z+0.3} & \dfrac{z}{z+0.3} \\ \dfrac{-z+2}{z+0.3} & \dfrac{3}{z+0.3} \end{bmatrix}$$

with common denominator $d(z) = z + 0.3$ and sample time of 0.2 seconds.

```
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 4

Compute the transfer function of the state-space model with the following data.

$$A = \begin{bmatrix} -2 & -1 \\ 1 & -2 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix}, \qquad C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \qquad D = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

To do this, type

```
sys = ss([-2 -1;1 -2],[1 1;2 -1],[1 0],[0 1])
tf(sys)

Transfer function from input 1 to output:
      s
-------------
s^2 + 4 s + 5

Transfer function from input 2 to output:
s^2 + 5 s + 8
-------------
s^2 + 4 s + 5
```

Example 5

You can use a `for` loop to specify a 10-by-1 array of SISO TF models.

```
s = tf('s')
H = tf(zeros(1,1,10));
   for k=1:10,
       H(:,:,k) = k/(s^2+s+k);
   end
```

The first statement pre-allocates the TF array and fills it with zero transfer functions.

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]`.

`tf` switches convention based on your choice of variable (value of the `'Variable'` property).

Variable	Convention
`'z'` (default)	Use the row vector `[ak ... a1 a0]` to specify the polynomial $a_k z^k + ... + a_1 z + a_0$ (coefficients ordered in *descending* powers of z).
`'z^-1'`, `'q'`	Use the row vector `[b0 b1 ... bk]` to specify the polynomial $b_0 + b_1 z^{-1} + ... + b_k z^{-k}$ (coefficients in *ascending* powers of z^{-1} or q).

For example,

```
g = 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 = 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 data so that 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 `zero` and `pole` to convert state-space models.

See Also

`filt`	Specify discrete transfer functions in DSP format
`frd`	Specify a frequency response data model
`get`	Get properties of LTI models
`set`	Set properties of LTI models
`ss`	Specify state-space models or convert to state space
`tfdata`	Retrieve transfer function data
`zpk`	Specify zero-pole-gain models or convert to ZPK

tfdata

Purpose

Quick access to transfer function data

Syntax

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

Description

`[num,den] = tfdata(sys)` returns the numerator(s) and denominator(s) of the transfer function for the TF, SS or ZPK model (or LTI array of TF, SS or ZPK models) `sys`. For single LTI models, the outputs `num` and `den` of `tfdata` are cell arrays with the following characteristics:

- `num` and `den` have as many rows as outputs and as many columns as inputs.
- The `(i,j)` entries `num{i,j}` and `den{i,j}` are row vectors specifying the numerator and denominator coefficients of the transfer function from input `j` to output `i`. These coefficients are ordered in *descending* powers of s or z.

For arrays `sys` of LTI models, `num` and `den` are multidimensional cell arrays with the same sizes as `sys`.

If `sys` is a state-space or zero-pole-gain model, it is first converted to transfer function form using `tf`. See Table 11-15, "LTI Properties," on page 11-194 for more information on the format of transfer function model data.

For SISO transfer functions, the syntax

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

forces `tfdata` to return the numerator and denominator directly as row vectors rather than as cell arrays (see example below).

`[num,den,Ts] = tfdata(sys)` also returns the sample time `Ts`.

You can access the remaining LTI properties of `sys` with `get` or by direct referencing, for example,

```
sys.Ts
sys.variable
```

Example

Given the SISO transfer function

```
h = tf([1 1],[1 2 5])
```

you can extract the numerator and denominator coefficients by typing

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

num =
     0     1     1

den =
     1     2     5
```

This syntax returns two row vectors.

If you turn h into a MIMO transfer function by typing

```
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 celldisp to visualize this data. Type

```
celldisp(num)
```

and MATLAB returns the numerator vectors of the entries of H.

```
num{1} =
     0     1     1

num{2} =
     0     1
```

Similarly, for the denominators, type

```
celldisp(den)

den{1} =
     1     2     5

den{2} =
     1     1
```

See Also

get	Get properties of LTI models
ssdata	Quick access to state-space data

`tf`	Specify transfer functions
`zpkdata`	Quick access to zero-pole-gain data

totaldelay

Purpose

Return the total combined I/O delays for an LTI model

Syntax

```
td = totaldelay(sys)
```

Description

`td = totaldelay(sys)` returns the total combined I/O delays for an LTI model `sys`. The matrix `td` combines contributions from the `InputDelay`, `OutputDelay`, and `ioDelay` properties, (see `set` on page 11-192 or type `ltiprops` for details on these properties).

Delays are expressed in seconds for continuous-time models, and as integer multiples of the sample period for discrete-time models. To obtain the delay times in seconds, multiply `td` by the sample time `sys.Ts`.

Example

```
sys = tf(1,[1 0]);  % TF of 1/s
sys.inputd = 2;                            % 2 sec input delay
sys.outputd = 1.5;                         % 1.5 sec output delay
td = totaldelay(sys)

td =
    3.5000
```

The resulting I/O map is

$$e^{-2s} \times \frac{1}{s} e^{-1.5s} = e^{-3.5s} \frac{1}{s}$$

This is equivalent to assigning an I/O delay of 3.5 seconds to the original model `sys`.

See Also

`delay2z`	Change transfer functions of discrete-time LTI models with delays to rational functions or absorbs FRD delays into the frequency response phase information
`hasdelay`	True for LTI models with delays

zero

Purpose

Transmission zeros of LTI models

Syntax

```
z = zero(sys)
[z,gain] = zero(sys)
```

Description

`zero` 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 = zero(sys)` returns the (transmission) zeros of the LTI model `sys` as a column vector.

`[z,gain] = zero(sys)` also returns the gain (in the zero-pole-gain sense) if `sys` is a SISO system.

Algorithm

The transmission zeros are computed using the algorithm in [1].

See Also

`pole`	Compute the poles of an LTI model
`pzmap`	Compute the pole-zero map

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, for root locus and pole-zero maps, 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 `Ts` is the sample time.

`zgrid([],[])` draws the unit circle.

Alternatively, you can select **Grid** from the right-click menu to generate the same z-plane grid.

Example

Plot z-plane grid lines on the root locus for the system

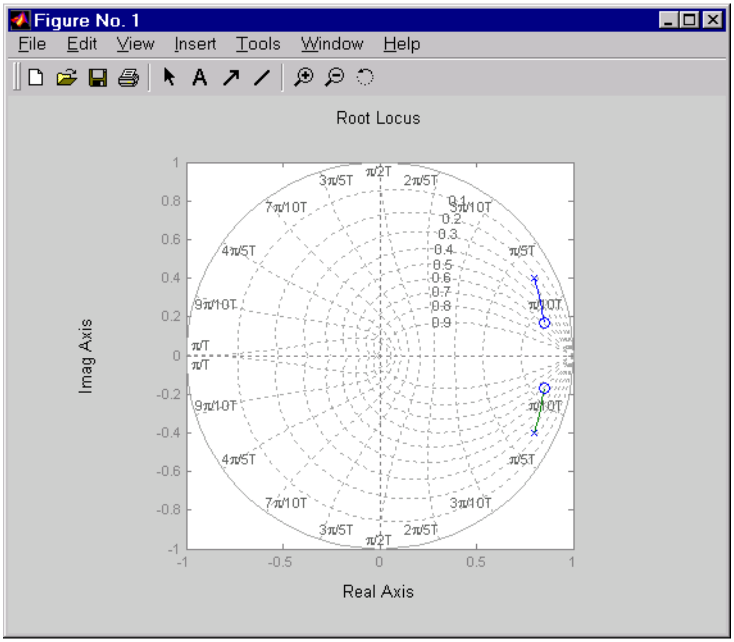
$$H(z) = \frac{2z^2 - 3.4z + 1.5}{z^2 - 1.6z + 0.8}$$

by typing

```
H = tf([2 -3.4 1.5],[1 -1.6 0.8],-1)

Transfer function:
2 z^2 - 3.4 z + 1.5
-------------------
 z^2 - 1.6 z + 0.8

Sampling time: unspecified
```

To see the z-plane grid on the root locus plot, type

```
rlocus(H)
zgrid
axis('square')
```



See Also

`pzmap`	Plot pole-zero map of LTI systems
`rlocus`	Plot root locus
`sgrid`	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,ltisys)

sys = zpk(z,p,k,'Property1',Value1,...,'PropertyN',ValueN)
sys = zpk(z,p,k,Ts,'Property1',Value1,...,'PropertyN',ValueN)

sys = zpk('s')
sys = zpk('z')

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 TF or SS 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 “LTI Objects” on page 2-3).

In the SISO case, z and p are the vectors of real- or complex-valued zeros and poles, and k is the real- or complex-valued scalar gain.

$$h(s) = k\frac{(s-z(1))(s-z(2))\ldots(s-z(m))}{(s-p(1))(s-p(2))\ldots(s-p(n))}$$

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).

You can also use rational expressions to create a ZPK model. To do so, use either:

- s = zpk('s') to specify a ZPK model from a rational transfer function of the Laplace variable, s.

- `z = zpk('z',Ts)` to specify a ZPK model with sample time `Ts` from a rational transfer function of the discrete-time variable, `z`.

Once you specify either of these variables, you can specify ZPK models directly as real- or complex-valued rational expressions in the variable `s` or `z`.

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 `Ts` (in seconds). Set `Ts = -1` or `Ts = []` 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,ltisys)` creates a zero-pole-gain model with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See "Generic Properties" on page 2-26 for an overview of generic LTI properties.

To create an array of ZPK models, use a `for` loop, or use multidimensional cell arrays for `z` and `p`, and a multidimensional array for `k`.

Any of the previous syntaxes can be followed by property name/property value pairs.

```
'PropertyName',PropertyValue
```

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 following sequence of commands.

```
sys = zpk(z,p,k)
set(sys,'Property1',Value1,...,'PropertyN',ValueN)
```

Zero-Pole-Gain Models as Rational Expressions in s or z

You can also use rational expressions to create a ZPK model. To do so, first type either:

- `s = zpk('s')` to specify a ZPK model using a rational function in the Laplace variable, `s`.
- `z = zpk('z',Ts)` to specify a ZPK model with sample time `Ts` using a rational function in the discrete-time variable, `z`.

Once you specify either of these variables, you can specify ZPK models directly as rational expressions in the variable `s` or `z` by entering your transfer function as a rational expression in either `s` or `z`.

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 `zero` 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 following zero-pole-gain model.

$$H(z) = \begin{bmatrix} \dfrac{1}{z-0.3} \\ \dfrac{2(z+0.5)}{(z-0.1+j)(z-0.1-j)} \end{bmatrix}$$

To do this, type

```
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

```
h = tf([-10 20 0],[1 7 20 28 19 5])

Transfer function:
              -10 s^2 + 20 s
----------------------------------------
s^5 + 7 s^4 + 20 s^3 + 28 s^2 + 19 s + 5
```

to zero-pole-gain form by typing

```
zpk(h)

Zero/pole/gain:
     -10 s (s-2)
----------------------
(s+1)^3 (s^2 + 4s + 5)
```

Example 3

Create a discrete-time ZPK model from a rational expression in the variable z, by typing

```
z = zpk('z',0.1);
H = (z+.1)*(z+.2)/(z^2+.6*z+.09)

Zero/pole/gain:
(z+0.1) (z+0.2)
---------------
```

```
      (z+0.3)^2

   Sampling time: 0.1
```

Algorithm

zpk uses the MATLAB function roots to convert transfer functions and the functions zero and pole to convert state-space models.

See Also

frd	Convert to frequency response data models
get	Get properties of LTI models
set	Set properties of LTI models
ss	Convert to state-space models
tf	Convert to TF transfer function models
zpkdata	Retrieve zero-pole-gain data

zpkdata

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`. See Table 11-15, "LTI Properties," on page 11-194 for more information on the format of state-space model data.

For SISO zero-pole-gain models, the 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 `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.Ts
sys.inputname
```

Example

Given a zero-pole-gain model with two outputs and one input

```
H = zpk({[0];[-0.5]},{[0.3];[0.1+i 0.1-i]},[1;2],-1)
```

```
Zero/pole/gain from input to output...
        1
 #1:  -------
      (z-0.3)

          2 (z+0.5)
 #2:  -------------------
      (z^2 - 0.2z + 1.01)

Sampling time: unspecified
```

you can extract the zero/pole/gain data embedded in `H` with

```
[z,p,k] = zpkdata(H)

z =
    [      0]
    [-0.5000]
p =
    [    0.3000]
    [2x1 double]
k =
     1
     2
```

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}

ans =
   -0.5000

p{2,1}

ans =
   0.1000+ 1.0000i
   0.1000- 1.0000i
```

See Also

`get`	Get properties of LTI models
`ssdata`	Quick access to state-space data
`tfdata`	Quick access to transfer function data

`zpk`	Specify zero-pole-gain models

Block Reference

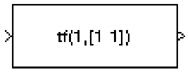
Introduction

The Control System Toolbox provides the LTI System block for use with Simulink. Its reference page contains the following information:

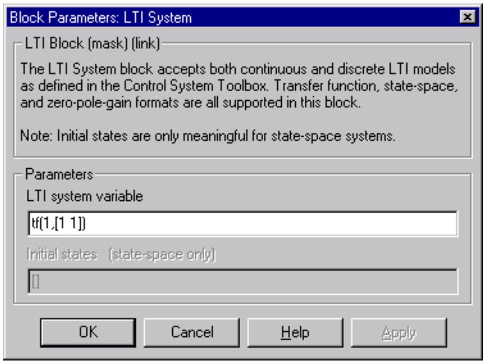
- The block name and icon
- The purpose of the block
- A description of the block
- The block parameters and dialog box including a brief description of each parameter

Purpose Import LTI System

Description The LTI System block imports linear, time-invariant (LTI) systems into Simulink.



Dialog Box



LTI system variable

Enter your LTI model. This block supports state-space, zero/pole/gain, and transfer function formats. Your model can be discrete- or continuous-time.

Initial states (state-space only)

If your model is in state-space format, you can specify the initial states in vector format. The default is zero for all states.

Symbols

A

B

C

D

E

F

L

M