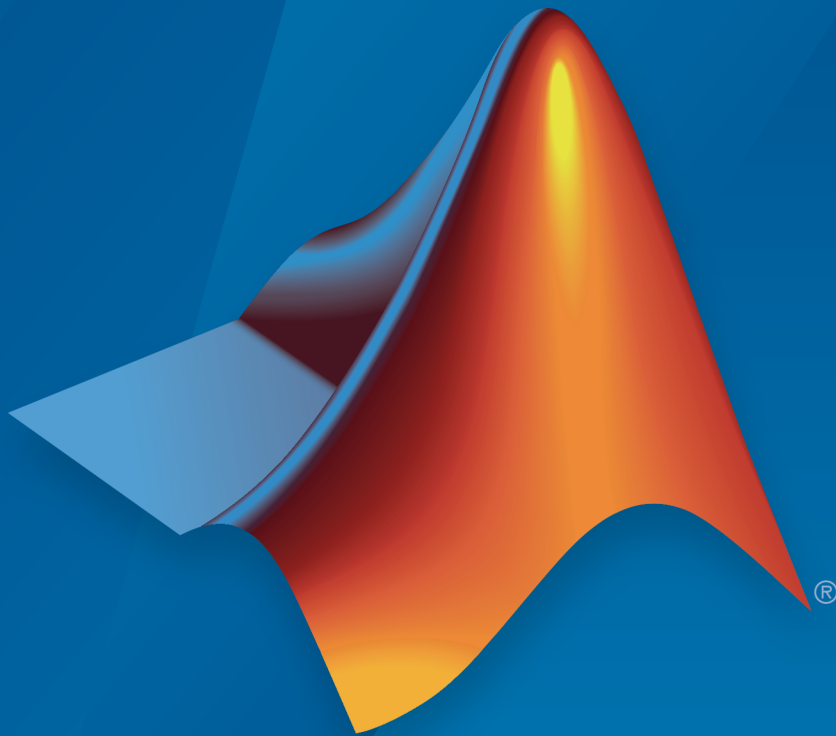


# Control System Toolbox™

## Reference



# MATLAB®

R2016b

 MathWorks®

## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



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

### *Control System Toolbox™ Reference*

© COPYRIGHT 2001–2016 by The MathWorks, Inc.

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

**FEDERAL ACQUISITION:** This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

### **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

### **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

June 2001	Online only	New for Version 5.1 (Release 12.1)
July 2002	Online only	Revised for Version 5.2 (Release 13)
June 2004	Online only	Revised for Version 6.0 (Release 14)
March 2005	Online only	Revised for Version 6.2 (Release 14SP2)
September 2005	Online only	Revised for Version 6.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 7.0 (Release 2006a)
September 2006	Online only	Revised for Version 7.1 (Release 2006b)
March 2007	Online only	Revised for Version 8.0 (Release 2007a)
September 2007	Online only	Revised for Version 8.0.1 (Release 2007b)
March 2008	Online only	Revised for Version 8.1 (Release 2008a)
October 2008	Online only	Revised for Version 8.2 (Release 2008b)
March 2009	Online only	Revised for Version 8.3 (Release 2009a)
September 2009	Online only	Revised for Version 8.4 (Release 2009b)
March 2010	Online only	Revised for Version 8.5 (Release 2010a)
September 2010	Online only	Revised for Version 9.0 (Release 2010b)
April 2011	Online only	Revised for Version 9.1 (Release 2011a)
September 2011	Online only	Revised for Version 9.2 (Release 2011b)
March 2012	Online only	Revised for Version 9.3 (Release 2012a)
September 2012	Online only	Revised for Version 9.4 (Release 2012b)
March 2013	Online only	Revised for Version 9.5 (Release 2013a)
September 2013	Online only	Revised for Version 9.6 (Release 2013b)
March 2014	Online only	Revised for Version 9.7 (Release 2014a)
October 2014	Online only	Revised for Version 9.8 (Release 2014b)
March 2015	Online only	Revised for Version 9.9 (Release 2015a)
September 2015	Online only	Revised for Version 9.10 (Release 2015b)
March 2016	Online only	Revised for Version 10.0 (Release 2016a)
September 2016	Online only	Revised for Version 10.1 (Release 2016b)



<b>1</b>	<u>Class Reference</u>
<b>2</b>	<u>Functions — Alphabetical List</u>
<b>3</b>	<u>Block Reference</u>



# Class Reference

---

TuningGoal.ConicSector  
TuningGoal.ControllerPoles  
TuningGoal.Gain  
TuningGoal.LoopShape  
TuningGoal.LQG  
TuningGoal.Margins  
TuningGoal.MinLoopGain  
TuningGoal.MaxLoopGain  
TuningGoal.Overshoot  
TuningGoal.Passivity  
TuningGoal.Poles  
TuningGoal.Rejection  
TuningGoal.Sensitivity  
TuningGoal.StepRejection  
TuningGoal.StepTracking  
TuningGoal.Tracking  
TuningGoal.Transient  
TuningGoal.Variance  
TuningGoal.WeightedPassivity  
TuningGoal.WeightedGain  
TuningGoal.WeightedVariance

# TuningGoal.ConicSector class

**Package:** TuningGoal

Sector bound for control system tuning

## Description

A conic sector bound is a restriction on the output trajectories of a system. If for all nonzero input trajectories  $u(t)$ , the output trajectory  $z(t) = (Hu)(t)$  of a linear system  $H$  satisfies:

$$\int_0^T z(t)^\top Q z(t) dt < 0,$$

for all  $T \geq 0$ , then the output trajectories of  $H$  lie in the conic sector described by the symmetric indefinite matrix  $Q$ . Selecting different  $Q$  matrices imposes different conditions on the system response.

When tuning a control system with `systemtune`, use `TuningGoal.ConicSector` to restrict the output trajectories of the response between specified inputs and outputs to a specified sector. For more information about sector bounds, see “About Sector Bounds and Sector Indices”.

## Construction

`Req = TuningGoal.ConicSector(inputname,outputname,Q)` creates a tuning goal for restricting the response  $H(s)$  from inputs `inputname` to outputs `outputname` to the conic sector specified by the symmetric matrix  $Q$ . The tuning goal constrains  $H$  such that its trajectories  $z(t) = (Hu)(t)$  satisfy:

$$\int_0^T z(t)^\top Q z(t) dt < 0,$$

for all  $T \geq 0$ . (See “About Sector Bounds and Sector Indices”.)

To specify frequency-dependent sector bounds, set  $Q$  to an LTI model that satisfies  $Q(s)' = Q(-s)$ .



## Input Arguments

### inputname

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink® model of a control system, then `inputname` can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1', 'u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **y1** and **y2**. Use **'y1'** to designate that point as an output signal when creating tuning goals. Use **{'y1','y2'}** to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **outputname** can include:
  - Any output of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **outputname** can be any output name in **T.OutputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **outputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **outputname** is an **AnalysisPoint** location of a generalized model, the output signal for the tuning goal is the implied output associated with the **AnalysisPoint** block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

## Q

Sector geometry, specified as:

- A matrix, for constant sector geometry. **Q** is a symmetric square matrix that is  $n_y$  on a side, where  $n_y$  is the number of signals in `outputname`. The matrix **Q** must be indefinite to describe a well-defined conic sector. An indefinite matrix has both positive and negative eigenvalues. In particular, **Q** must have as many negative eigenvalues as there are input channels specified in `inputname` (the size of the vector input signal  $u(t)$ ).
- An LTI model, for frequency-dependent sector geometry. **Q** satisfies  $Q(s)' = Q(-s)$ . In other words,  $Q(s)$  evaluates to a Hermitian matrix at each frequency.

For more information, see “About Sector Bounds and Sector Indices”.

## Properties

### Q

Sector geometry, specified as a matrix or an LTI model. The **Q** input argument sets initial value of **Q** when you create the tuning goal.

### Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the **Focus** property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (rad/TimeUnit). For example, suppose `Req` is a tuning goal that you want to

apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** [0, Inf] for continuous time; [0, pi/Ts] for discrete time, where Ts is the model sample time.

### **Input**

Input signal names, specified as a cell array of character vectors. The input signal names specify the inputs of the constrained response, initially populated by the `inputname` argument.

### **Output**

Output signal names, specified as a cell array of character vectors. The output signal names specify the outputs of the constrained response, initially populated by the `outputname` argument.

### **Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### **Openings**

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear

analysis point in an `slTuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `slTuner` interface. Use `getPoints` to get the list of analysis points available in an `slTuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Tips

- Suppose that

$$Q = W_1 W_1' - W_2 W_2'$$

is an indefinite factorization of  $Q$ , where  $W_1' W_2 = 0$ . This tuning goal imposes a minimum-phase condition on the transfer function  $W_2' H(s)$ , where  $H(s)$  is the transfer function between the inputs and outputs specified in the tuning goal.

(See “Algorithms” on page 1-8.) The transmission zeros of  $W_2' H(s)$  are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systuneOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systuneOptions` to change these defaults.

## Algorithms

Let

$$Q = W_1 W_1' - W_2 W_2'$$

be an indefinite factorization of  $Q$ , where  $W_1' W_2 = 0$ . If  $W_2' H(s)$  is square and minimum phase, then the time-domain sector bound

$$\int_0^T z(t)^\top Q z(t) dt < 0,$$

is equivalent to the frequency-domain sector condition,

$$H(-j\omega) Q H(j\omega) < 0$$

for all frequencies. The `TuningGoal.ConicSector` goal uses this equivalence to convert the time-domain characterization into a frequency-domain condition that `systemtuner` can handle in the same way it handles gain constraints. To secure this equivalence, `TuningGoal.ConicSector` also makes  $W_2' H(s)$  minimum phase by making all its zeros stable.

For sector bounds, the  $R$ -index plays the same role as the peak gain does for gain constraints (see “About Sector Bounds and Sector Indices”). The condition

$$H(-j\omega) Q H(j\omega) < 0$$

is satisfied at all frequencies if and only if the  $R$ -index is less than one. The `viewSpec` plot for `TuningGoal.ConicSector` shows the  $R$ -index value as a function of frequency (see `sectorplot`).

When you tune a control system using a `TuningGoal` object to specify a tuning goal, the software converts the tuning goal into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning goal is a hard constraint.

For the `TuningGoal.ConicSector` goal, for a closed-loop transfer function  $H(s, x)$  from `inputname` to `outputname`,  $f(x)$  is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

$R$  is the relative sector index (see `getSectorIndex`) of  $H(s, x)$ , for the sector represented by  $Q$ .

The dynamics of  $H$  affected by the minimum-phase condition are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### Conic Sector Goal

Create a tuning goal that restricts the response from an input or analysis point 'u' to an output or analysis point 'y' in a control system to the following sector:

$$S = \{(y, u) : 0.1u^2 < uy < 10u^2\}.$$

The  $Q$  matrix for this sector is given by:

```
a = 0.1;
b = 10;
Q = [1 -(a+b)/2 ; -(a+b)/2 a*b];
```

Use this  $Q$  matrix to create the tuning goal.

```
TG = TuningGoal.ConicSector('u', 'y', Q)
```

```
TG =
```

```
ConicSector with properties:
```

```
Q: [2x2 double]
Focus: [0 Inf]
Input: {'u'}
Output: {'y'}
```

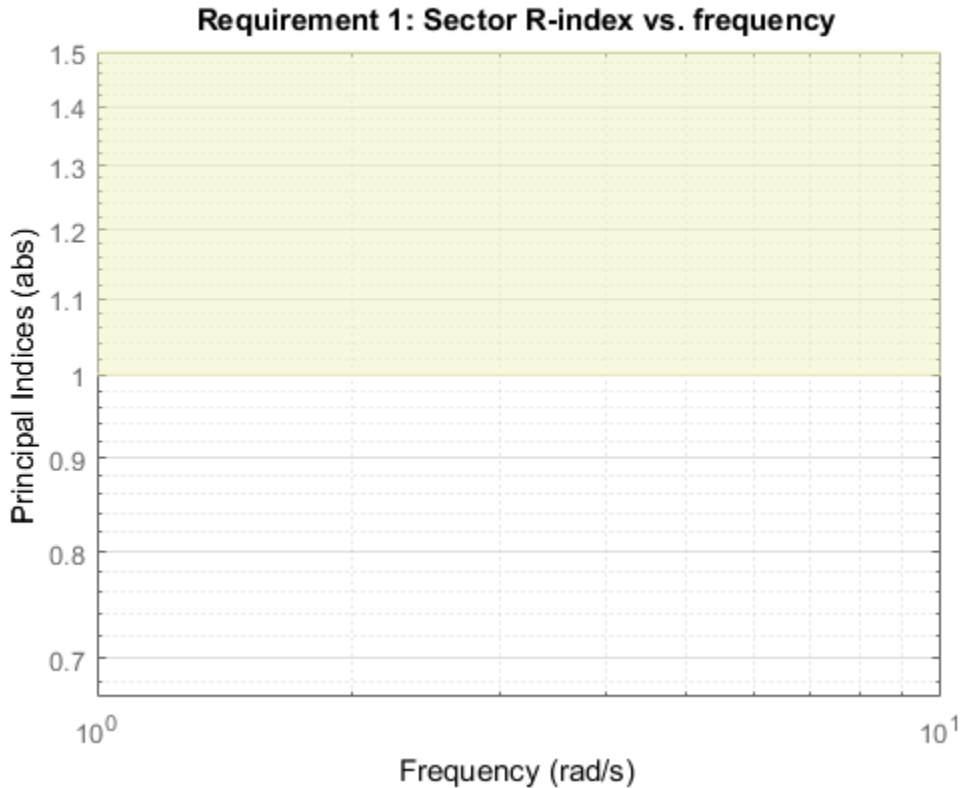
```
Models: NaN
Openings: {0x1 cell}
Name: ''
```

Set properties to further configure the tuning goal. For example, suppose the control system model has an analysis point called 'OuterLoop', and you want to enforce the tuning goal with the loop open at that point.

```
TG.Openings = 'OuterLoop';
```

Before or after tuning, use `viewSpec` to visualize the tuning goal.

```
viewSpec(TG)
```

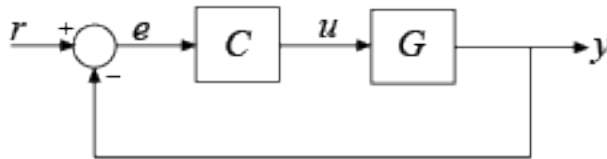




The goal is met when the relative sector index  $R < 1$  at all frequencies. The shaded area represents the region where the goal is not met. When you use this requirement to tune a control system CL, `viewSpec(TG,CL)` shows  $R$  for the specified inputs and outputs on this plot, enabling you to identify frequency ranges in which the goal is not met, and by how much.

### Constrain Input and Output Trajectories to Conic Sector

Consider the following control system.



Suppose that the signal  $u$  is marked as an analysis point in a Simulink model or `genss` model of the control system. Suppose also that  $G$  is the closed-loop transfer function from  $u$  to  $y$ . Create a tuning goal that constrains all I/O trajectories  $\{u(t), y(t)\}$  of  $G$  to satisfy:

$$\int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt < 0,$$

for all  $T \geq 0$ . For this example, use sector matrix that imposes input passivity with index 0.5.

```
nu = 0.5;
Q = [0 -1; -1 2*nu];
```

Constraining the I/O trajectories of  $G$  is equivalent to restricting the output trajectories  $z(t)$  of  $H = [G; I]$  to the sector defined by:

$$\int_0^T z(t)^T Q z(t) dt < 0.$$

(See “About Sector Bounds and Sector Indices” for more details about this equivalence.) To specify this constraint, create a tuning goal that constrains the transfer function  $H = [G; I]$ , which the transfer function from input  $u$  to outputs  $\{y; u\}$ .

```
TG = TuningGoal.ConicSector('u',{ 'y'; 'u' },Q);
```

When you specify the same signal 'u' as both input and output, the conic sector tuning goal sets the corresponding transfer function to the identity. Therefore, the transfer function constrained by TG is  $H = [G; I]$  as intended. This treatment is specific to the conic sector tuning goal. For other tuning goals, when the same signal appears in both inputs and outputs, the resulting transfer function is zero in the absence of feedback loops, or the complementary sensitivity at that location otherwise. This result occurs because when the software processes analysis points, it assumes the input is injected after the output. See “Marking Signals of Interest for Control System Analysis and Design” for more information about how analysis points work.

## See Also

systune (for sITuner) | getSectorIndex | systune | viewSpec | evalSpec | sITuner

## How To

- “About Sector Bounds and Sector Indices”
- “Tuning Control Systems with SYSTUNE”
- “Tune Control Systems in Simulink”

**Introduced in R2016b**

# TuningGoal.ControllerPoles class

**Package:** TuningGoal

Constraint on controller dynamics for control system tuning

## Description

Use the `TuningGoal.ControllerPoles` requirement object to specify a tuning requirement that constrains the dynamics of a tunable component in a control system model. Use this requirement for constraining the dynamics of tuned blocks identified in a `sITuner` interface to a Simulink model. If you are tuning a `genss` model of a control system, use the requirement to constrain tunable elements such as `tunableTF` or `tunableSS`. The `TuningGoal.ControllerPoles` requirement lets you control the minimum decay rate, minimum damping, and maximum natural frequency of the poles of the tunable element, ensuring that the controller is free of fast or resonant dynamics. The requirement can also ensure stability of the tuned value of the tunable element.

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-15 of the object.

## Construction

`Req = TuningGoal.ControllerPoles(blockID,mindecay,mindamping,maxfreq)` creates a tuning requirement that constrains the dynamics of a tunable component of a control system. The minimum decay rate, minimum damping constant, and maximum natural frequency define a region of the complex plane in which poles of the component must lie. A nonnegative minimum decay ensures stability of the tuned poles. The requirement applies to all poles in the block except fixed integrators, such as the I term of a PID controller.

## Input Arguments

### **blockID**

Tunable component to constrain, specified as a character vector. `blockID` designates one of the tuned blocks in the control system you are tuning.

- For tuning a Simulink model of a control system, `blockID` is a tuned block in the `sITuner` interface to the model. For example, suppose the `sITuner` interface has a tuned block called `Controller`. To constrain this block, use `'Controller'` for the `blockID` input argument.
- For tuning a `genss` model of a control system, `blockid` is one of the control design blocks of that model. For example, suppose the `genss` interface has a tunable block with name `C1`. To constrain this block, use `'C1'` for the `blockID` input argument.

### **mindecay**

Minimum decay rate of poles of tunable component, specified as a scalar value in the frequency units of the control system model you are tuning.

Specify `mindecay`  $\geq 0$  to ensure that the block is stable. If you specify a negative value, the tuned block can include unstable poles.

When you tune the control system using this requirement, all poles of the tunable component are constrained to satisfy:

- $\text{Re}(s) < -\text{mindecay}$ , for continuous-time systems.
- $\log(|z|) < -\text{mindecay} \cdot T_s$ , for discrete-time systems with sample time  $T_s$ .

**Default:** 0

### **mindamping**

Desired minimum damping ratio of poles of the tunable block, specified as a value between 0 and 1.

Poles of the block that depend on the tunable parameters are constrained to satisfy  $\text{Re}(s) < -\text{mindamping} \cdot |s|$ . In discrete time, the damping ratio is computed using  $s = \log(z) / T_s$ .

**Default:** 0

### **maxfreq**

Desired maximum natural frequency of poles of the tunable block, specified as a scalar value in the units of the control system model you are tuning.

Poles of the block are constrained to satisfy  $|s| < \text{maxfreq}$  for continuous-time blocks, or  $|\log(z)| < \text{maxfreq} \cdot T_s$  for discrete-time blocks with sample time  $T_s$ . This constraint prevents fast dynamics in the tunable block.

**Default:** Inf

## Properties

### Block

Name of tunable component to constrain, specified as a character vector. The `blockID` input argument sets the value of `Block`.

### MinDecay

Minimum decay rate of poles of tunable component, specified as a scalar value in the frequency units of the control system you are tuning. The initial value of this property is set by the `mindecay` input argument.

$\text{MinDecay} \geq 0$  to ensure that the block is stable. If you specify a negative value, the tuned block can include unstable poles.

When you tune the control system using this requirement, all poles of the tunable component are constrained to satisfy  $\text{Re}(s) < -\text{MinDecay}$  for continuous-time systems, or  $\log(|z|) < -\text{MinDecay} \cdot T_s$  for discrete-time systems with sample time  $T_s$ .

You can use dot notation to change the value of this property after you create the requirement. For example, suppose `Req` is a `TuningGoal.ControllerPoles` requirement. Change the minimum decay rate to 0.001:

```
Req.MinDecay = 0.001;
```

**Default:** 0

### MinDamping

Desired minimum damping ratio of poles of the tunable block, specified as a value between 0 and 1. The initial value of this property is set by the `mindamping` input argument.

Poles of the block that depend on the tunable parameters are constrained to satisfy  $\text{Re}(s) < -\text{MinDamping} \cdot |s|$ . In discrete time, the damping ratio is computed using  $s = \log(z) / T_s$ .

**Default:** 0

## MaxFrequency

Desired maximum natural frequency of poles of the tunable block, specified as a scalar value in the frequency units of the control system model you are tuning. The initial value of this property is set by the `maxfreq` input argument.

Poles of the block are constrained to satisfy  $|s| < \text{maxfreq}$  for continuous-time blocks, or  $|\log(z)| < \text{maxfreq} * T_s$  for discrete-time blocks with sample time  $T_s$ . This constraint prevents fast dynamics in the tunable block.

You can use dot notation to change the value of this property after you create the requirement. For example, suppose `Req` is a `TuningGoal.ControllerPoles` requirement. Change the maximum frequency to 1000:

```
Req.MaxFrequency = 1000;
```

**Default:** `Inf`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Examples

### Constrain Dynamics of Tunable Transfer Function

Create a tuning requirement that constrains the dynamics of a tunable transfer function block in a tuned control system.

For this example, suppose that you are tuning a control system that includes a compensator block parametrized as a second-order transfer function. Create a tuning requirement that restricts the poles of that transfer function to the region  $\text{Re}(s) < -0.1$ ,  $|s| < 30$ .

Create a tunable component that represents the compensator.

```
C = tunableTF('Compensator',2,2);
```

This command creates a Control Design Block named 'Compensator' with two poles and two zeroes. You can construct a tunable control system model, T, by interconnecting this Control Design Block with other tunable and numeric LTI models. If you tune T using `systemtune`, the values of these poles and zeroes are unconstrained by default.

Create a tuning requirement to constrain the dynamics of the compensator block. Set the minimum decay rate to 0.1 rad/s, and set the maximum frequency to 30 rad/s.

```
Req = TuningGoal.ControllerPoles('Compensator',0.1,0,30);
```

The `mindamping` input argument is 0, which imposes no constraint on the damping constant of the poles of the block.

If you tune T using `systemtune` and the tuning requirement Req, the poles of the compensator block are constrained satisfy these values. After you tune T, you can use `viewSpec` to validate the tuned control system against the requirement.

## Tips

- `TuningGoal.ControllerPoles` restricts the dynamics of a single tunable component of the control system. To ensure the stability or restrict the overall dynamics of the tuned control system, use `TuningGoal.Poles`.

## Algorithms

When you use a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ .  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$ , or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For `TuningGoal.ControllerPoles`,  $f(x)$  reflects the relative satisfaction or violation of the goal. For example, if you attempt to constrain the pole of a tuned block to a minimum damping of  $\zeta = 0.5$ , then:

- $f(x) = 1$  means the damping of the pole is  $\zeta = 0.5$  exactly.
- $f(x) = 1.1$  means the damping is  $\zeta = 0.5/1.1 = 0.45$ , roughly 10% less than the target.

- $f(x) = 0.9$  means the damping is  $\zeta = 0.5/0.9 = 0.55$ , roughly 10% better than the target.

## See Also

`systemtune` (for `slTuner`) | `TuningGoal.Poles` | `looptune` | `systemtune` | `looptune` (for `slTuner`) | `viewSpec` | `evalSpec` | `tunableTF` | `tunableSS`

## How To

- “System Dynamics Specifications”
- “Models with Tunable Coefficients”



# TuningGoal.Gain class

**Package:** TuningGoal

Gain constraint for control system tuning

## Description

Use the `TuningGoal.Gain` object to specify a constraint that limits the gain from a specified input to a specified output. Use this requirement for control system tuning with tuning commands such as `sys tune` or `looptune`.

When you use a `TuningGoal.Gain` requirement, the software attempts to tune the system so that the gain from the specified input to the specified output does not exceed the specified value. By default, the constraint is applied with the loop closed. To apply the constraint to an open-loop response, use the `Openings` property of the `TuningGoal.Gain` object.

You can use a gain constraint to:

- Enforce a design requirement of disturbance rejection across a particular input/output pair, by constraining the gain to be less than 1
- Enforce a custom roll-off rate in a particular frequency band, by specifying a gain profile in that band

## Construction

`Req = TuningGoal.Gain(inputname,outputname,gainvalue)` creates a tuning requirement `Req`. This requirement constrains the gain from `inputname` to `outputname` to remain below the value `gainvalue`.

You can specify the `inputname` or `outputname` as cell arrays (vector-valued signals). If you do so, then the tuning requirement constrains the largest singular value of the transfer matrix from `inputname` to `outputname`. See `sigma` for more information about singular values.

`Req = TuningGoal.Gain(inputname,outputname,gainprofile)` specifies the maximum gain as a function of frequency. You can specify the target gain profile

(maximum gain across the I/O pair) as a smooth transfer function. Alternatively, you can sketch a piecewise error profile using an `frd` model.

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1', 'u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### outputname

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `y1` and `y2`. Use `'y1'` to designate that point as an output signal when creating tuning goals. Use `{'y1', 'y2'}` to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then **outputname** can include:
  - Any output of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then **outputname** can be any output name in `T.OutputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then **outputname** can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `outputname` is an `AnalysisPoint` location of a generalized model, the output signal for the tuning goal is the implied output associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **gainvalue**

Maximum gain (linear). The gain constraint `Req` specifies that the gain from `inputname` to `outputname` is less than `gainvalue`.

`gainvalue` is a scalar value. If the signals `inputname` or `outputname` are vector-valued signals, then `gainvalue` constrains the largest singular value of the transfer matrix from `inputname` to `outputname`. See `sigma` for more information about singular values.

### **gainprofile**

Gain profile as a function of frequency. The gain constraint `Req` specifies that the gain from `inputname` to `outputname` at a particular frequency is less than `gainprofile`. You can specify `gainprofile` as a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise gain profile using a `frd` model or the `makeweight` function. When you do so, the software automatically maps the gain profile onto a `zpk` model. The magnitude of this `zpk` model approximates the desired gain profile. Use `viewSpec(Req)` to plot the magnitude of the `zpk` model.

`gainprofile` is a SISO transfer function. If `inputname` or `outputname` are cell arrays, `gainprofile` applies to all I/O pairs from `inputname` to `outputname`

## **Properties**

### **MaxGain**

Maximum gain as a function of frequency, expressed as a SISO `zpk` model.

The software automatically maps the `gainvalue` or `gainprofile` input arguments to a `zpk` model. The magnitude of this `zpk` model approximates the desired gain profile, and is stored in the `MaxGain` property. Use `viewSpec(Req)` to plot the magnitude of `MaxGain`.

### Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `Focus` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (`rad/TimeUnit`). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0,Inf]` for continuous time; `[0,pi/Ts]` for discrete time, where `Ts` is the model sample time.

### Stabilize

Stability requirement on closed-loop dynamics, specified as 1 (`true`) or 0 (`false`).

By default, `TuningGoal.Gain` imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain requirement. If stability is not required or cannot be achieved, set `Stabilize` to `false` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, set `Stabilize` to `false`.

**Default:** 1(`true`)

### InputScaling

Input signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued input signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from `Input` to `Output` when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from **Input** to **Output**. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the **OutputScaling** and **InputScaling** values on the diagonal, respectively.

The default value, `[]`, means no scaling.

**Default:** `[]`

### **OutputScaling**

Output signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued output signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from **Input** to **Output** when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from **Input** to **Output**. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the **OutputScaling** and **InputScaling** values on the diagonal, respectively.

The default value, `[]`, means no scaling.

**Default:** `[]`

### **Input**

Input signal names, specified as a cell array of character vectors that identify the inputs of the transfer function that the tuning requirement constrains. The initial value of the **Input** property is set by the `inputname` input argument when you construct the requirement object.

### **Output**

Output signal names, specified as a cell array of character vectors that identify the outputs of the transfer function that the tuning requirement constrains. The initial value of the **Output** property is set by the `outputname` input argument when you construct the requirement object.

### **Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systemtune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systemtune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** [ ]

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Gain` requirement,  $f(x)$  is given by:

$$f(x) = \left\| \frac{1}{\text{MaxGain}} D_o^{-1} T(s, x) D_i \right\|_{\infty}.$$

$T(s, x)$  is the closed-loop transfer function from `Input` to `Output`.  $D_o$  and  $D_i$  are diagonal matrices with the `OutputScaling` and `InputScaling` property values on the diagonal, respectively.  $\|\cdot\|_{\infty}$  denotes the  $H_{\infty}$  norm (see `norm`).

## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from `Input` to `Output`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### Disturbance Rejection Goal

Create a gain constraint that enforces a disturbance rejection requirement from a signal 'du' to a signal 'u'.



```
Req = TuningGoal.Gain('du','u',1);
```

This requirement specifies that the maximum gain of the response from 'du' to 'u' not exceed 1 (0 dB).

### Custom roll-off specification

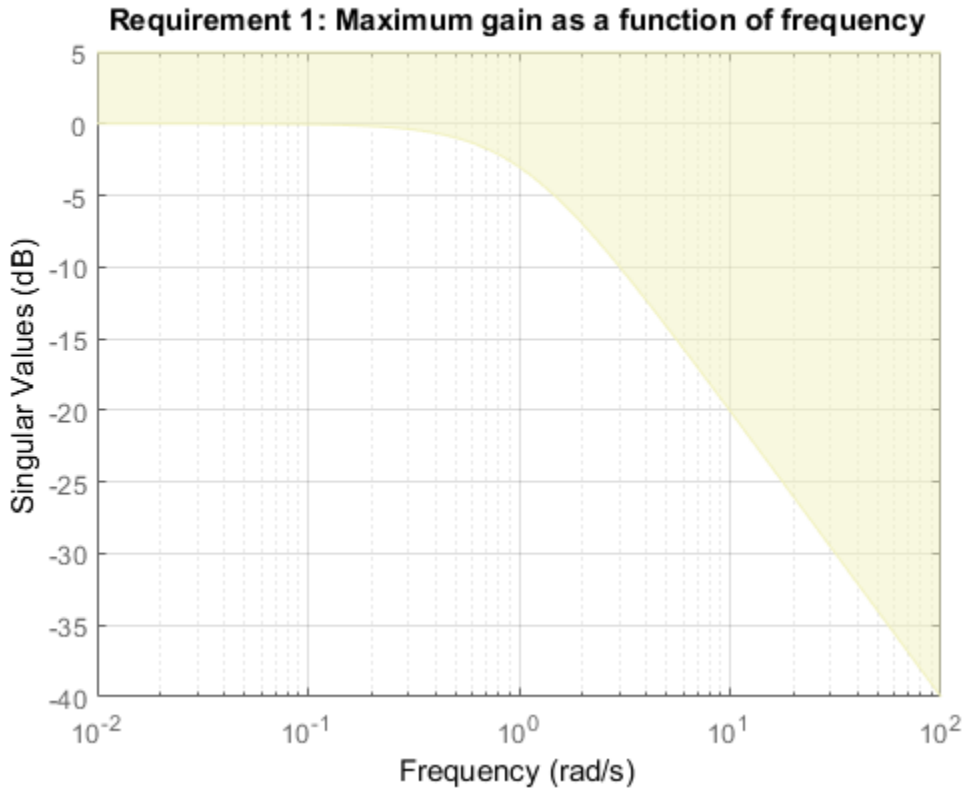
Create a gain constraint that constrains the response from a signal 'du' to a signal 'u' to roll off at 20 dB/decade at frequencies greater than 1. The gain constraint also specifies disturbance rejection (maximum gain of 1) in the frequency range [0,1].

```
gmax = frd([1 1 0.01],[0 1 100]);  
Req = TuningGoal.Gain('du','u',gmax);
```

These commands use a `frd` model to specify the gain profile as a function of frequency. The maximum gain of 1 dB at the frequency 1 rad/s, together with the maximum gain of 0.01 dB at the frequency 100 rad/s, specifies the desired rolloff of 20 dB/decade.

The software converts `gmax` into a smooth function of frequency that approximates the piecewise specified requirement. Display the error requirement using `viewSpec`.

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

### See Also

`systemtune` (for `sITuner`) | `TuningGoal.Tracking` | `looptune` | `viewSpec` | `systemtune` | `looptune` (for `sITuner`) | `TuningGoal.LoopShape` | `sITuner` | `makeweight`

### How To

- “Frequency-Domain Specifications”
- “Control of a Linear Electric Actuator”
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”

- “MIMO Control of Diesel Engine”

## TuningGoal.LoopShape class

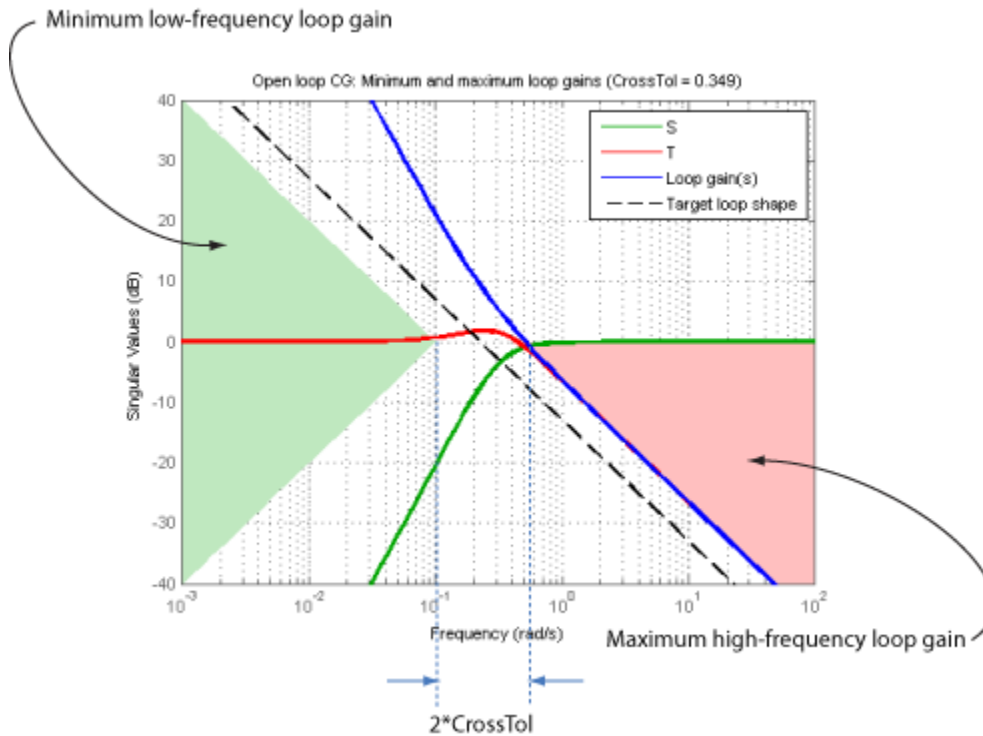
**Package:** TuningGoal

Target loop shape for control system tuning

### Description

Use the `TuningGoal.LoopShape` object to specify a target *gain profile* (gain as a function of frequency) of an open-loop response. The `TuningGoal.LoopShape` requirement constrains the open-loop, point-to-point response ( $L$ ) at a specified location in your control system. Use this requirement for control system tuning with tuning commands, such as `sys tune` or `looptune`.

When you tune a control system, the target open-loop gain profile is converted into constraints on the inverse sensitivity function  $\text{inv}(S) = (I + L)$  and the complementary sensitivity function  $T = 1 - S$ . These constraints are illustrated for a representative tuned system in the following figure.



Where  $L$  is much greater than 1, a minimum gain constraint on  $\text{inv}(S)$  (green shaded region) is equivalent to a minimum gain constraint on  $L$ . Similarly, where  $L$  is much smaller than 1, a maximum gain constraint on  $T$  (red shaded region) is equivalent to a maximum gain constraint on  $L$ . The gap between these two constraints is twice the `CrossTol` parameter, which specifies the frequency band where the loop gain can cross 0 dB.

For multi-input, multi-output (MIMO) control systems, values in the gain profile greater than 1 are interpreted as minimum performance requirements. Such values are lower bounds on the smallest singular value of the open-loop response. Gain profile values less than one are interpreted as minimum roll-off requirements, which are upper bounds on the largest singular value of the open-loop response. For more information about singular values, see `sigma`.

Use `TuningGoal.LoopShape` when the loop shape near crossover is simple or well understood (such as integral action). To specify only high gain or low gain constraints in

certain frequency bands, use `TuningGoal.MinLoopGain` and `TuningGoal.MaxLoopGain`. When you do so, the software determines the best loop shape near crossover.

## Construction

`Req = TuningGoal.LoopShape(location,loopgain)` creates a tuning requirement for shaping the open-loop response measured at the specified location. The magnitude of the single-input, single-output (SISO) transfer function `loopgain` specifies the target open-loop gain profile. You can specify the target gain profile (maximum gain across the I/O pair) as a smooth transfer function or sketch a piecewise error profile using an `frd` model.

`Req = TuningGoal.LoopShape(location,loopgain,crosstol)` specifies a tolerance on the location of the crossover frequency. `crosstol` expresses the tolerance in decades. For example, `crosstol = 0.5` allows gain crossovers within half a decade on either side of the target crossover frequency specified by `loopgain`. When you omit `crosstol`, the tuning requirement uses a default value of 0.1 decades. You can increase `crosstol` when tuning MIMO control systems. Doing so allows more widely varying crossover frequencies for different loops in the system.

`Req = TuningGoal.LoopShape(location,wc)` specifies just the target gain crossover frequency. This syntax is equivalent to specifying a pure integrator loop shape, `loopgain = wc/s`.

`Req = TuningGoal.LoopShape(location,wcrange)` specifies a range for the target gain crossover frequency. The range is a vector of the form `wcrange = [wc1,wc2]`. This syntax is equivalent to using the geometric mean `sqrt(wc1*wc2)` as `wc` and setting `crosstol` to the half-width of `wcrange` in decades. Using a range instead of a single `wc` value increases the ability of the tuning algorithm to enforce the target loop shape for all loops in a MIMO control system.

## Input Arguments

### **location**

Location where the open-loop response shape to be constrained is measured, specified as a character vector or cell array of character vectors that identify one or more locations in the control system to tune. What locations are available depends on what kind of system you are tuning:

- If you are tuning a Simulink model of a control system, you can use any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. For example, if the `sITuner` interface contains an analysis point `u`, you can use `'u'` to refer to that point when creating tuning goals. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.
- If you are tuning a generalized state-space (`genss`) model of a control system, you can use any `AnalysisPoint` location in the control system model. For example, the following code creates a PI loop with an analysis point at the plant input `'u'`.

```
AP = AnalysisPoint('u');
G = tf(1,[1 2]);
C = tunablePID('C','pi');
T = feedback(G*AP*C,1);
```

When creating tuning goals, you can use `'u'` to refer to the analysis point at the plant input. Use `getPoints` to get the list of analysis points available in a `genss` model.

The loop shape requirement applies to the point-to-point open-loop transfer function at the specified location. That transfer function is the open-loop response obtained by injecting signals at the location and measuring the return signals at the same point.

If `location` specifies multiple locations, then the loop-shape requirement applies to the MIMO open-loop transfer function.

### **loopgain**

Target open-loop gain profile as a function of frequency.

You can specify `loopgain` as a smooth SISO transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise gain profile using a `frd` model. When you do so, the software automatically maps your specified gain profile to a `zpk` model whose magnitude approximates the desired gain profile. Use `viewSpec(Req)` to plot the magnitude of that `zpk` model.

For multi-input, multi-output (MIMO) control systems, values in the gain profile greater than 1 are interpreted as minimum performance requirements. These values are lower bounds on the smallest singular value of  $L$ . Gain profile values less than one are interpreted as minimum roll-off requirements, which are upper bounds on the largest singular value of  $L$ . For more information about singular values, see `sigma`.

**crosstol**

Tolerance in the location of crossover frequency, in decades. specified as a scalar value. For example, `crosstol = 0.5` allows gain crossovers within half a decade on either side of the target crossover frequency specified by `loopgain`. Increasing `crosstol` increases the ability of the tuning algorithm to enforce the target loop shape for all loops in a MIMO control system.

**Default:** 0.1

**wc**

Target crossover frequency, specified as a positive scalar value. Express `wc` in units of `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the control system model you are tuning.

**wcrange**

Range for target crossover frequency, specified as a vector of the form `[wc1,wc2]`. Express `wc` in units of `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the control system model you are tuning.

## Properties

**LoopGain**

Target loop shape as a function of frequency, specified as a SISO `zpk` model.

The software automatically maps the input argument `loopgain` onto a `zpk` model. The magnitude of this `zpk` model approximates the desired gain profile. Use `viewSpec(Req)` to plot the magnitude of the `zpk` model `LoopGain`.

**CrossTol**

Tolerance on gain crossover frequency, in decades.

The initial value of `CrossTol` is set by the `crosstol` input when you create the requirement object.

**Default:** 0.1



## Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `Focus` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (rad/TimeUnit). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0,Inf]` for continuous time; `[0,pi/Ts]` for discrete time, where `Ts` is the model sample time.

## Stabilize

Stability requirement on closed-loop dynamics, specified as 1 (`true`) or 0 (`false`).

When `Stabilize` is `true`, this requirement stabilizes the specified feedback loop, as well as imposing gain or loop-shape requirements. Set `Stabilize` to `false` if stability for the specified loop is not required or cannot be achieved.

**Default:** 1 (`true`)

## LoopScaling

Toggle for automatically scaling loop signals, specified as `'on'` or `'off'`.

In multi-loop or MIMO control systems, the feedback channels are automatically rescaled to equalize the off-diagonal terms in the open-loop transfer function (loop interaction terms). Set `LoopScaling` to `'off'` to disable such scaling and shape the unscaled open-loop response.

**Default:** `'on'`

## Location

Location at which the open-loop response shape to be constrained is measured, specified as a cell array of character vectors that identify one or more analysis points in the control system to tune. For example, if `Location = {'u'}`, the tuning goal evaluates the open-

loop response measured at an analysis point 'u'. If `Location = {'u1', 'u2'}`, the tuning goal evaluates the MIMO open-loop response measured at analysis points 'u1' and 'u2'.

The initial value of the `Location` property is set by the `location` input argument when you create the `TuningGoal.LoopShape` requirement.

### **Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### **Openings**

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1' , 'u2' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.LoopShape` requirement,  $f(x)$  is given by:

$$f(x) = \left\| \frac{W_S S}{W_T T} \right\|_{\infty}.$$

$S = D^{-1}[I - L(s,x)]^{-1}D$  is the scaled sensitivity function.

$L(s,x)$  is the open-loop response being shaped.

$D$  is an automatically-computed loop scaling factor. (If the `LoopScaling` property is set to `'off'`, then  $D = I$ .)

$T = S - I$  is the complementary sensitivity function.

$W_S$  and  $W_T$  are weighting functions derived from the specified loop shape.

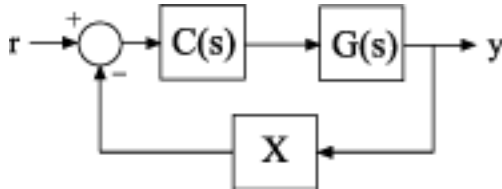
## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at `Location`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### Loop Shape and Crossover Tolerance

Create a target gain profile requirement for the following control system. Specify integral action, gain crossover at 1, and a roll-off requirement of 40 dB/decade.

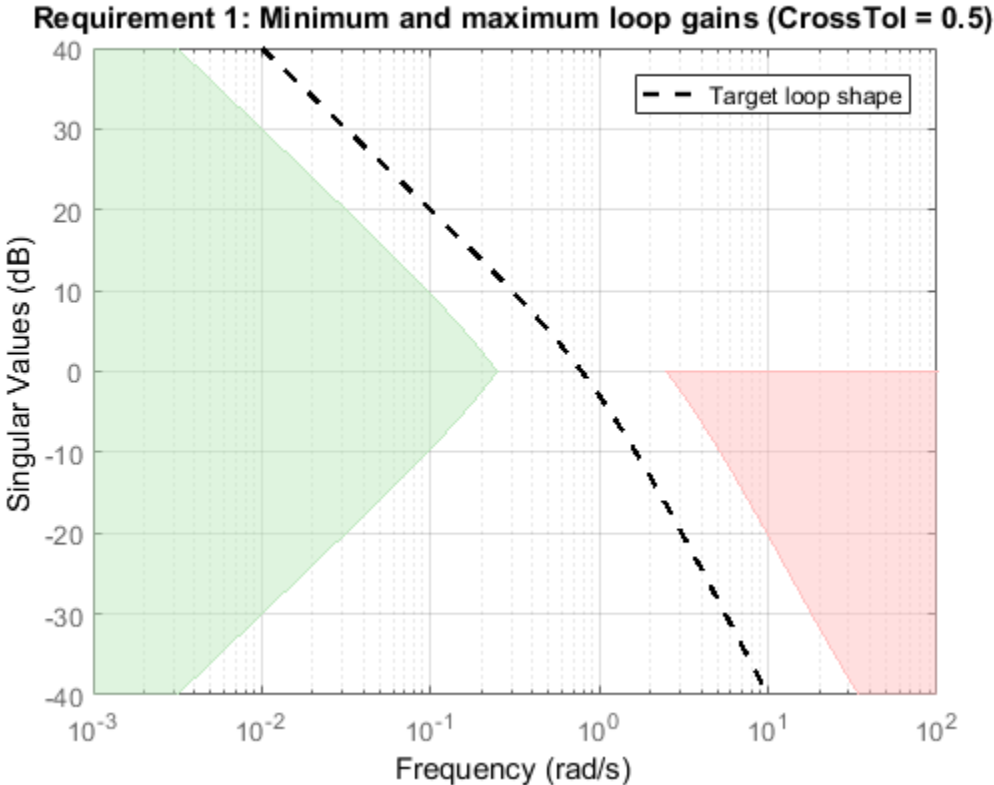


The requirement should apply to the open-loop response measured at the `AnalysisPoint` block X. Specify a crossover tolerance of 0.5 decades.

```
LS = frd([100 1 0.0001],[0.01 1 100]);  
Req = TuningGoal.LoopShape('X',LS,0.5);
```

The software converts `LS` into a smooth function of frequency that approximates the piecewise-specified requirement. Display the requirement using `viewSpec`.

```
viewSpec(Req)
```

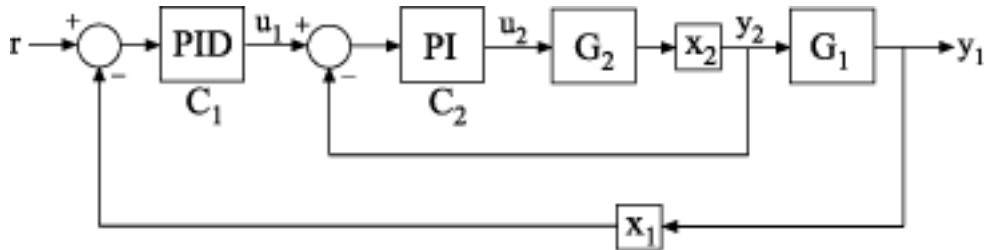


The green and red regions indicate the bounds for the inverse sensitivity,  $\text{inv}(S) = 1 - G \cdot C$ , and the complementary sensitivity,  $T = 1 - S$ , respectively. The gap between these regions at 0 dB gain reflects the specified crossover tolerance, which is half a decade to either side of the target loop crossover.

When you use `viewSpec(Req, CL)` to validate a tuned closed-loop model of this control system, `CL`, the tuned values of `S` and `T` are also plotted.

**Specify Different Loop Shapes for Multiple Loops**

Create separate loop shape requirements for the inner and outer loops of the following control system.



For the inner loop, specify a loop shape with integral action, gain crossover at 1, and a roll-off requirement of 40 dB/decade. Additionally, specify that this loop shape requirement should be enforced with the outer loop open.

```
LS2 = frd([100 1 0.0001],[0.01 1 100]);
Req2 = TuningGoal.LoopShape('X2',LS2);
Req2.Openings = 'X1';
```

Specifying 'X2' for the `location` indicates that `Req2` applies to the point-to point, open-loop transfer function at the location X2. Setting `Req2.Openings` indicates that the loop is opened at the analysis point X1 when `Req2` is enforced.

By default, `Req2` imposes a stability requirement on the inner loop as well as the loop shape requirement. In some control systems, however, inner-loop stability might not be required, or might be impossible to achieve. In that case, remove the stability requirement from `Req2` as follows.

```
Req2.Stabilize = false;
```

For the outer loop, specify a loop shape with integral action, gain crossover at 0.1, and a roll-off requirement of 20 dB/decade.

```
LS1 = frd([10 1 0.01],[0.01 0.1 10]);
Req1 = TuningGoal.LoopShape('X1',LS1);
```

Specifying 'X1' for the `location` indicates that `Req1` applies to the point-to point, open-loop transfer function at the location X1. You do not have to set `Req1.Openings` because this loop shape is enforced with the inner loop closed.

You might want to tune the control system with both loop shaping requirements `Req1` and `Req2`. To do so, use both requirements as inputs to the tuning command. For example, suppose `CLO` is a tunable `genss` model of the closed-loop control system. In that

case, use `[CL, fSoft] = systune(CLO, [Req1, Req2])` to tune the control system to both requirements.

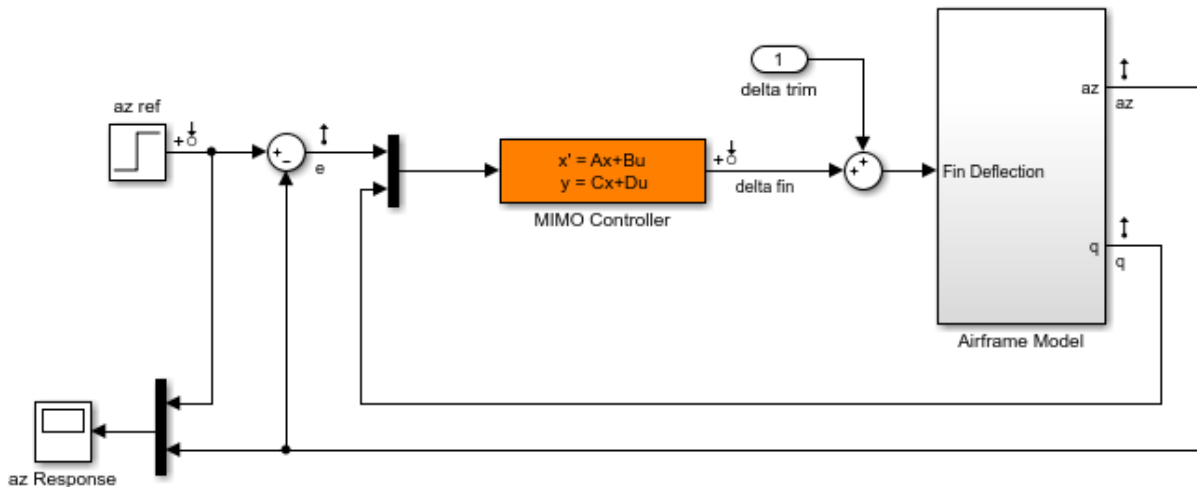
### Loop Shape for Tuning Simulink Model

Create a loop-shape requirement for the feedback loop on 'q' in the Simulink model `rct_airframe2`. Specify that the loop-shape requirement is enforced with the 'az' loop open.

Open the model.

```
open_system('rct_airframe2')
```

#### Two-loop autopilot for controlling the vertical acceleration of an airframe



Create a loop shape requirement that enforces integral action with a crossover a 2 rad/s for the 'q' loop. This loop shape corresponds to a loop shape of  $2/_s_$ .

```
s = tf('s');
shape = 2/s;
Req = TuningGoal.LoopShape('q', shape);
```

Specify the location at which to open an additional loop when enforcing the requirement.

```
Req.Openings = 'az';
```

To use this requirement to tune the Simulink model, create an `sITuner` interface to the model. Identify the block to tune in the interface.

```
ST0 = sITuner('rct_airframe2', 'MIMO Controller');
```

Designate both `az` and `q` as analysis points in the `sITuner` interface.

```
addPoint(ST0, {'az', 'q'});
```

This command makes `q` available as an analysis location. It also allows the tuning requirement to be enforced with the loop open at `az`.

You can now tune the model using `Req` and any other tuning requirements. For example:

```
[ST, fSoft] = systune(ST0, Req);
```

```
Final: Soft = 0.843, Hard = -Inf, Iterations = 47
```

### **Loop Shape Requirement with Crossover Range**

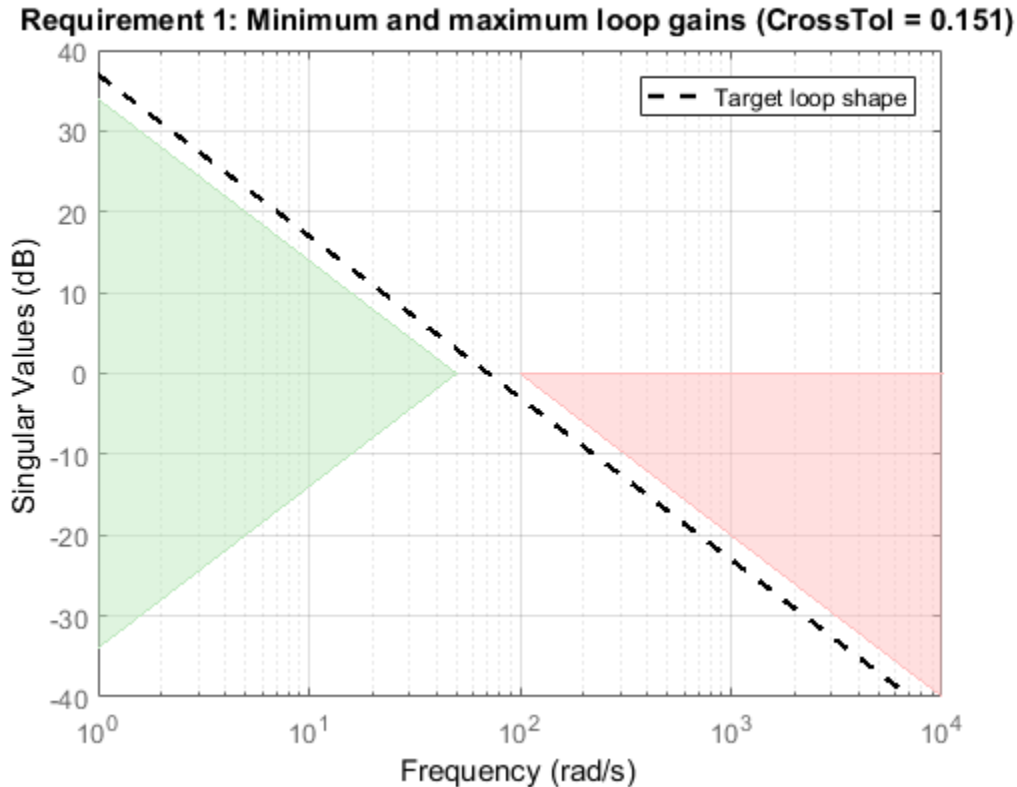
Create a tuning requirement specifying that the open-loop response of loop identified by `'X'` cross unity gain between 50 and 100 rad/s.

```
Req = TuningGoal.LoopShape('X', [50, 100]);
```

Examine the resulting requirement to see the target loop shape.

```
viewSpec(Req)
```





The plot shows that the requirement specifies an integral loop shape, with crossover around 70 rad/s, the geometrical mean of the range [50,100]. The gap at 0 dB between the minimum low-frequency gain (green region) and the maximum high-frequency gain (red region) reflects the allowed crossover range [50,100].

### See Also

looptune (for sITuner) | TuningGoal.MinLoopGain | TuningGoal.MaxLoopGain  
 | viewSpec | TuningGoal.Gain | sITuner | looptune | systune | systune (for  
 sITuner) | TuningGoal.Tracking | frd

## **How To**

- “Loop Shape and Stability Margin Specifications”
- “Tuning Multi-Loop Control Systems”
- “Tuning of a Digital Motion Control System”

# TuningGoal.LQG class

**Package:** TuningGoal

Linear-Quadratic-Gaussian (LQG) goal for control system tuning

## Description

Use the `TuningGoal.LQG` object to specify a tuning requirement for quantifying control performance as an LQG cost. It is applicable to any control structure, not just the classical observer structure of optimal LQG control. You can use this requirement for control system tuning with tuning commands, such as `systemtune` or `looptune`.

The LQG cost is given by:

$$J = E(z(t)' QZ z(t)).$$

$z(t)$  is the system response to a white noise input vector  $w(t)$ . The covariance of  $w(t)$  is given by:

$$E(w(t)w(t)') = QW.$$

The vector  $w(t)$  typically consists of external inputs to the system such as noise, disturbances, or command. The vector  $z(t)$  includes all the system variables that characterize performance, such as control signals, system states, and outputs.  $E(x)$  denotes the expected value of the stochastic variable  $x$ .

The cost function  $J$  can also be written as an average over time:

$$J = \lim_{T \rightarrow \infty} E \left( \frac{1}{T} \int_0^T z(t)' QZ z(t) dt \right).$$

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-49 of the object.

## Construction

`Req = TuningGoal.LQG(wname, zname, QW, QZ)` creates an LQG requirement. `wname` and `zname` specify the signals making up  $w(t)$  and  $z(t)$ . The matrices `QW` and `QZ` specify

the noise covariance and performance weight. These matrices must be symmetric nonnegative definite. Use scalar values for  $QW$  and  $QZ$  to specify multiples of the identity matrix.

## Input Arguments

### **wname**

Noise inputs,  $w(t)$ , specified as a character vector or a cell array of character vectors, that designate the signals making up  $w(t)$  by name, such as 'w' or { 'w' , 'v' }. The signals available to designate as noise inputs for the tuning requirement are as follows.

- If you are using the requirement to tune a Simulink model of a control system, then **wname** can include:
  - Any model input
  - Any linearization input point in the model
  - Any signal identified as a **Controls**, **Measurements**, **Switches**, or **IOs** signal in an **sITuner** interface associated with the Simulink model
- If you are using the requirement to tune a generalized state-space model (**genss**) of a control system using **systeme**, then **wname** can include:
  - Any input of the control system model
  - Any channel of an **AnalysisPoint** block in the control system model

For example, if you are tuning a control system model, **T**, then **wname** can be an input name contained in **T.InputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **X**, then **wname** can include **X**.

- If you are using the requirement to tune a controller model, **C0** for a plant **G0**, using **looptune**, then **wname** can include:
  - Any input of **C0** or **G0**
  - Any channel of an **AnalysisPoint** block in **C0** or **G0**

If **wname** is a channel of an **AnalysisPoint** block of a generalized model, the noise input for the requirement is the implied input associated with the switch:



### **zname**

Performance outputs,  $z(t)$ , specified as a character vector or a cell array of character vectors, that designate the signals making up  $z(t)$  by name, such as 'y' or {'y', 'u'}. The signals available to designate as performance outputs for the tuning requirement are as follows.

- If you are using the requirement to tune a Simulink model of a control system, then **zname** can include:
  - Any model output
  - Any linearization output point in the model
  - Any signal identified as a **Controls**, **Measurements**, **Switches**, or **IOs** signal in an **sITuner** interface associated with the Simulink model
- If you are using the requirement to tune a generalized state-space model (**genss**) of a control system using **systeme**, then **zname** can include:
  - Any output of the control system model
  - Any channel of an **AnalysisPoint** block in the control system model

For example, if you are tuning a control system model, **T**, then **zname** can be an output name contained in **T.OutputName**. Also, if **T** contains an **AnalysisPoint** block with a channel named **X**, then **zname** can include **X**.

- If you are using the requirement to tune a controller model, **C0** for a plant **G0**, using **looptune**, then **zname** can include:
  - Any input of **C0** or **G0**
  - Any channel of an **AnalysisPoint** block in **C0** or **G0**

If **zname** is a channel of an **AnalysisPoint** block of a generalized model, the performance output for the requirement is the implied output associated with the switch:

**QW**

Covariance of the white noise input vector  $w(t)$ , specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix with as many rows as there are entries in the vector  $w(t)$ . A diagonal matrix means the entries of  $w(t)$  are uncorrelated.

The covariance of  $w(t)$  is given by:

$$E(w(t)w(t)') = QW.$$

When you are tuning a control system in discrete time, the LQG requirement assumes:

$$E(w[k]w[k]') = QWT_s.$$

$T_s$  is the model sample time. This assumption ensures consistent results with tuning in the continuous-time domain. In this assumption,  $w[k]$  is discrete-time noise obtained by sampling continuous white noise  $w(t)$  with covariance  $QW$ . If in your system  $w[k]$  is a truly discrete process with known covariance  $QWd$ , use the value  $T_s * QWd$  for the  $QW$  value when creating the LQG goal.

**Default:**  $I$

**QZ**

Performance weights, specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix. Use a diagonal matrix to independently scale or penalize the contribution of each variable in  $z$ .

The performance weights contribute to the cost function according to:

$$J = E(z(t)' QZ z(t)).$$

When you use the LQG requirement as a hard goal, the software tries to drive the cost function  $J < 1$ . When you use it as a soft goal, the cost function  $J$  is minimized subject to any hard goals and its value is contributed to the overall objective function.

Therefore, select **QZ** values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

**Default:** *I*

## Properties

### NoiseCovariance

Covariance matrix of the noise inputs  $w(t)$ , specified as a matrix. The value of the **NoiseCovariance** property is set by the **WZ** input argument when you create the LQG requirement.

### PerformanceWeight

Weights for the performance signals  $z(t)$ , specified as a matrix. The value of the **PerformanceWeight** property is set by the **QZ** input argument when you create the LQG requirement.

### Input

Noise input signal names, specified as a cell array of character vectors. The input signal names specify the inputs of the transfer function that the tuning requirement constrains. The initial value of the **Input** property is set by the **wname** input argument when you construct the requirement object.

### Output

Performance output signal names, specified as a cell array of character vectors. The output signal names specify the outputs of the transfer function that the tuning requirement constrains. The initial value of the **Output** property is set by the **zname** input argument when you construct the requirement object.

### Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the **Models** property when tuning an array of control system models with **systeme**, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, **Req**, to the second, third, and fourth models in a model

array passed to `system`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1 ' , 'u2 ' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq' ;
```

**Default:** []



## Tips

- When you use this requirement to tune a continuous-time control system, `systune` attempts to enforce zero feedthrough ( $D = 0$ ) on the transfer that the requirement constrains. Zero feedthrough is imposed because the  $H_2$  norm, and therefore the value of the tuning goal, is infinite for continuous-time systems with nonzero feedthrough.

`systune` enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. `systune` returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, use the `Value` and `Free` properties of the block parametrization. For example, consider a tuned state-space block:

```
C = tunableSS('C',1,2,3);
```

To enforce zero feedthrough on this block, set its  $D$  matrix value to zero, and fix the parameter.

```
C.D.Value = 0;
C.D.Free = false;
```

For more information on fixing parameter values, see the Control Design Block reference pages, such as `tunableSS`.

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from `wname` to `zname`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systuneOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systuneOptions` to change these defaults.

## Algorithms

When you use a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ .  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$ , or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.LQG` requirement,  $f(x)$  is given by the cost function  $J$ :  
 $J = E(z(t)' QZ z(t))$ .

When you use the LQG requirement as a hard goal, the software tries to drive the cost function  $J < 1$ . When you use it as a soft goal, the cost function  $J$  is minimized subject to any hard goals and its value is contributed to the overall objective function. Therefore, select `QZ` values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

### See Also

`systeme` | `systeme (for sITuner)` | `viewSpec` | `TuningGoal.WeightedVariance` | `sITuner` | `evalSpec` | `TuningGoal.Variance`

### How To

- “Vibration Control in Flexible Beam”
- “Time-Domain Specifications”

# TuningGoal.Margins class

**Package:** TuningGoal

Stability margin requirement for control system tuning

## Description

Use the `TuningGoal.Margins` requirement object to specify a tuning requirement for the gain and phase margins of a SISO or MIMO feedback loop. You can use this requirement for validating a tuned control system with `viewSpec`. You can also use the requirement for control system tuning with tuning commands such as `sysTune` or `looptune`.

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-55 of the object.

After using the requirement to tune a control system, you can visualize the requirement and the tuned value using the `viewSpec` command. For information about interpreting the margins goal, see “Interpreting Stability Margins in Control System Tuning”.

## Construction

`Req = TuningGoal.Margins(location, gainmargin, phasemargin)` creates a tuning requirement that specifies the minimum gain and phase margins at the specified location in the control system.

## Input Arguments

### **location**

Location in the control system at which the minimum gain and phase margins apply, specified as a character vector or cell array of character vectors that identify one or more locations in the control system to tune. What locations are available depends on what kind of system you are tuning:

- If you are tuning a Simulink model of a control system, you can use any linear analysis point marked in the model, or any linear analysis point in an `sITuner`

interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. For example, if the `sITuner` interface contains an analysis point `u`, you can use `'u'` to refer to that point when creating tuning goals. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

- If you are tuning a generalized state-space (`genss`) model of a control system, you can use any `AnalysisPoint` location in the control system model. For example, the following code creates a PI loop with an analysis point at the plant input `'u'`.

```
AP = AnalysisPoint('u');  
G = tf(1,[1 2]);  
C = tunablePID('C','pi');  
T = feedback(G*AP*C,1);
```

When creating tuning goals, you can use `'u'` to refer to the analysis point at the plant input. Use `getPoints` to get the list of analysis points available in a `genss` model.

The margin requirements apply to the point-to-point, open-loop transfer function at the specified loop-opening location. That transfer function is the open-loop response obtained by injecting signals at the specified location, and measuring the return signals at the same point.

If `location` is a cell array, then the margin requirement applies to the MIMO open-loop transfer function.

### **gainmargin**

Required minimum gain margin for the feedback loop, specified as a scalar value in dB.

For MIMO feedback loops, the gain margin is based upon the notion of disk margins, which guarantee stability for concurrent gain and phase variations of  $\pm$ `gainmargin` and  $\pm$ `phasemargin` in all feedback channels. See `loopmargin` for more information about disk margins.

### **phasemargin**

Required minimum phase margin for the feedback loop, specified as a scalar value in degrees.

For MIMO feedback loops, the phase margin is based upon the notion of disk margins, which guarantee stability for concurrent gain and phase variations of  $\pm$ `gainmargin` and

$\pm$ phasemargin in all feedback channels. See loopmargin for more information about disk margins.

## Properties

### GainMargin

Required minimum gain margin for the feedback loop, specified as a scalar value in decibels (dB).

The value of the GainMargin property is set by the gainmargin input argument when you create the TuningGoal.Margins requirement.

### PhaseMargin

Required minimum phase margin for the feedback loop, specified as a scalar value in degrees.

The value of the PhaseMargin property is set by the phasemargin input argument when you create the TuningGoal.Margins requirement.

### ScalingOrder

Controls the order (number of states) of the scalings involved in computing MIMO stability margins. Static scalings (ScalingOrder = 0) are used by default. Increasing the order may improve results at the expense of increased computations. Use viewSpec to assess the gap between optimized and actual margins. If this gap is too large, consider increasing the scaling order. See “Interpreting Stability Margins in Control System Tuning”.

**Default:** 0 (static scaling)

### Focus

Frequency band in which tuning requirement is enforced, specified as a row vector of the form [min,max].

Set the Focus property to limit enforcement of the requirement to a particular frequency band. For best results with stability margin requirements, pick a frequency band extending about one decade on each side of the gain crossover frequencies. For example, suppose Req is a TuningGoal.Margins requirement that you are using to tune

a system with approximately 10 rad/s bandwidth. To limit the enforcement of the requirement, use the following command:

```
Req.Focus = [1,100];
```

**Default:** [0, Inf] for continuous time; [0, pi/Ts] for discrete time, where Ts is the model sample time.

### **Location**

Location at which the minimum gain and phase margins apply, specified as a cell array of character vectors that identify one or more analysis points in the control system to tune. For example, if `Location = {'u'}`, the tuning goal enforces the minimum gain and phase margins at an analysis point 'u'.

The value of the `Location` property is set by the `location` input argument when you create the `TuningGoal.Margins` requirement.

### **Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### **Openings**

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `slTuner` interface associated with the Simulink model. Use

`addPoint` to add analysis points and loop openings to the `slTuner` interface. Use `getPoints` to get the list of analysis points available in an `slTuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1' , 'u2' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Margins` requirement,  $f(x)$  is given by:

$$f(x) = \|2\alpha S - \alpha I\|_{\infty}.$$

$S = D^{-1}[I - L(s,x)]^{-1}D$  is the scaled sensitivity function.

$L(s,x)$  is the open-loop response being shaped.

$D$  is an automatically-computed loop scaling factor.

$\alpha$  is a scalar parameter computed from the specified gain and phase margin.

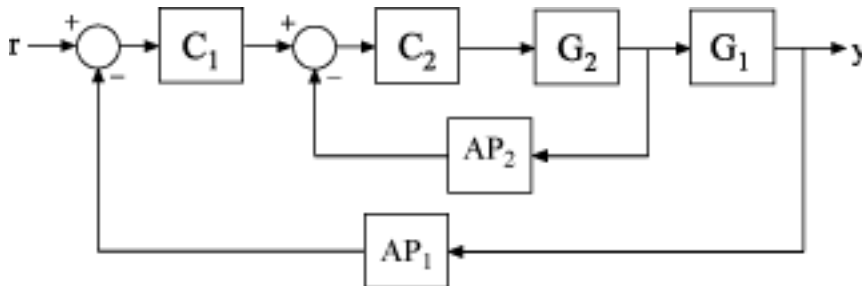
## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at `Location`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### SISO Margin Requirement Evaluated with Additional Loop Opening

Create a margin requirement for the inner loop of the following control system. The requirement imposes a minimum gain margin of 5 dB and a minimum phase margin of 40 degrees.



Create a model of the system. To do so, specify and connect the numeric plant models `G1` and `G2`, and the tunable controllers `C1` and `C2`. Also specify and connect the `AnalysisPoint` blocks `AP1` and `AP2` that mark points of interest for analysis and tuning.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
```



```
AP1 = AnalysisPoint('AP1');
AP2 = AnalysisPoint('AP2');
T = feedback(G1*feedback(G2*C2,AP2)*C1,AP1);
```

Create a tuning requirement object.

```
Req = TuningGoal.Margins('AP2',5,40);
```

This requirement imposes the specified stability margins on the feedback loop identified by the `AnalysisPoint` channel 'AP2', which is the inner loop.

Specify that these margins are evaluated with the outer loop of the control system open.

```
Req.Openings = {'AP1'};
```

Adding 'AP1' to the `Openings` property of the tuning requirements object ensures that `systemtune` evaluates the requirement with the loop open at that location.

Use `systemtune` to tune the free parameters of T to meet the tuning requirement specified by Req. You can then use `viewSpec` to validate the tuned control system against the requirement.

### MIMO Margin Requirement in Frequency Band

Create a requirement that sets minimum gain and phase margins for the loop defined by three loop-opening locations in a control system to tune. Because this loop is defined by three loop-opening locations, it is a MIMO loop.

The requirement sets a minimum gain margin of 10 dB and a minimum phase margin of 40 degrees, within the band between 0.1 and 10 rad/s.

```
Req = TuningGoal.Margins({'r', 'theta', 'phi'},10,40);
```

The names 'r', 'theta', and 'phi' must specify valid loop-opening locations in the control system that you are tuning.

Limit the requirement to the frequency band between 0.1 and 10 rad/s.

```
Req.Focus = [0.1 10];
```

### See Also

| `systemtune` (for `sITuner`) | `looptune` | `systemtune` | `looptune` (for `sITuner`) | `viewSpec` | `evalSpec`

## **How To**

- “Loop Shape and Stability Margin Specifications”
- “Tuning Control Systems with SYSTUNE”
- “Digital Control of Power Stage Voltage”
- “Tuning of a Two-Loop Autopilot”
- “Fixed-Structure Autopilot for a Passenger Jet”
- “Interpreting Stability Margins in Control System Tuning”

# TuningGoal.MinLoopGain class

**Package:** TuningGoal

Minimum loop gain constraint for control system tuning

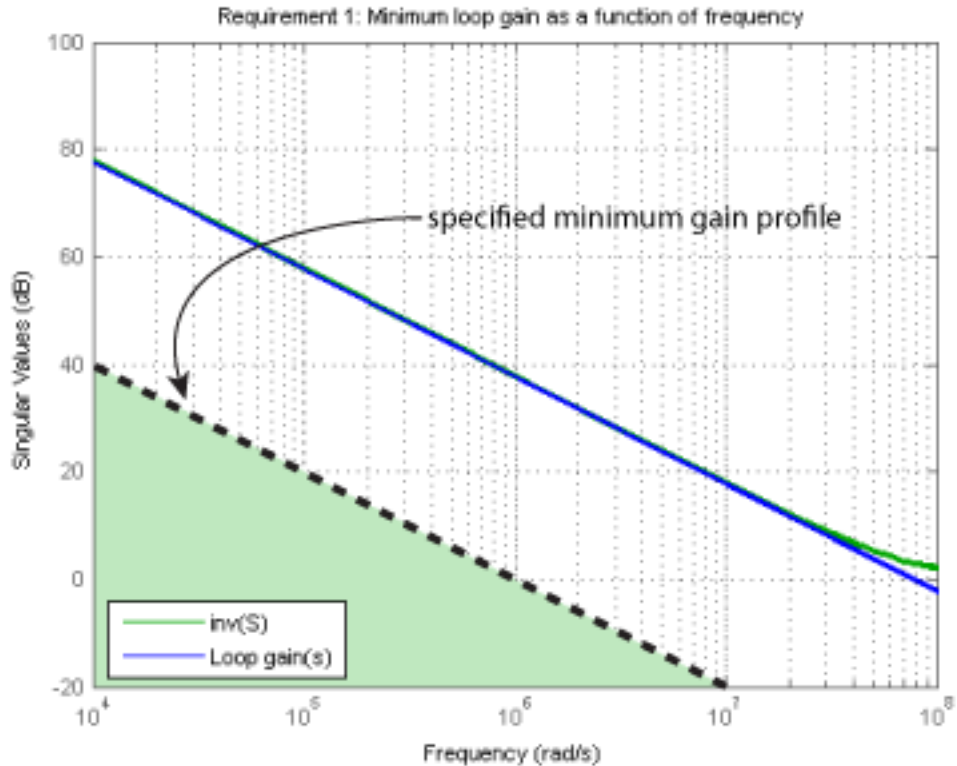
## Description

Use the `TuningGoal.MinLoopGain` object to enforce a minimum loop gain in a particular frequency band. Use this requirement with control system tuning commands such as `systune` or `looptune`.

This requirement imposes a minimum gain on the open-loop frequency response ( $L$ ) at a specified location in your control system. You specify the minimum open-loop gain as a function of frequency (a minimum *gain profile*). For MIMO feedback loops, the specified gain profile is interpreted as a lower bound on the smallest singular value of  $L$ .

When you tune a control system, the minimum gain profile is converted to a minimum gain constraint on the inverse of the sensitivity function,  $\text{inv}(S) = (I + L)$ .

The following figure shows a typical specified minimum gain profile (dashed line) and a resulting tuned loop gain,  $L$  (blue line). The green region represents gain profile values that are forbidden by this requirement. The figure shows that when  $L$  is much larger than 1, imposing a minimum gain on  $\text{inv}(S)$  is a good proxy for a minimum open-loop gain.



`TuningGoal.MinLoopGain` and `TuningGoal.MaxLoopGain` specify only low-gain or high-gain constraints in certain frequency bands. When you use these requirements, `systeme` and `looptune` determine the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use `TuningGoal.LoopShape` to specify that target loop shape.

## Construction

`Req = TuningGoal.MinLoopGain(location, loopgain)` creates a tuning requirement for boosting the gain of a SISO or MIMO feedback loop. The requirement

specifies that the open-loop frequency response ( $L$ ) measured at the specified locations exceeds the minimum gain profile specified by `loopgain`.

You can specify the minimum gain profile as a smooth transfer function or sketch a piecewise error profile using an `frd` model or the `makeweight` command. Only gain values greater than 1 are enforced.

For MIMO feedback loops, the specified gain profile is interpreted as a lower bound on the smallest singular value of  $L$ .

`Req = TuningGoal.MinLoopGain(location, fmin, gmin)` specifies a minimum gain profile of the form `loopgain = K/s` (integral action). The software chooses  $K$  such that the gain value is `gmin` at the specified frequency, `fmin`.

## Input Arguments

### **location**

Location at which the maximum open-loop gain is constrained, specified as a character vector or cell array of character vectors that identify one or more locations in the control system to tune. What loop-opening locations are available depends on what kind of system you are tuning:

- If you are tuning a Simulink model of a control system, you can use any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. For example, if the `sITuner` interface contains an analysis point `u`, you can use `'u'` to refer to that point when creating tuning goals. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.
- If you are tuning a generalized state-space (`genss`) model of a control system, you can use any `AnalysisPoint` location in the control system model. For example, the following code creates a PI loop with an analysis point at the plant input `'u'`.

```
AP = AnalysisPoint('u');
G = tf(1,[1 2]);
C = tunablePID('C','pi');
T = feedback(G*AP*C,1);
```

When creating tuning goals, you can use `'u'` to refer to the analysis point at the plant input. Use `getPoints` to get the list of analysis points available in a `genss` model.

If `location` is a cell array of loop-opening locations, then the minimum gain requirement applies to the resulting MIMO loop.

### **loopgain**

Minimum open-loop gain as a function of frequency.

You can specify `loopgain` as a smooth SISO transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise gain profile using a `frd` model or the `makeweight` command. For example, the following `frd` model specifies a minimum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of  $-20$  dB/dec at higher frequencies.

```
loopgain = frd([100 100 10],[0 1e-1 1]);
```

When you use an `frd` model to specify `loopgain`, the software automatically maps your specified gain profile to a `zpk` model. The magnitude of this model approximates the desired gain profile. Use `viewSpec(Req)` to plot the magnitude of that `zpk` model.

Only gain values larger than 1 are enforced. For multi-input, multi-output (MIMO) feedback loops, the gain profile is interpreted as a lower bound on the smallest singular value of  $L$ . For more information about singular values, see `sigma`.

### **fmin**

Frequency of minimum gain `gmin`, specified as a scalar value in rad/s.

Use this argument to specify a minimum gain profile of the form `loopgain = K/s` (integral action). The software chooses  $K$  such that the gain value is `gmin` at the specified frequency, `fmin`.

### **gmin**

Value of minimum gain occurring at `fmin`, specified as a scalar absolute value.

Use this argument to specify a minimum gain profile of the form `loopgain = K/s` (integral action). The software chooses  $K$  such that the gain value is `gmin` at the specified frequency, `fmin`.

## **Properties**

### **MinGain**

Minimum open-loop gain as a function of frequency, specified as a SISO `zpk` model.

The software automatically maps the input argument `loopgain` onto a `zpk` model. The magnitude of this `zpk` model approximates the desired gain profile. Alternatively, if you use the `fmin` and `gmin` arguments to specify the gain profile, this property is set to `K/s`. The software chooses `K` such that the gain value is `gmin` at the specified frequency, `fmin`.

Use `viewSpec(Req)` to plot the magnitude of the open-loop minimum gain profile.

### Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `Focus` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (`rad/TimeUnit`). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0,Inf]` for continuous time; `[0,pi/Ts]` for discrete time, where `Ts` is the model sample time.

### Stabilize

Stability requirement on closed-loop dynamics, specified as 1 (`true`) or 0 (`false`).

When `Stabilize` is `true`, this requirement stabilizes the specified feedback loop, as well as imposing gain or loop-shape requirements. Set `Stabilize` to `false` if stability for the specified loop is not required or cannot be achieved.

**Default:** 1 (`true`)

### LoopScaling

Toggle for automatically scaling loop signals, specified as `'on'` or `'off'`.

In multi-loop or MIMO control systems, the feedback channels are automatically rescaled to equalize the off-diagonal terms in the open-loop transfer function (loop interaction terms). Set `LoopScaling` to `'off'` to disable such scaling and shape the unscaled open-loop response.

**Default:** `'on'`

## Location

Location at which minimum loop gain is constrained, specified as a cell array of character vectors that identify one or more analysis points in the control system to tune. For example, if `Location = {'u'}`, the tuning goal evaluates the open-loop response measured at an analysis point 'u'. If `Location = {'u1', 'u2'}`, the tuning goal evaluates the MIMO open-loop response measured at analysis points 'u1' and 'u2'.

The value of the `Location` property is set by the `location` input argument when you create the requirement.

## Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

## Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control



system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ . Here,  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.MinLoopGain` requirement,  $f(x)$  is given by:

$$f(x) = \left\| W_S (D^{-1} S D) \right\|_{\infty}.$$

$W_S$  is the minimum loop gain profile, `MaxGain`.  $D$  is a diagonal scaling (for MIMO loops).  $S$  is the sensitivity function at `Location`.

Although  $S$  is a closed-loop transfer function, driving  $f(x) < 1$  is equivalent to enforcing a lower bound on the open-loop transfer function,  $L$ , in a frequency band where the gain of  $L$  is greater than 1. To see why, note that  $S = 1/(1 + L)$ . For SISO loops, when  $|L| \gg 1$ ,  $|S| \approx 1/|L|$ . Therefore, enforcing the open-loop minimum gain requirement,  $|L| > |W_S|$ , is roughly equivalent to enforcing  $|W_S S| < 1$ . For MIMO loops, similar reasoning applies, with  $\|S\| \approx 1/\sigma_{\min}(L)$ , where  $\sigma_{\min}$  is the smallest singular value.

For an example illustrating the constraint on  $S$ , see “Minimum Loop Gain as Constraint on Sensitivity Function” on page 1-70.

## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at `Location`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systuneOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systuneOptions` to change these defaults.

## Examples

### Minimum Loop Gain Requirement

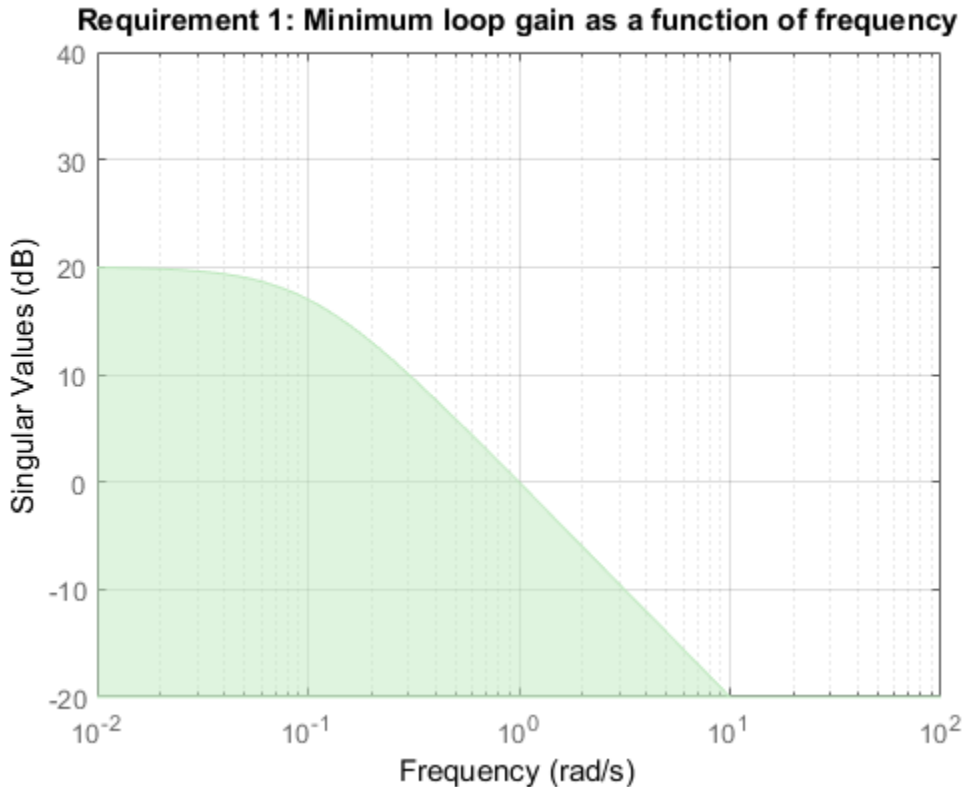
Create a requirement that boosts the open-loop gain of a feedback loop to greater than a specified profile.

Suppose that you are tuning a control system that has a loop-opening location identified by `PILoop`. Specify that the open-loop gain measured at that location exceed a minimum gain of 10 (20 dB) below 0.1 rad/s, rolling off at a rate of -20 dB/dec at higher frequencies. Use an `frd` model to sketch this gain profile.

```
loopgain = frd([10 10 0.1],[0 1e-1 10]);  
Req = TuningGoal.MinLoopGain('PILoop',loopgain);
```

The software converts `loopgain` into a smooth function of frequency that approximates the piecewise-specified requirement. Display the requirement using `viewSpec`.

```
viewSpec(Req)
```



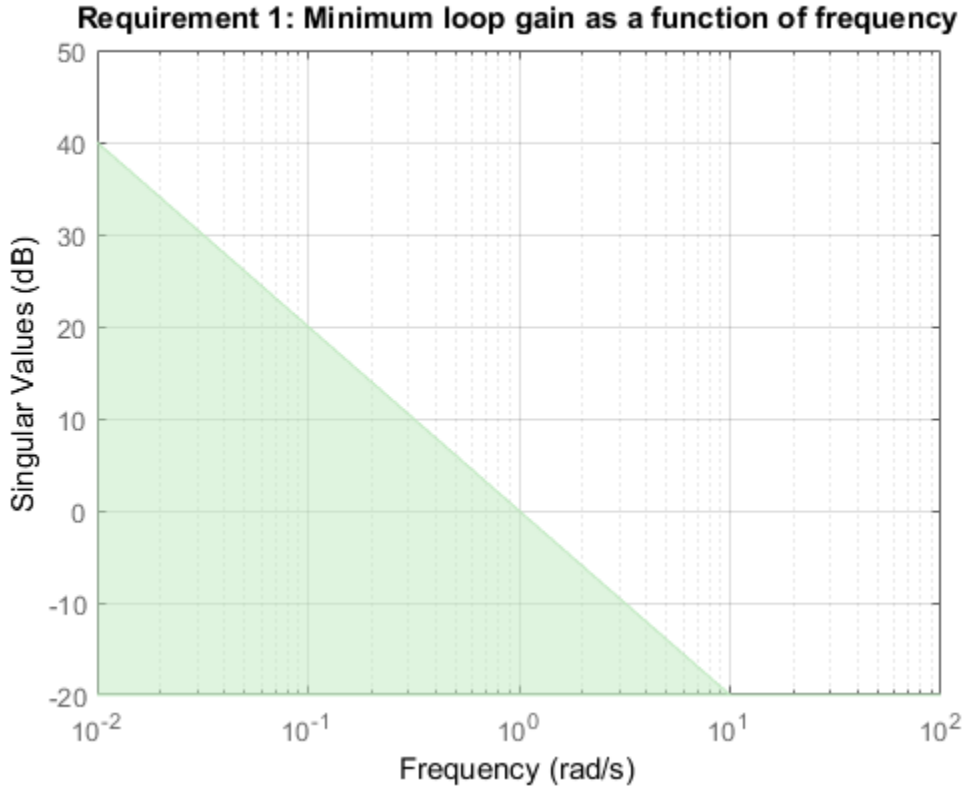
The green region indicates where the requirement is violated, except that gain values less than 1 are not enforced. Therefore, this requirement only specifies a minimum gain at frequencies below 1 rad/s.

You can use `Req` as an input to `looptune` or `systemtune` when tuning the control system.

### Integral Minimum Gain Specified as Gain Value at Single Frequency

Create a requirement that specifies a minimum loop gain profile of the form  $L = K / s$ . The gain profile attains the value of -20 dB (0.01) at 100 rad/s.

```
Req = TuningGoal.MinLoopGain('X',100,0.01);
viewSpec(Req)
```

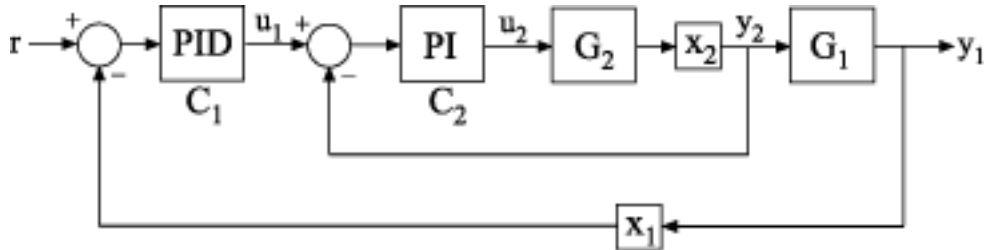


viewSpec confirms that the requirement is correctly specified. You can use this requirement to tune a control system that has a loop-opening location identified as 'X'. Since loop gain values less than 1 are ignored, this requirement specifies minimum gain only below 1 rad/s, with no restriction on loop gain at higher frequency.

### Minimum Loop Gain as Constraint on Sensitivity Function

Examine a minimum loop gain requirement against the tuned loop gain. A minimum loop gain requirement is converted to a constraint on the gain of the sensitivity function at the requirement location.

To see this relationship between the requirement and the sensitivity function, tune the following closed-loop system with analysis points at X1 and X2. The control system has tunable PID controllers C1 and C2.



Create a model of the control system.

```
G2 = zpk([], -2, 3);
G1 = zpk([], [-1 -1 -1], 10);
C20 = tunablePID('C2', 'pi');
C10 = tunablePID('C1', 'pid');
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
InnerLoop = feedback(X2*G2*C20, 1);
CL0 = feedback(G1*InnerLoop*C10, X1);
CL0.InputName = 'r';
CL0.OutputName = 'y';
```

Specify some tuning requirements, including a minimum loop gain requirement. Tune the control system to these requirements.

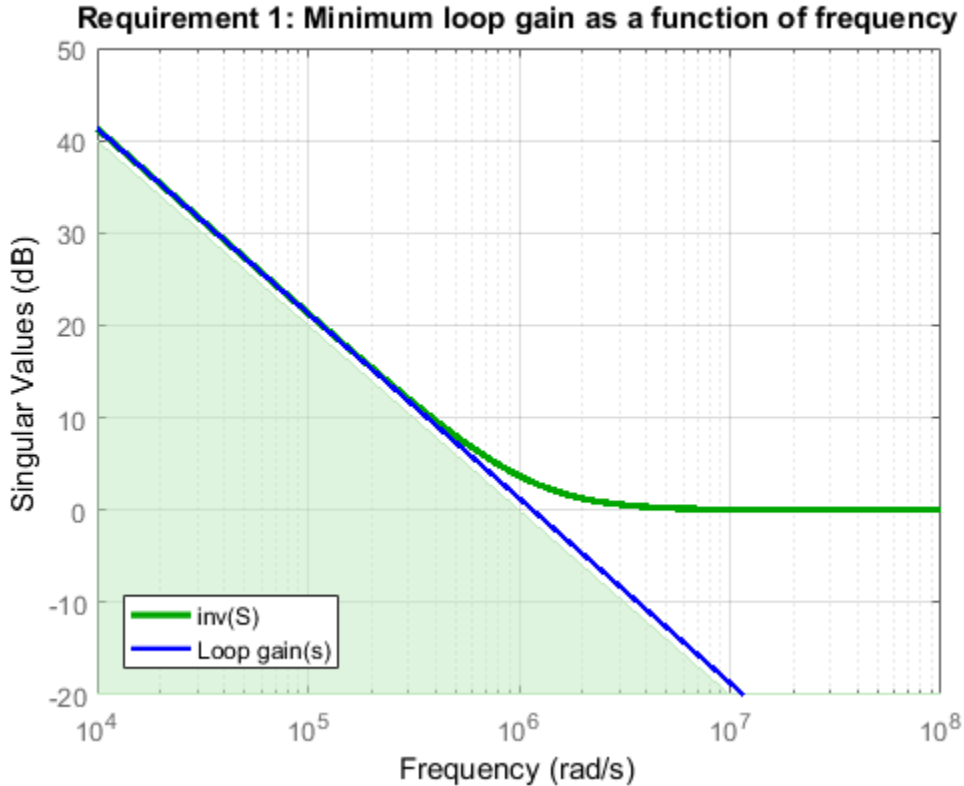
```
Rtrack = TuningGoal.Tracking('r', 'y', 10, 0.01);
Rreject = TuningGoal.Gain('X2', 'y', 0.1);
Rgain = TuningGoal.MinLoopGain('X2', 100, 10000);
Rgain.Openings = 'X1';

[CL, fSoft] = systune(CL0, [Rtrack, Rreject, Rgain]);

Final: Soft = 1.04, Hard = -Inf, Iterations = 149
```

Examine the TuningGoal.MinLoopGain requirement against the corresponding tuned response.

```
viewSpec(Rgain, CL)
```



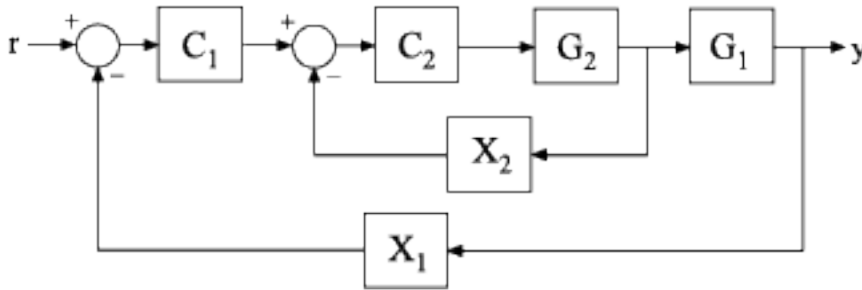
The plot shows the achieved loop gain for the loop at X2 (blue line). The plot also shows the inverse of the achieved sensitivity function,  $S$ , at the location X2 (green line). The inverse sensitivity function at this location is given by  $\text{inv}(S) = I+L$ . Here,  $L$  is the open-loop point-to-point loop transfer measured at X2.

The minimum loop gain requirement  $R_{\text{gain}}$  is constraint on  $\text{inv}(S)$ , represented in the plot by the green shaded region. The constraint on  $\text{inv}(S)$  can be thought of as a

minimum gain constraint on  $L$  that applies where the gain of  $L$  (or the smallest singular value of  $L$ , for MIMO loops) is greater than 1.

### Loop-Gain Requirement without Stability Constraint on Inner Loop

Create requirements that specify a minimum loop gain of 20 dB (100) at 50 rad/s and a maximum loop gain of  $-20$  dB (0.01) at 1000 rad/s on the inner loop of the following control system.



Create the maximum and minimum loop gain requirements.

```
RMinGain = TuningGoal.MinLoopGain('X2',50,100);
RMaxGain = TuningGoal.MaxLoopGain('X2',1000,0.01);
```

Configure the requirements to apply to the loop gain of the inner loop measured with the outer loop open.

```
RMinGain.Openings = 'X1';
RMaxGain.Openings = 'X1';
```

Setting `Req.Openings` tells the tuning algorithm to enforce the requirements with a loop open at the specified location. With the outer loop open, the requirements apply only to the inner loop.

By default, tuning using `TuningGoal.MinLoopGain` or `TuningGoal.MaxLoopGain` imposes a stability requirement as well as the minimum or maximum loop gain.

Practically, in some control systems it is not possible to achieve a stable inner loop. In that case, remove the stability requirement for the inner loop by setting the `Stabilize` property to `false`.

```
RMinGain.Stabilize = false;  
RMaxGain.Stabilize = false;
```

When you tune using either of these requirements, the tuning algorithm still imposes a stability requirement on the overall tuned control system, but not on the inner loop alone.

## See Also

`systemtune` (for `slTuner`) | `TuningGoal.Gain` | `TuningGoal.MaxLoopGain` | `TuningGoal.Margins` | `slTuner` | `looptune` | `systemtune` | `looptune` (for `slTuner`) | `viewSpec` | `evalSpec` | `TuningGoal.LoopShape` | `sigma`

## How To

- “Loop Shape and Stability Margin Specifications”
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”



# TuningGoal.MaxLoopGain class

**Package:** TuningGoal

Maximum loop gain constraint for control system tuning

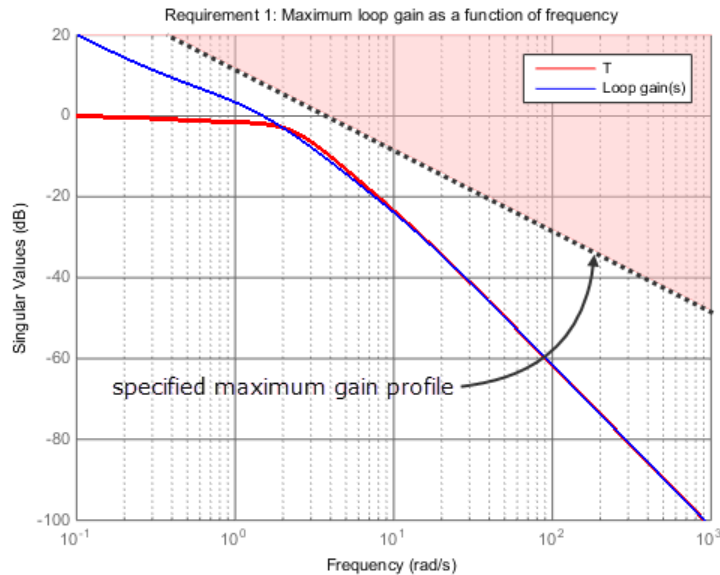
## Description

Use the `TuningGoal.MaxLoopGain` object to enforce a maximum loop gain and desired roll-off in a particular frequency band. Use this requirement with control system tuning commands such as `systune` or `looptune`.

This requirement imposes a maximum gain on the open-loop frequency response ( $L$ ) at a specified location in your control system. You specify the maximum open-loop gain as a function of frequency (a maximum *gain profile*). For MIMO feedback loops, the specified gain profile is interpreted as an upper bound on the largest singular value of  $L$ .

When you tune a control system, the maximum gain profile is converted to a maximum gain constraint on the complementary sensitivity function,  $T = L/(I + L)$ .

The following figure shows a typical specified maximum gain profile (dashed line) and a resulting tuned loop gain,  $L$  (blue line). The pink region represents gain profile values that are forbidden by this requirement. The figure shows that when  $L$  is much smaller than 1, imposing a maximum gain on  $T$  is a good proxy for a maximum open-loop gain.



`TuningGoal.MaxLoopGain` and `TuningGoal.MinLoopGain` specify only high-gain or low-gain constraints in certain frequency bands. When you use these requirements, `systemtune` and `looptune` determine the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use `TuningGoal.LoopShape` to specify that target loop shape.

## Construction

`Req = TuningGoal.MaxLoopGain(location, loopgain)` creates a tuning requirement for limiting the gain of a SISO or MIMO feedback loop. The requirement limits the open-loop frequency response measured at the specified locations to the maximum gain profile specified by `loopgain`. You can specify the maximum gain profile as a smooth transfer function or sketch a piecewise error profile using an `frd` model or the `makeweight` command. Only gain values smaller than 1 are enforced.

`Req = TuningGoal.MaxLoopGain(location, fmax, gmax)` specifies a maximum gain profile of the form  $\text{loopgain} = K/s$  (integral action). The software chooses  $K$  such that the gain value is `gmax` at the specified frequency, `fmax`.

## Input Arguments

### location

Location at which the maximum open-loop gain is constrained, specified as a character vector or cell array of character vectors that identify one or more locations in the control system to tune. What loop-opening locations are available depends on what kind of system you are tuning:

- If you are tuning a Simulink model of a control system, you can use any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. For example, if the `sITuner` interface contains an analysis point `u`, you can use `'u'` to refer to that point when creating tuning goals. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.
- If you are tuning a generalized state-space (`genss`) model of a control system, you can use any `AnalysisPoint` location in the control system model. For example, the following code creates a PI loop with an analysis point at the plant input `'u'`.

```
AP = AnalysisPoint('u');
G = tf(1,[1 2]);
C = tunablePID('C','pi');
T = feedback(G*AP*C,1);
```

When creating tuning goals, you can use `'u'` to refer to the analysis point at the plant input. Use `getPoints` to get the list of analysis points available in a `genss` model.

If `location` is a cell array of loop-opening locations, then the maximum gain requirement applies to the resulting MIMO loop.

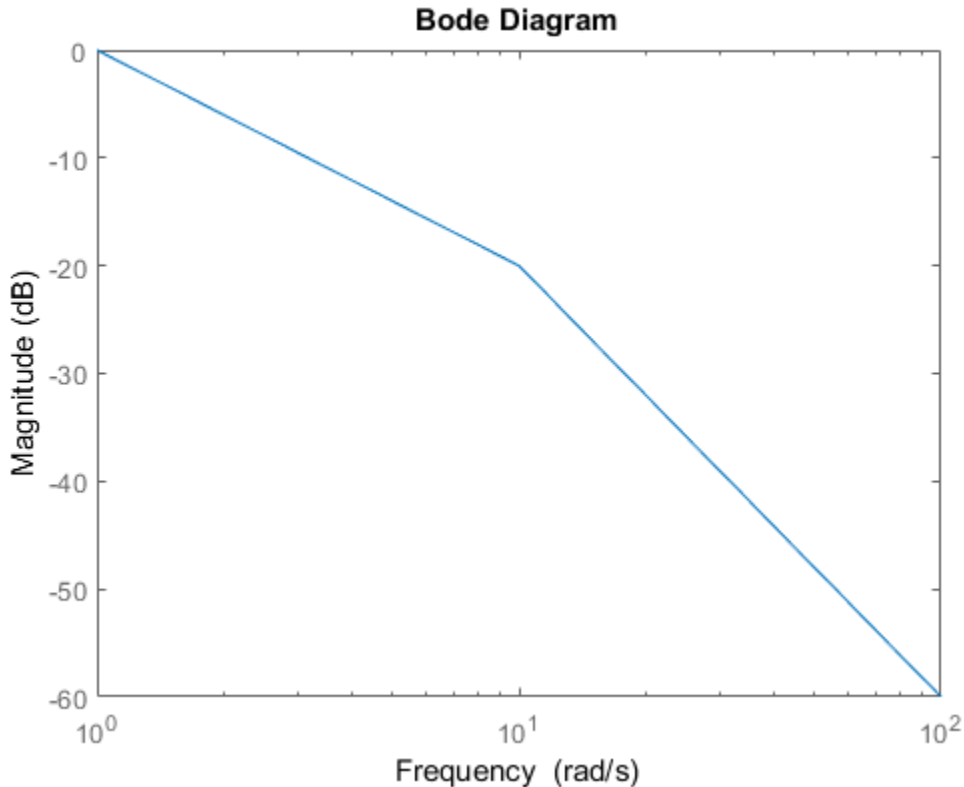
### loopgain

Maximum open-loop gain as a function of frequency.

You can specify `loopgain` as a smooth SISO transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise gain profile using a `frd` model or the `makeweight` command. For example, the following `frd` model specifies a maximum gain of 1 (0 dB) at 1 rad/s, rolling off at a rate of  $-20$  dB/dec up to 10 rad/s, and a rate of  $-40$  dB/dec at higher frequencies.

```
loopgain = frd([1 1e-1 1e-3],[1 10 100]);
```

```
bodemag(loopgain)
```



When you use an `frd` model to specify `loopgain`, the software automatically maps your specified gain profile to a `zpk` model. The magnitude of this model approximates the desired gain profile. Use `viewSpec(Req)` to plot the magnitude of that `zpk` model.

Only gain values smaller than 1 are enforced. For multi-input, multi-output (MIMO) feedback loops, the gain profile is interpreted as a minimum roll-off requirement, which is an upper bound on the largest singular value of  $L$ . For more information about singular values, see `sigma`.

**fmax**

Frequency of maximum gain `gmax`, specified as a scalar value in rad/s.

Use this argument to specify a maximum gain profile of the form  $\text{loopgain} = K/s$  (integral action). The software chooses  $K$  such that the gain value is  $g_{\text{max}}$  at the specified frequency,  $f_{\text{max}}$ .

### **gmax**

Value of maximum gain occurring at  $f_{\text{max}}$ , specified as a scalar absolute value.

Use this argument to specify a maximum gain profile of the form  $\text{loopgain} = K/s$  (integral action). The software chooses  $K$  such that the gain value is  $g_{\text{max}}$  at the specified frequency,  $f_{\text{max}}$ .

## **Properties**

### **MaxGain**

Maximum open-loop gain as a function of frequency, specified as a SISO zpk model.

The software automatically maps the input argument `loopgain` onto a zpk model. The magnitude of this zpk model approximates the desired gain profile. Alternatively, if you use the `fmax` and `gmax` arguments to specify the gain profile, this property is set to  $K/s$ . The software chooses  $K$  such that the gain value is  $g_{\text{max}}$  at the specified frequency,  $f_{\text{max}}$ .

Use `viewSpec(Req)` to plot the magnitude of the open-loop maximum gain profile.

### **Focus**

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `Focus` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning ( $\text{rad}/\text{TimeUnit}$ ). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0,Inf]` for continuous time; `[0,pi/Ts]` for discrete time, where `Ts` is the model sample time.

**Stabilize**

Stability requirement on closed-loop dynamics, specified as 1 (**true**) or 0 (**false**).

When **Stabilize** is **true**, this requirement stabilizes the specified feedback loop, as well as imposing gain or loop-shape requirements. Set **Stabilize** to **false** if stability for the specified loop is not required or cannot be achieved.

**Default:** 1 (**true**)

**LoopScaling**

Toggle for automatically scaling loop signals, specified as 'on' or 'off'.

In multi-loop or MIMO control systems, the feedback channels are automatically rescaled to equalize the off-diagonal terms in the open-loop transfer function (loop interaction terms). Set **LoopScaling** to 'off' to disable such scaling and shape the unscaled open-loop response.

**Default:** 'on'

**Location**

Location at which minimum loop gain is constrained, specified as a cell array of character vectors that identify one or more analysis points in the control system to tune. For example, if **Location** = {'u'}, the tuning goal evaluates the open-loop response measured at an analysis point 'u'. If **Location** = {'u1', 'u2'}, the tuning goal evaluates the MIMO open-loop response measured at analysis points 'u1' and 'u2'.

The value of the **Location** property is set by the **location** input argument when you create the requirement.

**Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the **Models** property when tuning an array of control system models with **systune**, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, **Req**, to the second, third, and fourth models in a model array passed to **systune**. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then **Openings** can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then **Openings** can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1' , 'u2' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** []

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ .

Here,  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.MaxLoopGain` requirement,  $f(x)$  is given by:

$$f(x) = \left\| W_T \left( D^{-1} T D \right) \right\|_{\infty}.$$

$W_T$  is the reciprocal of the maximum loop gain profile, `MaxGain`.  $D$  is a diagonal scaling (for MIMO loops).  $T$  is the complementary sensitivity function at `Location`.

Although  $T$  is a closed-loop transfer function, driving  $f(x) < 1$  is equivalent to enforcing an upper bound on the open-loop transfer,  $L$ , in a frequency band where the gain of  $L$  is less than one. To see why, note that  $T = L/(I + L)$ . For SISO loops, when  $|L| \ll 1$ ,  $|T| \approx |L|$ . Therefore, enforcing the open-loop maximum gain requirement,  $|L| < 1/|W_T|$ , is roughly equivalent to enforcing  $|W_T T| < 1$ . For MIMO loops, similar reasoning applies, with  $\|T\| \approx \sigma_{\max}(L)$ , where  $\sigma_{\max}$  is the largest singular value.

## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at `Location`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### Maximum Loop Gain Requirement

Create a requirement that limits the maximum open-loop gain of a feedback loop to a specified profile.

Suppose that you are tuning a control system that has a loop-opening location identified by `PILoop`. Limit the open-loop gain measured at that location to 1 (0 dB) at 1 rad/

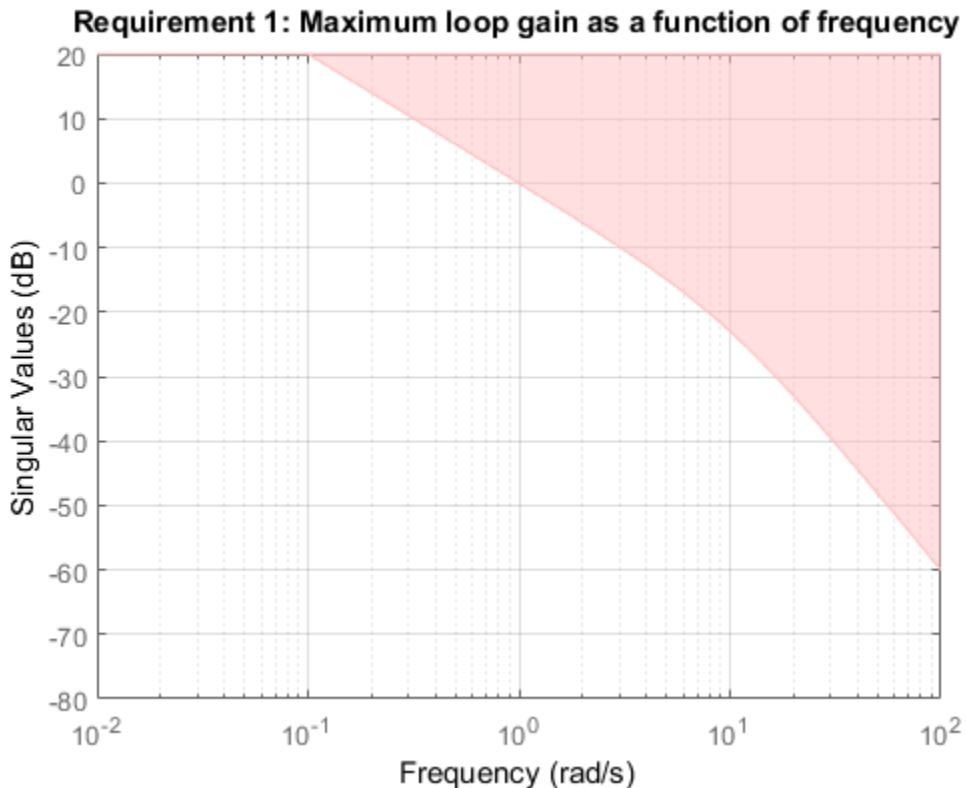


s, rolling off at a rate of -20 dB/dec up to 10 rad/s, and a rate of -40 dB/dec at higher frequencies. Use an `frd` model to sketch this gain profile.

```
loopgain = frd([1 1e-1 1e-3],[1 10 100]);
Req = TuningGoal.MaxLoopGain('PILoop',loopgain);
```

The software converts `loopgain` into a smooth function of frequency that approximates the piecewise-specified requirement. Display the requirement using `viewSpec`.

```
viewSpec(Req)
```



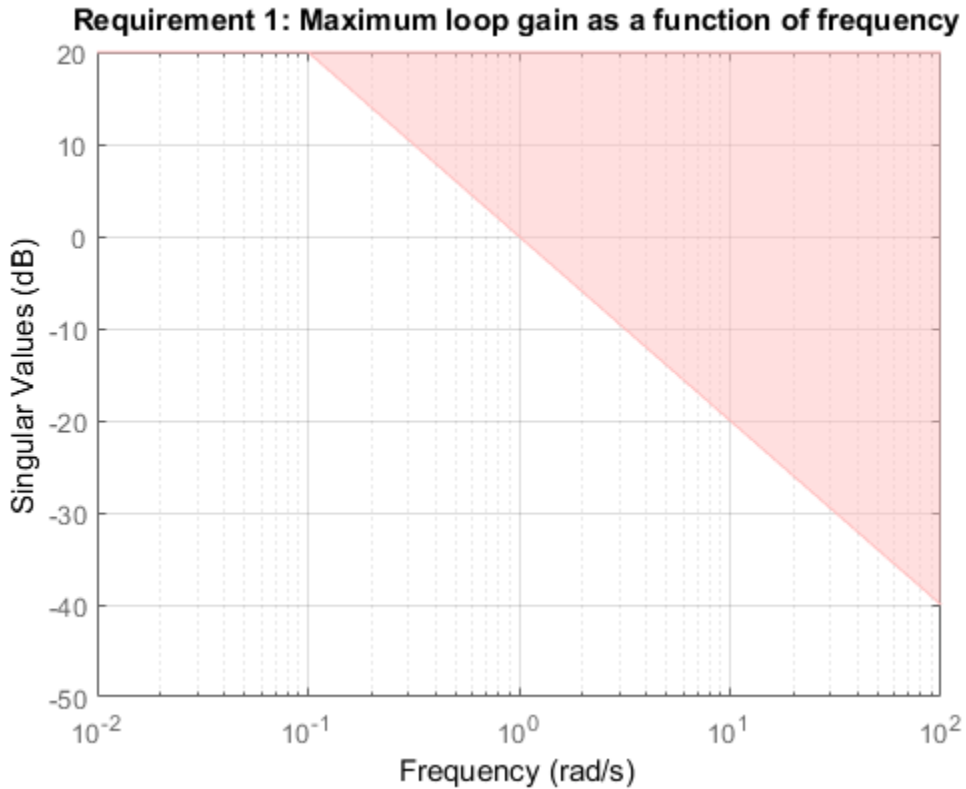
The yellow region indicates where the requirement is violated, except that gain values greater than 1 are not enforced. Therefore, this requirement only specifies minimum roll-off rates at frequencies above 1 rad/s.

You can use `Req` as an input to `looptune` or `systemtune` when tuning the control system.

### Integral Loop Gain Specified as Gain Value at Single Frequency

Create a requirement that specifies a maximum loop gain of the form  $L = K / s$ . The maximum gain attains the value of -20 dB (0.01) at 100 rad/s.

```
Req = TuningGoal.MaxLoopGain('X',100,0.01);  
viewSpec(Req)
```

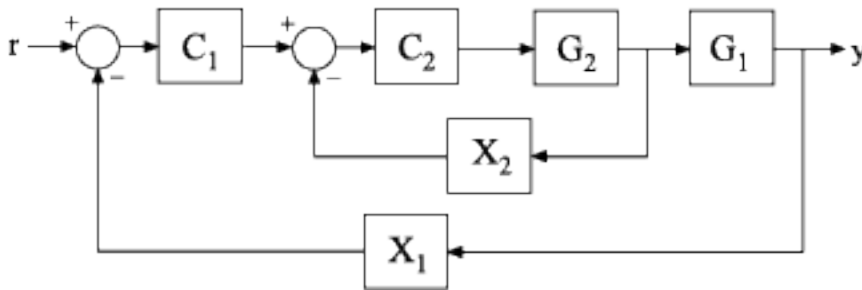


`viewSpec` confirms that the requirement is correctly specified. You can use this requirement to tune a control system that has a loop-opening location identified as 'X'.

Since loop gain values greater than 1 are ignored, this requirement specifies a rolloff of 20 dB/decade above 1 rad/s, with no restriction on loop gain below that frequency.

### Loop-Gain Requirement without Stability Constraint on Inner Loop

Create requirements that specify a minimum loop gain of 20 dB (100) at 50 rad/s and a maximum loop gain of  $-20$  dB (0.01) at 1000 rad/s on the inner loop of the following control system.



Create the maximum and minimum loop gain requirements.

```
RMinGain = TuningGoal.MinLoopGain('X2',50,100);
RMaxGain = TuningGoal.MaxLoopGain('X2',1000,0.01);
```

Configure the requirements to apply to the loop gain of the inner loop measured with the outer loop open.

```
RMinGain.Openings = 'X1';
RMaxGain.Openings = 'X1';
```

Setting `Req.Openings` tells the tuning algorithm to enforce the requirements with a loop open at the specified location. With the outer loop open, the requirements apply only to the inner loop.

By default, tuning using `TuningGoal.MinLoopGain` or `TuningGoal.MaxLoopGain` imposes a stability requirement as well as the minimum or maximum loop gain.

Practically, in some control systems it is not possible to achieve a stable inner loop. In that case, remove the stability requirement for the inner loop by setting the `Stabilize` property to `false`.

```
RMinGain.Stabilize = false;  
RMaxGain.Stabilize = false;
```

When you tune using either of these requirements, the tuning algorithm still imposes a stability requirement on the overall tuned control system, but not on the inner loop alone.

## See Also

`looptune` (for `sITuner`) | `TuningGoal.Gain` | `TuningGoal.MinLoopGain` | `TuningGoal.Margins` | `sITuner` | `looptune` | `system` | `system` (for `sITuner`) | `viewSpec` | `evalSpec` | `TuningGoal.LoopShape` | `sigma`

## How To

- “Loop Shape and Stability Margin Specifications”
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”
- “MIMO Control of Diesel Engine”
- “Tuning of a Two-Loop Autopilot”

# TuningGoal.Overshoot class

**Package:** TuningGoal

Overshoot constraint for control system tuning

## Description

Use the `TuningGoal.Overshoot` object to limit the overshoot in the step response from specified inputs to specified outputs of a control system. Use this requirement for control system tuning with tuning commands such as `systemtune` or `looptune`.

## Construction

`Req = TuningGoal.Overshoot(inputname,outputname,maxpercent)` creates a tuning requirement for limiting the overshoot in the step response between the specified signal locations. The scalar `maxpercent` specifies the maximum overshoot as a percentage.

When you use `TuningGoal.Overshoot` for tuning, the software maps overshoot constraints to peak gain constraints assuming second-order system characteristics. Therefore, the mapping is only approximate for higher-order systems. In addition, this requirement cannot reliably reduce the overshoot below 5%.

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input.
  - Any linear analysis point marked in the model.

- Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1', 'u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `outputname` can include:
  - Any model output.

- Any linear analysis point marked in the model.
- Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `y1` and `y2`. Use `'y1'` to designate that point as an output signal when creating tuning goals. Use `{'y1','y2'}` to designate a two-channel output.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `outputname` can include:

- Any output of the `genss` model
- Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `outputname` can be any output name in `T.OutputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `outputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `outputname` is an `AnalysisPoint` location of a generalized model, the output signal for the tuning goal is the implied output associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **maxpercent**

Maximum percent overshoot, specified as a scalar value. For example, the following code specifies a maximum 5% overshoot in the step response from `'r'` to `'y'`.

```
Req = TuningGoal.Overshoot('r','y',5);
```

TuningGoal.OverShoot cannot reliably reduce the overshoot below 5%.

## Properties

### MaxOvershoot

Maximum percent overshoot, specified as a scalar value. For example, the scalar value 5 means the overshoot should not exceed 5%. The initial value of the MaxOvershoot property is set by the maxpercent input argument when you construct the requirement object.

### InputScaling

Reference signal scaling, specified as a vector of positive real values.

For a MIMO tuning requirement, when the choice of units results in a mix of small and large signals in different channels of the response, use this property to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

For example, suppose that Req is a requirement that signals { 'y1' , 'y2' } track reference signals { 'r1' , 'r2' }. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If r1 and r2 have comparable amplitudes, then it is sufficient to keep the gains from r1 to y2 and r2 and y1 below 0.1. However, if r1 is 100 times larger than r2, the gain from r1 to y2 must be less than 0.001 to ensure that r1 changes y2 by less than 10% of the r2 target. To ensure this result, set the InputScaling property as follows.

```
Req.InputScaling = [100,1];
```

This tells the software to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, [ ] , means no scaling.

**Default:** [ ]

### Input

Input signal names, specified as a cell array of character vectors that identify the inputs of the transfer function that the tuning requirement constrains. The initial value of



the `Input` property is set by the `inputname` input argument when you construct the requirement object.

### Output

Output signal names, specified as a cell array of character vectors that identify the outputs of the transfer function that the tuning requirement constrains. The initial value of the `Output` property is set by the `outputname` input argument when you construct the requirement object.

### Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `sysTune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `sysTune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1', 'u2' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### **Name**

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## **Examples**

### **Overshoot Constraint**

Create a requirement that limits the overshoot of the step response from signals named `'r'` to `'y'` in a control system to 8 percent.

```
Req = TuningGoal.Overshoot('r','y',8);
```

You can use `Req` as an input to `looptune` or `systemtune` when tuning the control system.

Configure the requirement to apply only to the second model in a model array to tune. Also, configure the requirement to be evaluated with a loop open in the control system.

```
Req.Models = 2;  
Req.Openings = 'OuterLoop';
```

Setting the `Models` property restricts application of the requirement to the second model in an array, when you use the requirement to tune an array of control system models. Setting the `Openings` property specifies that requirement is evaluated with a loop opened at the location in the control system identified by `'OuterLoop'`.

## **Tips**

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from `Input` to `Output`, evaluated with loops opened at the points

identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Algorithms

When you use a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ .  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$ , or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For `TuningGoal.Overshoot`,  $f(x)$  reflects the relative satisfaction or violation of the goal. The percent deviation from  $f(x) = 1$  roughly corresponds to the percent deviation from the specified overshoot target. For example,  $f(x) = 1.2$  means the actual overshoot exceeds the target by roughly 20%, and  $f(x) = 0.8$  means the actual overshoot is about 20% less than the target.

`TuningGoal.Overshoot` uses  $\|T\|_{\infty}$  as a proxy for the overshoot, based on second-order model characteristics. Here,  $T$  is the closed-loop transfer function that the requirement constrains. The overshoot is tuned in the range from 5% ( $\|T\|_{\infty} = 1$ ) to 100% ( $\|T\|_{\infty}$ ). `TuningGoal.Overshoot` is ineffective at forcing the overshoot below 5%.

## See Also

`system` (for `sITuner`) | `TuningGoal.Gain` | `looptune` | `system` | `looptune` (for `sITuner`) | `viewSpec` | `evalSpec` | `TuningGoal.Sensitivity` | `sITuner`

## How To

- “Time-Domain Specifications”
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”

## TuningGoal.Passivity class

**Package:** TuningGoal

Passivity constraint for control system tuning

### Description

A system is *passive* if all its I/O trajectories  $(u(t),y(t))$  satisfy:

$$\int_0^T y(t)^\top u(t) dt > 0,$$

for all  $T > 0$ . Equivalently, a system is passive if its frequency response is positive real, which means that for all  $\omega > 0$ ,

$$G(j\omega) + G(j\omega)^H > 0$$

Use `TuningGoal.Passivity` to enforce passivity of the response between specified inputs and outputs, when using a control system tuning command such as `systemtune`. You can also use `TuningGoal.Passivity` to ensure a particular excess or shortage of passivity (see `getPassiveIndex`).

### Construction

`Req = TuningGoal.Passivity(inputname,outputname)` creates a tuning goal for enforcing passivity of the response from the specified inputs to the specified outputs.

`Req = TuningGoal.Passivity(inputname,outputname,nu,rho)` creates a tuning goal for enforcing:

$$\int_0^T y(t)^\top u(t) dt > \nu \int_0^T u(t)^\top u(t) dt + \rho \int_0^T y(t)^\top y(t) dt,$$

for all  $T > 0$ . This requirement enforces an excess of passivity at the inputs or outputs when  $\nu > 0$  or  $\rho > 0$ , respectively. The requirement allows for a shortage of input passivity when  $\nu < 0$ . See `getPassiveIndex` for more information about these indices.

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1', 'u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **y1** and **y2**. Use **'y1'** to designate that point as an output signal when creating tuning goals. Use **{'y1','y2'}** to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **outputname** can include:
  - Any output of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **outputname** can be any output name in **T.OutputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **outputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **outputname** is an **AnalysisPoint** location of a generalized model, the output signal for the tuning goal is the implied output associated with the **AnalysisPoint** block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **nu**

Target passivity at the inputs listed in `inputname`, specified as a scalar value. The input passivity index is defined as the largest value of  $\nu$  for which:

$$\int_0^T y(t)^\top u(t) dt > \nu \int_0^T u(t)^\top u(t) dt,$$

for all  $T > 0$ . Equivalently, `nu` is the largest  $\nu$  for which:

$$G(j\omega) + G(j\omega)^H > 2\nu I$$

When you set a target `nu` in a `TuningGoal.Passivity` goal, the tuning software:

- Ensures that the specified response is input strictly passive when `nu` > 0. The magnitude of `nu` sets the required excess of passivity.
- Allows the response to be not input strictly passive when `nu` < 0. The magnitude of `nu` sets the permitted shortage of passivity.

**Default:** 0

### **rho**

Target passivity at the outputs listed in `outputname`, specified as a scalar value. The output passivity index is defined as the largest value of  $\rho$  for which:

$$\int_0^T y(t)^\top u(t) dt > \rho \int_0^T y(t)^\top y(t) dt,$$

for all  $T > 0$ .

When you set a target `rho` in a `TuningGoal.Passivity` goal, the tuning software:

- Ensures that the specified response is output strictly passive when `rho > 0`. The magnitude of `rho` sets the required excess of passivity.
- Allows the response to be not output strictly passive when `rho < 0`. The magnitude of `rho` sets the permitted shortage of passivity.

**Default:** 0

## Properties

### IPX

Target passivity at the inputs, stored as a scalar value. This value specifies the required amount of passivity at the inputs listed in `inputname`. The initial value of this property is set by the input argument `nu` when you create the `TuningGoal.Passivity` goal.

**Default:** 0

### OPX

Target passivity at the outputs, stored as a scalar value. This value specifies the required amount of passivity at the outputs listed in `outputname`. The initial value of this property is set by the input argument `rho` when you create the `TuningGoal.Passivity` goal.

**Default:** 0

### Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `FOCUS` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (`rad/TimeUnit`). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:



```
Req.Focus = [1,100];
```

**Default:** [0, Inf] for continuous time; [0, pi/Ts] for discrete time, where Ts is the model sample time.

### Input

Input signal names, specified as a cell array of character vectors. The input signal names specify the input locations for determining passivity, initially populated by the `inputname` argument.

### Output

Output signal names, specified as a cell array of character vectors. The output signal names specify the output locations for determining passivity, initially populated by the `outputname` argument.

### Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with  `systune` , to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear

analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning goal, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning goal is a hard constraint.

For the `TuningGoal.Passivity` goal, for a closed-loop transfer function  $G(s, x)$  from `inputname` to `outputname`,  $f(x)$  is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

$R$  is the relative sector index (see `getSectorIndex`) of  $[G(s, x); I]$ , for the sector represented by:

$$Q = \begin{pmatrix} 2\rho & -I \\ -I & 2\nu \end{pmatrix},$$

using the values of the OPX and IPX properties for  $\rho$  and  $\nu$ , respectively.

## Tips

- Use `viewSpec` to visualize this tuning goal. For enforcing passivity with `nu = 0` and `rho = 0`, `viewSpec` plots the relative passivity indices as a function of frequency (see `passiveplot`). These are the singular values of  $(I - G(j\omega))(I - G(j\omega))^{-1}$ . The transfer function  $G$  from `inputname` to `outputname` (evaluated with loops open as specified in `Openings`) is passive when the largest singular value is less than 1 at all frequencies.

For nonzero `nu` or `rho`, `viewSpec` plots the relative index as described in “Algorithms” on page 1-100.

- This tuning goal imposes an implicit minimum-phase constraint on the transfer function  $G + I$ . The transmission zeros of  $G + I$  are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

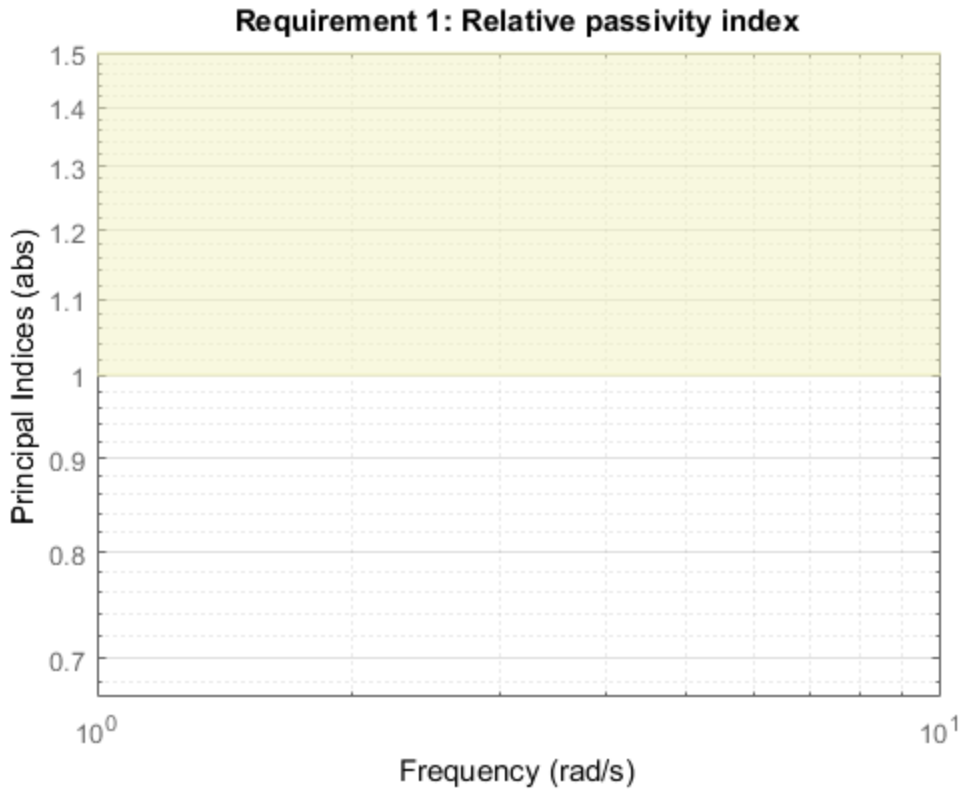
### Passivity Requirement

Create a requirement that ensures passivity in the response from an input or analysis point 'u' to an output or analysis point 'y' in a control system.

```
TG = TuningGoal.Passivity('u', 'y');
```

Use `viewSpec` to visualize the tuning goal.

```
viewSpec(TG)
```



The requirement is satisfied when the relative passivity index  $R < 1$  at all frequencies. The shaded area represents the region where the requirement is violated. When you use this requirement to tune a control system CL, `viewSpec(TG, CL)` shows  $R$  for the specified inputs and outputs on this plot, enabling you to identify frequency ranges in which the passivity requirement is violated.

### Input Passivity in Specified Frequency Range

Create a requirement that ensures that the response from an input 'u' to an output 'y' is input strictly passive, with an excess of passivity of 2.

```
TGi = TuningGoal.Passivity('u', 'y', 2, 0);
```

Restrict the requirement to apply only within the frequency range between 0 and 10 rad/s.

```
TGi.Focus = [0 10];
```

## See Also

`systemtune` (for `sITuner`) | `TuningGoal.WeightedPassivity` | `looptune` | `systemtune` | `looptune` (for `sITuner`) | `viewSpec` | `evalSpec` | `sITuner` | `getPassiveIndex` | `passiveplot`

## How To

- “About Passivity and Passivity Indices”
- “Tuning Control Systems with SYSTUNE”
-

# TuningGoal.Poles class

**Package:** TuningGoal

Constraint on control system dynamics

## Description

Use the `TuningGoal.Poles` object to specify a tuning requirement for constraining the closed-loop dynamics of a control system or of specific feedback loops within the control system. You can use this requirement for control system tuning with tuning commands, such as `sys tune` or `looptune`. A `TuningGoal.Poles` requirement can ensure a minimum decay rate or minimum damping of the poles of the control system or loop. The requirement can also eliminate fast dynamics in the tuned system.

## Construction

`Req = TuningGoal.Poles(mindecay, mindamping, maxfreq)` creates a default template for constraining the closed-loop pole locations. The minimum decay rate, minimum damping constant, and maximum natural frequency define a region of the complex plane in which poles of the component must lie. Set `mindecay = 0`, `mindamping = 0`, or `maxfreq = Inf` to skip any of the three constraints.

`Req = TuningGoal.Poles(location, mindecay, mindamping, maxfreq)` constrains the poles of the sensitivity function measured at a specified location in the control system. (See `getSensitivity` for information about sensitivity functions.) Use this syntax to narrow the scope of the requirement to a particular feedback loop.

If you want to constrain the poles of the system with one or more feedback loops opened, set the `Openings` property. To limit the enforcement of this requirement to poles having natural frequency within a specified frequency range, set the `Focus` property. (See “Properties” on page 1-106.)

## Input Arguments

### **mindecay**

Minimum decay rate of poles of tunable component, specified as a nonnegative scalar value in the frequency units of the control system model you are tuning.

When you tune the control system using this requirement, the closed-loop poles of the control system are constrained to satisfy:

- $\text{Re}(s) < -\text{mindecay}$ , for continuous-time systems.
- $\log(|z|) < -\text{mindecay} \cdot T_s$ , for discrete-time systems with sample time  $T_s$ .

Set `mindecay = 0` to impose no constraint on the decay rate.

### **mindamping**

Desired minimum damping ratio of the closed-loop poles, specified as a value between 0 and 1.

Poles that depend on the tunable parameters are constrained to satisfy  $\text{Re}(s) < -\text{mindamping} \cdot |s|$ . In discrete time, the damping ratio is computed using  $s = \log(z) / T_s$ .

Set `mindamping = 0` to impose no constraint on the damping ratio.

### **maxfreq**

Desired maximum natural frequency of closed-loop poles, specified as a scalar value in the frequency units of the control system model you are tuning.

Poles are constrained to satisfy  $|s| < \text{maxfreq}$  for continuous time, or  $|\log(z)| < \text{maxfreq} \cdot T_s$  for discrete-time systems with sample time  $T_s$ . This constraint prevents fast dynamics in the closed-loop system.

Set `maxfreq = Inf` to impose no constraint on the natural frequency.

### **location**

Location at which poles are assessed, specified as a character vector or cell array of character vectors that identify one or more locations in the control system to tune. When you use this input, the requirement constrains the poles of the sensitivity function

measured at this location. (See `getSensitivity` for information about sensitivity functions.) What locations are available depends on what kind of system you are tuning:

- If you are tuning a Simulink model of a control system, you can use any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. For example, if the `sITuner` interface contains an analysis point `u`, you can use `'u'` to refer to that point when creating tuning goals. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.
- If you are tuning a generalized state-space (`genss`) model of a control system, you can use any `AnalysisPoint` location in the control system model. For example, the following code creates a PI loop with an analysis point at the plant input `'u'`.

```
AP = AnalysisPoint('u');  
G = tf(1,[1 2]);  
C = tunablePID('C','pi');  
T = feedback(G*AP*C,1);
```

When creating tuning goals, you can use `'u'` to refer to the analysis point at the plant input. Use `getPoints` to get the list of analysis points available in a `genss` model.

If `location` specifies multiple locations, then the poles constraint applies to the sensitivity of the MIMO loop.

## Properties

### **MinDecay**

Minimum decay rate of closed-loop poles of tunable component, specified as a positive scalar value in the frequency units of the control system you are tuning. The initial value of this property is set by the `mindecay` input argument.

When you tune the control system using this requirement, closed-loop poles are constrained to satisfy  $\text{Re}(s) < -\text{MinDecay}$  for continuous-time systems, or  $\log(|z|) < -\text{MinDecay} \cdot T_s$  for discrete-time systems with sample time  $T_s$ .

You can use dot notation to change the value of this property after you create the requirement. For example, suppose `Req` is a `TuningGoal.Poles` requirement. Change the minimum decay rate to 0.001:



```
Req.MinDecay = 0.001;
```

**Default:** 0

### MinDamping

Desired minimum damping ratio of closed-loop poles, specified as a value between 0 and 1. The initial value of this property is set by the `mindamping` input argument.

Poles that depend on the tunable parameters are constrained to satisfy  $\text{Re}(s) < -\text{MinDamping} * |s|$ . In discrete time, the damping ratio is computed using  $s = \log(z) / T_s$ .

**Default:** 0

### MaxFrequency

Desired maximum natural frequency of closed-poles, specified as a scalar value in the frequency units of the control system model you are tuning. The initial value of this property is set by the `maxfreq` input argument.

Poles of the block are constrained to satisfy  $|s| < \text{maxfreq}$  for continuous-time systems, or  $|\log(z)| < \text{maxfreq} * T_s$  for discrete-time systems with sample time  $T_s$ . This constraint prevents fast dynamics in the tuned control system.

You can use dot notation to change the value of this property after you create the requirement. For example, suppose `Req` is a `TuningGoal.ControllerPoles` requirement. Change the maximum frequency to 1000:

```
Req.MaxFrequency = 1000;
```

**Default:** Inf

### Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `Focus` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (rad/TimeUnit). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:**  $[0, \text{Inf}]$  for continuous time;  $[0, \pi/T_s]$  for discrete time, where  $T_s$  is the model sample time.

### Location

Location at which poles are assessed, specified as a cell array of character vectors that identify one or more analysis points in the control system to tune. For example, if `Location = {'u'}`, the tuning goal evaluates the open-loop response measured at an analysis point 'u'. If `Location = {'u1', 'u2'}`, the tuning goal evaluates the MIMO open-loop response measured at analysis points 'u1' and 'u2'.

The initial value of the `Location` property is set by the `location` input argument when you create the `TuningGoal.Poles` requirement.

### Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **Openings** can include any **AnalysisPoint** location in the control system model. Use **getPoints** to get the list of analysis points available in the **genss** model.

For example, if **Openings** = { 'u1' , 'u2' }, then the tuning goal is evaluated with loops open at analysis points **u1** and **u2**.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if **Req** is a tuning goal:

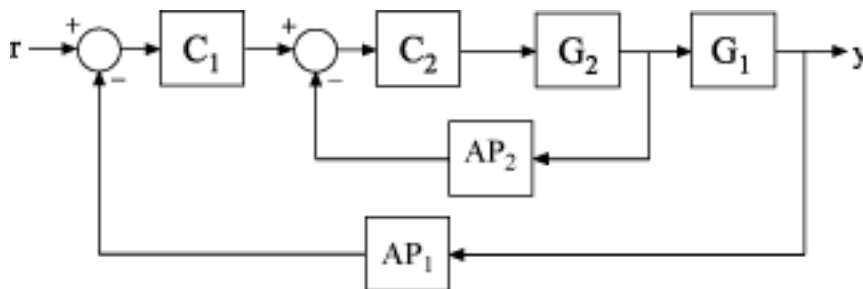
```
Req.Name = 'LoopReq';
```

**Default:** []

## Examples

### Constrain Closed-Loop Dynamics of Specified Loop of System to Tune

Create a requirement that constrains the inner loop of the following control system to be stable and free of fast dynamics. Specify that the constraint is evaluated with the outer loop open.



Create a model of the system. To do so, specify and connect the numeric plant models, **G1** and **G2**, and the tunable controllers **C1** and **C2**. Also, create and connect the

AnalysisPoint blocks, AP1 and AP2, which mark points of interest for analysis and tuning.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
AP1 = AnalysisPoint('AP1');
AP2 = AnalysisPoint('AP2');
T = feedback(G1*feedback(G2*C2,AP2)*C1,AP1);
```

Create a tuning requirement that constrains the dynamics of the closed-loop poles.

Restrict the poles of the inner loop to the region  $\text{Re}(s) < -0.1, |s| < 30$ .

```
Req = TuningGoal.Poles(0.1,0,30);
```

Setting the minimum damping to zero imposes no constraint on the damping constants for the poles.

Specify that the constraint on the tuned system poles is applied with the outer loop open.

```
Req.Openings = 'AP1';
```

When you tune T using this requirement, the constraint applies to the poles of the entire control system evaluated with the loop open at 'AP1'. In other words, the poles of the inner loop plus the poles of C1 and G1 are all considered.

After you tune T, you can use `viewSpec` to validate the tuned control system against the requirement.

### Constrain Dynamics of Specified Feedback Loop

Create a requirement that constrains the inner loop of the system of the previous example to be stable and free of fast dynamics. Specify that the constraint is evaluated with the outer loop open.

Create a tuning requirement that constrains the dynamics of the inner feedback loop, the loop identified by AP2. Restrict the poles of the inner loop to the region  $\text{Re}(s) < -0.1, |s| < 30$ .

```
Req = TuningGoal.Poles('AP2',0.1,0,30);
```

Specify that the constraint on the tuned system poles is applied with the outer loop open.

```
Req.Openings = 'AP1';
```

When you tune **T** using this requirement, the constraint applies only to the poles of the inner loop, evaluated with the outer loop open. In this case, since **G1** and **C1** do not contribute to the sensitivity function at **AP2** when the outer loop is open, the requirement constrains only the poles of **G2** and **C2**.

After you tune **T**, you can use `viewSpec` to validate the tuned control system against the requirement.

## Tips

- `TuningGoal.Poles` restricts the closed-loop dynamics of the tuned control system. To constrain the dynamics or ensure the stability of a single tunable component, use `TuningGoal.ControllerPoles`.

## Algorithms

When you use a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ .  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$ , or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For `TuningGoal.Poles`,  $f(x)$  reflects the relative satisfaction or violation of the goal. For example, if you attempt to constrain the closed-loop poles of a feedback loop to a minimum damping of  $\zeta = 0.5$ , then:

- $f(x) = 1$  means the smallest damping among the constrained poles is  $\zeta = 0.5$  exactly.
- $f(x) = 1.1$  means the smallest damping  $\zeta = 0.5/1.1 = 0.45$ , roughly 10% less than the target.
- $f(x) = 0.9$  means the smallest damping  $\zeta = 0.5/0.9 = 0.55$ , roughly 10% better than the target.

## See Also

`looptune` | `looptune` (for `sITuner`) | `TuningGoal.ControllerPoles` | `systune` | `systune` (for `sITuner`) | `viewSpec` | `evalSpec` | `tunableTF` | `tunableSS`

## **How To**

- “System Dynamics Specifications”
- “Digital Control of Power Stage Voltage”
- “Multi-Loop Control of a Helicopter”

# TuningGoal.Rejection class

**Package:** TuningGoal

Disturbance rejection requirement for control system tuning

## Description

Use the `TuningGoal.Rejection` object to specify the minimum attenuation of a disturbance injected at a specified location in a control system. This requirement helps you tune control systems with tuning commands such as `systemtune` or `looptune`.

When you use a `TuningGoal.Rejection` requirement, the software attempts to tune the system so that the attenuation of a disturbance at the specified location exceeds the minimum attenuation factor you specify. This attenuation factor is the ratio between the open- and closed-loop sensitivities to the disturbance and is a function of frequency. You can achieve disturbance attenuation only inside the control bandwidth. The loop gain must be larger than one for the disturbance to be attenuated (attenuation factor > 1).

## Construction

`Req = TuningGoal.Rejection(distloc,attfact)` creates a tuning requirement for rejecting a disturbance entering at `distloc`. This requirement constrains the minimum disturbance attenuation factor to the frequency-dependent value, `attfact`.

## Input Arguments

### **distloc**

Disturbance location, specified as a character vector or, for multiple-input requirements, a cell array of character vectors.

- If you are using the requirement to tune a Simulink model of a control system, then `distloc` can include any signal identified as an analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as the disturbance input when creating tuning goals. Use `{'u1','u2'}` to designate a two-channel disturbance input.

- If you are using the requirement to tune a generalized state-space model (`genss`) of a control system, then `inputname` can include any `AnalysisPoint` channel in the model. For example, if you are tuning a control system model, `T`, which contains an `AnalysisPoint` block with a location named `AP_u`, then `distloc` can include `'AP_u'`. (Use `getPoints` to get a list of analysis points available in a `genss` model.) The constrained disturbance location is injected at the implied input associated with the analysis point, and measured at the implied output:



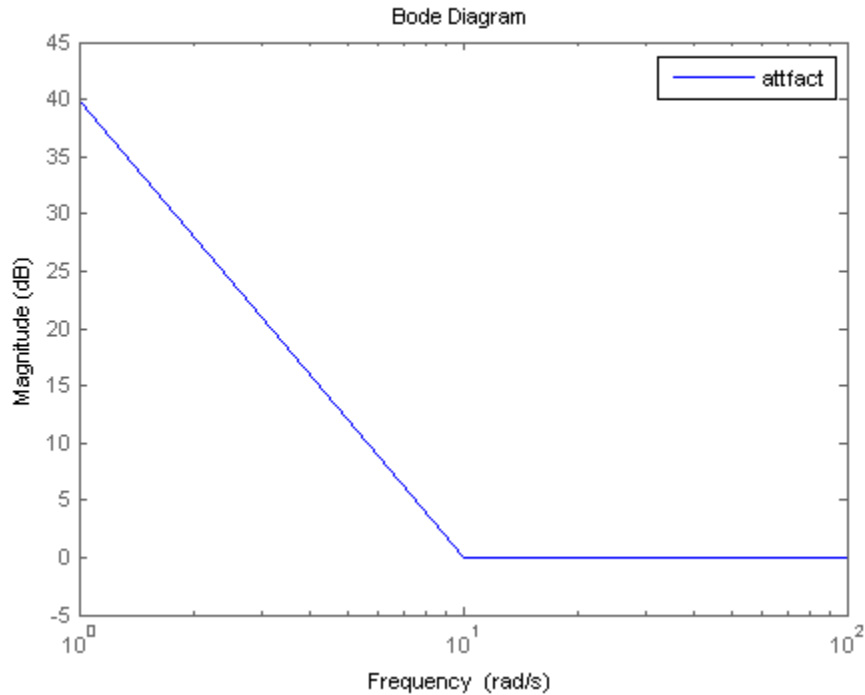
### **attfact**

Attenuation factor as a function of frequency, specified as a numeric LTI model.

The `TuningGoal.Rejection` requirement constrains the minimum disturbance attenuation to the frequency-dependent value `attfact`. You can specify `attfact` as a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can specify a piecewise gain profile using a `frd` model. For example, the following code specifies an attenuation factor of 100 (40 dB) below 1 rad/s, gradually dropping to 1 (0 dB) past 10 rad/s, for a disturbance injected at `u`.

```
attfact = frd([100 100 1 1],[0 1 10 100]);  
Req = TuningGoal.Rejection('u',attfact);  
bodemag(attfact)
```





When you use an `frd` model to specify `attfact`, the gain profile is automatically mapped onto a `zpk` model. The magnitude of this `zpk` model approximates the desired gain profile. Use `viewSpec(Req)` to visualize the resulting attenuation profile.

## Properties

### MinAttenuation

Minimum disturbance attenuation as a function of frequency, expressed as a SISO `zpk` model.

The software automatically maps the `attfact` input argument to a `zpk` model. The magnitude of this `zpk` model approximates the desired attenuation factor and is stored in the `MinAttenuation` property. Use `viewSpec(Req)` to plot the magnitude of `MinAttenuation`.

**Focus**

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the **Focus** property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (`rad/TimeUnit`). For example, suppose **Req** is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0,Inf]` for continuous time; `[0,pi/Ts]` for discrete time, where **Ts** is the model sample time.

**LoopScaling**

Toggle for automatically scaling loop signals, specified as `'on'` or `'off'`.

For multiloop or MIMO disturbance rejection requirements, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Set **LoopScaling** to `'off'` to disable such scaling and shape the unscaled open-loop response.

**Default:** `'on'`

**Location**

Location of disturbance, specified as a specified as a cell array of character vectors that identify one or more analysis points in the control system to tune. For example, if **Location** = `{'u'}`, the tuning goal evaluates disturbance rejection at an analysis point `'u'`. If **Location** = `{'u1','u2'}`, the tuning goal evaluates the rejection at based on the MIMO open-loop response measured at analysis points `'u1'` and `'u2'`.

The initial value of the **Location** property is set by the **distloc** input argument when you create the requirement.

**Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the **Models** property when tuning an array of control system models with **systune**, to enforce a tuning goal for a subset of models in the array. For example, suppose you

want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then **Openings** can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then **Openings** can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** []

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the requirement is converted into a normalized scalar value  $f(x)$ . In this case,  $x$  is the vector of free (tunable) parameters in the control system. The parameter values are adjusted automatically to minimize  $f(x)$  or drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Rejection` requirement,  $f(x)$  is given by:

$$f(x) = \max_{\omega \in \Omega} \|W(j\omega)S(j\omega, x)\|.$$

$W(j\omega)$  is the rational transfer function of the `MinAttenuation` property. This transfer function's magnitude approximates the minimum disturbance attenuation that you specify for the requirement.  $S(j\omega, x)$  is the closed-loop sensitivity function measured at the disturbance location.  $\Omega$  is the frequency interval over which the requirement is enforced, specified in the `Focus` property.

## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at `Location`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### Constant Minimum Attenuation in Frequency Band

Create a tuning requirement that enforces a attenuation of at least a factor of 10 between 0 and 5 rad/s. The requirement applies to a disturbance entering a control system at a point identified as 'u'.

```
Req = TuningGoal.Rejection('u',10);  
Req.Name = 'Rejection spec';  
Req.Focus = [0 5]
```

## Frequency-Dependent Attenuation Profile

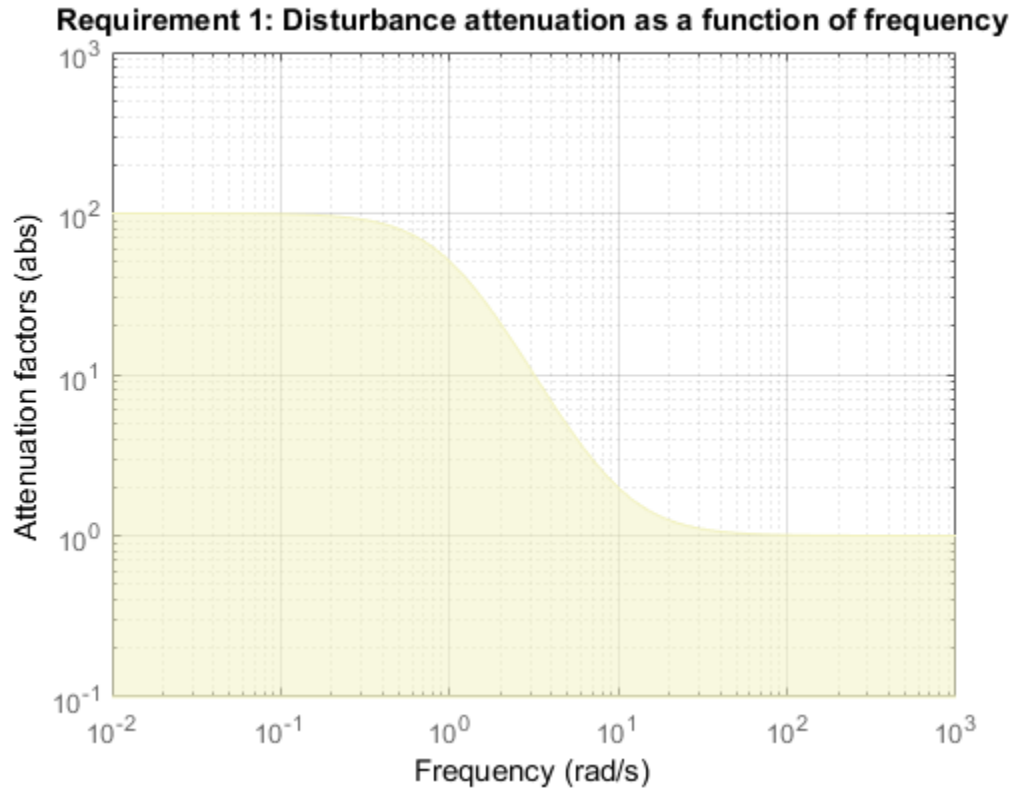
Create a tuning requirement that enforces an attenuation factor of at least 100 (40 dB) below 1 rad/s, gradually dropping to 1 (0 dB) past 10 rad/s. The requirement applies to a disturbance entering a control system at a point identified as 'u'.

```
attfact = frd([100 100 1 1],[0 1 10 100]);  
Req = TuningGoal.Rejection('u',attfact);
```

These commands use a `frd` model to specify the minimum attenuation profile as a function of frequency. The minimum attenuation of 100 below 1 rad/s, together with the minimum attenuation of 1 at the frequencies of 10 and 100 rad/s, specifies the desired rolloff of the requirement.

`attfact` is converted into a smooth function of frequency that approximates the piecewise specified requirement. Display the error requirement using `viewSpec`.

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

### See Also

`systeme` (for `sITuner`) | `TuningGoal.Tracking` | `looptune` | `viewSpec` | `systeme` | `looptune` (for `sITuner`) | `TuningGoal.LoopShape` | `sITuner`

### How To

- “Time-Domain Specifications”
- “Decoupling Controller for a Distillation Column”
- “Tuning of a Two-Loop Autopilot”

# TuningGoal.Sensitivity class

**Package:** TuningGoal

Sensitivity requirement for control system tuning

## Description

Use the `TuningGoal.Sensitivity` object to limit the sensitivity of a feedback loop to disturbances. Constrain the sensitivity to be smaller than one at frequencies where you need good disturbance rejection. Use this requirement for control system tuning with tuning commands such as `systune` or `looptune`.

## Construction

`Req = TuningGoal.Sensitivity(location,maxsens)` creates a tuning requirement for limiting the sensitivity to disturbances entering a feedback loop at the specified location. `maxsens` specifies the maximum sensitivity as a function of frequency. You can specify the maximum sensitivity profile as a smooth transfer function or sketch a piecewise error profile using an `frd` model or the `makeweight` command.

See `getSensitivity` for more information about sensitivity functions.)

## Input Arguments

### **location**

Location at which the sensitivity to disturbances is constrained, specified as a character vector or cell array of character vectors that identify one or more locations in the control system to tune. What locations are available depends on what kind of system you are tuning:

- If you are tuning a Simulink model of a control system, you can use any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. For example, if the `sITuner` interface contains an analysis

point `u`, you can use `'u'` to refer to that point when creating tuning goals. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

- If you are tuning a generalized state-space (`genss`) model of a control system, you can use any `AnalysisPoint` location in the control system model. For example, the following code creates a PI loop with an analysis point at the plant input `'u'`.

```
AP = AnalysisPoint('u');
G = tf(1,[1 2]);
C = tunablePID('C','pi');
T = feedback(G*AP*C,1);
```

When creating tuning goals, you can use `'u'` to refer to the analysis point at the plant input. Use `getPoints` to get the list of analysis points available in a `genss` model.

If `location` is a cell array, then the sensitivity requirement applies to the MIMO loop.

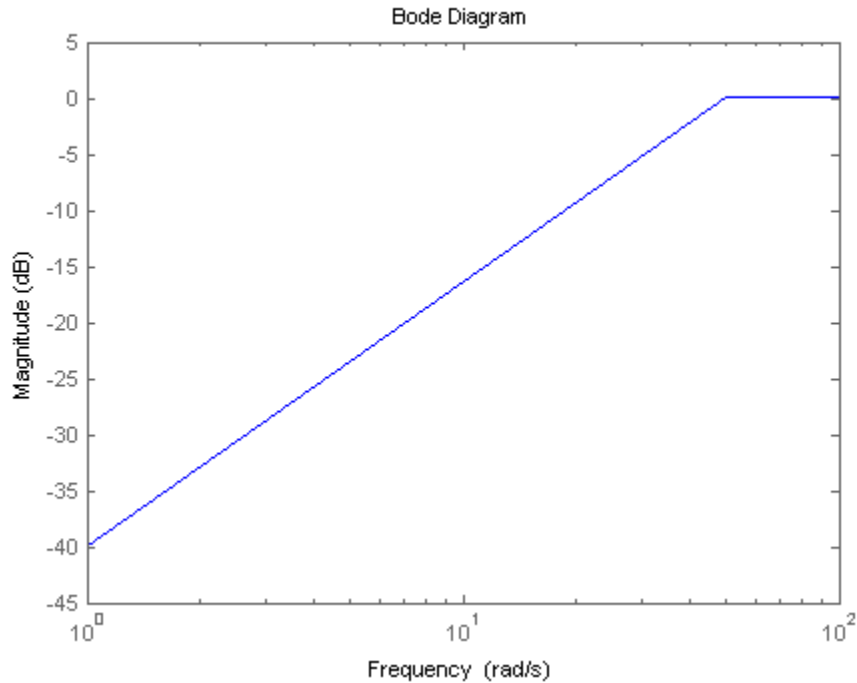
### **maxsens**

Maximum sensitivity to disturbances as a function of frequency.

You can specify `maxsens` as a smooth SISO transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise gain profile using a `frd` model or the `makeweight` command. For example, the following `frd` model specifies a maximum sensitivity of 0.01 (–40 dB) at 1 rad/s, increasing to 1 (0 dB) past 50 rad/s.

```
maxsens = frd([0.01 1 1],[1 50 100]);
bodemag(maxsens)
ylim([-45,5])
```





When you use an `frd` model to specify `maxsens`, the software automatically maps your specified gain profile to a `zpk` model whose magnitude approximates the desired gain profile. Use `viewSpec(Req)` to plot the magnitude of that `zpk` model.

## Properties

### **MaxSensitivity**

Maximum sensitivity as a function of frequency, specified as a SISO `zpk` model.

The software automatically maps the input argument `maxsens` onto a `zpk` model. The magnitude of this `zpk` model approximates the desired gain profile. Use `viewSpec(Req)` to plot the magnitude of the `zpk` model `MaxSensitivity`.

**Focus**

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the **Focus** property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (`rad/TimeUnit`). For example, suppose **Req** is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0,Inf]` for continuous time; `[0,pi/Ts]` for discrete time, where **Ts** is the model sample time.

**LoopScaling**

Toggle for automatically scaling loop signals, specified as `'on'` or `'off'`.

In multi-loop or MIMO control systems, the feedback channels are automatically rescaled to equalize the off-diagonal terms in the open-loop transfer function (loop interaction terms). Set **LoopScaling** to `'off'` to disable such scaling and shape the unscaled sensitivity function.

**Default:** `'on'`

**Location**

Location of disturbance, specified as a cell array of character vectors that identify one or more analysis points in the control system to tune. For example, if **Location** = `{'u'}`, the tuning goal evaluates the open-loop response measured at an analysis point `'u'`. If **Location** = `{'u1','u2'}`, the tuning goal evaluates the MIMO open-loop response measured at analysis points `'u1'` and `'u2'`.

The initial value of the **Location** property is set by the `location` input argument when you create the requirement.

**Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the **Models** property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you

want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then **Openings** can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then **Openings** can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** []

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Sensitivity` requirement,  $f(x)$  is given by:

$$f(x) = \left\| \frac{1}{\text{MaxSensitivity}} S(s,x) \right\|_{\infty}.$$

$S(s,x)$  is the sensitivity function of the control system at `location`.

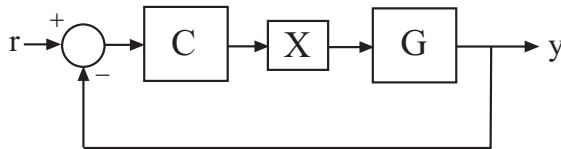
## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at `Location`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### Disturbance Sensitivity at Plant Input

Create a requirement that limits the sensitivity to disturbance at the plant input of the following control system. The control system contains an `AnalysisPoint` block at the plant input named 'X'.

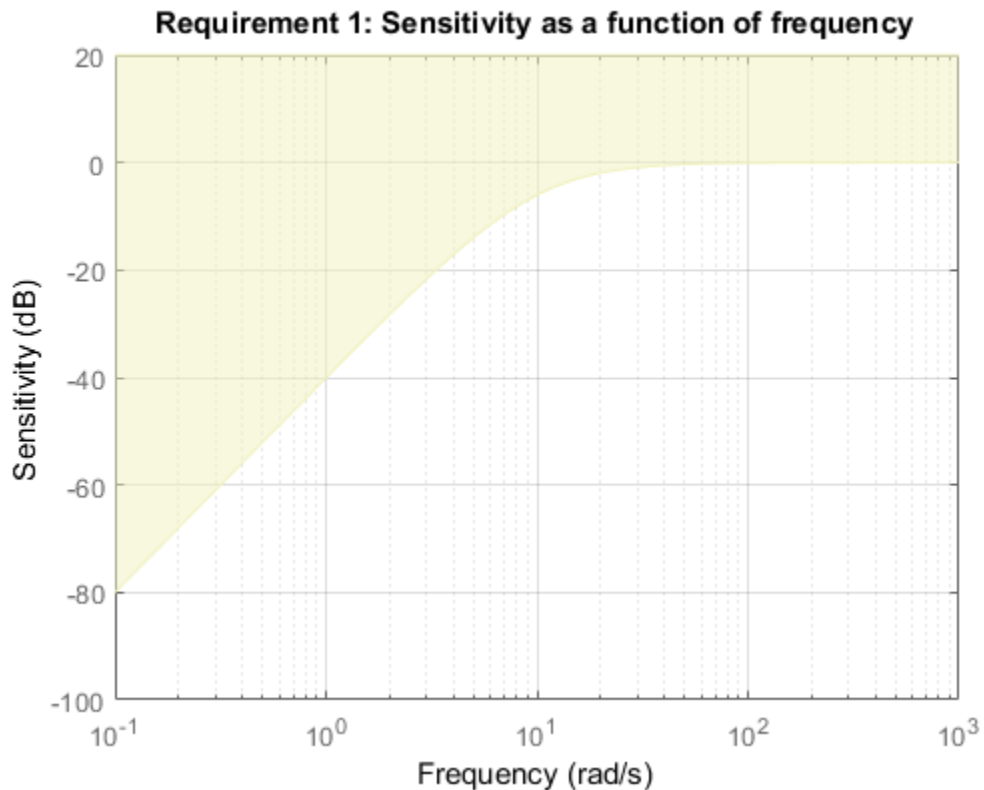


Specify a maximum sensitivity of 0.01 (–40 dB) at 1 rad/s, increasing to 1 (0 dB) past 10 rad/s. Use an `frd` model to sketch this target sensitivity.

```
maxsens = frd([0.01 1 1],[1 10 100]);
Req = TuningGoal.Sensitivity('X',maxsens);
```

The software converts `maxsens` into a smooth function of frequency that approximates the piecewise-specified requirement. Display the requirement using `viewSpec`.

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

You can use `Req` as an input to `looptune` or `systune` when tuning the control system.

## Requirement with Limited Frequency Range and Model Application

Create a requirement that specifies a maximum sensitivity of 0.1 (10%) at frequencies below 5 rad/s. Configure the requirement to apply only to the second and third plant models.

```
Req = TuningGoal.Sensitivity('u',0.1);  
Req.Focus = [0 5];  
Req.Models = [2 3];
```

You can use `Req` as an input to `looptune` or `systune` when tuning a control system that has an analysis point called 'u'. Setting the `Focus` property limits the application of the requirement to frequencies between 0 and 5 rad/s. Setting the `Models` property restricts application of the requirement to the second and third models in an array, when you use the requirement to tune an array of control system models.

### See Also

`looptune` (for `slTuner`) | `TuningGoal.Gain` | `TuningGoal.Rejection` | `TuningGoal.MaxLoopGain` | `looptune` | `systune` | `systune` (for `slTuner`) | `viewSpec` | `evalSpec` | `TuningGoal.LoopShape` | `TuningGoal.MinLoopGain` | `slTuner`

### How To

- “Frequency-Domain Specifications”

# TuningGoal.StepRejection class

**Package:** TuningGoal

Step disturbance rejection requirement for control system tuning

## Description

Use the `TuningGoal.StepRejection` object to specify how a step disturbance injected at a specified location in your control system affects the signal at a specified output location. Use this requirement with control system tuning commands such as `sys tune` or `looptune`.

You can specify the desired response in time-domain terms of peak value, settling time, and damping ratio. Alternatively, you can specify the response as a stable reference model having DC-gain. In that case, the tuning goal is to reject the disturbance as well as or better than the reference model.

To specify disturbance rejection in terms of a frequency-domain attenuation profile, use `TuningGoal.Rejection`.

## Construction

`Req = TuningGoal.StepRejection(inputname,outputname,refsys)` creates a tuning requirement that constrains how a step disturbance injected at a location `inputname` affects the response at `outputname`. The requirement is that the disturbance be rejected as well as or better than the reference system. `inputname` and `outputname` can describe a SISO or MIMO response of your control system. For MIMO responses, the number of inputs must equal the number of outputs.

`Req = TuningGoal.StepRejection(inputname,outputname,peak,tSettle)` specifies an oscillation-free response in terms of a peak value and a settling time.

`Req = TuningGoal.StepRejection(inputname,outputname,peak,tSettle,zeta)` allows for damped oscillations with a damping ratio of at least `zeta`.

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **inputname** can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **u1** and **u2**. Use **'u1'** to designate that point as an input signal when creating tuning goals. Use **{'u1', 'u2'}** to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **inputname** can include:
  - Any input of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **inputname** can be any input name in **T.InputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **inputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **inputname** is an **AnalysisPoint** location of a generalized model, the input signal for the tuning goal is the implied input associated with the **AnalysisPoint** block:





For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **y1** and **y2**. Use **'y1'** to designate that point as an output signal when creating tuning goals. Use **{'y1','y2'}** to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **outputname** can include:
  - Any output of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **outputname** can be any output name in **T.OutputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **outputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **outputname** is an **AnalysisPoint** location of a generalized model, the output signal for the tuning goal is the implied output associated with the **AnalysisPoint** block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **refsys**

Reference system for target step rejection, specified as a SISO dynamic system model, such as a `tf`, `zpk`, or `ss` model. `refsys` must be stable and proper, and must have zero DC gain. This restriction ensures perfect rejection of the steady-state disturbance.

`refsys` can be continuous or discrete. If `refsys` is discrete, it can include time delays which are treated as poles at  $z = 0$ .

For best results, `refsys` and the open-loop response from the disturbance to the output should have similar gains at the frequency where the reference model gain peaks. You can check the peak gain and peak frequency using `getPeakGain`. For example:

```
[gmax,fmax] = getPeakGain(refsys);
```

Use `getIOTransfer` to extract the open-loop response from the disturbance to the output.

### **peak**

Peak absolute value of target response to disturbance, specified as a scalar value.

### **tSettle**

Target settling time of the response to disturbance, specified as a positive scalar value, in the time units of the control system you are tuning.

### **zeta**

Minimum damping ratio of oscillations in the response to disturbance, specified as a value between 0 and 1.

**Default:** 1

## Properties

### ReferenceModel

Reference system for target response to step disturbance, specified as a SISO (zpk) model. The step response of this model specifies how the output signals specified by `outputname` should respond to the step disturbance at `inputname`.

If you use the `refsys` input argument to create the tuning requirement, then the value of `ReferenceModel` is `zpk(refsys)`.

If you use the `peak`, `tSample`, and `zeta` input arguments, then `ReferenceModel` is a `zpk` representation of the first-order or second-order transfer function whose step response has the specified characteristics.

### InputScaling

Input signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued input signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from `Input` to `Output` when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from `Input` to `Output`. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the `OutputScaling` and `InputScaling` values on the diagonal, respectively.

The default value, `[]`, means no scaling.

**Default:** `[]`

### OutputScaling

Output signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued output signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from `Input` to `Output` when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from **Input** to **Output**. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the **OutputScaling** and **InputScaling** values on the diagonal, respectively.

The default value, [ ] , means no scaling.

**Default:** [ ]

### **Input**

Names of disturbance input locations, specified as a cell array of character vectors. This property is initially populated by the **inputname** argument when you create the tuning requirement.

### **Output**

Names of locations at which response to step disturbance is measured, specified as a cell array of character vectors. This property is initially populated by the **outputname** argument when you create the tuning requirement.

### **Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the **Models** property when tuning an array of control system models with **systemtune**, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, **Req**, to the second, third, and fourth models in a model array passed to **systemtune**. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When **Models** = NaN, the tuning goal applies to all models.

**Default:** NaN

### **Openings**

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then **Openings** can include any linear analysis point marked in the model, or any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points and loop openings to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **Openings** can include any **AnalysisPoint** location in the control system model. Use **getPoints** to get the list of analysis points available in the **genss** model.

For example, if **Openings** = { 'u1' , 'u2' }, then the tuning goal is evaluated with loops open at analysis points **u1** and **u2**.

**Default:** {}

#### **Name**

Name of the tuning goal, specified as a character vector.

For example, if **Req** is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** []

## **Algorithms**

When you tune a control system using a **TuningGoal** object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

The **TuningGoal.StepRejection** requirement aims to keep the gain from disturbance to output below the gain of the reference model. The scalar value of the requirement  $f(x)$  is given by:

$$f(x) = \left\| \frac{T(s, x)}{T_{ref}(s)} \right\|_{\infty}.$$

$T(s,x)$  is the closed-loop transfer function from **Input** to **Output**.  $T_{ref}(s)$  is the reference model.  $\|\cdot\|_{\infty}$  denotes the  $H_{\infty}$  norm (see `norm`).

## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from **Input** to **Output**, evaluated with loops opened at the points identified in **Openings**. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systuneOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systuneOptions` to change these defaults.

## Examples

### Specify First-Order or Second-Order Step Disturbance Response Characteristics

Create a requirement that specifies the step disturbance response in terms of peak time-domain response, settling time, and damping of oscillations.

Suppose you want the response at 'y' to a disturbance injected at 'd' to never exceed an absolute value of 0.25, and to settle within 5 seconds. Create a `TuningGoal.StepRejection` requirement that captures these specifications and also specifies non-oscillatory response.

```
Req1 = TuningGoal.StepRejection('d','y',0.25,5);
```

Omitting an explicit value for the damping ratio, `zeta`, is equivalent to setting `zeta = 1`. Therefore, `Req` specifies a non-oscillatory response. The software converts the peak value and settling time into a reference transfer function whose step response has the desired time-domain profile. This transfer function is stored in the `ReferenceModel` property of `Req`.

```
Req1.ReferenceModel
```

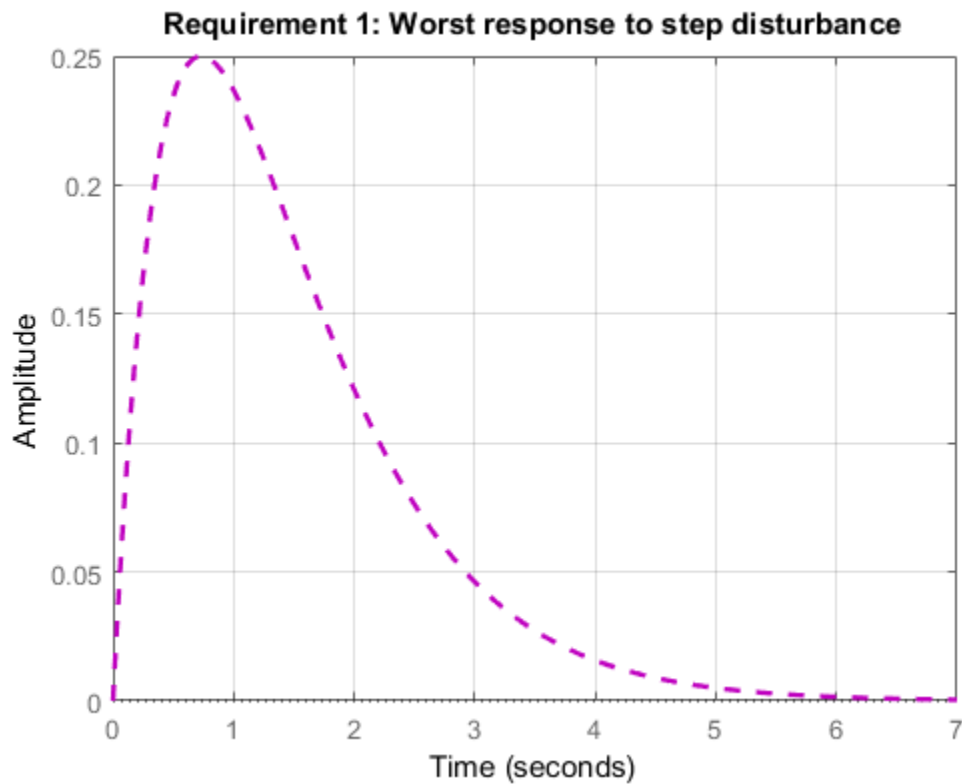
```
ans =  
    0.92883 s  
-----
```

```
(s+1.367)^2
```

```
Continuous-time zero/pole/gain model.
```

Confirm the target response by displaying Req.

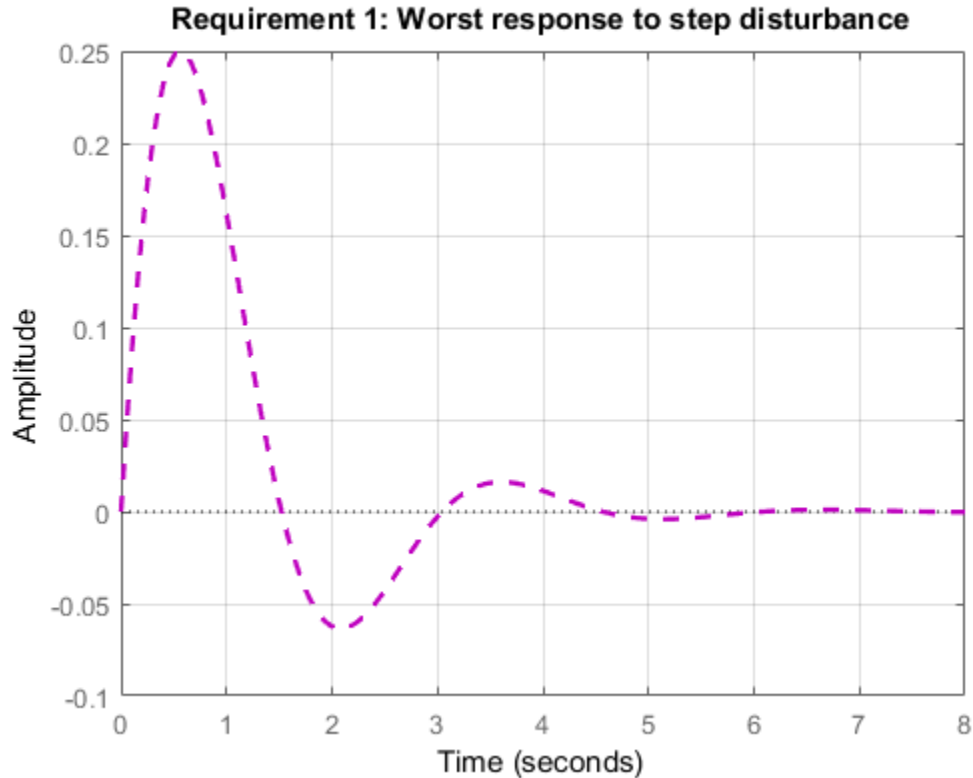
```
figure()  
viewSpec(Req1)
```



Suppose your application can tolerate oscillations provided the damping ratio is less than 0.4. Create a requirement that specifies this disturbance response.

```
Req2 = TuningGoal.StepRejection('d','y',0.25,5,0.4);  
figure()
```

```
viewSpec(Req2)
```



### Step Disturbance Rejection with Custom Reference Model

Create a requirement that specifies the step disturbance response as a transfer function.

Suppose you want the response to a disturbance injected at an analysis point `d` in your control system and measured at a point `'y'` to be rejected at least as well as the transfer function

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

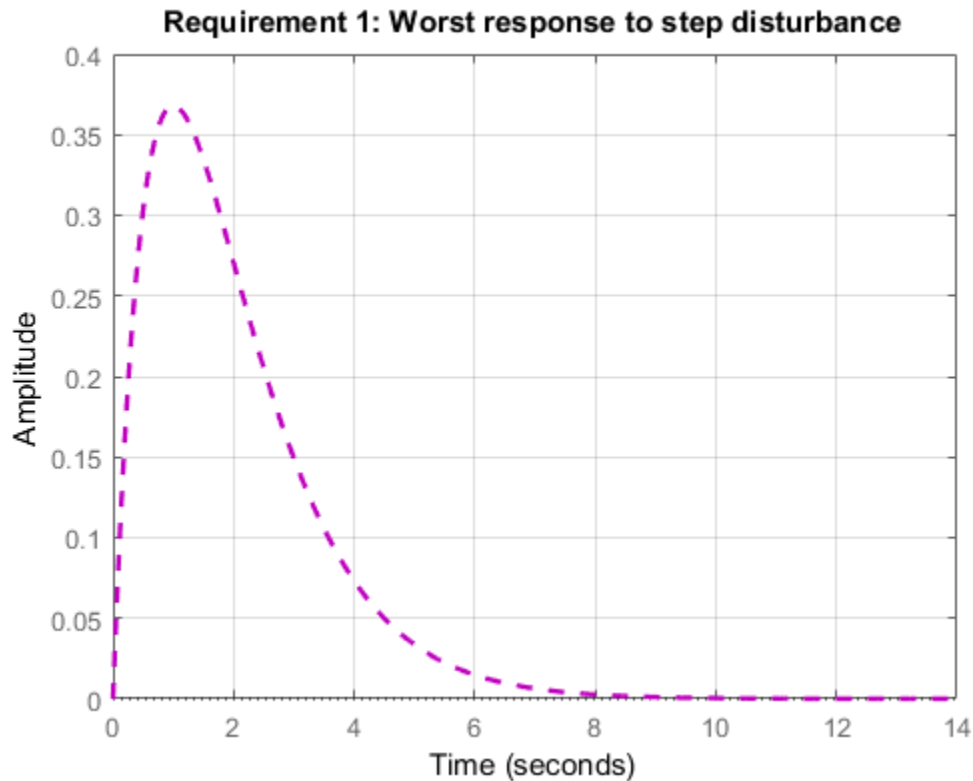
Create a `TuningGoal.StepRejection` requirement.



```
H = tf([1 0],[1 2 1]);  
Req = TuningGoal.StepRejection('d','y',H);
```

Display the requirement.

```
viewSpec(Req)
```



The plot displayed by `viewSpec` shows the step response of the specified transfer function. This response is the target time-domain response to disturbance.

## See Also

[TuningGoal.Gain](#) | [TuningGoal.LoopShape](#) | [evalSpec](#) | [looptune](#) | [looptune](#) (for `sITuner`) | [sITuner](#) | [systune](#) | [systune](#) (for `sITuner`) | [viewSpec](#)

## **More About**

- “Time-Domain Specifications”
- “Tuning Control Systems with SYSTUNE”
- “Tune Control Systems in Simulink”

# TuningGoal.StepTracking class

**Package:** TuningGoal

Step response requirement for control system tuning

## Description

Use the `TuningGoal.StepTracking` object to specify a target step response from specified inputs to specified outputs of a control system. Use this requirement with control system tuning commands such as `systemtune` or `looptune`.

## Construction

`Req = TuningGoal.StepTracking(inputname,outputname,refsys)` creates a tuning requirement that constrains the step response between the specified signal locations to match the step response of a stable reference system, `refsys`. The constraint is satisfied when the relative difference between the tuned and target responses falls within a tolerance specified by the `RelGap` property of the requirement (see “Properties” on page 1-145). `inputname` and `outputname` can describe a SISO or MIMO response of your control system. For MIMO responses, the number of inputs must equal the number of outputs.

`Req = TuningGoal.StepTracking(inputname,outputname,tau)` specifies the desired step response as a first-order response with time constant `tau`:

$$\text{Req.ReferenceModel} = \frac{1 / \tau}{s + 1 / \tau}.$$

`Req = TuningGoal.StepTracking(inputname,outputname,tau,overshoot)` specifies the desired step response as a second-order response with natural period `tau`, natural frequency `1/tau`, and percent overshoot `overshoot`:

$$\text{Req.ReferenceModel} = \frac{(1 / \tau)^2}{s^2 + 2(\zeta / \tau)s + (1 / \tau)^2}.$$

The damping is given by `zeta = cos(atan2(pi, -log(overshoot/100)))`.

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **inputname** can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **u1** and **u2**. Use **'u1'** to designate that point as an input signal when creating tuning goals. Use **{'u1', 'u2'}** to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **inputname** can include:
  - Any input of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **inputname** can be any input name in **T.InputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **inputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **inputname** is an **AnalysisPoint** location of a generalized model, the input signal for the tuning goal is the implied input associated with the **AnalysisPoint** block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **y1** and **y2**. Use **'y1'** to designate that point as an output signal when creating tuning goals. Use **{'y1','y2'}** to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **outputname** can include:
  - Any output of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **outputname** can be any output name in **T.OutputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **outputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **outputname** is an **AnalysisPoint** location of a generalized model, the output signal for the tuning goal is the implied output associated with the **AnalysisPoint** block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **refsys**

Reference system for target step response, specified as a dynamic system model, such as a `tf`, `zpk`, or `ss` model. `refsys` must be stable and must have DC gain of 1 (zero steady-state error).

`refsys` can be continuous or discrete. If `refsys` is discrete, it can include time delays which are treated as poles at  $z = 0$ .

`refsys` can be MIMO, provided that it is square and that its DC singular value (`sigma`) is 1. If `refsys` is a MIMO model, then its number of inputs and outputs must match the dimensions of `inputname` and `outputname`.

For best results, `refsys` should also include intrinsic system characteristics such as non-minimum-phase zeros (undershoot).

### **tau**

Time constant or natural period of target step response, specified as a positive scalar.

If you use the syntax `Req = TuningGoal.StepTracking(inputname,outputname,tau)` to specify a first-order target response, then `tau` is the time constant of the response decay. In that case, the target is the step response of the system given by:

$$\text{Req.ReferenceModel} = \frac{1 / \text{tau}}{s + 1 / \text{tau}}.$$

If you use the syntax `Req = TuningGoal.StepTracking(inputname,outputname,tau,overshoot)` to specify a second-order target response, then `tau` is the inverse of the natural frequency of the response. In that case, the target is the step response of the system given by:

$$\text{Req.ReferenceModel} = \frac{(1/\tau)^2}{s^2 + 2(\zeta/\tau)s + (1/\tau)^2}.$$

The damping of the system is given by  $\zeta = \cos(\text{atan2}(\pi, -\log(\text{overshoot}/100)))$ .

### overshoot

Percent overshoot of target step response, specified as a scalar value in the range (0,100).

## Properties

### ReferenceModel

Reference system for target step response, specified as a SISO or MIMO state-space (ss) model. When you use the requirement to tune a control system, the step response from `inputname` to `outputname` is tuned to match this target response to within the tolerance specified by the `RelGap` property.

If you use the `refsys` input argument to create the tuning requirement, then the value of `ReferenceModel` is `ss(refsys)`.

If you use the `tau` or `tau` and `overshoot` input arguments, then `ReferenceModel` is a state-space representation of the corresponding first-order or second-order transfer function.

`ReferenceModel` must be stable and have unit DC gain (zero steady-state error). For best results, `ReferenceModel` should also include intrinsic system characteristics such as non-minimum-phase zeros (undershoot).

### RelGap

Maximum relative matching error, specified as a positive scalar value. This property specifies the matching tolerance as the maximum relative gap between the target and actual step responses. The relative gap is defined as:

$$\text{gap} = \frac{\|y(t) - y_{ref}(t)\|_2}{\|1 - y_{ref}(t)\|_2}.$$

$y(t) - y_{ref}(t)$  is the response mismatch, and  $1 - y_{ref}(t)$  is the step-tracking error of the target model.  $\|\cdot\|_2$  denotes the signal energy (2-norm).

Increase the value of `RelGap` to loosen the matching tolerance.

**Default:** 0.1

### **InputScaling**

Reference signal scaling, specified as a vector of positive real values.

For a MIMO tuning requirement, when the choice of units results in a mix of small and large signals in different channels of the response, use this property to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

For example, suppose that `Req` is a requirement that signals `{ 'y1', 'y2' }` track reference signals `{ 'r1', 'r2' }`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set the `InputScaling` property as follows.

```
Req.InputScaling = [100,1];
```

This tells the software to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, `[]`, means no scaling.

**Default:** `[]`

### **Input**

Input signal names, specified as a cell array of character vectors that identify the inputs of the transfer function that the tuning requirement constrains. The initial value of the `Input` property is set by the `inputname` input argument when you construct the requirement object.



## Output

Output signal names, specified as a cell array of character vectors that identify the outputs of the transfer function that the tuning requirement constrains. The initial value of the `Output` property is set by the `outputname` input argument when you construct the requirement object.

## Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

## Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `slTuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `slTuner` interface. Use `getPoints` to get the list of analysis points available in an `slTuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1' , 'u2' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ . Here,  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.StepTracking` requirement,  $f(x)$  is given by:

$$f(x) = \frac{\left\| T(s,x) - \frac{1}{s} \text{ReferenceModel} \right\|_2}{\text{RelGap} \left\| \frac{1}{s} (\text{ReferenceModel} - I) \right\|_2}.$$

$T(s,x)$  is the closed-loop transfer function from `Input` to `Output` with parameter values  $x$ .  $\|\cdot\|_2$  denotes the  $H_2$  norm (see `norm`).

## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from `Input` to `Output`, evaluated with loops opened at the points

identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systuneOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systuneOptions` to change these defaults.

## Examples

### Step Response Requirement with Specified Tolerance

Create a requirement for the step response from a signal named 'r' to a signal named 'y'. Constrain the step response to match the transfer function  $H = 10/(s+10)$ , but allow 20% relative variation between the target the tuned responses.

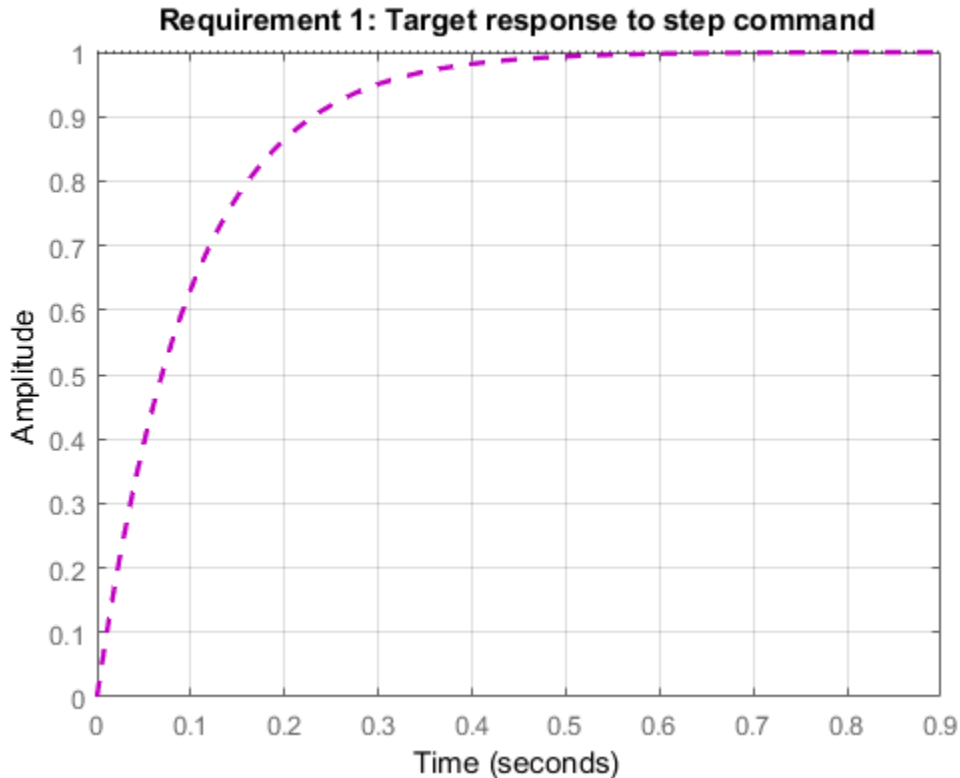
```
H = tf(10,[1 10]);  
Req = TuningGoal.StepResp('r','y',H);
```

By default, this requirement allows a relative gap of 0.1 between the target and tuned responses. To change the relative gap to 20%, set the `RelGap` property of the requirement.

```
Req.RelGap = 0.2;
```

Examine the requirement.

```
viewSpec(Req);
```



The dashed line shows the target step response specified by this requirement. You can use this requirement to tune a control system model, `T`, that contains valid input and output locations named `'r'` and `'y'`. If you do so, the command `viewSpec(Req,T)` plots the achieved step response from `'r'` to `'y'` for comparison to the target response.

## First-Order Step Response With Known Time Constant

Create a requirement that specifies a first-order step response with time constant of 5 seconds. Create the requirement for the step response from a signal named `'r'` to a signal named `'y'`.

```
Req = TuningGoal.StepResp('r','y',5);
```

When you use this requirement to tune a control system model,  $T$ , the time constant 5 is taken to be expressed in the prevailing units of the control system. For example, if  $T$  is a `genss` model and the property `T.TimeUnit` is `'seconds'`, then this requirement specifies a target time constant of 5 seconds for the response from the input `'r'` to the output `'y'` of `'T'`.

The specified time constant is converted into a reference state-space model stored in the `ReferenceModel` property of the requirement.

```
refsys = tf(Req.ReferenceModel)
```

```
refsys =
```

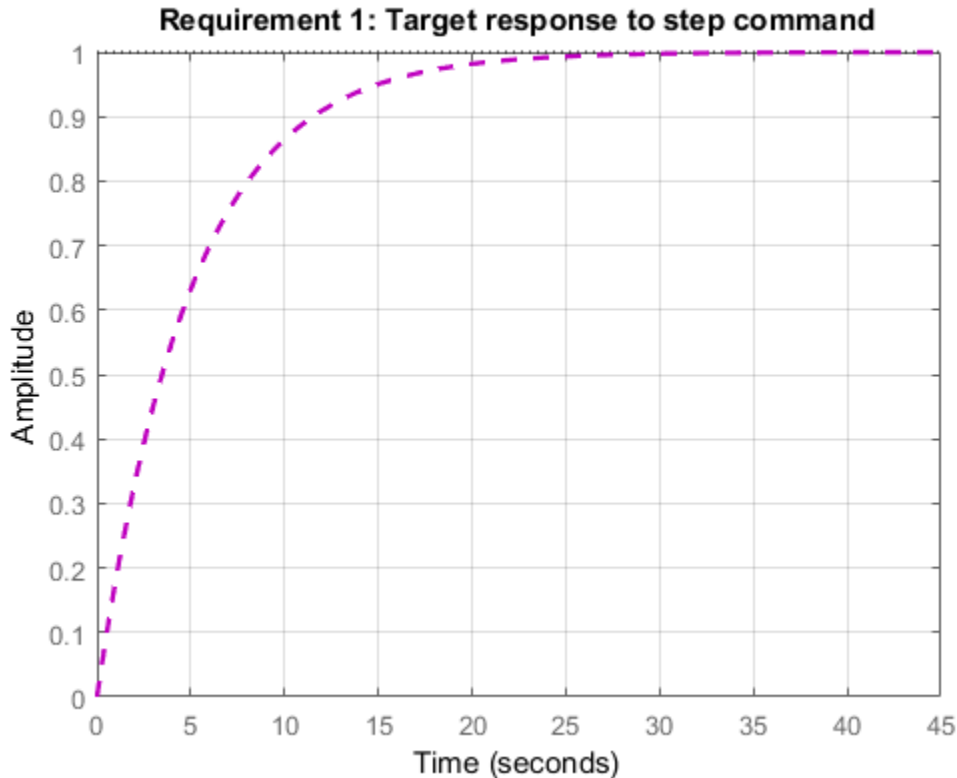
```
    0.2  
-----  
s + 0.2
```

Continuous-time transfer function.

As expected, `refsys` is a first-order model.

Examine the requirement. The `viewSpec` command displays the target response, which is the step response of the reference model.

```
viewSpec(Req);
```



The dashed line shows the target step response specified by this requirement, a first-order response with a time constant of five seconds.

## Second-Order Step Response With Known Natural Period and Overshoot

Create a requirement that specifies a second-order step response with a natural period of 5 seconds, and a 10% overshoot. Create the requirement for the step response from a signal named 'r' to a signal named 'y'.

```
Req = TuningGoal.StepResp('r', 'y', 5, 10);
```

When you use this requirement to tune a control system model, T, the natural period 5 is taken to be expressed in the prevailing units of the control system. For example, if T

is a `genss` model and the property `T.TimeUnit` is `'seconds'`, then this requirement specifies a target natural period of 5 seconds for the response from the input `'r'` to the output `'y'` of `'T'`.

The specified parameters of the response is converted into a reference state-space model stored in the `ReferenceModel` property of the requirement.

```
refsys = tf(Req.ReferenceModel)
```

```
refsys =
```

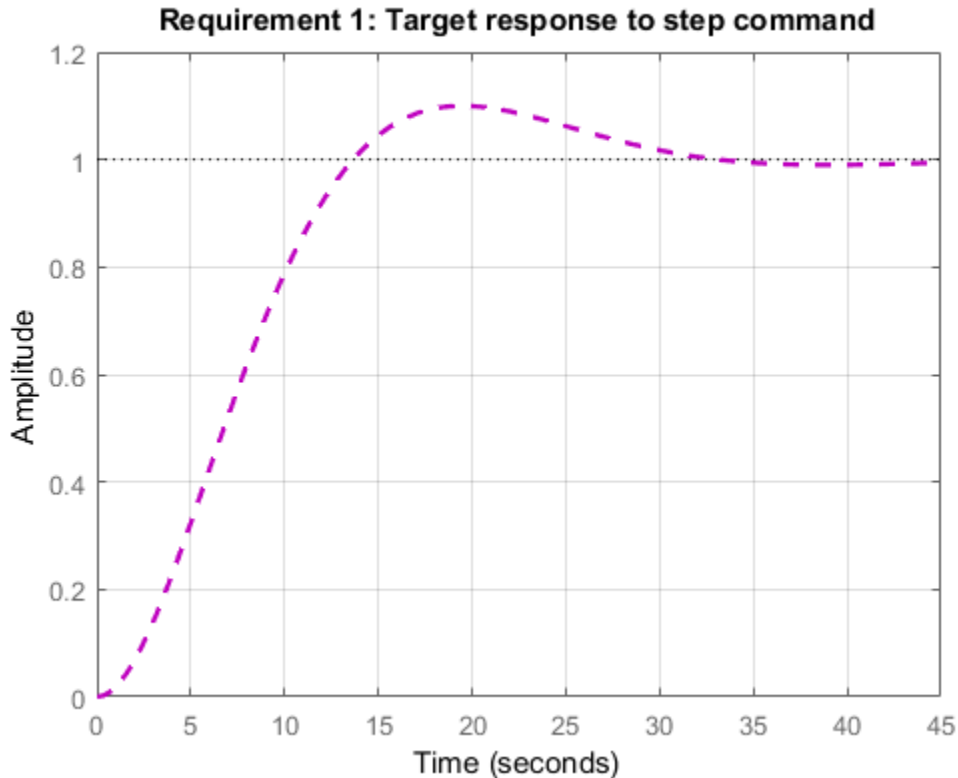
$$\frac{0.04}{s^2 + 0.2365 s + 0.04}$$

Continuous-time transfer function.

As expected, `refsys` is a second-order model.

Examine the requirement. The `viewSpec` command displays the target response, which is the step response of the reference model.

```
viewSpec(Req);
```



The dashed line shows the target step response specified by this requirement, a second-order response with 10% overshoot and a natural period of five seconds.

## Requirement with Limited Model Application and Additional Loop Openings

Create a requirement that specifies a first-order step response with time constant of 5 seconds. Set the `Models` and `Openings` properties to further configure the requirement's applicability.

```
Req = TuningGoal.StepTracking('r','y',5);  
Req.Models = [2 3];
```



```
Req.Openings = 'OuterLoop'
```

When tuning a control system that has an input 'r', an output 'y', and an analysis-point location 'OuterLoop', you can use `Req` as an input to `looptune` or `systune`. Setting the `Openings` property specifies that the step response from 'r' to 'y' is measured with the loop opened at 'OuterLoop'. When tuning an array of control system models, setting the `Models` property restricts how the requirement is applied. In this example, the requirement applies only to the second and third models in an array.

## See Also

`looptune` (for `sITuner`) | `TuningGoal.Tracking` | `looptune` | `systune` | `systune` (for `sITuner`) | `viewSpec` | `evalSpec` | `TuningGoal.Overshoot`

## How To

- “Time-Domain Specifications”
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”

## TuningGoal.Tracking class

**Package:** TuningGoal

Tracking requirement for control system tuning

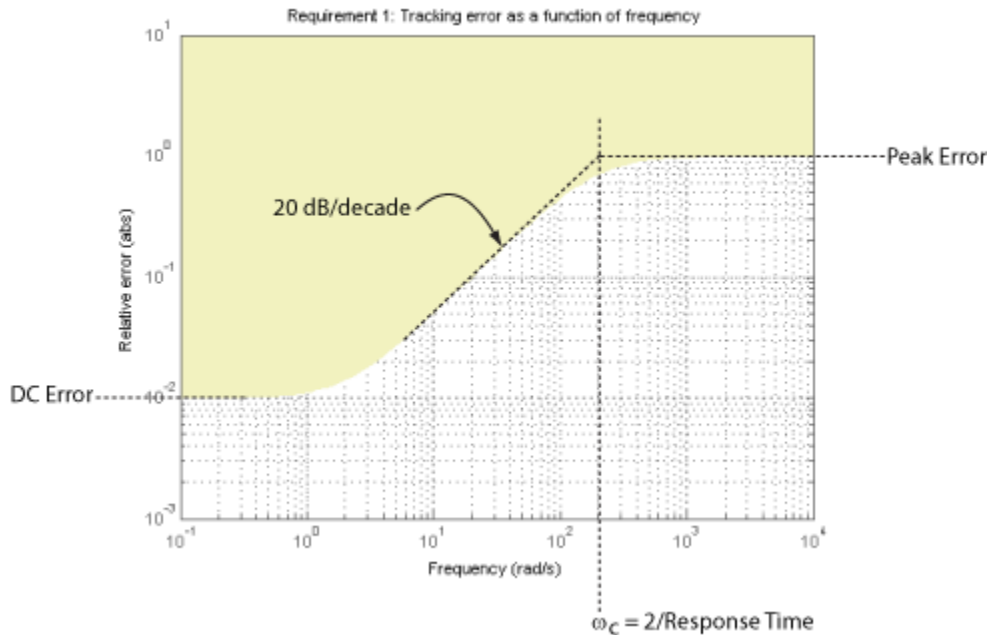
### Description

Use the `TuningGoal.Tracking` object to specify a frequency-domain tracking requirement between specified inputs and outputs. This requirement specifies the maximum relative error (gain from reference input to tracking error) as a function of frequency. Use this requirement for control system tuning with tuning commands such as `systemtune` or `looptune`.

You can specify the maximum error profile directly by providing a transfer function. Alternatively, you can specify a target DC error, peak error, and response time. These parameters are converted to the following transfer function that describes the maximum frequency-domain tracking error:

$$\text{MaxError} = \frac{(\text{PeakError})s + \omega_c (\text{DCError})}{s + \omega_c}.$$

Here,  $\omega_c$  is  $2/(\text{response time})$ . The following plot illustrates these relationships for an example set of values.



## Construction

`Req = TuningGoal.Tracking(inputname,outputname,responsetime,dcerror,peakerror)` creates a tuning requirement `Req` that constrains the tracking performance from `inputname` to `outputname` in the frequency domain. This tuning requirement specifies a maximum error profile as a function of frequency given by:

$$\text{MaxError} = \frac{(\text{PeakError})s + \omega_c (\text{DCError})}{s + \omega_c}.$$

The tracking bandwidth  $\omega_c = 2/\text{responsetime}$ . The maximum relative steady-state error is given by `dcerror`, and `peakerror` gives the peak relative error across all frequencies.

You can specify a MIMO tracking requirement by specifying signal names or a cell array of multiple signal names for `inputname` or `outputname`. For MIMO tracking

requirements, use the `InputScaling` property to help limit cross-coupling. See “Properties” on page 1-161.

`Req = TuningGoal.Tracking(inputname,outputname,maxerror)` specifies the maximum relative error as a function of frequency. You can specify the target error profile (maximum gain from reference signal to tracking error signal) as a smooth transfer function. Alternatively, you can sketch a piecewise error profile using an `frd` model.

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1','u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `outputname` can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `y1` and `y2`. Use `'y1'` to designate that point as an output signal when creating tuning goals. Use `{'y1', 'y2'}` to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `outputname` can include:
  - Any output of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `outputname` can be any output name in `T.OutputName`. Also, if `T` contains an `AnalysisPoint` block

with a location named `AP_u`, then `outputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `outputname` is an `AnalysisPoint` location of a generalized model, the output signal for the tuning goal is the implied output associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **responsetime**

Target response time, specified as a positive scalar value. The tracking bandwidth is given by  $\omega_c = 2/\text{responsetime}$ . Express the target response time in the time units of the models to be tuned. For example, when tuning a model `T`, if `T.TimeUnit` is `'minutes'`, then express the target response time in minutes.

### **dcerror**

Maximum steady-state fractional tracking error, specified as a positive scalar value. For example, `dcerror = 0.01` sets a maximum steady-state error of 1%.

If `inputname` or `outputname` are vector-valued, `dcerror` applies to all I/O pairs from `inputname` to `outputname`.

**Default:** 0.001

### **peakerror**

Maximum fractional tracking error across all frequencies, specified as a positive scalar value greater than 1.

**Default:** 1

### **maxerror**

Target tracking error profile as a function of frequency, specified as a SISO numeric LTI model.

`maxerror` is the maximum gain from reference signal to tracking error signal. You can specify `maxerror` as a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise error profile using a `frd` model. When you do so, the software automatically maps the error profile to a `zpk` model. The magnitude of the `zpk` model approximates the desired error profile. Use `show(Req)` to plot the magnitude of the `zpk` model.

`maxerror` must be a SISO LTI model. If `inputname` or `outputname` are cell arrays, `maxerror` applies to all I/O pairs from `inputname` to `outputname`.

## Properties

### MaxError

Maximum error as a function of frequency, expressed as a SISO `zpk` model. This property stores the maximum tracking error as a function of frequency (maximum gain from reference signal to tracking error signal).

If you use the syntax `Req = TuningGoal.Tracking(inputname,outputname,maxerror)`, then the `MaxError` property is the `zpk` equivalent or approximation of the LTI model you supplied as the `maxerror` input argument.

If you use the syntax `Req = TuningGoal.Tracking(inputname,outputname,resptime,dcerror,peakerror)`, then the `MaxError` is a `zpk` transfer function given by:

$$\text{MaxError} = \frac{(\text{PeakError})s + \omega_c (\text{DCError})}{s + \omega_c}$$

`MaxError` is a SISO LTI model. If `inputname` or `outputname` are cell arrays, `MaxError` applies to all I/O pairs from `inputname` to `outputname`.

Use `show(Req)` to plot the magnitude of `MaxError`.

### Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the **Focus** property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (rad/TimeUnit). For example, suppose **Req** is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** [0, Inf] for continuous time; [0, pi/Ts] for discrete time, where Ts is the model sample time.

### **InputScaling**

Reference signal scaling, specified as a vector of positive real values.

For a MIMO tuning requirement, when the choice of units results in a mix of small and large signals in different channels of the response, use this property to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

For example, suppose that **Req** is a requirement that signals { 'y1', 'y2' } track reference signals { 'r1', 'r2' }. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If r1 and r2 have comparable amplitudes, then it is sufficient to keep the gains from r1 to y2 and r2 and y1 below 0.1. However, if r1 is 100 times larger than r2, the gain from r1 to y2 must be less than 0.001 to ensure that r1 changes y2 by less than 10% of the r2 target. To ensure this result, set the **InputScaling** property as follows.

```
Req.InputScaling = [100,1];
```

This tells the software to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, [ ] , means no scaling.

**Default:** [ ]

### **Input**

Reference signal names, specified as a character vector or cell array of character vectors specifying the names of the signals to be tracked, populated by the **inputname** argument.



## Output

Output signal names, specified as a character vector or cell array of character vectors specifying the names of the signals that must track the reference signals, populated by the `outputname` argument.

## Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systemtune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systemtune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

## Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** {}

**Name**

Name of the tuning goal, specified as a character vector.

For example, if Req is a tuning goal:

```
Req.Name = 'LoopReq' ;
```

**Default:** []

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Tracking` requirement,  $f(x)$  is given by:

$$f(x) = \left\| \frac{1}{\text{MaxError}} (T(s, x) - I) \right\|_{\infty}.$$

$T(s, x)$  is the closed-loop transfer function from `Input` to `Output`.  $\|\cdot\|_{\infty}$  denotes the  $H_{\infty}$  norm (see `norm`).

## Tips

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from `Input` to `Output`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Examples

### Tracking Requirement With Response Time and Maximum Steady-State Tracking Error

Create a tracking requirement specifying that a signal 'theta' track a signal 'theta\_ref'. The required response time is 2, in the time units of the control system you are tuning. The maximum steady-state error is 0.1%.

```
Req = TuningGoal.Tracking('theta_ref','theta',2,0.001);
```

Since `peakerror` is unspecified, this requirement uses the default value, 1.

### Tracking Requirement With Maximum Tracking Error as a Function of Frequency

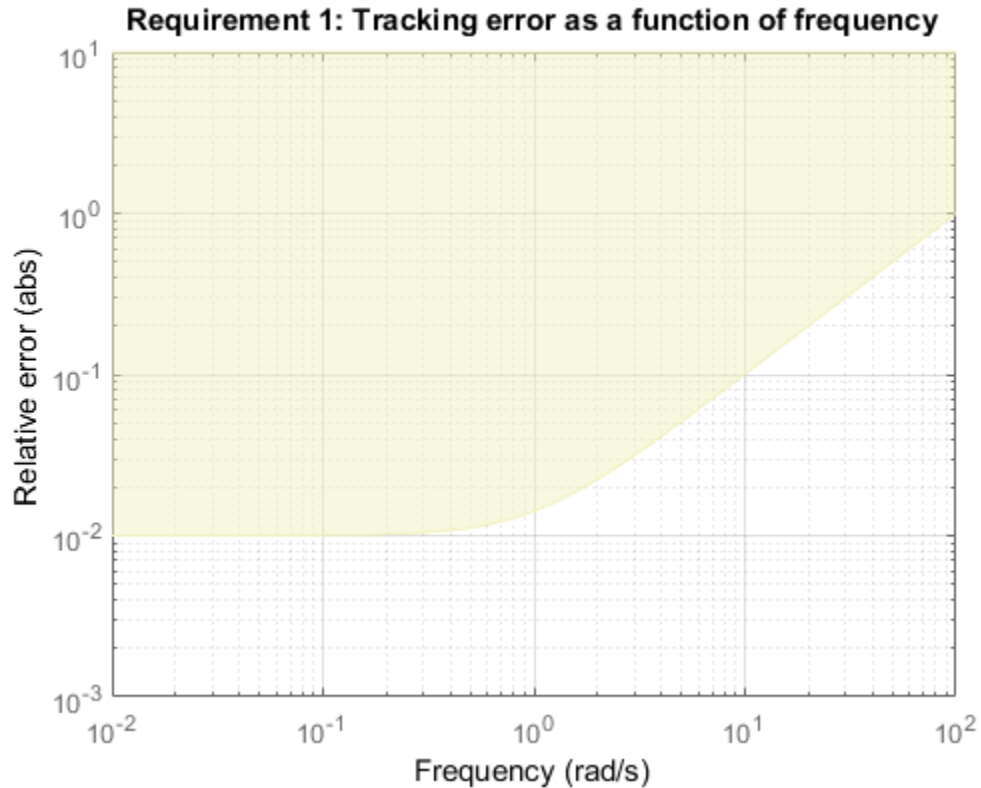
Create a tracking requirement specifying that a signal 'theta' track a signal 'theta\_ref'. The maximum relative error is 0.01 (1%) in the frequency range [0,1]. The relative error increases to 1 (100%) at the frequency 100.

Use an `frd` model to specify the error profile as a function of frequency.

```
err = frd([0.01 0.01 1],[0 1 100]);
Req = TuningGoal.Tracking('theta_ref','theta',err);
```

The software converts `err` into a smooth function of frequency that approximates the piecewise specified requirement. Display the error requirement using `viewSpec`.

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

### See Also

[systune](#) (for [slTuner](#)) | [TuningGoal.Gain](#) | [looptune](#) | [systune](#) | [looptune](#) (for [slTuner](#)) | [viewSpec](#) | [evalSpec](#) | [TuningGoal.LoopShape](#) | [slTuner](#)

### How To

- “Time-Domain Specifications”
- “Tuning Control Systems with SYSTUNE”
- “Tune Control Systems in Simulink”

- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”
- “Decoupling Controller for a Distillation Column”
- “Digital Control of Power Stage Voltage”
- “Tuning of a Two-Loop Autopilot”

# TuningGoal.Transient class

**Package:** TuningGoal

Transient matching requirement for control system tuning

## Description

Use the `TuningGoal.Transient` object to constrain the transient response from specified inputs to specified outputs. This requirement specifies that the transient response closely match the response of a reference model. Specify the closeness of the required match using the `RelGap` property of the requirement (see “Properties” on page 1-172). You can constrain the response to an impulse, step, or ramp input signal. You can also constrain the response to an input signal given by the impulse response of an input filter you specify.

## Construction

`Req = TuningGoal.Transient(inputname,outputname,refsys)` requires that the impulse response from `inputname` to `outputname` closely matches the impulse response of the reference model `refsys`. Specify the closeness of the required match using the `RelGap` property of the requirement (see “Properties” on page 1-172). `inputname` and `outputname` can describe a SISO or MIMO response of your control system. For MIMO responses, the number of inputs must equal the number of outputs.

`Req = TuningGoal.Transient(inputname,outputname,refsys,inputtype)` specifies whether the input signal that generates the constrained transient response is and impulse, step, or ramp signal.

`Req = TuningGoal.Transient(inputname,outputname,refsys,inputfilter)` specifies the input signal for generating the transient response that the requirement constrains. Specify the input signal as a SISO transfer function, `inputfilter`, that is the Laplace transform of the desired time-domain input signal. The impulse response of `inputfilter` is the desired input signal.

## Input Arguments

### inputname

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1','u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **y1** and **y2**. Use **'y1'** to designate that point as an output signal when creating tuning goals. Use **{'y1','y2'}** to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **outputname** can include:
  - Any output of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **outputname** can be any output name in **T.OutputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **outputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **outputname** is an **AnalysisPoint** location of a generalized model, the output signal for the tuning goal is the implied output associated with the **AnalysisPoint** block:





For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### refsys

Reference system for target transient response, specified as a dynamic system model, such as a `tf`, `zpk`, or `ss` model. The desired transient response is the response of this model to the input signal specified by `inputtype` or `inputfilter`. The reference model must be stable, and the series connection of the reference model with the input shaping filter must have no feedthrough term.

### inputtype

Type of input signal that generates the constrained transient response, specified as one of the following values:

- `'impulse'` — Constrain the response at `outputname` to a unit impulse applied at `inputname`.
- `'step'` — Constrain the response to a unit step. Using `'step'` is equivalent to using the `TuningGoal.StepTracking` design goal.
- `'ramp'` — Constrain the response to a unit ramp,  $u = t$ .

**Default:** `'impulse'`

### inputfilter

Custom input signal for generating the transient response, specified as a SISO transfer function (`tf` or `zpk`) model that represents the Laplace transform of the desired input signal. `inputfilter` must be continuous, and can have no poles in the open right-half plane.

The frequency response of `inputfilter` gives the signal spectrum of the desired input signal, and the impulse response of `inputfilter` is the time-domain input signal.

For example, to constrain the transient response to a unit-amplitude sine wave of frequency  $w$ , set `inputfilter` to `tf(w, [1,0,w^2])`. This transfer function is the Laplace transform of  $\sin(wt)$ .

The series connection of `refsys` with `inputfilter` must have no feedthrough term.

## Properties

### ReferenceModel

Reference system for target transient response, specified as a SISO or MIMO state-space (SS) model. When you use the requirement to tune a control system, the transient response from `inputname` to `outputname` is tuned to match this target response to within the tolerance specified by the `RelGap` property.

The `refsys` argument to `TuningGoal.Transient` sets the value of `ReferenceModel` to `ss(refsys)`.

### InputShaping

Input signal for generating the transient response, specified as a SISO zpk model that represents the Laplace transform of the time-domain input signal. `InputShaping` must be continuous, and can have no poles in the open right-half plane. The value of this property is populated using the `inputtype` or `inputfilter` arguments used when creating the requirement.

For requirements created using the `inputtype` argument, `InputShaping` takes the following values:

<b>inputtype</b>	<b>InputShaping</b>
'impulse'	1
'step'	1/s
'ramp'	1/s <sup>2</sup>

For requirements created using an `inputfilter` transfer function, `InputShaping` takes the value `zpk(inputfilter)`.

The series connection of `ReferenceModel` with `InputShaping` must have no feedthrough term.

**Default:** 1

### RelGap

Maximum relative matching error, specified as a positive scalar value. This property specifies the matching tolerance as the maximum relative gap between the target and actual transient responses. The relative gap is defined as:

$$\text{gap} = \frac{\|y(t) - y_{ref}(t)\|_2}{\|y_{ref(tr)}(t)\|_2}.$$

$y(t) - y_{ref}(t)$  is the response mismatch, and  $1 - y_{ref(tr)}(t)$  is the transient portion of  $y_{ref}$  (deviation from steady-state value or trajectory).  $\|\cdot\|_2$  denotes the signal energy (2-norm).

The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient

Increase the value of **RelGap** to loosen the matching tolerance.

**Default:** 0.1

### InputScaling

Input signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued input signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from **Input** to **Output** when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from **Input** to **Output**. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the **OutputScaling** and **InputScaling** values on the diagonal, respectively.

The default value, [ ] , means no scaling.

**Default:** [ ]

### OutputScaling

Output signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued output signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from **Input** to **Output** when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from **Input** to **Output**. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the **OutputScaling** and **InputScaling** values on the diagonal, respectively.

The default value, [ ] , means no scaling.

**Default:** [ ]

### **Input**

Input signal names, specified as a cell array of character vectors that indicate the inputs for the transient responses that the tuning goal constrains. The initial value of the **Input** property is populated by the `inputname` argument when you create the tuning goal.

### **Output**

Output signal names, specified as a cell array of character vectors that indicate the outputs where transient responses that the tuning goal constrains are measured. The initial value of the **Output** property is populated by the `outputname` argument when you create the tuning goal.

### **Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the **Models** property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, **Req**, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When **Models** = NaN, the tuning goal applies to all models.

**Default:** NaN

## Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then **Openings** can include any linear analysis point marked in the model, or any linear analysis point in an **slTuner** interface associated with the Simulink model. Use **addPoint** to add analysis points and loop openings to the **slTuner** interface. Use **getPoints** to get the list of analysis points available in an **slTuner** interface to your model.

If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **Openings** can include any **AnalysisPoint** location in the control system model. Use **getPoints** to get the list of analysis points available in the **genss** model.

For example, if **Openings** = { 'u1' , 'u2' }, then the tuning goal is evaluated with loops open at analysis points **u1** and **u2**.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if **Req** is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** []

## Tips

- When you use this requirement to tune a continuous-time control system, **systune** attempts to enforce zero feedthrough ( $D = 0$ ) on the transfer that the requirement constrains. Zero feedthrough is imposed because the  $H_2$  norm, and therefore the value of the tuning goal (see “Algorithms” on page 1-176), is infinite for continuous-time systems with nonzero feedthrough.

`system` enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. `system` returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, use the `Value` and `Free` properties of the block parametrization. For example, consider a tuned state-space block:

```
C = tunableSS('C',1,2,3);
```

To enforce zero feedthrough on this block, set its  $D$  matrix value to zero, and fix the parameter.

```
C.D.Value = 0;  
C.D.Free = false;
```

For more information on fixing parameter values, see the Control Design Block reference pages, such as `tunableSS`.

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from **Input** to **Output**, evaluated with loops opened at the points identified in **Openings**. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ ,

where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Transient` requirement,  $f(x)$  is based upon the relative gap between the tuned response and the target response:

$$\text{gap} = \frac{\|y(t) - y_{ref}(t)\|_2}{\|y_{ref(tr)}(t)\|_2}.$$

$y(t) - y_{ref}(t)$  is the response mismatch, and  $1 - y_{ref(tr)}(t)$  is the transient portion of  $y_{ref}$  (deviation from steady-state value or trajectory).  $\|\cdot\|_2$  denotes the signal energy (2-norm).

The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient

## Examples

### Transient Response Requirement with Specified Input Type and Tolerance

Create a requirement for the transient response from a signal named 'r' to a signal named 'u'. Constrain the impulse response to match the response of transfer function  $refsys = 1/(s + 1)$ , but allow 20% relative variation between the target and tuned responses.

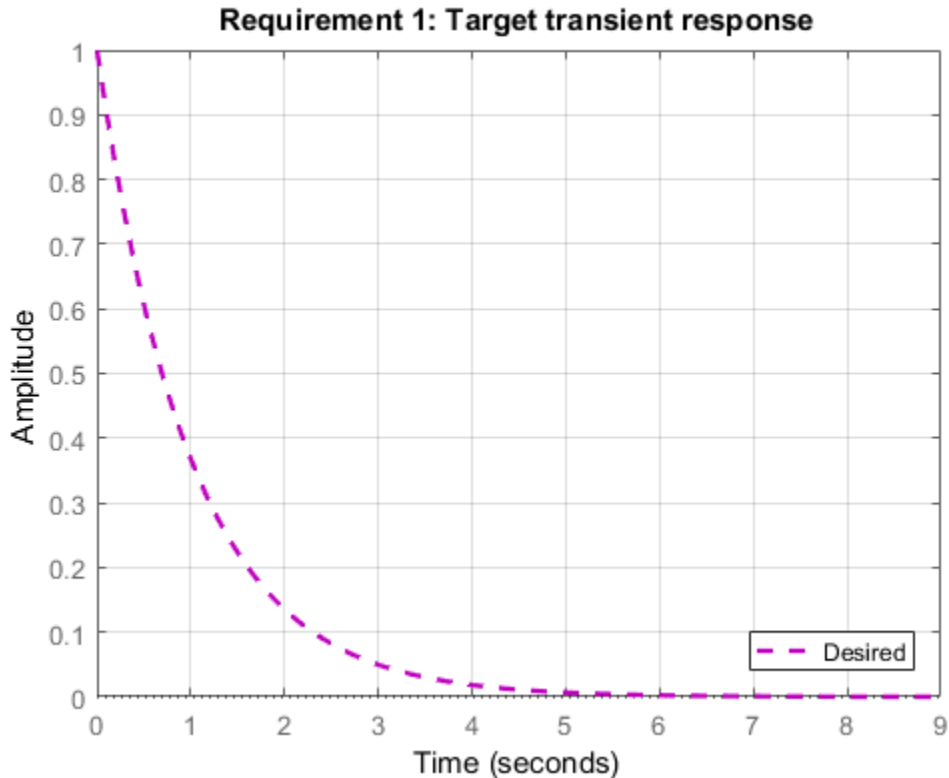
```
refsys = tf(1,[1 1]);
Req1 = TuningGoal.Transient('r','u',refsys);
```

When you do not specify a response type, the requirement constrains the transient response. By default, the requirement allows a relative gap of 0.1 between the target and tuned responses. To change the relative gap to 20%, set the `RelGap` property of the requirement.

```
Req1.RelGap = 0.2;
```

Examine the requirement.

```
viewSpec(Req1)
```



The dashed line shows the target impulse response specified by this requirement. You can use this requirement to tune a control system model, `T`, that contains valid input and output locations named `'r'` and `'u'`. If you do so, the command `viewSpec(Req1,T)` plots the achieved impulse response from `'r'` to `'u'` for comparison to the target response.

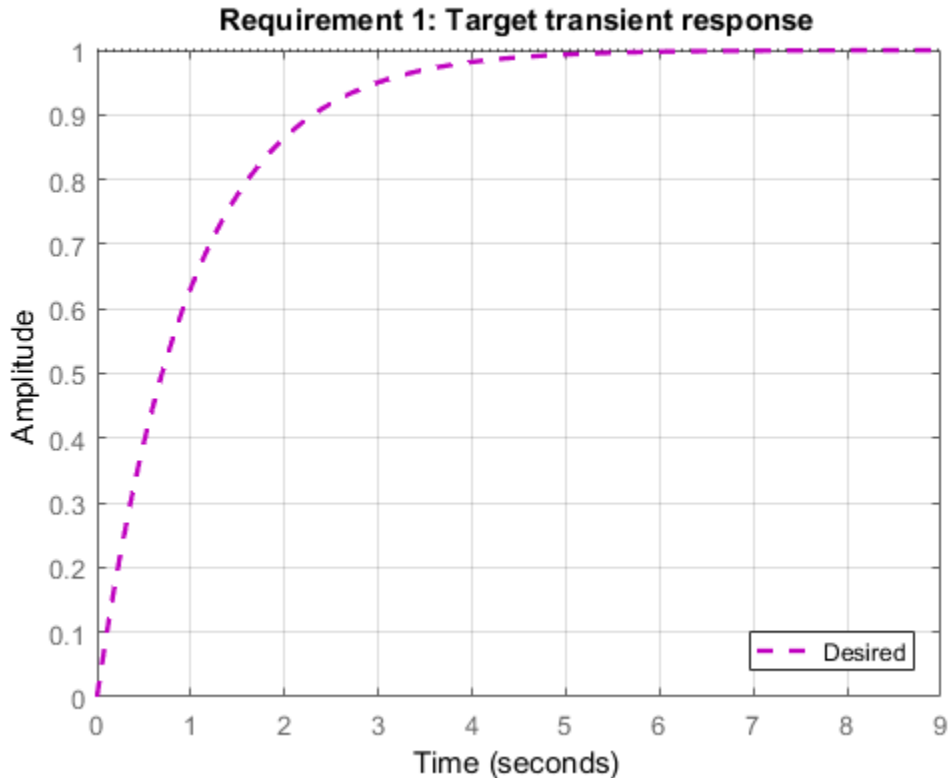
Create a requirement that constrains the response to a step input, instead of the impulse response.

```
Req2 = TuningGoal.Transient('r','u',refsys,'step');
```

Examine this requirement.

```
viewSpec(Req2)
```





Req2 is equivalent to the following step tracking requirement:

```
Req3 = TuningGoal.StepTracking('r','u',refsys);
```

## Constrain Transient Response to Custom Input Signal

Create a requirement for the transient response from 'r' to 'u'. Constrain the response to a sinusoidal input signal, rather than to an input, step, or ramp.

To specify a custom input signal, set the input filter to the Laplace transform of the desired signal. For example, suppose you want to constrain the response to a signal of  $\sin \omega t$ . The Laplace transform of this signal is given by:

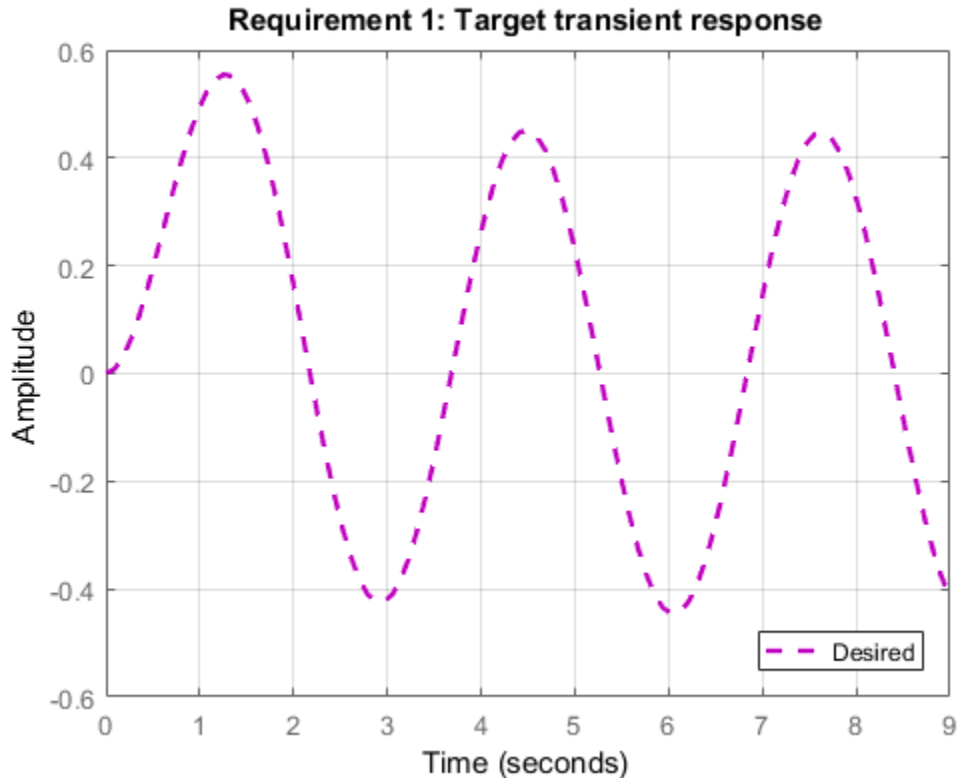
$$\text{inputfilter} = \frac{\omega}{s^2 + \omega^2}.$$

Create a requirement that constrains the response at 'u' to a sinusoidal input of natural frequency 2 rad/s at 'r'. The response should match that of the reference system  $refsys = 1/(s + 1)$ .

```
refsys = tf(1,[1 1]);  
w = 2;  
inputfilter = tf(w,[1 0 w^2]);  
Req = TuningGoal.Transient('u','r',refsys,inputfilter);
```

Examine the requirement to see the shape of the target response.

```
viewSpec(Req)
```



## Requirement with Limited Model Application and Additional Loop Openings

Create a requirement that constrains the impulse response. Set the `Models` and `Openings` properties to further configure the requirement's applicability.

```
refsys = tf(1,[1 1]);
Req = TuningGoal.Transient('r','u',refsys);
Req.Models = [2 3];
Req.Openings = 'OuterLoop'
```

When tuning a control system that has an input (or analysis point) 'r', an output (or analysis point) 'u', and another analysis point at location 'OuterLoop', you

can use `Req` as an input to `looptune` or `systune`. Setting the `Openings` property specifies that the impulse response from 'r' to 'y' is computed with the loop opened at 'OuterLoop'. When tuning an array of control system models, setting the `Models` property restricts how the requirement is applied. In this example, the requirement applies only to the second and third models in an array.

## See Also

`systune` (for `sITuner`) | `TuningGoal.StepTracking` | `TuningGoal.StepRejection` | `sITuner` | `looptune` | `systune` | `looptune` (for `sITuner`) | `viewSpec` | `evalSpec`

## How To

- “Time-Domain Specifications”
- “Tuning Control Systems with SYSTUNE”
- “Tune Control Systems in Simulink”

# TuningGoal.Variance class

**Package:** TuningGoal

Noise amplification constraint for control system tuning

## Description

Use the `TuningGoal.Variance` object to specify a tuning requirement that limits the noise amplification from specified inputs to outputs. The noise amplification is defined as either:

- The square root of the output variance, for a unit-variance white-noise input
- The root-mean-square of the output, for a unit-variance white-noise input
- The  $H_2$  norm of the transfer function from the specified inputs to outputs, which equals the total energy of the impulse response

These definitions are different interpretations of the same quantity. `TuningGoal.Variance` imposes the same limit on these quantities.

You can use the `TuningGoal.Variance` requirement for control system tuning with tuning commands, such as `sys tune` or `looptune`. Specifying this requirement allows you to tune the system response to white-noise inputs. For stochastic inputs with a nonuniform spectrum (colored noise), use `TuningGoal.WeightedVariance` instead.

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-186 of the object.

## Construction

`Req = TuningGoal.Variance(inputname,outputname,maxamp)` creates a tuning requirement. This tuning requirement limits the noise amplification of the transfer function from `inputname` to `outputname` to the scalar value `maxamp`.

When you tune a control system in discrete time, this requirement assumes that the physical plant and noise process are continuous. To ensure that continuous-time and

discrete-time tuning give consistent results, `maxamp` is interpreted as a constraint on the continuous-time  $H_2$  norm. If the plant and noise processes are truly discrete and you want to constrain the discrete-time  $H_2$  norm instead, multiply `maxamp` by  $\sqrt{T_s}$ .  $T_s$  is the sample time of the model you are tuning.

## Input Arguments

### `inputname`

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1', 'u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### outputname

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `y1` and `y2`. Use `'y1'` to designate that point as an output signal when creating tuning goals. Use `{'y1', 'y2'}` to designate a two-channel output.

•

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then **outputname** can include:

- Any output of the `genss` model
- Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then **outputname** can be any output name in `T.OutputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then **outputname** can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `outputname` is an `AnalysisPoint` location of a generalized model, the output signal for the tuning goal is the implied output associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **maxamp**

Maximum noise amplification from `inputname` to `outputname`, specified as a positive scalar value. This value specifies the maximum value of the output variance at the signals specified in `outputname`, for unit-variance white noise signal at `inputname`. This value corresponds to the maximum  $H_2$  norm from `inputname` to `outputname`.

When you tune a control system in discrete time, this requirement assumes that the physical plant and noise process are continuous, and interprets `maxamp` as a bound on the continuous-time  $H_2$  norm. This ensures that continuous-time and discrete-time tuning give consistent results. If the plant and noise processes are truly discrete, and you want to bound the discrete-time  $H_2$  norm instead, specify the value `maxamp`/ $\sqrt{T_s}$ .  $T_s$  is the sample time of the model you are tuning.

## **Properties**

### **MaxAmplification**

Maximum noise amplification, specified as a positive scalar value. This property specifies the maximum value of the output variance at the signals specified in `Output`, for unit-variance white noise signal at `Input`. This value corresponds to the maximum  $H_2$  norm from `Input` to `Output`. The initial value of `MaxAmplification` is set by the `maxamp` input argument when you construct the requirement.



## InputScaling

Input signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued input signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from **Input** to **Output** when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from **Input** to **Output**. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the **OutputScaling** and **InputScaling** values on the diagonal, respectively.

The default value, [ ] , means no scaling.

**Default:** [ ]

## OutputScaling

Output signal scaling, specified as a vector of positive real values.

Use this property to specify the relative amplitude of each entry in vector-valued output signals when the choice of units results in a mix of small and large signals. This information is used to scale the closed-loop transfer function from **Input** to **Output** when the tuning requirement is evaluated.

Suppose  $T(s)$  is the closed-loop transfer function from **Input** to **Output**. The requirement is evaluated for the scaled transfer function  $D_o^{-1}T(s)D_i$ . The diagonal matrices  $D_o$  and  $D_i$  have the **OutputScaling** and **InputScaling** values on the diagonal, respectively.

The default value, [ ] , means no scaling.

**Default:** [ ]

## Input

Input signal names, specified as a cell array of character vectors that identify the inputs of the transfer function that the tuning requirement constrains. The initial value of the **Input** property is set by the `inputname` input argument when you construct the requirement object.

## Output

Output signal names, specified as a cell array of character vectors that identify the outputs of the transfer function that the tuning requirement constrains. The initial value of the `Output` property is set by the `outputname` input argument when you construct the requirement object.

## Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

## Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `slTuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `slTuner` interface. Use `getPoints` to get the list of analysis points available in an `slTuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1' , 'u2' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Tips

- When you use this requirement to tune a continuous-time control system, `systemtune` attempts to enforce zero feedthrough ( $D = 0$ ) on the transfer that the requirement constrains. Zero feedthrough is imposed because the  $H_2$  norm, and therefore the value of the tuning goal (see “Algorithms” on page 1-190), is infinite for continuous-time systems with nonzero feedthrough.

`systemtune` enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. `systemtune` returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software’s approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, use the `Value` and `Free` properties of the block parametrization. For example, consider a tuned state-space block:

```
C = tunableSS('C',1,2,3);
```

To enforce zero feedthrough on this block, set its  $D$  matrix value to zero, and fix the parameter.

```
C.D.Value = 0;  
C.D.Free = false;
```

For more information on fixing parameter values, see the Control Design Block reference pages, such as `tunableSS`.

- This tuning goal imposes an implicit stability constraint on the closed-loop transfer function from `Input` to `Output`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systuneOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systuneOptions` to change these defaults.

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ . The vector  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Variance` requirement,  $f(x)$  is given by:

$$f(x) = \left\| \frac{1}{\text{MaxAmplification}} T(s, x) \right\|_2.$$

$T(s, x)$  is the closed-loop transfer function from `Input` to `Output`.  $\|\cdot\|_2$  denotes the  $H_2$  norm (see `norm`).

For tuning discrete-time control systems,  $f(x)$  is given by:

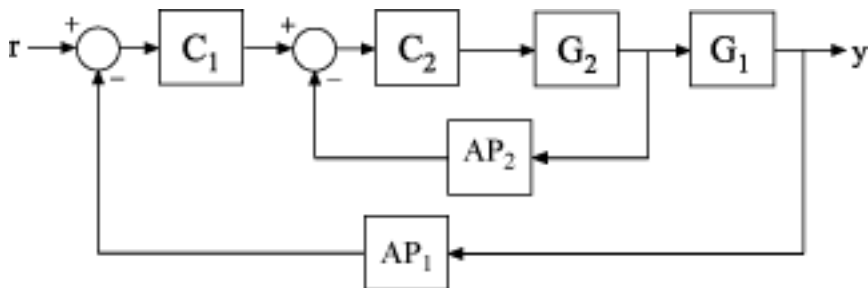
$$f(x) = \left\| \frac{1}{\text{MaxAmplification} \sqrt{T_s}} T(z, x) \right\|_2.$$

$T_s$  is the sample time of the discrete-time transfer function  $T(z,x)$ .

## Examples

### Constrain Noise Amplification Evaluated with a Loop Opening

Create a requirement that constrains the amplification of the variance from the analysis point AP2 to the output  $y$  of the following control system, measured with the outer loop open.



Create a model of the system. To do so, specify and connect the numeric plant models G1 and G2, and the tunable controllers C1 and C2. Also specify and connect the AnalysisPoint blocks AP1 and AP2 that mark points of interest for analysis and tuning.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
AP1 = AnalysisPoint('AP1');
AP2 = AnalysisPoint('AP2');
T = feedback(G1*feedback(G2*C2,AP2)*C1,AP1);
```

Create a tuning requirement that constrains the noise amplification from the implicit input associated with the analysis point, AP2, to the output  $y$ .

```
Req = TuningGoal.Variance('AP2','y',0.1);
```

This constraint limits the amplification to a factor of 0.1.

Specify that the transfer function from AP2 to  $y$  is evaluated with the outer loop open when tuning to this constraint.

```
Req.Openings = {'AP1'};
```

Use `system` to tune the free parameters of `T` to meet the tuning requirement specified by `Req`. You can then validate the tuned control system against the requirement using `viewSpec(Req,T,Info)`.

## See Also

`looptune` (for `sITuner`) | `TuningGoal.WeightedVariance` | `looptune` | `system` | `system` (for `sITuner`) | `sITuner` | `viewSpec` | `evalSpec` | `norm`

## How To

- “Frequency-Domain Specifications”
- “Active Vibration Control in Three-Story Building”
- “Fault-Tolerant Control of a Passenger Jet”

# TuningGoal.WeightedPassivity class

**Package:** TuningGoal

Frequency-weighted passivity constraint

## Description

A system is *passive* if all its I/O trajectories  $(u(t),y(t))$  satisfy:

$$\int_0^T y(t)^\top u(t) dt > 0,$$

for all  $T > 0$ . TuningGoal.WeightedPassivity enforces the passivity of the transfer function:

$$H(s) = W_L(s)T(s)W_R(s),$$

where  $T_s$  is a closed-loop response in the control system being tuned.  $W_L$  and  $W_R$  are weighting functions used to emphasize particular frequency bands. Use TuningGoal.WeightedPassivity with control system tuning commands such as systune.

## Construction

`Req = TuningGoal.WeightedPassivity(inputname,outputname,WL,WR)` creates a tuning goal for enforcing passivity of the transfer function:

$$H(s) = W_L(s)T(s)W_R(s),$$

where  $T_s$  is the closed-loop transfer function from the specified inputs to the specified outputs. The weights WL and WR can be matrices or LTI models.

By default, the tuning goal enforces passivity of the weighted transfer function  $H$ . You can also enforce input and output passivity indices, with a specified excess or shortage of passivity. (See `getPassiveIndex` for more information about passivity indices.) To do so, set the IPX and OPX properties of the tuning goal. See “Weighted Passivity and Input Passivity” on page 1-201.

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **inputname** can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **u1** and **u2**. Use **'u1'** to designate that point as an input signal when creating tuning goals. Use **{'u1', 'u2'}** to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **inputname** can include:
  - Any input of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **inputname** can be any input name in **T.InputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **inputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **inputname** is an **AnalysisPoint** location of a generalized model, the input signal for the tuning goal is the implied input associated with the **AnalysisPoint** block:





For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **outputname** can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **y1** and **y2**. Use **'y1'** to designate that point as an output signal when creating tuning goals. Use **{'y1','y2'}** to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **outputname** can include:
  - Any output of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **outputname** can be any output name in **T.OutputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **outputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If **outputname** is an **AnalysisPoint** location of a generalized model, the output signal for the tuning goal is the implied output associated with the **AnalysisPoint** block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **WL, WR**

Input and output weighting functions, specified as scalars, matrices, or SISO or MIMO numeric LTI models.

The functions WL and WR provide the weights for the tuning requirement. The tuning requirement ensures passivity of the weighted transfer function:

$$H(s) = W_L(s)T(s)W_R(s),$$

where  $T(s)$  is the transfer function from `inputname` to `outputname`. The function WL provides the weighting for the output channels of  $T(s)$ , and WR provides the weighting for the input channels. You can specify:

- Scalar weighting — use a scalar or numeric matrix.
- Frequency-dependent weighting — use a SISO or MIMO numeric LTI model. For example:

```
WL = tf(1,[1 0.01]);  
WR = 10;
```

If WL or WR is a matrix or a MIMO model, then `inputname` and `outputname` must be vector signals. The dimensions of the vector signals must be such that the dimensions of  $T(s)$  are commensurate with the dimensions of WL and WR. For example, if you specify `WR = diag([1 10])`, then `inputname` must include two signals. Scalar values and SISO LTI models, however, automatically expand to any input or output dimension.

A value of `WL = []` or `WR = []` is interpreted as the identity.

**Default:** []

## Properties

### WL

Frequency-weighting function for the output channels of the transfer function to constrain, specified as a scalar, a matrix, or a SISO or MIMO numeric LTI model. The initial value of this property is set by the WL input argument when you construct the tuning goal.

### WR

Frequency-weighting function for the input channels of the transfer function to constrain, specified as a scalar, a matrix, or a SISO or MIMO numeric LTI model. The initial value of this property is set by the WR input argument when you construct the tuning goal.

### IPX

Target passivity at the inputs listed in `inputname`, specified as a scalar value. The input passivity index is defined as the largest value of  $\nu$  for which the trajectories  $\{u(t), y(t)\}$  of the weighted transfer function  $H$  satisfy:

$$\int_0^T y(t)^\top u(t) dt > \nu \int_0^T u(t)^\top u(t) dt,$$

for all  $T > 0$ .

By default, the tuning goal enforces strict passivity of the weighted transfer function. To enforce an input passivity index with a specified excess or shortage of passivity, set the IPX property of the tuning goal. When you do so, the tuning software:

- Ensures that the weighted response is input strictly passive when  $IPX > 0$ . The magnitude of IPX sets the required excess of passivity.
- Allows the weighted response to be not input strictly passive when  $IPX < 0$ . The magnitude of IPX sets the permitted shortage of passivity.

See “Weighted Passivity and Input Passivity” on page 1-201 for an example. See `getPassiveIndex` for more information about passivity indices.

**Default:** 0

**OPX**

Target passivity at the outputs listed in `outputname`, specified as a scalar value. The output passivity index is defined as the largest value of  $\rho$  for which the trajectories  $\{u(t), y(t)\}$  of the weighted transfer function  $H$  satisfy:

$$\int_0^T y(t)^\top u(t) dt > \rho \int_0^T y(t)^\top y(t) dt,$$

for all  $T > 0$ .

By default, the tuning goal enforces strict passivity of the weighted transfer function. To enforce an output passivity index with a specified excess or shortage of passivity, set the `OPX` property of the tuning goal. When you do so, the tuning software:

- Ensures that the weighted response is output strictly passive when `OPX > 0`. The magnitude of `IPX` sets the required excess of passivity.
- Allows the weighted response to be not output strictly passive when `OPX < 0`. The magnitude of `IPX` sets the permitted shortage of passivity.

See “Weighted Passivity and Input Passivity” on page 1-201 for an example. See `getPassiveIndex` for more information about passivity indices.

**Default:** 0

**Focus**

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `Focus` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are tuning (`rad/TimeUnit`). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0, Inf]` for continuous time; `[0, pi/Ts]` for discrete time, where `Ts` is the model sample time.

## Input

Input signal names, specified as a cell array of character vectors. The input signal names specify the input locations for determining passivity, initially populated by the `inputname` argument.

## Output

Output signal names, specified as a cell array of character vectors. The output signal names specify the output locations for determining passivity, initially populated by the `outputname` argument.

## Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

## Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `slTuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `slTuner` interface. Use `getPoints` to get the list of analysis points available in an `slTuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = {'u1', 'u2'}`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** `{}`

### Name

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** `[]`

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ , where  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.WeightedPassivity` requirement, for a closed-loop transfer function  $T(s, x)$  from `inputname` to `outputname`, and the weighted transfer function  $H(s, x) = WL * T(s, x) * WR$ ,  $f(x)$  is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

$R$  is the relative sector index (see `getSectorIndex`) of  $[H(s, x); I]$ , for the sector represented by:

$$Q = \begin{pmatrix} 2\rho & -I \\ -I & 2\nu \end{pmatrix}$$

using the values of the OPX and IPX properties for  $\rho$  and  $\nu$ , respectively.  $R_{\max}$  is fixed at  $10^6$ , included to avoid numerical errors for very large  $R$ .

## Tips

- Use `viewSpec` to visualize this tuning goal. For enforcing passivity with `IPX = 0` and `OPX = 0`, `viewSpec` plots the relative passivity indices as a function of frequency (see `passiveplot`). These are the singular values of  $(I - H(j\omega))(I - H(j\omega))^{-1}$ . The weighted transfer function  $H$  is passive when the largest singular value is less than 1 at all frequencies.

For nonzero IPX or OPX, `viewSpec` plots the relative index as described in “Algorithms” on page 1-200.

- This tuning goal imposes an implicit minimum-phase constraint on the transfer function  $H + I$ , where  $H$  is the weighted closed-loop transfer function from `Input` to `Output`, evaluated with loops opened at the points identified in `Openings`. The transmission zeros of  $H + I$  are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systuneOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systuneOptions` to change these defaults.

## Examples

### Weighted Passivity and Input Passivity

Create a tuning goal that enforces the passivity of the transfer function:

$$H(s) = \begin{bmatrix} 1 & 0 \\ 0 & 10 \end{bmatrix} T(s) \begin{pmatrix} 1 \\ s \end{pmatrix},$$

where  $T(s)$  is the transfer function from an input 'd' to outputs ['y'; 'z'] in a control system model.

```
WL = tf(1,[1 0]);
WR = diag([1 10]);
```

```
TG = TuningGoal.WeightedPassivity('d',{ 'y', 'z' },WL,WR);
```

Use **TG** with `system` to enforce that weighted passivity requirement.

Suppose that instead of enforcing overall passivity of the weighted transfer function  $H$ , you want to ensure that  $H$  is input strictly passive with an input feedforward passivity index of at least 0.1. To do so, set the **IPX** property of **TG**.

```
TG.IPX = 0.1;
```

### See Also

`system` (for `sITuner`) | `TuningGoal.Passivity` | `sITuner` | `looptune` | `system` | `looptune` (for `sITuner`) | `viewSpec` | `evalSpec` | `getPassiveIndex` | `passiveplot`

### How To

- “About Passivity and Passivity Indices”
- “Vibration Control in Flexible Beam”
- “Tuning Control Systems with SYSTUNE”
- “Tune Control Systems in Simulink”



# TuningGoal.WeightedGain class

**Package:** TuningGoal

Frequency-weighted gain constraint for control system tuning

## Description

Use the `TuningGoal.WeightedGain` object to specify a tuning requirement that limits the weighted gain from specified inputs to outputs. The weighted gain is the maximum across frequency of the gain from input to output, multiplied by weighting functions that you specify. You can use the `TuningGoal.WeightedGain` requirement for control system tuning with tuning commands such as `systemtune` or `looptune`.

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-206 of the object.

## Construction

`Req = TuningGoal.WeightedGain(inputname,outputname,WL,WR)` creates a tuning requirement. This tuning requirement specifies that the closed-loop transfer function,  $H(s)$ , from the specified input to output meets the requirement:

$$\| | W_L(s)H(s)W_R(s) | |_{\infty} < 1.$$

The notation  $\| \cdot \|_{\infty}$  denotes the maximum gain across frequency (the  $H_{\infty}$  norm).

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input.

- Any linear analysis point marked in the model.
- Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `u1` and `u2`. Use `'u1'` to designate that point as an input signal when creating tuning goals. Use `{'u1', 'u2'}` to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `inputname` can include:
  - Any input of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `inputname` can be any input name in `T.InputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `inputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `outputname` can include:

- Any model output.
- Any linear analysis point marked in the model.
- Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `y1` and `y2`. Use `'y1'` to designate that point as an output signal when creating tuning goals. Use `{'y1', 'y2'}` to designate a two-channel output.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `outputname` can include:

- Any output of the `genss` model
- Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `outputname` can be any output name in `T.OutputName`. Also, if `T` contains an `AnalysisPoint` block with a location named `AP_u`, then `outputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `outputname` is an `AnalysisPoint` location of a generalized model, the output signal for the tuning goal is the implied output associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **WL,WR**

Frequency-weighting functions, specified as scalars, matrices, or SISO or MIMO numeric LTI models.

The functions `WL` and `WR` provide the weights for the tuning requirement. The tuning requirement ensures that the gain  $H(s)$  from the specified input to output satisfies the inequality:

$$\| |WL(s)H(s)WR(s)| |_{\infty} < 1.$$

`WL` provides the weighting for the output channels of  $H(s)$ , and `WR` provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model. For example:

```
WL = tf(1,[1 0.01]);  
WR = 10;
```

If you specify MIMO weighting functions, then `inputname` and `outputname` must be vector signals. The dimensions of the vector signals must be such that the dimensions of  $H(s)$  are commensurate with the dimensions of `WL` and `WR`. For example, if you specify `WR = diag([1 10])`, then `inputname` must include two signals. Scalar values, however, automatically expand to any input or output dimension.

A value of `WL = []` or `WR = []` is interpreted as the identity.

## Properties

### WL

Frequency-weighting function for the output channels of the transfer function to constrain, specified as a scalar, a matrix, or a SISO or MIMO numeric LTI model. The initial value of this property is set by the `WL` input argument when you construct the tuning goal.

### WR

Frequency-weighting function for the input channels of the transfer function to constrain, specified as a scalar, a matrix, or a SISO or MIMO numeric LTI model. The initial value of this property is set by the `WR` input argument when you construct the tuning goal.

### Focus

Frequency band in which tuning goal is enforced, specified as a row vector of the form `[min,max]`.

Set the `Focus` property to limit enforcement of the tuning goal to a particular frequency band. Express this value in the frequency units of the control system model you are

tuning (rad/TimeUnit). For example, suppose `Req` is a tuning goal that you want to apply only between 1 and 100 rad/s. To restrict the tuning goal to this band, use the following command:

```
Req.Focus = [1,100];
```

**Default:** `[0, Inf]` for continuous time; `[0, pi/Ts]` for discrete time, where `Ts` is the model sample time.

### **Stabilize**

Stability requirement on closed-loop dynamics, specified as 1 (**true**) or 0 (**false**).

By default, `TuningGoal.Gain` imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain requirement. If stability is not required or cannot be achieved, set **Stabilize** to **false** to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, set **Stabilize** to **false**.

**Default:** 1(**true**)

### **Input**

Input signal names, specified as a cell array of character vectors that identify the inputs of the transfer function that the tuning requirement constrains. The initial value of the **Input** property is set by the `inputname` input argument when you construct the requirement object.

### **Output**

Output signal names, specified as a cell array of character vectors that identify the outputs of the transfer function that the tuning requirement constrains. The initial value of the **Output** property is set by the `outputname` input argument when you construct the requirement object.

### **Models**

Models to which the tuning goal applies, specified as a vector of indices.

Use the **Models** property when tuning an array of control system models with `systemtune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model

array passed to `system`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

### **Openings**

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then `Openings` can include any linear analysis point marked in the model, or any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points and loop openings to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `Openings` can include any `AnalysisPoint` location in the control system model. Use `getPoints` to get the list of analysis points available in the `genss` model.

For example, if `Openings = { 'u1 ' , 'u2 ' }`, then the tuning goal is evaluated with loops open at analysis points `u1` and `u2`.

**Default:** {}

### **Name**

Name of the tuning goal, specified as a character vector.

For example, if `Req` is a tuning goal:

```
Req.Name = 'LoopReq' ;
```

**Default:** []

## Tips

- This tuning goal imposes an implicit stability constraint on the weighted closed-loop transfer function from **Input** to **Output**, evaluated with loops opened at the points identified in **Openings**. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value  $f(x)$ .  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.WeightedGain` requirement,  $f(x)$  is given by:

$$f(x) = \|W_L T(s, x) W_R\|_\infty.$$

$T(s, x)$  is the closed-loop transfer function from **Input** to **Output**.  $\|\cdot\|_\infty$  denotes the  $H_\infty$  norm (see `norm`).

## Examples

### Constrain Weighted Gain of Closed-Loop System

Create a tuning goal requirement that constrains the gain of a closed-loop SISO system from its input,  $r$ , to its output,  $y$ . Weight the gain at its input by a factor of 10 and at its output by the frequency-dependent weight  $1/(s + 0.01)$ .

```
WL = tf(1, [1 0.01]);
WR = 10;
```

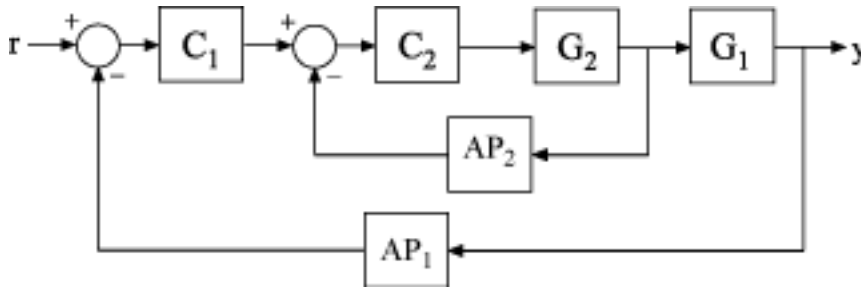
```
Req = TuningGoal.WeightedGain('r','y',WL,WR);
```

You can use the requirement `Req` with `systemtune` to tune the free parameters of any control system model that has an input signal named `'r'` and an output signal named `'y'`.

You can then use `viewSpec` to validate the tuned control system against the requirement.

### Constrain Weighted Gain Evaluated with a Loop Opening

Create a requirement that constrains the gain of the outer loop of the following control system, evaluated with the inner loop open.



Create a model of the system. To do so, specify and connect the numeric plant models, `G1` and `G2`, the tunable controllers `C1` and `C2`. Also, create and connect the `AnalysisPoint` blocks that mark points of interest for analysis or tuning, `AP1` and `AP2`.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
AP1 = AnalysisPoint('AP1');
AP2 = AnalysisPoint('AP2');
T = feedback(G1*feedback(G2*C2,AP2)*C1,AP1);
T.InputName = 'r';
T.OutputName = 'y';
```

Create a tuning requirement that constrains the gain of this system from `r` to `y`. Weight the gain at the output by  $s/(s + 0.5)$ .

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



```
Req = TuningGoal.WeightedGain('r','y',WL,[]);
```

This requirement is equivalent to `Req = TuningGoal.Gain('r','y',1/WL)`. However, for MIMO systems, you can use `TuningGoal.WeightedGain` to create channel-specific weightings that cannot be expressed as `TuningGoal.Gain` requirements.

Specify that the transfer function from  $r$  to  $y$  be evaluated with the outer loop open for the purpose of tuning to this constraint.

```
Req.Openings = 'AP1';
```

By default, tuning using `TuningGoal.WeightedGain` imposes a stability requirement as well as the gain requirement. Practically, in some control systems it is not possible to achieve a stable inner loop. When this occurs, remove the stability requirement for the inner loop by setting the `Stabilize` property to `false`.

```
Req.Stabilize = false;
```

The tuning algorithm still imposes a stability requirement on the overall tuned control system, but not on the inner loop alone.

Use `systemtune` to tune the free parameters of `T` to meet the tuning requirement specified by `Req`. You can then validate the tuned control system against the requirement using the command `viewSpec(Req,T,Info)`.

## See Also

`looptune` (for `sITuner`) | `looptune` | `systemtune` | `systemtune` (for `sITuner`) | `sITuner` | `viewSpec` | `evalSpec`

## How To

- “Frequency-Domain Specifications”

# TuningGoal.WeightedVariance class

**Package:** TuningGoal

Frequency-weighted  $H_2$  norm constraint for control system tuning

## Description

Use the `TuningGoal.WeightedVariance` object to specify a tuning requirement that limits the weighted  $H_2$  norm of the transfer function from specified inputs to outputs. The  $H_2$  norm measures:

- The total energy of the impulse response, for deterministic inputs to the transfer function.
- The square root of the output variance for a unit-variance white-noise input, for stochastic inputs to the transfer function. Equivalently, the  $H_2$  norm measures the root-mean-square of the output for such input.

You can use the `TuningGoal.WeightedVariance` requirement for control system tuning with tuning commands, such as `sysstune` or `looptune`. By specifying this requirement, you can tune the system response to stochastic inputs with a nonuniform spectrum such as colored noise or wind gusts. You can also use `TuningGoal.WeightedVariance` to specify LQG-like performance objectives.

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-216 of the object.

## Construction

`Req = TuningGoal.Variance(inputname,outputname,WL,WR)` creates a tuning requirement `Req`. This tuning requirement specifies that the closed-loop transfer function  $H(s)$  from the specified input to output meets the requirement:

$$\| | W_L(s)H(s)W_R(s) | | _2 < 1.$$

The notation  $\| | \cdot | | _2$  denotes the  $H_2$  norm.

When you are tuning a discrete-time system, `Req` imposes the following constraint:

$$\frac{1}{\sqrt{T_s}} \|W_L(z)T(z,x)W_R(z)\|_2 < 1.$$

The  $H_2$  norm is scaled by the square root of the sample time  $T_s$  to ensure consistent results with tuning in continuous time. To constrain the true discrete-time  $H_2$  norm, multiply either  $W_L$  or  $W_R$  by  $\sqrt{T_s}$ .

## Input Arguments

### **inputname**

Input signals for the tuning goal, specified as a character vector or, for multiple-input tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then **inputname** can include:
  - Any model input.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an **sITuner** interface associated with the Simulink model. Use **addPoint** to add analysis points to the **sITuner** interface. Use **getPoints** to get the list of analysis points available in an **sITuner** interface to your model.

For example, suppose that the **sITuner** interface contains analysis points **u1** and **u2**. Use **'u1'** to designate that point as an input signal when creating tuning goals. Use **{'u1','u2'}** to designate a two-channel input.

- If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **inputname** can include:
  - Any input of the **genss** model
  - Any **AnalysisPoint** location in the control system model

For example, if you are tuning a control system model, **T**, then **inputname** can be any input name in **T.InputName**. Also, if **T** contains an **AnalysisPoint** block with a location named **AP\_u**, then **inputname** can include **'AP\_u'**. Use **getPoints** to get a list of analysis points available in a **genss** model.

If `inputname` is an `AnalysisPoint` location of a generalized model, the input signal for the tuning goal is the implied input associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

### **outputname**

Output signals for the tuning goal, specified as a character vector or, for multiple-output tuning goals, a cell array of character vectors.

- If you are using the tuning goal to tune a Simulink model of a control system, then `outputname` can include:
  - Any model output.
  - Any linear analysis point marked in the model.
  - Any linear analysis point in an `sITuner` interface associated with the Simulink model. Use `addPoint` to add analysis points to the `sITuner` interface. Use `getPoints` to get the list of analysis points available in an `sITuner` interface to your model.

For example, suppose that the `sITuner` interface contains analysis points `y1` and `y2`. Use `'y1'` to designate that point as an output signal when creating tuning goals. Use `{'y1', 'y2'}` to designate a two-channel output.

- If you are using the tuning goal to tune a generalized state-space (`genss`) model of a control system, then `outputname` can include:
  - Any output of the `genss` model
  - Any `AnalysisPoint` location in the control system model

For example, if you are tuning a control system model, `T`, then `outputname` can be any output name in `T.OutputName`. Also, if `T` contains an `AnalysisPoint` block

with a location named `AP_u`, then `outputname` can include `'AP_u'`. Use `getPoints` to get a list of analysis points available in a `genss` model.

If `outputname` is an `AnalysisPoint` location of a generalized model, the output signal for the tuning goal is the implied output associated with the `AnalysisPoint` block:



For more information about analysis points in control system models, see “Marking Signals of Interest for Control System Analysis and Design”.

## WL, WR

Frequency-weighting functions, specified as scalars, matrices, or SISO or MIMO numeric LTI models.

The functions `WL` and `WR` provide the weights for the tuning requirement. The tuning requirement ensures that the gain  $H(s)$  from the specified input to output satisfies the inequality:

$$\| |W_L(s)H(s)W_R(s)| \|_2 < 1.$$

`WL` provides the weighting for the output channels of  $H(s)$ , and `WR` provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model. For example:

```
WL = tf(1,[1 0.01]);
WR = 10;
```

If you specify MIMO weighting functions, then `inputname` and `outputname` must be vector signals. The dimensions of the vector signals must be such that the dimensions of  $H(s)$  are commensurate with the dimensions of `WL` and `WR`. For example, if you specify `WR = diag([1 10])`, then `inputname` must include two signals. Scalar values, however, automatically expand to any input or output dimension.

When you are tuning a discrete-time system, `WL` and `WR` must be either scalars or discrete-time models having the same sample time (`Ts`) as the model you are tuning.

A value of `WL = []` or `WR = []` is interpreted as the identity.

## Properties

### WL

Frequency-weighting function for the output channels of the transfer function to constrain, specified as a scalar, a matrix, or a SISO or MIMO numeric LTI model. The initial value of this property is set by the WL input argument when you construct the tuning goal.

### WR

Frequency-weighting function for the input channels of the transfer function to constrain, specified as a scalar, a matrix, or a SISO or MIMO numeric LTI model. The initial value of this property is set by the WR input argument when you construct the tuning goal.

### Input

Input signal names, specified as a cell array of character vectors that identify the inputs of the transfer function that the tuning requirement constrains. The initial value of the Input property is set by the `inputname` input argument when you construct the requirement object.

### Output

Output signal names, specified as a cell array of character vectors that identify the outputs of the transfer function that the tuning requirement constrains. The initial value of the Output property is set by the `outputname` input argument when you construct the requirement object.

### Models

Models to which the tuning goal applies, specified as a vector of indices.

Use the `Models` property when tuning an array of control system models with `systemtune`, to enforce a tuning goal for a subset of models in the array. For example, suppose you want to apply the tuning goal, `Req`, to the second, third, and fourth models in a model array passed to `systemtune`. To restrict enforcement of the tuning goal, use the following command:

```
Req.Models = 2:4;
```

When `Models = NaN`, the tuning goal applies to all models.

**Default:** NaN

## Openings

Feedback loops to open when evaluating the tuning goal, specified as a cell array of character vectors that identify loop-opening locations. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify.

If you are using the tuning goal to tune a Simulink model of a control system, then **Openings** can include any linear analysis point marked in the model, or any linear analysis point in an **slTuner** interface associated with the Simulink model. Use **addPoint** to add analysis points and loop openings to the **slTuner** interface. Use **getPoints** to get the list of analysis points available in an **slTuner** interface to your model.

If you are using the tuning goal to tune a generalized state-space (**genss**) model of a control system, then **Openings** can include any **AnalysisPoint** location in the control system model. Use **getPoints** to get the list of analysis points available in the **genss** model.

For example, if **Openings** = { 'u1' , 'u2' }, then the tuning goal is evaluated with loops open at analysis points **u1** and **u2**.

**Default:** {}

### Name

Name of the tuning goal, specified as a character vector.

For example, if **Req** is a tuning goal:

```
Req.Name = 'LoopReq';
```

**Default:** []

## Tips

- When you use this requirement to tune a continuous-time control system, **systune** attempts to enforce zero feedthrough ( $D = 0$ ) on the transfer that the requirement constrains. Zero feedthrough is imposed because the  $H_2$  norm, and therefore the value of the tuning goal (see “Algorithms” on page 1-218), is infinite for continuous-time systems with nonzero feedthrough.

`system` enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. `system` returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, use the `Value` and `Free` properties of the block parametrization. For example, consider a tuned state-space block:

```
C = tunableSS('C',1,2,3);
```

To enforce zero feedthrough on this block, set its  $D$  matrix value to zero, and fix the parameter.

```
C.D.Value = 0;  
C.D.Free = false;
```

For more information on fixing parameter values, see the Control Design Block reference pages, such as `tunableSS`.

- This tuning goal imposes an implicit stability constraint on the weighted closed-loop transfer function from `Input` to `Output`, evaluated with loops opened at the points identified in `Openings`. The dynamics affected by this implicit constraint are the *stabilized dynamics* for this tuning goal. The `MinDecay` and `MaxRadius` options of `systemOptions` control the bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, use `systemOptions` to change these defaults.

## Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the software converts the requirement into a normalized scalar value



$f(x)$ .  $x$  is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize  $f(x)$  or to drive  $f(x)$  below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.WeightedVariance` requirement,  $f(x)$  is given by:

$$f(x) = \|W_L T(s, x) W_R\|_2.$$

$T(s, x)$  is the closed-loop transfer function from **Input** to **Output**.  $\|\cdot\|_2$  denotes the  $H_2$  norm (see `norm`).

For tuning discrete-time control systems,  $f(x)$  is given by:

$$f(x) = \frac{1}{\sqrt{T_s}} \|W_L(z) T(z, x) W_R(z)\|_2.$$

$T_s$  is the sample time of the discrete-time transfer function  $T(z, x)$ .

## Examples

### Weighted Constraint on H2 Norm

Create a constraint for a transfer function with one input,  $r$ , and two outputs,  $e$  and  $y$ , that limits the  $H_2$  norm as follows:

$$\left\| \begin{array}{c} \frac{1}{s + 0.001} T_{re} \\ \frac{s}{0.001s + 1} T_{ry} \end{array} \right\|_2 < 1.$$

$T_{re}$  is the closed-loop transfer function from  $r$  to  $e$ , and  $T_{ry}$  is the closed-loop transfer function from  $r$  to  $y$ .

```
s = tf('s');
WL = blkdiag(1/(s+0.001), s/(0.001*s+1));
```

```
Req = TuningGoal.WeightedVariance('r',{ 'e', 'y'},WL,[]);
```

## See Also

systemtune (for sITuner) | TuningGoal.Gain | TuningGoal.Variance | systemtune |  
looptune | looptune (for sITuner) | TuningGoal.LoopShape | sITuner | norm

## How To

- “Frequency-Domain Specifications”
- “Fault-Tolerant Control of a Passenger Jet”

# Functions — Alphabetical List

---

# abs

Entrywise magnitude of frequency response

## Syntax

```
absfrd = abs(sys)
```

## Description

`absfrd = abs(sys)` computes the magnitude of the frequency response contained in the FRD model `sys`. For MIMO models, the magnitude is computed for each entry. The output `absfrd` is an FRD object containing the magnitude data across frequencies.

## See Also

`bodemag` | `sigma` | `fnorm`

**Introduced in R2006a**

# absorbDelay

Replace time delays by poles at  $z = 0$  or phase shift

## Syntax

```
sysnd = absorbDelay(sysd)
[sysnd,G] = absorbDelay(sysd)
```

## Description

`sysnd = absorbDelay(sysd)` absorbs all time delays of the dynamic system model `sysd` into the system dynamics or the frequency response data.

For discrete-time models (other than frequency response data models), a delay of  $k$  sampling periods is replaced by  $k$  poles at  $z = 0$ . For continuous-time models (other than frequency response data models), time delays have no exact representation with a finite number of poles and zeros. Therefore, use `pade` to compute a rational approximation of the time delay.

For frequency response data models in both continuous and discrete time, `absorbDelay` absorbs all time delays into the frequency response data as a phase shift.

`[sysnd,G] = absorbDelay(sysd)` returns the matrix `G` that maps the initial states of the ss model `sysd` to the initial states of the `sysnd`.

## Examples

### Absorb Time Delay into System Dynamics

Create a discrete-time transfer function that has a time delay.

```
z = tf('z',-1);
sysd = (-0.4*z -0.1)/(z^2 + 1.05*z + 0.08);
sysd.InputDelay = 3
```

```
sysd =
```

$$z^{-3} * \frac{-0.4 z - 0.1}{z^2 + 1.05 z + 0.08}$$

Sample time: unspecified  
Discrete-time transfer function.

The display of `sysd` represents the `InputDelay` as a factor of  $z^{-3}$ , separate from the system poles that appear in the transfer function denominator.

Absorb the time delay into the system dynamics as poles at  $z = 0$ .

```
sysnd = absorbDelay(sysd)
```

```
sysnd =
```

$$\frac{-0.4 z - 0.1}{z^5 + 1.05 z^4 + 0.08 z^3}$$

Sample time: unspecified  
Discrete-time transfer function.

The display of `sysnd` shows that the factor of  $z^{-3}$  has been absorbed as additional poles in the denominator.

Verify that `sysnd` has no input delay.

```
sysnd.InputDelay
```

```
ans =
```

```
0
```

### Convert Leading Structural Zeros of Polynomial Model to Regular Coefficients

Create a discrete-time polynomial model.

```
m = idpoly(1,[0 0 0 2 3]);
```

Convert `m` to a transfer function model.

```
sys = tf(m)
```

```
sys =
```

$$z^{-2} * (2 z^{-1} + 3 z^{-2})$$

```
Sample time: unspecified
Discrete-time transfer function.
```

The numerator of the transfer function, `sys`, is `[0 2 3]` and the transport delay, `sys.IODelay`, is 2. This is because the value of the B polynomial, `m.B`, has 3 leading zeros. The first fixed zero shows lack of feedthrough in the model. The two zeros after that are treated as input-output delays.

Use `absorbDelay` to treat the leading zeros as regular B coefficients.

```
m2 = absorbDelay(m);
sys2 = tf(m2)
```

```
sys2 =
```

$$2 z^{-3} + 3 z^{-4}$$

```
Sample time: unspecified
Discrete-time transfer function.
```

The numerator of `sys2` is `[0 0 0 2 3]` and transport delay is 0. The model `m2` treats the leading zeros as regular coefficients by freeing their values. `m2.Structure.B.Free(2:3)` is TRUE while `m.Structure.B.Free(2:3)` is FALSE.

## See Also

`totaldelay` | `hasdelay` | `pade`

**Introduced in R2011b**

## allmargin

Gain margin, phase margin, delay margin and crossover frequencies

### Syntax

```
S = allmargin(sys)
S = allmargin(mag,phase,w,ts)
```

### Description

`S = allmargin(sys)` computes the gain margin, phase margin, delay margin and the corresponding crossover frequencies of the SISO open-loop model `sys`. The `allmargin` command is applicable to any SISO model, including models with delays.

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

- **GMFrequency** — All  $-180^\circ$  (modulo  $360^\circ$ ) crossover frequencies in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.
- **GainMargin** — Corresponding gain margins, defined as  $1/G$ , where  $G$  is the gain at the  $-180^\circ$  crossover frequency. Gain margins are in absolute units.
- **PMFrequency** — All 0 dB crossover frequencies in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.
- **PhaseMargin** — Corresponding phase margins in degrees.
- **DMFrequency** and **DelayMargin** — Critical frequencies and the corresponding delay margins. Delay margins are specified in the time units of the system 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.

Where stability cannot be assessed, `Stable` is set to `NaN`. In general, stability cannot be assessed for an `frd` system.

`S = allmargin(mag,phase,w,ts)` computes the stability margins from the frequency response data `mag`, `phase`, `w`, and the sample time, `ts`. Provide magnitude values `mag`



in absolute units, and phase values `phase` in degrees. You can provide the frequency vector `w` in any units; `allmargin` returns frequencies in the same units. `allmargin` interpolates between frequency points to approximate the true stability margins.

## See Also

Linear System Analyzer | `margin`

**Introduced before R2006a**

# AnalysisPoint

Points of interest for linear analysis

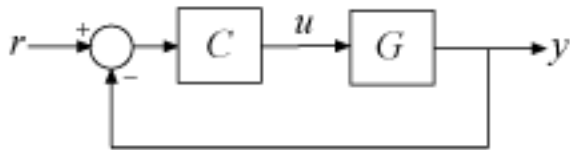
## Syntax

```
AP = AnalysisPoint(name)
AP = AnalysisPoint(name,N)
```

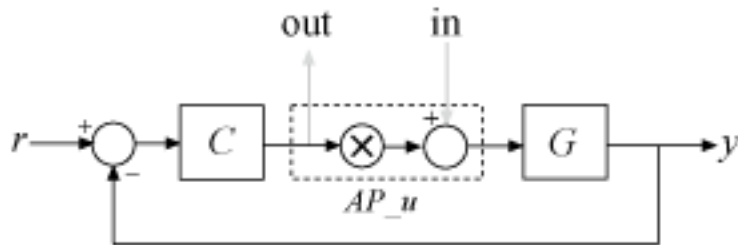
## Description

`AnalysisPoint` is a Control Design Block for marking a location in a control system model as a point of interest for linear analysis and controller tuning. You can combine an `AnalysisPoint` block with numeric LTI models, tunable LTI models, and other Control Design Blocks to build tunable models of control systems. `AnalysisPoint` locations are available for analysis with commands such as `getIOTransfer` or `getLoopTransfer`. Such locations are also available for specifying design goals for control system tuning.

For example, consider the following control system.



Suppose that you are interested in the effects of disturbance injected at  $u$  in this control system. Inserting an `AnalysisPoint` block at the location  $u$  associates an implied input, implied output, and the option to open the loop at that location, as in the following diagram.



Suppose that  $T$  is a model of the control system including the `AnalysisPoint` block, `AP_u`. In this case, the command `getIOTransfer(T, 'AP_u', 'y')` returns a model of the closed-loop transfer function from  $u$  to  $y$ . Likewise, the command `getLoopTransfer(T, 'AP_u', -1)` returns a model of the negative-feedback open-loop response,  $CG$ , measured at the location  $u$ .

`AnalysisPoint` blocks are also useful when tuning a control system using tuning commands such as `sys tune`. You can use an `AnalysisPoint` block to mark a loop-opening location for open-loop tuning requirements such as `TuningGoal.LoopShape` or `TuningGoal.Margins`. You can also use an `AnalysisPoint` block to mark the specified input or output for tuning requirements such as `TuningGoal.Gain`. For example, `Req = TuningGoal.Margins('AP_u', 5, 40)` constrains the gain and phase margins at the location  $u$ .

You can create `AnalysisPoint` blocks explicitly using the `AnalysisPoint` command and connect them with other block diagram components using model interconnection commands. For example, the following code creates a model of the system illustrated above. (See “Construction” on page 2-10 and “Examples” on page 2- below for more information.)

```
G = tf(1,[1 2]);
C = tunablePID('C','pi');
AP_u = AnalysisPoint('u');
T = feedback(G*AP_u*C,1);           % closed loop r->y
```

You can also create analysis points implicitly, using the `connect` command. The following syntax creates a dynamic system model with analysis points, by interconnecting multiple models `sys1`, `sys2`, . . . , `sysN`:

```
sys = connect(sys1,sys2,...,sysN,inputs,outputs,APs);
```

`APs` lists the signal locations at which to insert analysis points. The software automatically creates and inserts an `AnalysisPoint` block with channels corresponding to these locations. See `connect` for more information.

## Construction

`AP = AnalysisPoint(name)` creates a single-channel analysis point. Insert `AP` anywhere in the generalized model of your control system to mark a point of interest for linear analysis or controller tuning. `name` specifies the block name.

`AP = AnalysisPoint(name,N)` creates a multi-channel analysis point with `N` channels. Use this block to mark a vector-valued signal as a point of interest or to bundle together several points of interest.

## Input Arguments

### **name**

Analysis point name, specified as a character vector such as `'AP'`. This input argument sets the value of the `Name` property of the `AnalysisPoint` block. (See “Properties” on page 2-10.) When you build a control system model using the block, the `Name` property is what appears in the `Blocks` list of the resulting `genss` model.

### **N**

Number of channels for a multichannel analysis point, specified as a scalar integer.

## Properties

### **Location**

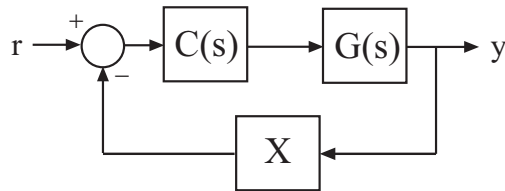
Names of channels in the `AnalysisPoint` blocks, specified as a character vector or a cell array of character vectors.

By default, the analysis-point channels are named after the `name` argument. For example, if you have a single-channel analysis point, AP, that has name 'AP', then `AP.Location = 'AP'` by default. If you have a multi-channel analysis point, then `AP.Location = {'AP(1)', 'AP(2)', ...}` by default. Set `AP.Location` to a different value if you want to customize the channel names.

## Open

Loop-opening state, specified as a logical value or vector of logical values. This property tracks whether the loop is open or closed at the analysis point.

For example, consider the feedback loop of the following illustration.



You can model this feedback loop as follows.

```
G = tf(1,[1 2]);
C = tunablePID('C','pi');
X = AnalysisPoint('X');
T = feedback(G*C,X);
```

You can get the transfer function from  $r$  to  $y$  with the feedback loop open at  $X$  as follows.

```
Try = getIOTransfer(T,'r','y','X');
```

In the resulting generalized state-space (`genss`) model, the `AnalysisPoint` block 'X' is marked open. In other words, `Try.Blocks.X.Open = 1`.

For a multi-channel analysis point, then `Open` is a logical vector with as many entries as the analysis point has channels.

**Default:** 0 for all channels

## Ts

Sample time. For `AnalysisPoint` blocks, the value of this property is automatically set to the sample time of other blocks and models you connect it with.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to {'controls(1)'; 'controls(2)'}.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** `''` for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### OutputName

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all output channels

### OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** `''` for all output channels

### OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as



a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### Name

System name, specified as a character vector. For example, `'system_1'`.

**Default:** `''`

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** `{}`

### UserData

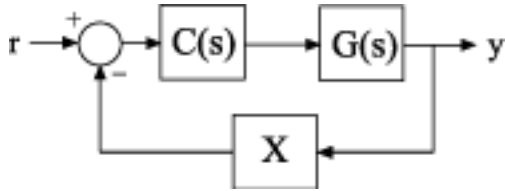
Any type of data you want to associate with system, specified as any MATLAB® data type.

**Default:** `[]`

## Examples

### Feedback Loop with Analysis Point

Create a model of the following feedback loop with an analysis point in the feedback path.



For this example, the plant model is  $G = 1/(s + 2)$ .  $C$  is a tunable PI controller, and  $X$  is the analysis point.

```
G = tf(1,[1 2]);
C = tunablePID('C','pi');
X = AnalysisPoint('X');
T = feedback(G*C,X);
T.InputName = 'r';
T.OutputName = 'y';
```

$T$  is a tunable `genss` model.  $T.Blocks$  contains the Control Design Blocks of the model, which are the controller,  $C$ , and the analysis point,  $X$ .

$T.Blocks$

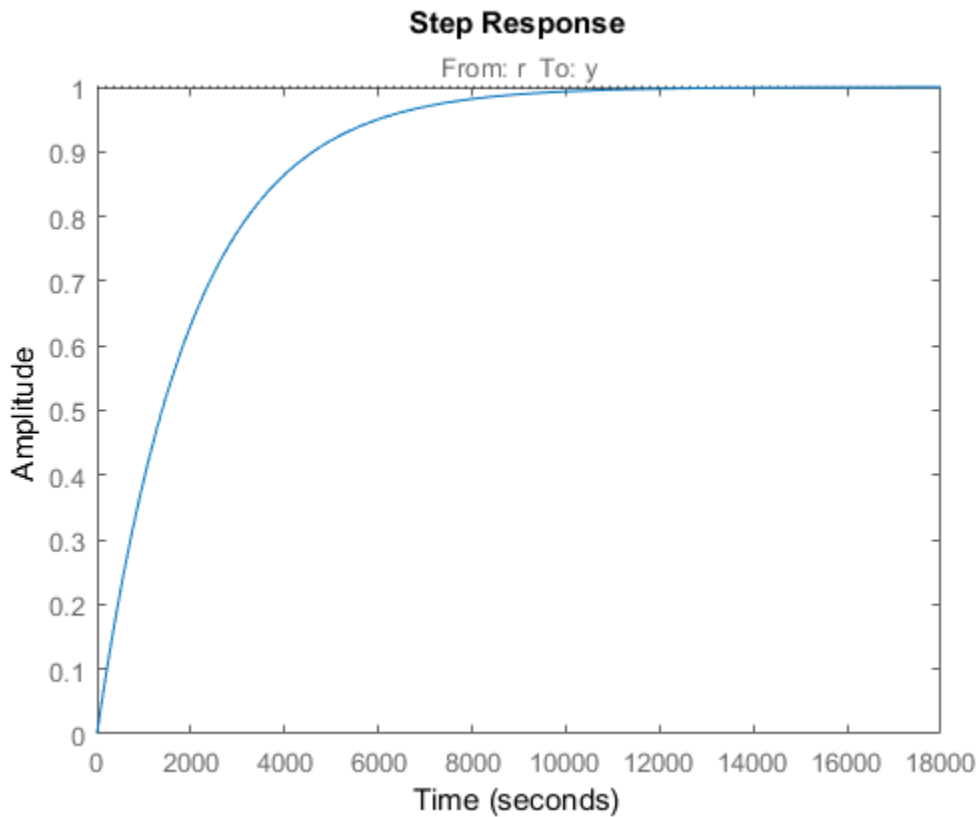
```
ans =

    struct with fields:

        C: [1x1 tunablePID]
        X: [1x1 AnalysisPoint]
```

Examine the step response of  $T$ .

```
stepplot(T)
```



The presence of the `AnalysisPoint` block does not change the dynamics of the model.

You can use the analysis point for linear analysis of the system. For instance, extract the system response at 'y' to a disturbance injected at the analysis point.

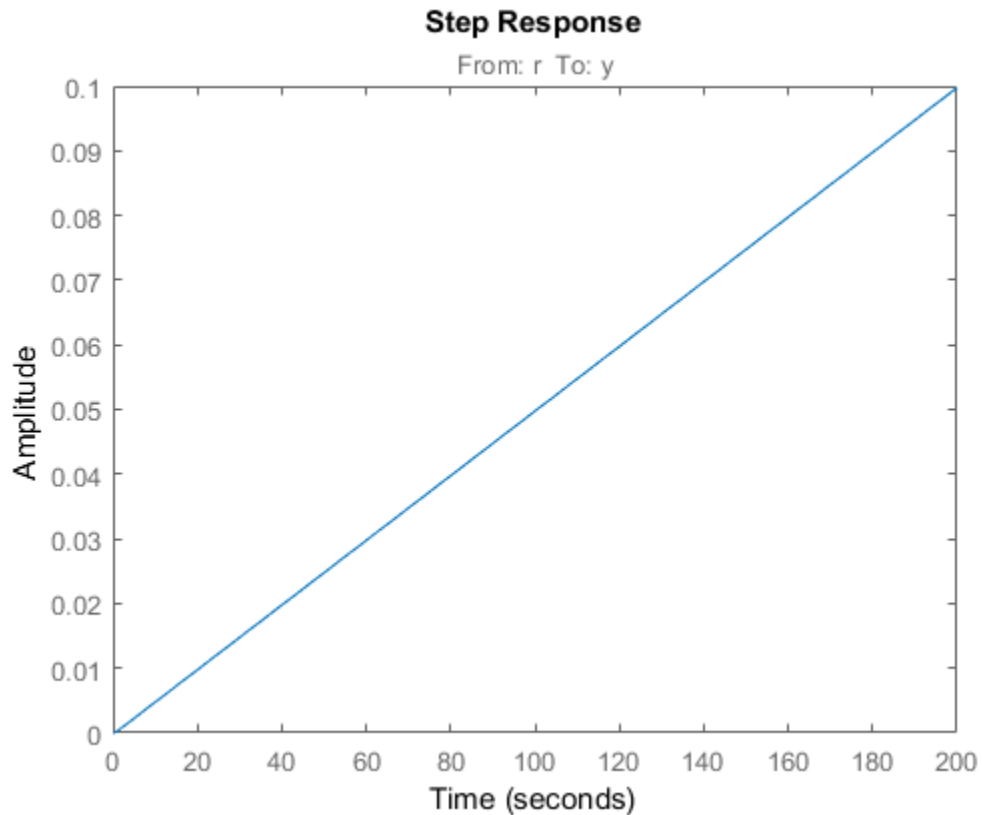
```
Txy = getIOTransfer(T, 'X', 'y');
```

The `AnalysisPoint` block also allows you to temporarily open the feedback loop at that point. For example, compute the open-loop response from 'r' to 'y'.

```
Try_open = getIOTransfer(T, 'r', 'y', 'X');
```

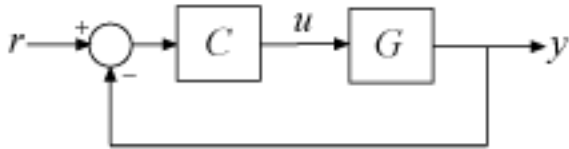
Specifying the analysis point name as the last argument to `getIOTransfer` extracts the response with the loop open at that point. Examine the step response of `Try_open` to verify that it is the open-loop response.

```
stepplot(Try_open);
```



### Feedback Loop With Analysis Point Inserted by connect

Create a model of the following block diagram from  $r$  to  $y$ . Insert an analysis point at an internal location,  $u$ .



Create C and G, and name the inputs and outputs.

```
C = pid(2,1);
C.InputName = 'e';
C.OutputName = 'u';
G = zpk([],[-1,-1],1);
G.InputName = 'u';
G.OutputName = 'y';
```

Create the summing junction.

```
Sum = sumblk('e = r - y');
```

Combine C, G, and the summing junction to create the aggregate model, with an analysis point at  $u$ .

```
T = connect(G,C,Sum, 'r', 'y', 'u')
```

```
T =
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, and 1 analysis points.
AnalysisPoints_: Analysis point, 1 channels, 1 occurrences.
```

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

The resulting T is a `genss` model. The `connect` command creates the `AnalysisPoint` block, `AnalysisPoints_`, and inserts it into T. To see the name of the analysis point channel in `AnalysisPoints_`, use `getPoints`.

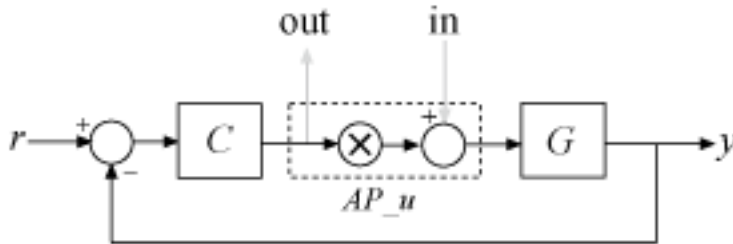
```
getPoints(T)
```

```
ans =
    cell
    'u'
```

The analysis point channel is named 'u'. You can use this analysis point to extract system responses. For example, the following commands extract the open-loop transfer at  $u$  and the closed-loop response at  $y$  to a disturbance injected at  $u$ .

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

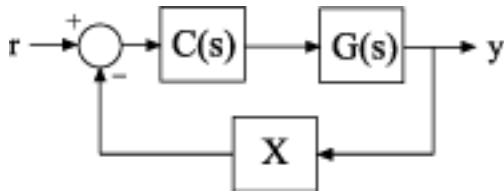
T is equivalent to the following block diagram, where  $AP_u$  designates the AnalysisPoint block AnalysisPoints\_ with channel name  $u$ .



### Multi-Channel Analysis Points

Create a block for marking two analysis points in a MIMO model.

In the control system of the following illustration, consider each signal a vector-valued signal of size 2. In other words, the signal  $r$  represents  $\{r(1), r(2)\}$ ,  $y$  represents  $\{y(1), y(2)\}$ , and so on.



The feedback signal is therefore also a vector-valued signal of size 2. Create a block for marking the two analysis points in the feedback path.

```
AP = AnalysisPoint('X',2)
```

```
AP =
```

```
Multi-channel analysis point at locations:
```

```
  X(1)
  X(2)
```

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

The **AnalysisPoint** block is stored as a variable in the MATLAB® workspace called **AP**. In addition, the **Name** property of the block is set to **X**. When you interconnect the block with numeric LTI models or other Control Design Blocks, this analysis-point block is identified in the **Blocks** property of the resulting **genss** model as **X**. The block name **X** is automatically expanded to generate the channel names **X(1)** and **X(2)**.

It is sometimes convenient to change the channel names to match the names of the signals they correspond to in a block diagram of your model. For example, suppose the points of interest you want to mark in your model are signals named **L** and **V**. Change the **Location** property of **AP** to make the names match those signals.

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

```
AP =
```

```
Multi-channel analysis point at locations:
```

```
  L
  V
```

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

Although the channel names have changed, the block name remains X.

AP . Name

```
ans =
```

```
X
```

Therefore, the **Blocks** property of a **genss** model you build with this block still identifies the block as X. Use `getPoints` to find the channel names of available analysis points in a **genss** model.

- “Control System with Multichannel Analysis Points”

## More About

- “Control Design Blocks”
- “Models with Tunable Coefficients”
- “Marking Signals of Interest for Control System Analysis and Design”

## See Also

`genss` | `getPoints` | `connect`

**Introduced in R2014b**



## append

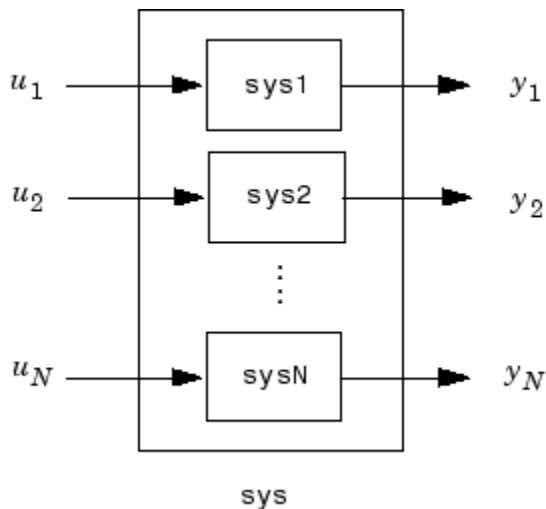
Group models by appending their inputs and outputs

### Syntax

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

### Description

`sys = append(sys1,sys2,...,sysN)` appends the inputs and outputs of the models `sys1,...,sysN` to form the augmented model `sys` depicted below.



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

$$\begin{bmatrix} H_1(s) & 0 & \dots & 0 \\ 0 & H_2(s) & \dots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \dots & 0 & H_N(s) \end{bmatrix}$$

For state-space models `sys1` and `sys2` with data  $(A_1, B_1, C_1, D_1)$  and  $(A_2, B_2, C_2, D_2)$ , `append(sys1, sys2)` produces the 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 model objects of any type. Regular matrices are also accepted as a representation of static gains, but there should be at least one model in the input list. The 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 “Rules That Determine Model Type” for details).

There is no limitation on the number of inputs.

## Examples

### Append Inputs and Outputs of Models

Create a SISO transfer function.

```
sys1 = tf(1,[1 0]);  
size(sys1)
```

Transfer function with 1 outputs and 1 inputs.

Create a SISO continuous-time state-space model.

```
sys2 = ss(1,2,3,4);  
size(sys2)
```

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

Append the inputs and outputs of `sys1`, a SISO static gain system, and `sys2`. The resulting model should be a 3-input, 3-output state-space model.

```
sys = append(sys1,10,sys2)
size(sys)
```

```
sys =
```

```
A =
```

	x1	x2
x1	0	0
x2	0	1

```
B =
```

	u1	u2	u3
x1	1	0	0
x2	0	0	2

```
C =
```

	x1	x2
y1	1	0
y2	0	0
y3	0	3

```
D =
```

	u1	u2	u3
y1	0	0	0
y2	0	10	0
y3	0	0	4

Continuous-time state-space model.

State-space model with 3 outputs, 3 inputs, and 2 states.

## See Also

`connect` | `feedback` | `parallel` | `series`

**Introduced before R2006a**

## augstate

Append state vector to output vector

### Syntax

```
asys = augstate(sys)
```

### Description

`asys = augstate(sys)` appends the state vector to the outputs of a state-space model.

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

$$\begin{aligned} \dot{x} &= Ax + Bu \\ \begin{bmatrix} y \\ x \end{bmatrix} &= \begin{bmatrix} C \\ I \end{bmatrix} x + \begin{bmatrix} D \\ 0 \end{bmatrix} u \end{aligned}$$

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` | `parallel` | `series`

**Introduced before R2006a**

## balreal

Gramian-based input/output balancing of state-space realizations

### Syntax

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

### Description

`[sysb,g] = balreal(sys)` computes a balanced realization **sysb** for the stable portion of the LTI model **sys**. `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`.

For stable systems, **sysb** is an equivalent realization for which the controllability and observability Gramians are equal and diagonal, their diagonal entries forming the vector **g** of Hankel singular values. Small entries in **g** indicate states that can be removed to simplify the model (use `modred` to reduce the model order).

If **sys** has unstable poles, its stable part is isolated, balanced, and added back to its unstable part to form **sysb**. The entries of **g** corresponding to unstable modes are set to `Inf`.

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

If the system is normalized properly, the diagonal **g** of the joint Gramian 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.

`[ ___ ] = balreal(sys,opts)` computes the balanced realization using options that you specify using `hsvdOptions`. Options include offset and tolerance options for

computing the stable-unstable decompositions. The options also allow you to limit the Gramian computation to particular time and frequency intervals. See `hsvdOptions` for details.

## Examples

### Balanced Realization of Stable System

Consider the following zero-pole-gain model, with near-canceling pole-zero pairs:

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

```
sys =
```

$$\frac{(s+10)(s+20.01)}{(s+5)(s+9.9)(s+20.1)}$$

Continuous-time zero/pole/gain model.

A state-space realization with balanced gramians is obtained by

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

The diagonal entries of the joint gramian are

```
g'
```

```
ans =
```

```
0.1006    0.0001    0.0000
```

This 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');
```

This yields the following first-order approximation of the original system.

```
zpk(sysr)
```

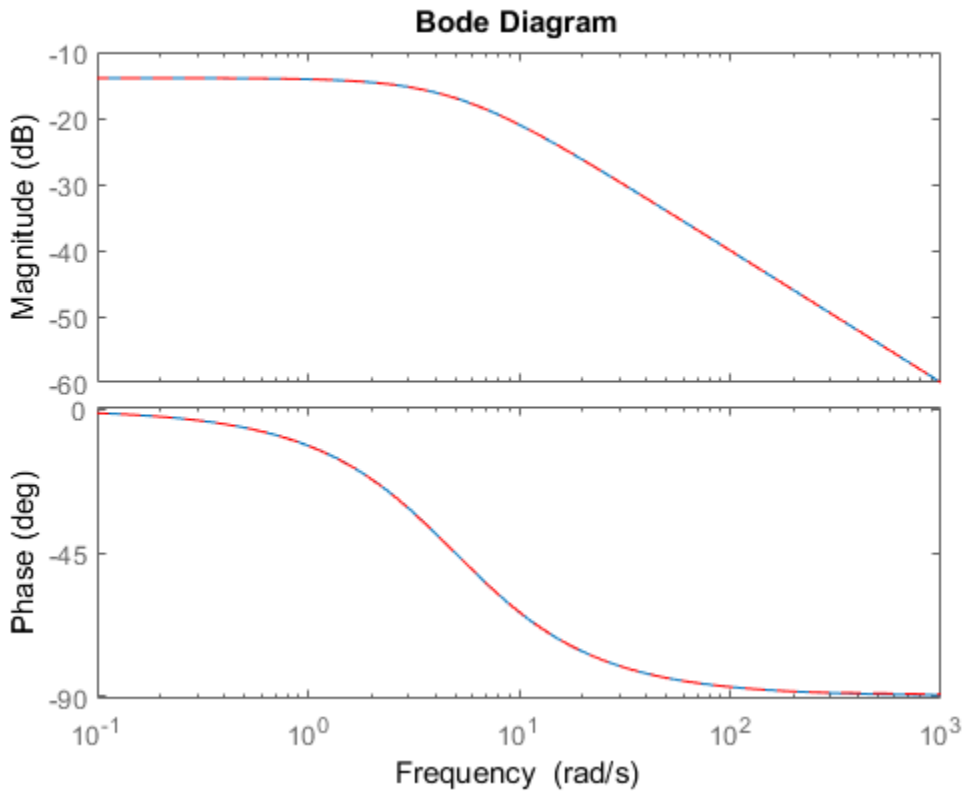
```
ans =
```

```
1.0001  
-----  
(s+4.97)
```

Continuous-time zero/pole/gain model.

Compare the Bode responses of the original and reduced-order models.

```
bodeplot(sys,sysr,'r--')
```





The plots shows that removing the second and third states does not have much effect on system dynamics.

### Balanced Realization of Unstable System

Create an unstable system.

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

```
sys =
```

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

Continuous-time transfer function.

Apply `balreal` to create a balanced-gramian realization.

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

```
sysbal =
```

```
A =
      x1  x2
x1    1   0
x2    0  -1
```

```
B =
      u1
x1  0.7071
x2  0.7071
```

```
C =
      x1      x2
y1  0.7071 -0.7071
```

```
D =
      u1
y1   0
```

Continuous-time state-space model.

```
g =
      Inf
    0.2500
```

The unstable pole shows up as `Inf` in the vector `g`.

## More About

### Algorithms

Consider the model

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

with controllability and observability Gramians  $W_c$  and  $W_o$ . The state coordinate transformation  $\bar{x} = Tx$  produces the equivalent model

$$\begin{aligned}\dot{\bar{x}} &= TAT^{-1}\bar{x} + TBu \\ y &= CT^{-1}\bar{x} + Du\end{aligned}$$

and transforms the Gramians to

$$\bar{W}_c = TW_cT^T, \quad \bar{W}_o = T^{-T}W_oT^{-1}$$

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

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

See [1], [2] for details on the algorithm.

If you use the `TimeIntervals` or `FreqIntervals` options of `hsvdOptions`, then `balreal` bases the balanced realization on time-limited or frequency-limited

controllability and observability Gramians. For information about calculating time-limited and frequency-limited Gramians, see `gram` and [4].

## 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<sup>®</sup> 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.
- [4] Gawronski, W. and J.N. Juang. "Model Reduction in Limited Time and Frequency Intervals." *International Journal of Systems Science*. Vol. 21, Number 2, 1990, pp. 349-376.

## See Also

`balred` | `hsvd` | `gram` | `hsvdOptions` | `modred`

**Introduced before R2006a**

## balred

Model order reduction

### Syntax

```
rsys = balred(sys,ORDERS)  
rsys = balred(sys,ORDERS,BALDATA)  
rsys = balred( ____,opts)
```

### Description

*rsys* = balred(*sys*,*ORDERS*) computes a reduced-order approximation *rsys* of the LTI model *sys*. The desired order (number of states) for *rsys* is specified by *ORDERS*. You can try multiple orders at once by setting *ORDERS* to a vector of integers, in which case *rsys* is a vector of reduced-order models. balred uses implicit balancing techniques to compute the reduced-order approximation *rsys*. Use *hsvd* to plot the Hankel singular values and pick an adequate approximation order. States with relatively small Hankel singular values can be safely discarded.

When *sys* has unstable poles, it is first decomposed into its stable and unstable parts using *stabsep*, and only the stable part is approximated. Use *balredOptions* to specify additional options for the stable/unstable decomposition.

When you have System Identification Toolbox™ software installed, *sys* can only be an identified state-space model (*idss*). The reduced-order model is also an *idss* model.

*rsys* = balred(*sys*,*ORDERS*,*BALDATA*) uses balancing data returned by *hsvd*. Because *hsvd* does most of the work needed to compute *rsys*, this syntax is more efficient when using *hsvd* and *balred* jointly.

*rsys* = balred( \_\_\_\_,*opts*) computes the model reduction using options that you specify using *balredOptions*. Options include offset and tolerance options for computing the stable-unstable decompositions. There also options for emphasizing particular time or frequency intervals. See *balredOptions* for details.

---

**Note:** The order of the approximate model is always at least the number of unstable poles and at most the minimal order of the original model (number NNZ of nonzero Hankel singular values using an eps-level relative threshold)

---

## Examples

### Reduced-Order Approximation with Offset Option

Compute a reduced-order approximation of the system given by:

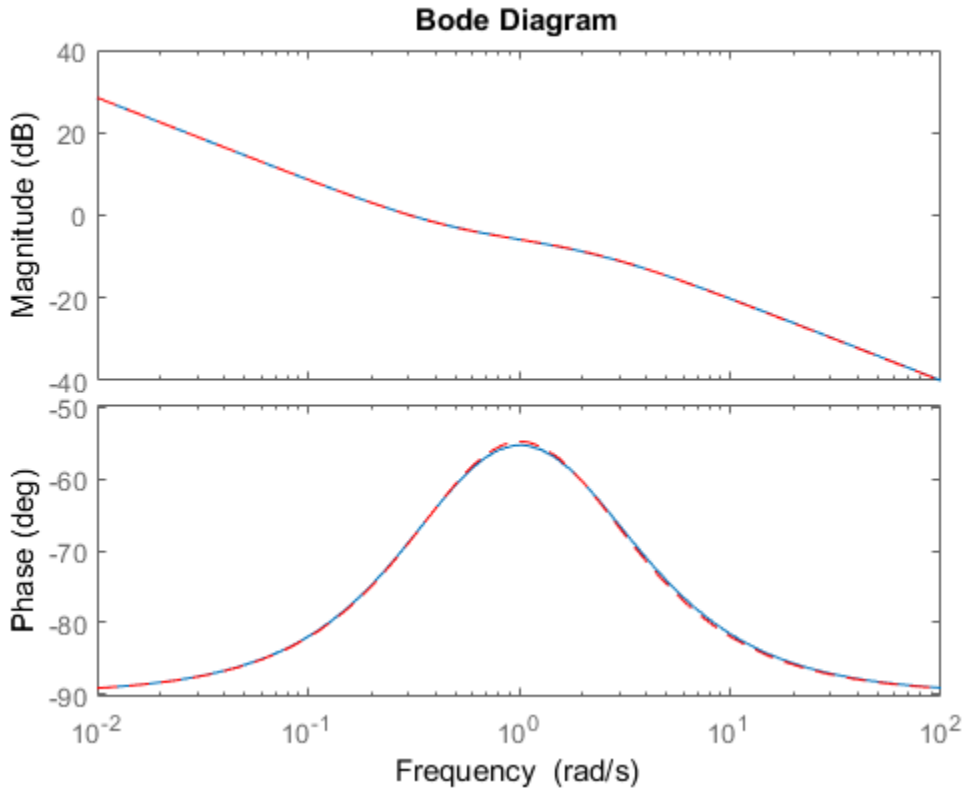
$$G(s) = \frac{(s + 0.5)(s + 1.1)(s + 2.9)}{(s + 10^{-6})(s + 1)(s + 2)(s + 3)}.$$

Use the `Offset` option to exclude the pole at  $s = 10^{-6}$  from the stable term of the stable/unstable decomposition.

```
sys = zpk([- .5 -1.1 -2.9],[-1e-6 -2 -1 -3],1);
% Create balredOptions
opt = balredOptions('Offset',.001,'StateElimMethod','Truncate');
% Compute second-order approximation
rsys = balred(sys,2,opt);
```

Compare the responses of the original and reduced-order models.

```
bodeplot(sys,rsys,'r--')
```

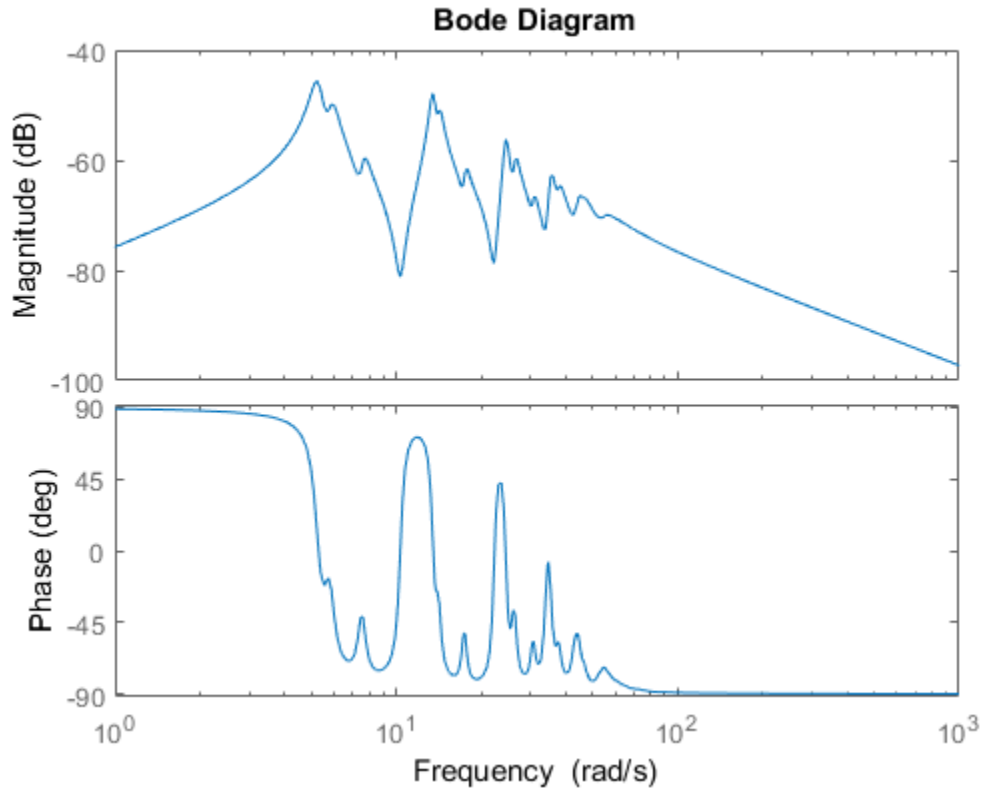


### Model Reduction in a Particular Frequency Band

Reduce a high-order model with a focus on the dynamics in a particular frequency range.

Load a model and examine its frequency response.

```
load(fullfile(matlabroot, 'examples', 'control', 'build.mat'), 'G')  
bodeplot(G)
```



$G$  is a 48th-order model with several large peak regions around 5.2 rad/s, 13.5 rad/s, and 24.5 rad/s, and smaller peaks scattered across many frequencies. Suppose that for your application you are only interested in the dynamics near the second large peak, between 10 rad/s and 22 rad/s. Focus the model reduction on the region of interest to obtain a good match with a low-order approximation. Use `balredOptions` to specify the frequency interval for `balred`.

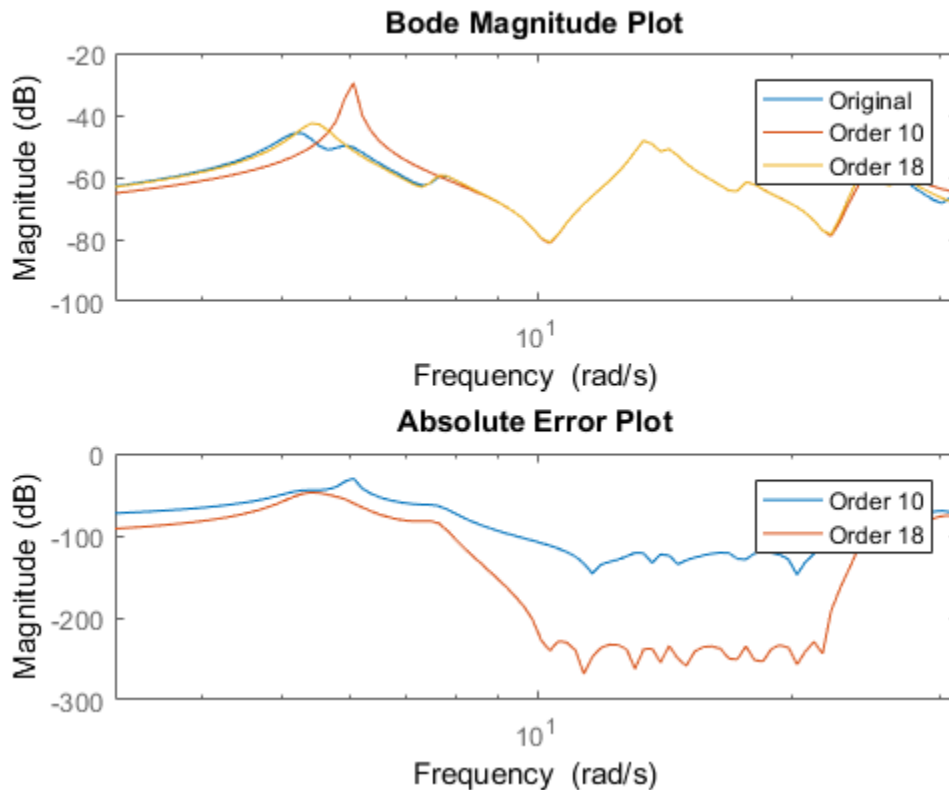
```
bopt = balredOptions('FreqIntervals',[10,22]);
GLim10 = balred(G,10,bopt);
GLim18 = balred(G,18,bopt);
```

Examine the frequency responses of the reduced-order models. Also, examine the difference between those responses and the original response (the absolute error).

```

subplot(2,1,1);
bodemag(G,GLim10,GLim18,logspace(0.5,1.5,100));
title('Bode Magnitude Plot')
legend('Original','Order 10','Order 18');
subplot(2,1,2);
bodemag(G-GLim10,G-GLim18,logspace(0.5,1.5,100));
title('Absolute Error Plot')
legend('Order 10','Order 18');

```



With the frequency-limited energy computation, even the 10th-order approximation is quite good in the region of interest.

- “Balanced Truncation Model Reduction”
- “Model Reduction Basics”



## References

- [1] Varga, A., "Balancing-Free Square-Root Algorithm for Computing Singular Perturbation Approximations," Proc. of 30th IEEE CDC, Brighton, UK (1991), pp. 1062-1065.

## See Also

balredOptions | hsvd | Model Reducer

**Introduced before R2006a**

## balredOptions

Create option set for model order reduction

### Syntax

```
opts = balredOptions  
opts = balredOptions('OptionName', OptionValue)
```

### Description

`opts = balredOptions` returns the default option set for the `balred` command.

`opts = balredOptions('OptionName', OptionValue)` accepts one or more comma-separated name/value pairs. Specify *OptionName* inside single quotes.

### Input Arguments

#### Name-Value Pair Arguments

##### 'FreqIntervals'

Frequency intervals for computing frequency-limited Hankel singular values, specified as a matrix with two columns. Each row specifies a frequency interval [*fmin* *fmax*], where *fmin* and *fmax* are nonnegative frequencies, expressed in the frequency unit of the model. When identifying low-energy states to truncate, the software computes state contributions to system behavior in these frequency ranges only. For example:

- To restrict the computation to the range between 3 rad/s and 15 rad/s, assuming the frequency unit of the model is rad/s, set `FreqIntervals` to `[3 15]`.
- To restrict the computation to two frequency intervals, 3-15 rad/s and 40-60 rad/s, use `[3 15; 40 60]`.
- To specify all frequencies below a cutoff frequency `fcut`, use `[0 fcut]`.

- To specify all frequencies above the cutoff, use `[fcut Inf]` in continuous time, or `[fcut pi/Ts]` in discrete time, where `Ts` is the sample time of the model.

The default value, `[]`, imposes no frequency limitation and is equivalent to `[0 Inf]` in continuous time or `[0 pi/Ts]` in discrete time. However, if you specify a `TimeIntervals` value other than `[]`, then this limit overrides `FreqIntervals = []`. If you specify both a `TimeIntervals` value and a `FreqIntervals` value, then the computation uses the union of these intervals.

If the frequency intervals exclude 0, then `balred` does not attempt to match the DC gain of the original and reduced models, even if `StateElimMethod = 'MatchDC'`. To force a DC match with frequency intervals that otherwise exclude 0, include an interval `[0 fLo]`, where `fLo` is a frequency that is small compared to the frequency ranges of interest.

If both the frequency and time intervals do include DC, you can still set `StateElimMethod = 'Truncate'` to improve the match at other frequencies and times.

**Default:** `[]`

### 'TimeIntervals'

Time intervals for computing time-limited Hankel singular values, specified as a matrix with two columns. Each row specifies a time interval `[tmin tmax]`, where `tmin` and `tmax` are nonnegative times, expressed in the time unit of the model. When identifying low-energy states to truncate, the software computes state contributions to the system's impulse response in these time intervals only. For example:

- To restrict the computation to the range between 3 s and 15 s, assuming the time unit of the model is seconds, set `TimeIntervals` to `[3 15]`.
- To restrict the computation to two time intervals, 3-15 s and 40-60 s, use `[3 15; 40 60]`.
- To specify all times from zero up to a cutoff time `tcut`, use `[0 tcut]`. To specify all times after the cutoff, use `[tcut Inf]`.

The default value, `[]`, imposes no time limitation and is equivalent to `[0 Inf]`. However, if you specify a `FreqIntervals` value other than `[]`, then this limit overrides `Timeintervals = []`. If you specify both a `TimeIntervals` value and a `FreqIntervals` value, then the computation uses the union of these intervals.

If the time intervals exclude `Inf`, then `balred` does not attempt to match the DC gain of the original and reduced models, even if `StateElimMethod = 'MatchDC'`. To force a DC match with time intervals that otherwise exclude `Inf`, include an interval `[ tHi Inf ]`, where `tHi` is a time that is long compared to the time intervals of interest.

If both the frequency and time intervals do include DC, you can still set `StateElimMethod = 'Truncate'` to improve the match at other frequencies and times.

**Default:** `[]`

**'StateElimMethod'**

State elimination method. Specifies how to eliminate the weakly coupled states (states with smallest Hankel singular values). Specified as one of the following values:

- |                         |   |
|-------------------------|---|
| <code>'MatchDC'</code>  | Discards the specified states and alters the remaining states to preserve the DC gain.  |
| <code>'Truncate'</code> | Discards the specified states without altering the remaining states. This method tends to product a better approximation in the frequency domain, but the DC gains are not guaranteed to match. |

**Default:** `'MatchDC'`

**'AbsTol, RelTol'**

Absolute and relative error tolerance for stable/unstable decomposition. Positive scalar values. For an input model  $G$  with unstable poles, `balred` first extracts the stable dynamics by computing the stable/unstable decomposition  $G \rightarrow GS + GU$ . The `AbsTol` and `RelTol` tolerances control the accuracy of this decomposition by ensuring that the frequency responses of  $G$  and  $GS + GU$  differ by no more than `AbsTol + RelTol*abs(G)`. Increasing these tolerances helps separate nearby stable and unstable modes at the expense of accuracy. See `stabsep` for more information.

**Default:** `AbsTol = 0; RelTol = 1e-8`

**'Offset'**

Offset for the stable/unstable boundary. Positive scalar value. In the stable/unstable decomposition, the stable term includes only poles satisfying

- $\text{Re}(s) < -\text{Offset} * \max(1, |\text{Im}(s)|)$  (Continuous time)
- $|z| < 1 - \text{Offset}$  (Discrete time)

Increase the value of `Offset` to treat poles close to the stability boundary as unstable.

**Default:** 1e-8

For additional information on the options and how to use them, see the `balred` reference page.

## Examples

### Reduced-Order Approximation with Offset Option

Compute a reduced-order approximation of the system given by:

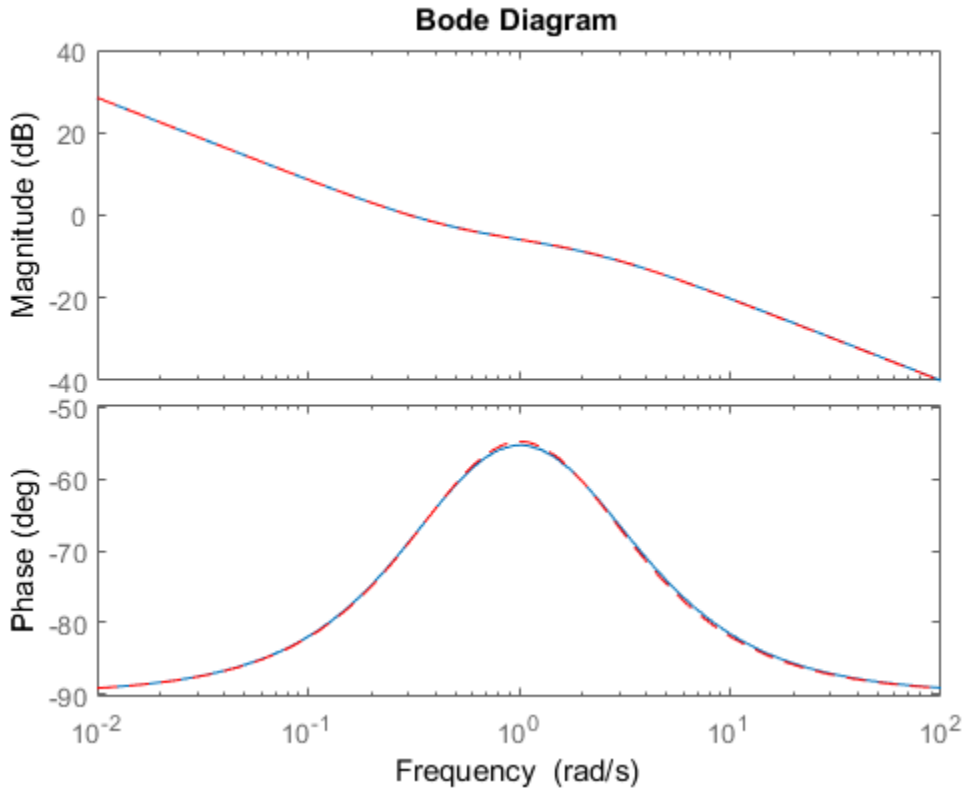
$$G(s) = \frac{(s + 0.5)(s + 1.1)(s + 2.9)}{(s + 10^{-6})(s + 1)(s + 2)(s + 3)}.$$

Use the `Offset` option to exclude the pole at  $s = 10^{-6}$  from the stable term of the stable/unstable decomposition.

```
sys = zpk([- .5 -1.1 -2.9],[-1e-6 -2 -1 -3],1);
% Create balredOptions
opt = balredOptions('Offset',.001,'StateElimMethod','Truncate');
% Compute second-order approximation
rsys = balred(sys,2,opt);
```

Compare the responses of the original and reduced-order models.

```
bodeplot(sys,rsys,'r--')
```

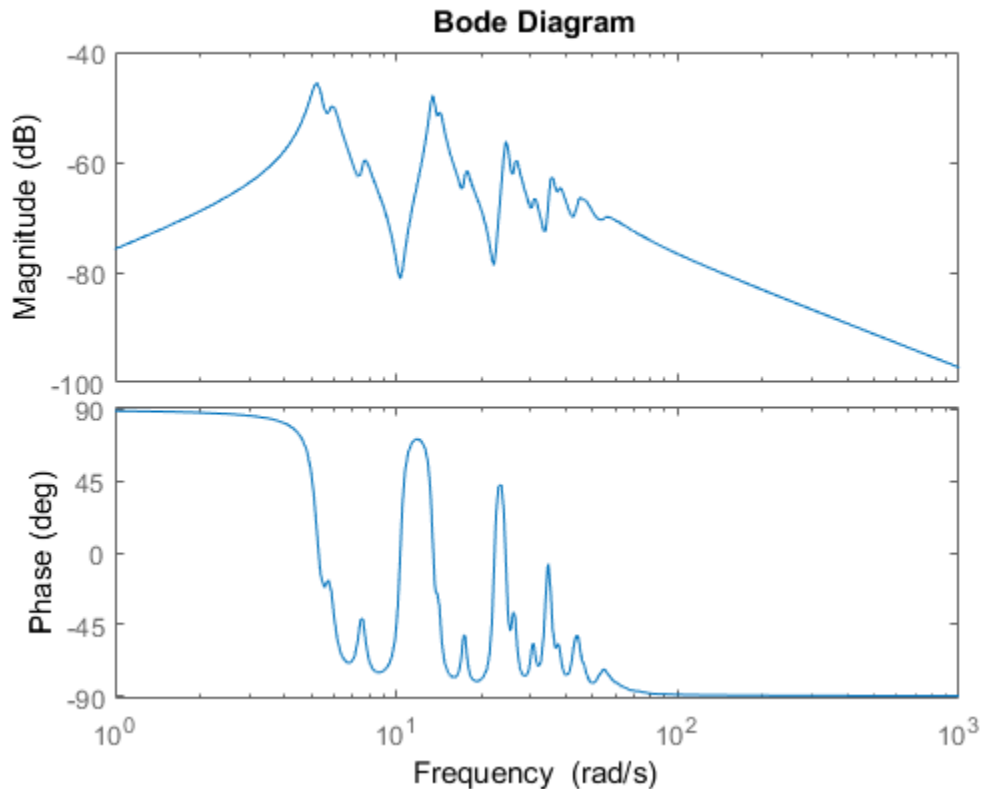


### Model Reduction in a Particular Frequency Band

Reduce a high-order model with a focus on the dynamics in a particular frequency range.

Load a model and examine its frequency response.

```
load(fullfile(matlabroot, 'examples', 'control', 'build.mat'), 'G')  
bodeplot(G)
```



$G$  is a 48th-order model with several large peak regions around 5.2 rad/s, 13.5 rad/s, and 24.5 rad/s, and smaller peaks scattered across many frequencies. Suppose that for your application you are only interested in the dynamics near the second large peak, between 10 rad/s and 22 rad/s. Focus the model reduction on the region of interest to obtain a good match with a low-order approximation. Use `balredOptions` to specify the frequency interval for `balred`.

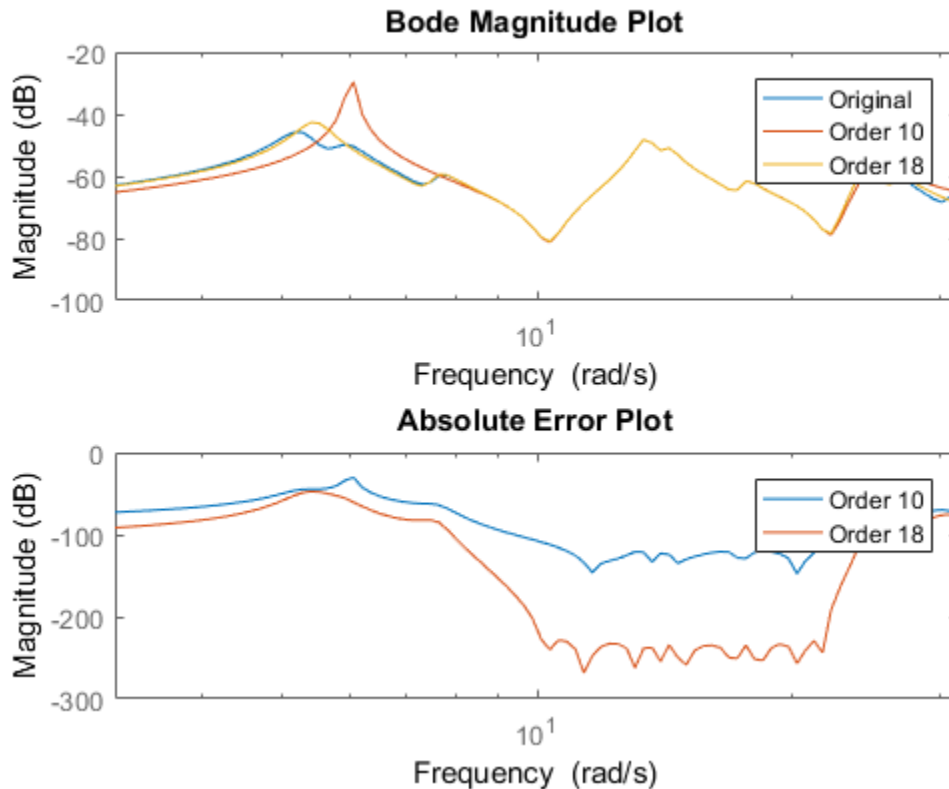
```
bopt = balredOptions('FreqIntervals',[10,22]);
GLim10 = balred(G,10,bopt);
GLim18 = balred(G,18,bopt);
```

Examine the frequency responses of the reduced-order models. Also, examine the difference between those responses and the original response (the absolute error).

```

subplot(2,1,1);
bodemag(G,GLim10,GLim18,logspace(0.5,1.5,100));
title('Bode Magnitude Plot')
legend('Original','Order 10','Order 18');
subplot(2,1,2);
bodemag(G-GLim10,G-GLim18,logspace(0.5,1.5,100));
title('Absolute Error Plot')
legend('Order 10','Order 18');

```



With the frequency-limited energy computation, even the 10th-order approximation is quite good in the region of interest.

- “Balanced Truncation Model Reduction”



## **See Also**

gramOptions | hsvdOptions | balred | stabsep

**Introduced in R2010a**

## bandwidth

Frequency response bandwidth

### Syntax

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

### Description

`fb = bandwidth(sys)` computes the bandwidth `fb` of the SISO dynamic system 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 `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

For FRD models, `bandwidth` uses the first frequency point to approximate the DC gain.

`fb = bandwidth(sys,dbdrop)` 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 models, `bandwidth` returns an array of the same size such that

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

### See Also

`issiso` | `dcgain`

Introduced before R2006a

# bdschur

Block-diagonal Schur factorization

## Syntax

```
[T,B,BLKS] = bdschur(A,CONDMAX)
[T,B] = bdschur(A,[],BLKS)
```

## Description

`[T,B,BLKS] = bdschur(A,CONDMAX)` computes a transformation matrix  $T$  such that  $B = T \setminus A * T$  is block diagonal and each diagonal block is a quasi upper-triangular Schur matrix.

`[T,B] = bdschur(A,[],BLKS)` pre-specifies the desired block sizes. The input matrix  $A$  should already be in Schur form when you use this syntax.

## Input Arguments

- **A**: Matrix for block-diagonal Schur factorization.
- **CONDMAX**: Specifies an upper bound on the condition number of  $T$ . By default,  $\text{CONDMAX} = 1/\text{sqrt}(\text{eps})$ . Use **CONDMAX** to control the tradeoff between block size and conditioning of  $T$  with respect to inversion. When **CONDMAX** is a larger value, the blocks are smaller and  $T$  becomes more ill-conditioned.

## Output Arguments

- **T**: Transformation matrix.
- **B**: Matrix  $B = T \setminus A * T$ .
- **BLKS**: Vector of block sizes.

## See Also

ordschur | schur

**Introduced in R2008a**

# blkdiag

Block-diagonal concatenation of models

## Syntax

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

## Description

`sys = blkdiag(sys1,sys2,...,sysN)` produces the aggregate system

$$\begin{bmatrix} \text{sys1} & 0 & \dots & 0 \\ 0 & \text{sys2} & \cdot & \vdots \\ \vdots & \cdot & \cdot & 0 \\ 0 & \dots & 0 & \text{sysN} \end{bmatrix}$$

`blkdiag` is equivalent to `append`.

## Examples

The commands

```
sys1 = tf(1,[1 0]);
sys2 = ss(1,2,3,4);
sys = blkdiag(sys1,10,sys2)
```

produce the state-space model

a =

```
      x1  x2
x1    0   0
x2    0   1
```

b =

```
      u1  u2  u3
```

```
x1  1  0  0
x2  0  0  2
```

c =

```
      x1  x2
y1   1  0
y2   0  0
y3   0  3
```

d =

```
      u1  u2  u3
y1   0  0  0
y2   0 10  0
y3   0  0  4
```

Continuous-time model.

### See Also

`append` | `series` | `parallel` | `feedback`

**Introduced in R2009a**

# bode

Bode plot of frequency response, magnitude and phase of frequency response

## Syntax

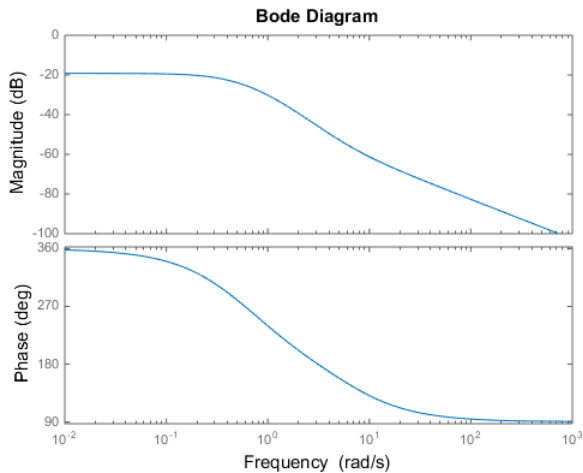
```
bode(sys)
bode(sys1,...,sysN)
bode(sys1,PlotStyle1,...,sysN,PlotStyleN)
bode(...,w)
[mag,phase] = bode(sys,w)
[mag,phase,wout] = bode(sys)
[mag,phase,wout,sdmag,sdphase] = bode(sys)
```

## Description

`bode(sys)` creates a Bode plot of the frequency response of a dynamic system model `sys`. The plot displays the magnitude (in dB) and phase (in degrees) of the system response as a function of frequency.

When `sys` is a multi-input, multi-output (MIMO) model, `bode` produces an array of Bode plots, each plot showing the frequency response of one I/O pair.

`bode` automatically determines the plot frequency range based on system dynamics.



`bode(sys1, ..., sysN)` plots the frequency response of multiple dynamic systems in a single figure. All systems must have the same number of inputs and outputs.

`bode(sys1, PlotStyle1, ..., sysN, PlotStyleN)` specifies a color, linestyle, and marker for each system in the plot.

`bode(..., w)` plots system responses at frequencies determined by `w`.

- If `w` is a cell array `{wmin, wmax}`, `bode(sys, w)` plots the system response at frequency values in the range `{wmin, wmax}`.
- If `w` is a vector of frequencies, `bode(sys, w)` plots the system response at each of the frequencies specified in `w`.

`[mag, phase] = bode(sys, w)` returns magnitudes `mag` in absolute units and phase values `phase` in degrees. The response values in `mag` and `phase` correspond to the frequencies specified by `w` as follows:

- If `w` is a cell array `{wmin, wmax}`, `[mag, phase] = bode(sys, w)` returns the system response at frequency values in the range `{wmin, wmax}`.
- If `w` is a vector of frequencies, `[mag, phase] = bode(sys, w)` returns the system response at each of the frequencies specified in `w`.

`[mag, phase, wout] = bode(sys)` returns magnitudes, phase values, and frequency values `wout` corresponding to `bode(sys)`.



[mag, phase, wout, sdmag, sdphase] = bode(sys) additionally returns the estimated standard deviation of the magnitude and phase values when **sys** is an identified model and [] otherwise.

## Input Arguments

### **sys**

Dynamic system model, such as a Numeric LTI model, or an array of such models.

### **PlotStyle**

Line style, marker, and color of both the line and marker, specified as a vector of one, two, or three characters. The characters can appear in any order. For example, 'r:' specifies a red dotted line. For more information about configuring the **PlotStyle** argument, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

### **w**

Input frequency values, specified as a row vector or a two-element cell array.

Possible values of **w**:

- Two-element cell array {wmin, wmax}, where wmin is the minimum frequency value and wmax is the maximum frequency value.
- Row vector of frequency values.

For example, use `logspace` to generate a row vector with logarithmically-spaced frequency values.

Specify frequency values in radians per **TimeUnit**, where **TimeUnit** is the time units of the input dynamic system, specified in the **TimeUnit** property of **sys**.

## Output Arguments

### **mag**

Bode magnitude of the system response in absolute units, returned as a 3-D array with dimensions (number of outputs) × (number of inputs) × (number of frequency points).

- For a single-input, single-output (SISO) `sys`, `mag(1, 1, k)` gives the magnitude of the response at the `k`th frequency.
- For MIMO systems, `mag(i, j, k)` gives the magnitude of the response from the `j`th input to the `i`th output.

You can convert the magnitude from absolute units to decibels using:

```
magdb = 20*log10(mag)
```

### **phase**

Phase of the system response in degrees, returned as a 3-D array with dimensions are (number of outputs) × (number of inputs) × (number of frequency points).

- For SISO `sys`, `phase(1, 1, k)` gives the phase of the response at the `k`th frequency.
- For MIMO systems, `phase(i, j, k)` gives the phase of the response from the `j`th input to the `i`th output.

### **wout**

Response frequencies, returned as a row vector of frequency points. Frequency values are in radians per `TimeUnit`, where `TimeUnit` is the value of the `TimeUnit` property of `sys`.

### **sdmag**

Estimated standard deviation of the magnitude. `sdmag` has the same dimensions as `mag`.

If `sys` is not an identified LTI model, `sdmag` is `[]`.

### **sdphase**

Estimated standard deviation of the phase. `sdphase` has the same dimensions as `phase`.

If `sys` is not an identified LTI model, `sdphase` is `[]`.

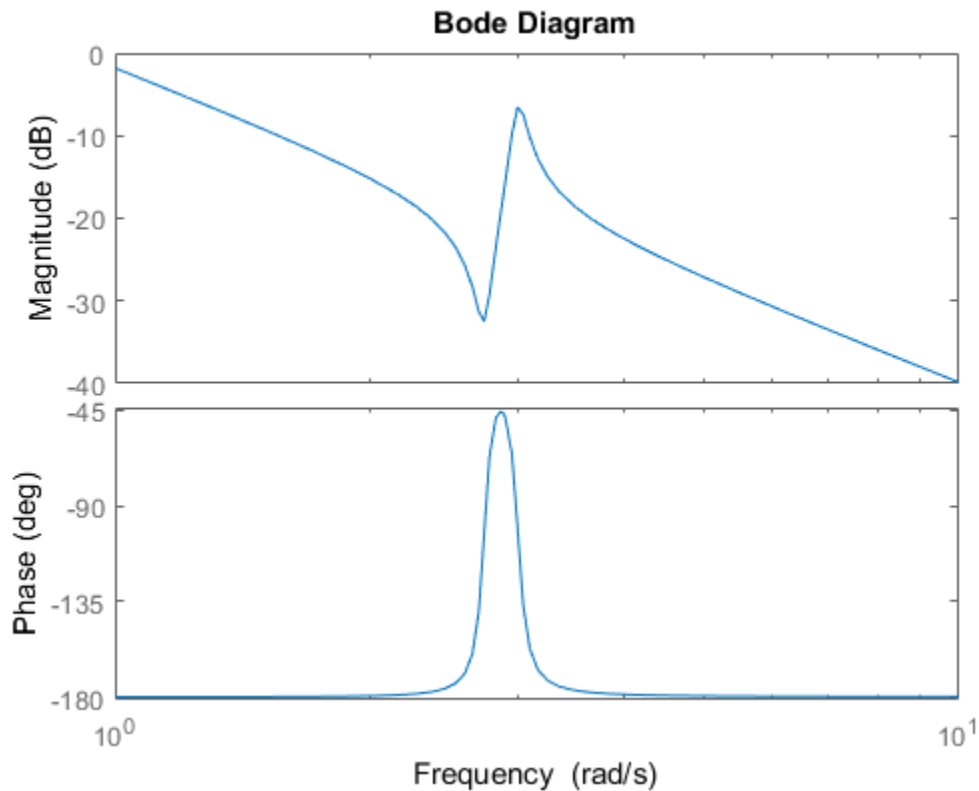
## **Examples**

### **Bode Plot of Dynamic System**

Create a Bode plot of the following continuous-time SISO dynamic system.

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

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

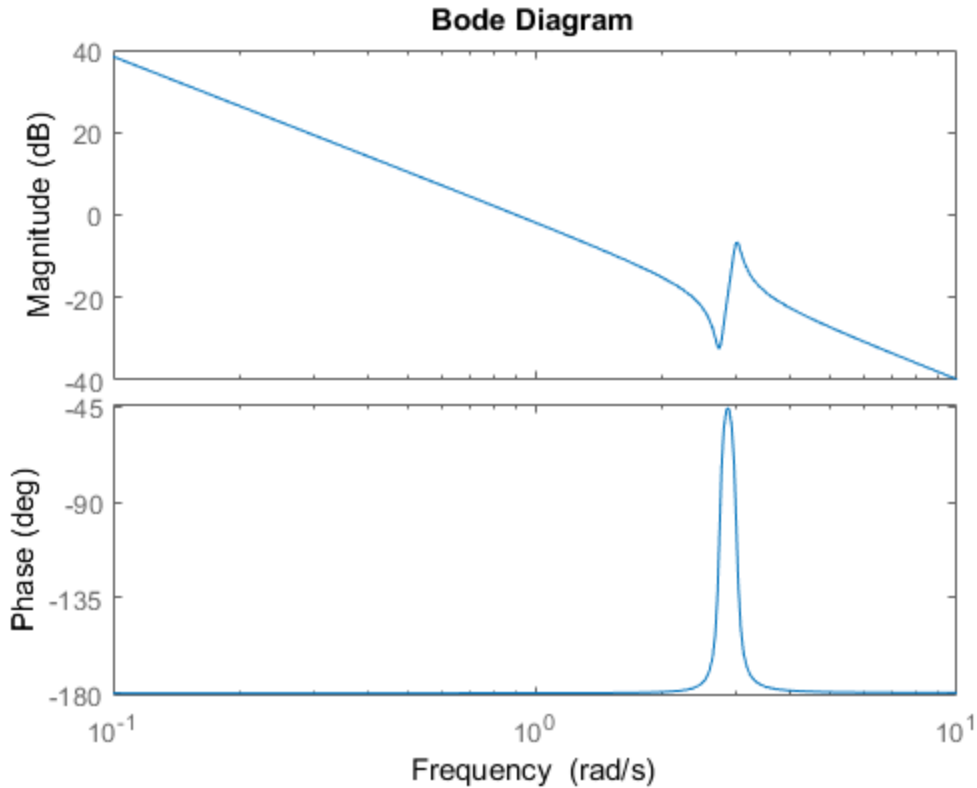


bode automatically selects the plot range based on the system dynamics.

## Bode Plot at Specified Frequencies

Create a Bode plot over a specified frequency range. Use this approach when you want to focus on the dynamics in a particular range of frequencies.

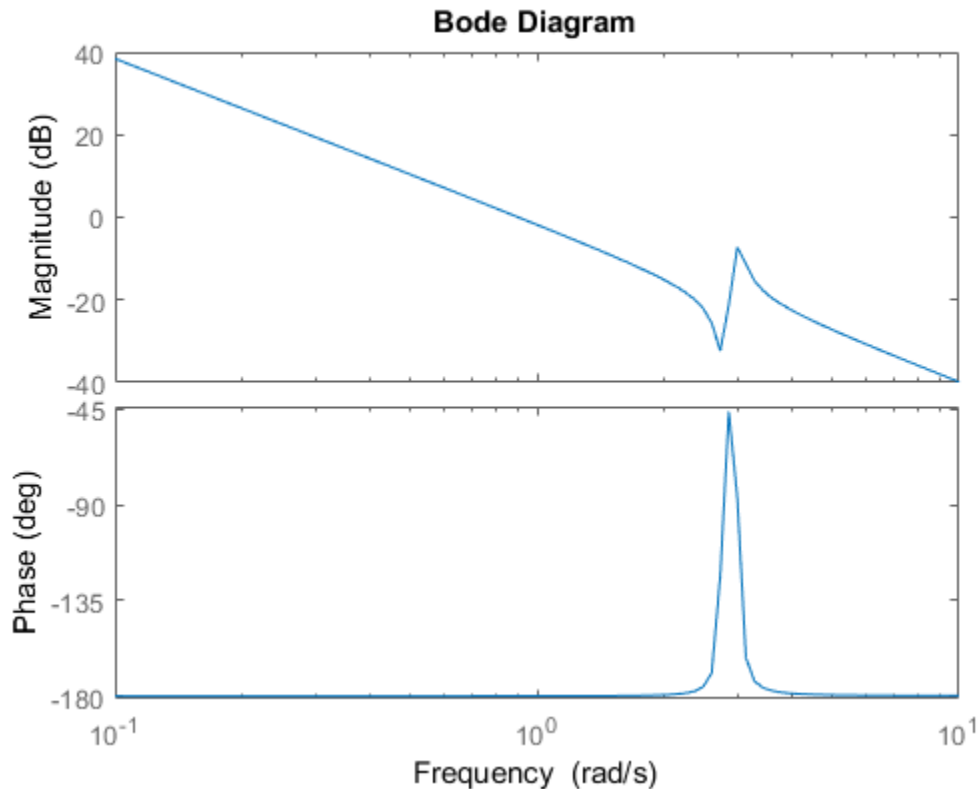
```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);  
bode(H,{0.1,10})
```



The cell array `{0.1,10}` specifies the minimum and maximum frequency values in the Bode plot. When you provide frequency bounds in this way, the software selects intermediate points for frequency response data.

Alternatively, specify a vector of frequency points to use for evaluating and plotting the frequency response.

```
w = logspace(-1,1,100);  
bode(H,w)
```



`logspace` defines a logarithmically spaced frequency vector in the range of 0.1-10 rad/s.

## Compare Bode Plots of Several Dynamic Systems

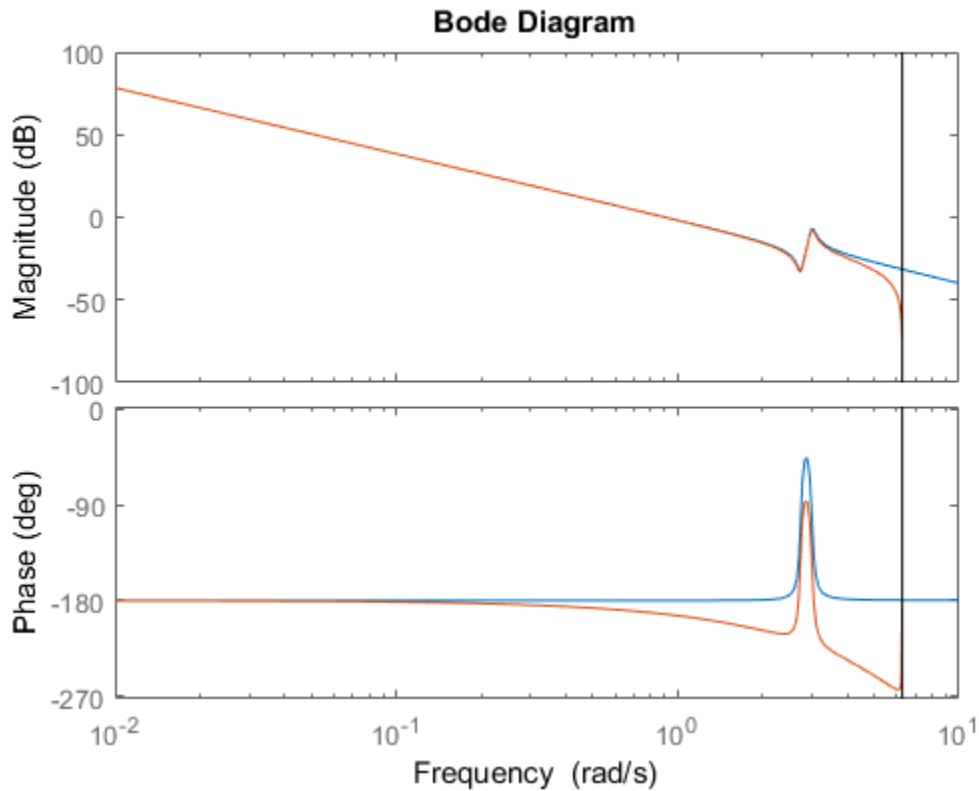
Compare the frequency response of a continuous-time system to an equivalent discretized system on the same Bode plot.

Create continuous-time and discrete-time dynamic systems.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);
Hd = c2d(H,0.5,'zoh');
```

Create a Bode plot that displays both systems.

`bode(H,Hd)`

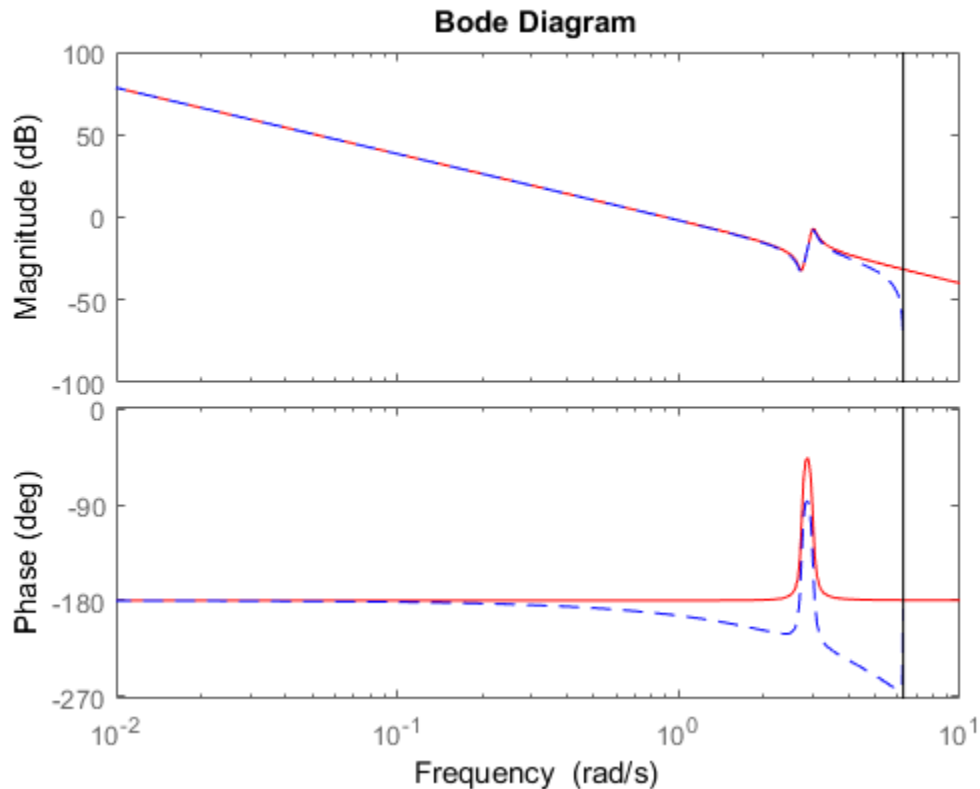


The Bode plot of a discrete-time system includes a vertical line marking the Nyquist frequency of the system.

### Bode Plot with Specified Line and Marker Attributes

Specify the color, linestyle, or marker for each system in a Bode plot using the `PlotStyle` input arguments.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);
Hd = c2d(H,0.5,'zoh');
bode(H,'r',Hd,'b--')
```



The first PlotStyle, 'r', specifies a solid red line for the response of H. The second PlotStyle, 'b--', specifies a dashed blue line for the response of Hd.

## Obtain Magnitude and Phase Data

Compute the magnitude and phase of the frequency response of a dynamic system.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);
[mag phase wout] = bode(H);
```

Because H is a SISO model, the first two dimensions of mag and phase are both 1. The third dimension is the number of frequencies in wout.

### Bode Plot of Identified Model

Compare the frequency response of a parametric model, identified from input/output data, to a nonparametric model identified using the same data.

Identify parametric and non-parametric models based on data.

```
load iddata2 z2;  
w = linspace(0,10*pi,128);  
sys_np = spa(z2,[],w);  
sys_p = tfest(z2,2);
```

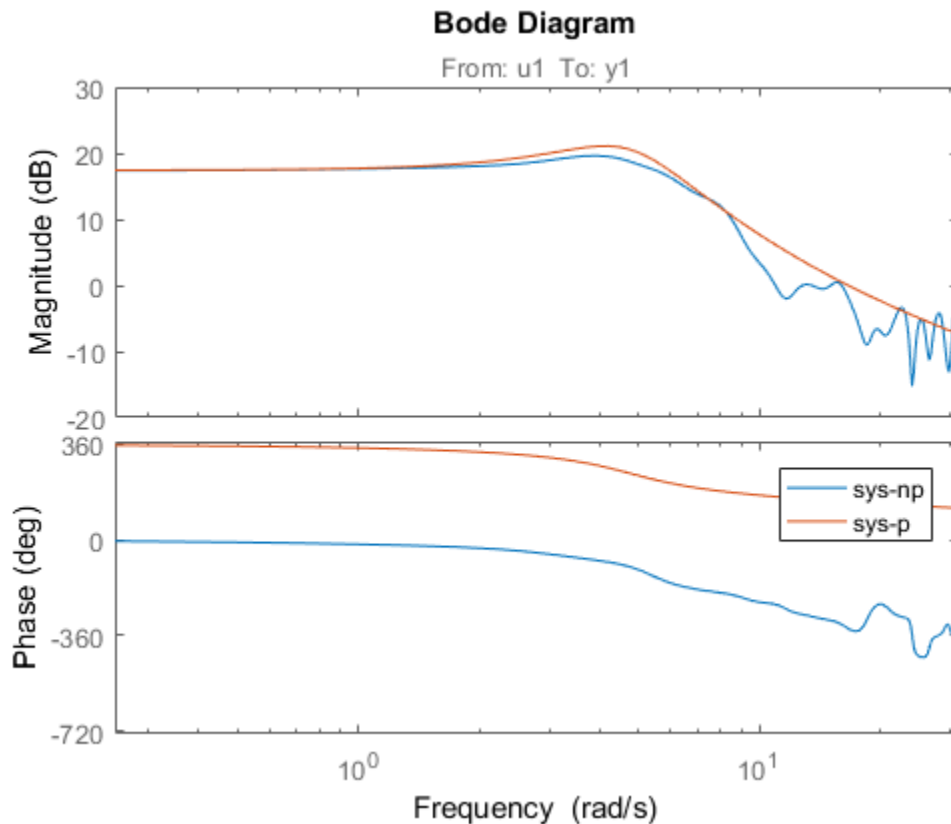
Using the `spa` and `tfest` commands requires System Identification Toolbox™ software.

`sys_np` is a non-parametric identified model. `sys_p` is a parametric identified model.

Create a Bode plot that includes both systems.

```
bode(sys_np,sys_p,w);  
legend('sys-np','sys-p')
```





## Obtain Magnitude and Phase Standard Deviation Data of Identified Model

Compute the standard deviation of the magnitude and phase of an identified model. Use this data to create a  $3\sigma$  plot of the response uncertainty.

Identify a transfer function model based on data. Obtain the standard deviation data for the magnitude and phase of the frequency response.

```
load iddata2 z2;
sys_p = tfest(z2,2);
w = linspace(0,10*pi,128);
```

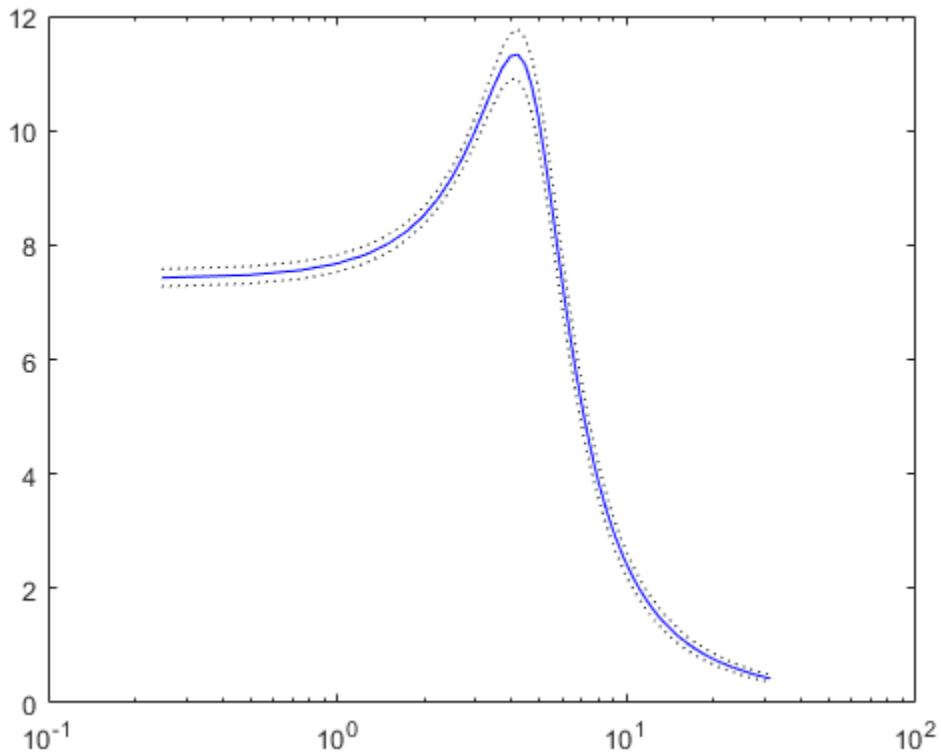
```
[mag,ph,w,sdmag,sdphase] = bode(sys_p,w);
```

Using the `tfest` command requires System Identification Toolbox™ software.

`sys_p` is an identified transfer function model. `sdmag` and `sdphase` contain the standard deviation data for the magnitude and phase of the frequency response, respectively.

Use the standard deviation data to create a  $3\sigma$  plot corresponding to the confidence region.

```
mag = squeeze(mag);  
sdmag = squeeze(sdmag);  
semilogx(w,mag,'b',w,mag+3*sdmag,'k:',w,mag-3*sdmag,'k:');
```



You can also right-click the Bode plot and select **Characteristics > Confidence Region** or use the `showConfidence` command to plot the confidence region.

## Alternatives

Use `bodeplot` when you need additional plot customization options.

## More About

### Algorithms

`bode` computes the frequency response using these steps:

- 1 Computes the zero-pole-gain (zpk) representation of the dynamic system.
- 2 Evaluates the gain and phase of the frequency response based on the zero, pole, and gain data for each input/output channel of the system.
  - a For continuous-time systems, `bode` evaluates the frequency response on the imaginary axis  $s = j\omega$  and considers only positive frequencies.
  - b For discrete-time systems, `bode` evaluates the frequency response on the unit circle. To facilitate interpretation, the command parameterizes the upper half of the unit circle as

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

where  $T_s$  is the sample time.  $\omega_N$  is the *Nyquist frequency*. The equivalent continuous-time frequency  $\omega$  is then used as the  $x$ -axis variable. Because

$H(e^{j\omega T_s})$  is periodic and has a period  $2\omega_N$ , `bode` plots the response only up to the Nyquist frequency  $\omega_N$ . If you do not specify a sample time, `bode` uses  $T_s = 1$ .

- “Dynamic System Models”

### See Also

`freqresp` | `nyquist` | `bodeplot` | `nichols`

**Introduced before R2006a**

# bodemag

Bode magnitude response of LTI models

## Syntax

```
bodemag(sys)
bodemag(sys,{wmin,wmax})
bodemag(sys,w)
bodemag(sys1,sys2,...,sysN,w)
```

## Description

`bodemag(sys)` plots the magnitude of the frequency response of the dynamic system 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 `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`).

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

`bodemag(sys1,sys2,...,sysN,w)` shows the frequency response magnitude of several 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. For example:

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

## See Also

`bode` | Linear System Analyzer

Introduced before R2006a

## bodeoptions

Create list of Bode plot options

### Syntax

```
P = bodeoptions
P = bodeoptions('cstprefs')
```

### Description

`P = bodeoptions` returns a default set of plot options for use with the `bodeplot`. You can use these options to customize the Bode plot appearance using the command line. This syntax is useful when you want to write a script to generate plots that look the same regardless of the preference settings of the MATLAB session in which you run the script.

`P = bodeoptions('cstprefs')` initializes the plot options with the options you selected in the Control System and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation. This syntax is useful when you want to change a few plot options but otherwise use your default preferences. A script that uses this syntax may generate results that look different when run in a session with different preferences.

The following table summarizes the Bode plot options.

Option	Description
Title, XLabel, YLabel	Label text and style, specified as a structure with the following fields: <ul style="list-style-type: none"> <li>• <b>String</b> — Label text, specified as a character vector, for example 'Amplitude'.</li> <li>• <b>FontSize</b> — <b>Default:</b> 8</li> <li>• <b>FontWeight</b> — <b>Default:</b> 'Normal'</li> <li>• <b>Font Angle</b> — <b>Default:</b> 'Normal'</li> <li>• <b>Color</b> — Vector of RGB values ranging from 0 to 1. <b>Default:</b> [0,0,0]</li> <li>• <b>Interpreter</b> — <b>Default:</b> 'tex'</li> </ul>

Option	Description
TickLabel	Tick label style, specified as a structure with the following fields: <ul style="list-style-type: none"> <li>• <b>FontSize</b> <b>Default:</b> 8</li> <li>• <b>FontWeight</b> — <b>Default:</b> 'Normal'</li> <li>• <b>Font Angle</b> — <b>Default:</b> 'Normal'</li> <li>• <b>Color</b> — Vector of RGB values ranging from 0 to 1. <b>Default:</b> [0,0,0]</li> </ul>
Grid	Show or hide the grid Specified as one of the following values: 'off'   'on' <b>Default:</b> 'off'
GridColor	Color of the grid lines Specified as one of the following: Vector of RGB values in the range [0,1]   character vector of color name   'none'. For example, for yellow color, specify as one of the following: [1 1 0], 'yellow', or 'y'. <b>Default:</b> [0.15,0.15,0.15]
XlimMode, YlimMode	Axis limit modes. <b>Default:</b> 'auto'
Xlim, Ylim	Axes limits, specified as an array of the form [min,max]
IOGrouping	Grouping of input-output pairs Specified as one of the following values: 'none'   'inputs'   'outputs'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels
ConfidenceRegionNumk	Number of standard deviations to use to plotting the response confidence region (identified models only).  <b>Default:</b> 1.

Option	Description
FreqUnits	Frequency units, specified as one of the following values: <ul style="list-style-type: none"> <li>• 'Hz'</li> <li>• 'rad/second'</li> <li>• 'rpm'</li> <li>• 'kHz'</li> <li>• 'MHz'</li> <li>• 'GHz'</li> <li>• 'rad/nanosecond'</li> <li>• 'rad/microsecond'</li> <li>• 'rad/millisecond'</li> <li>• 'rad/minute'</li> <li>• 'rad/hour'</li> <li>• 'rad/day'</li> <li>• 'rad/week'</li> <li>• 'rad/month'</li> <li>• 'rad/year'</li> <li>• 'cycles/nanosecond'</li> <li>• 'cycles/microsecond'</li> <li>• 'cycles/millisecond'</li> <li>• 'cycles/hour'</li> <li>• 'cycles/day'</li> <li>• 'cycles/week'</li> <li>• 'cycles/month'</li> <li>• 'cycles/year'</li> </ul>
FreqScale	Frequency scale Specified as one of the following values: 'linear'   'log' <b>Default:</b> 'log'



Option	Description
MagUnits	Magnitude units Specified as one of the following values: 'dB'   'abs' <b>Default:</b> 'dB'
MagScale	Magnitude scale Specified as one of the following values: 'linear'   'log' <b>Default:</b> 'linear'
MagVisible	Magnitude plot visibility Specified as one of the following values: 'on'   'off' <b>Default:</b> 'on'
MagLowerLimMode	Enables a lower magnitude limit Specified as one of the following values: 'auto'   'manual' <b>Default:</b> 'auto'
MagLowerLim	Specifies the lower magnitude limit
PhaseUnits	Phase units Specified as one of the following values: 'deg'   'rad' <b>Default:</b> 'deg'
PhaseVisible	Phase plot visibility Specified as one of the following values: 'on'   'off' <b>Default:</b> 'on'
PhaseWrapping	Enables phase wrapping Specified as one of the following values: 'on'   'off' When you set <b>PhaseWrapping</b> to 'on', the plot wraps accumulated phase at the value specified by the <b>PhaseWrappingBranch</b> property. <b>Default:</b> 'off'
PhaseWrappingBranch	Phase value at which the plot wraps accumulated phase when <b>PhaseWrapping</b> is set to 'on'. <b>Default:</b> -180 (phase wraps into the interval [-180°,180°))
PhaseMatching	Enables phase matching Specified as one of the following values: 'on'   'off' <b>Default:</b> 'off'
PhaseMatchingFreq	Frequency for matching phase
PhaseMatchingValue	The value to which phase responses are matched closely

## Examples

### Create Bode Plot with Custom Settings

Create a Bode plot that suppresses the phase plot and uses frequency units Hz instead of the default radians/second. Otherwise, the plot uses the settings that are saved in the toolbox preferences.

First, create an options set based on the toolbox preferences.

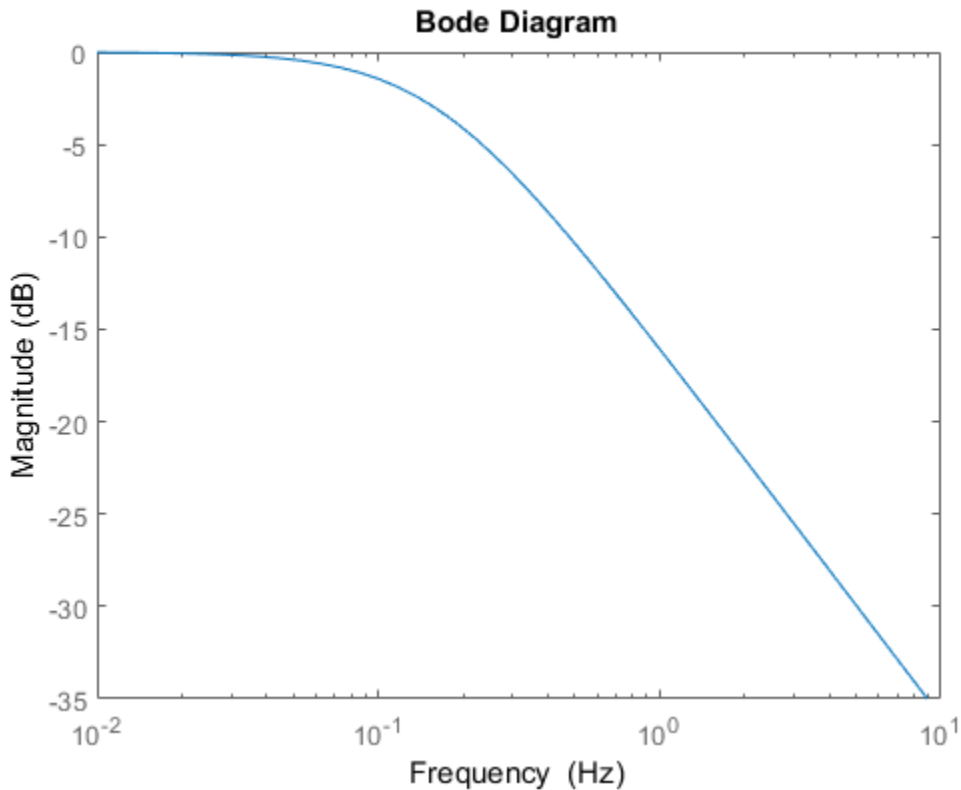
```
opts = bodeoptions('cstprefs');
```

Change properties of the options set.

```
opts.PhaseVisible = 'off';  
opts.FreqUnits = 'Hz';
```

Create a plot using the options.

```
h = bodeplot(tf(1,[1,1]),opts);
```



Depending on your own toolbox preferences, the plot you obtain might look different from this plot. Only the properties that you set explicitly, in this example `PhaseVisible` and `FreqUnits`, override the toolbox preferences.

### Custom Plot Settings Independent of Preferences

Create a Bode plot that uses 14-point red text for the title. This plot should look the same, regardless of the preferences of the MATLAB session in which it is generated.

First, create a default options set.

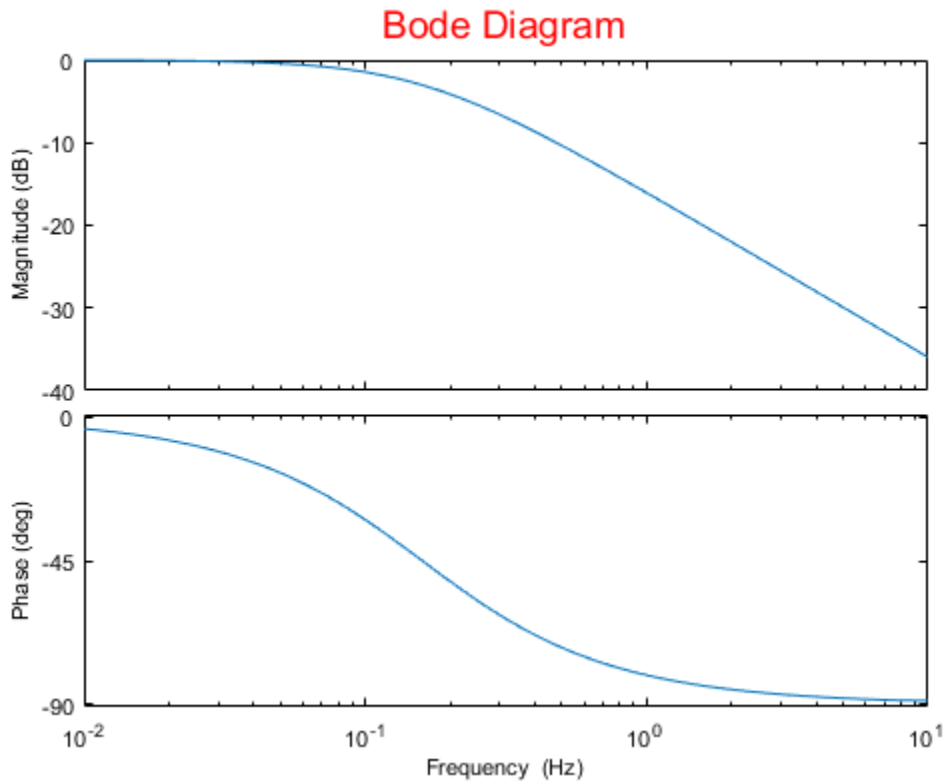
```
opts = bodeoptions;
```

Change properties of the options set.

```
opts.Title.FontSize = 14;  
opts.Title.Color = [1 0 0];  
opts.FreqUnits = 'Hz';
```

Create a plot using the options.

```
h = bodeplot(tf(1,[1,1]),opts);
```



Because `opts` begins with a fixed set of options, the plot result is independent of the toolbox preferences of the MATLAB session.

### See Also

`bodeplot` | `getoptions` | `setoptions` | `bode`

**Introduced in R2008a**

## bodeplot

Plot Bode frequency response with additional plot customization options

### Syntax

```
h = bodeplot(sys)
bodeplot(sys)
bodeplot(sys1,sys2,...)
bodeplot(AX,...)
bodeplot(..., plotoptions)
bodeplot(sys,w)
```

### Description

`h = bodeplot(sys)` plot the Bode magnitude and phase of the dynamic system model `sys` and returns the plot handle `h` to the plot. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands.

`bodeplot(sys)` draws the Bode plot of the model `sys`. The frequency range and number of points are chosen automatically.

`bodeplot(sys1,sys2,...)` graphs the Bode response of multiple models `sys1,sys2,...` on a single plot. You can specify a color, line style, and marker for each model, as in

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

`bodeplot(AX,...)` plots into the axes with handle `AX`.

`bodeplot(..., plotoptions)` plots the Bode response with the options specified in `plotoptions`. Type

```
help bodeoptions
```

for a list of available plot options. See “Match Phase at Specified Frequency” on page 2-79 for an example of phase matching using the `PhaseMatchingFreq` and `PhaseMatchingValue` options.

`bodeplot(sys,w)` draws the Bode plot for frequencies specified by `w`. When `w = {wmin,wmax}`, the Bode plot is drawn for frequencies between `wmin` and `wmax` (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`). When `w` is a user-supplied vector `w` of frequencies, in `rad/TimeUnit`, the Bode response is drawn for the specified frequencies.

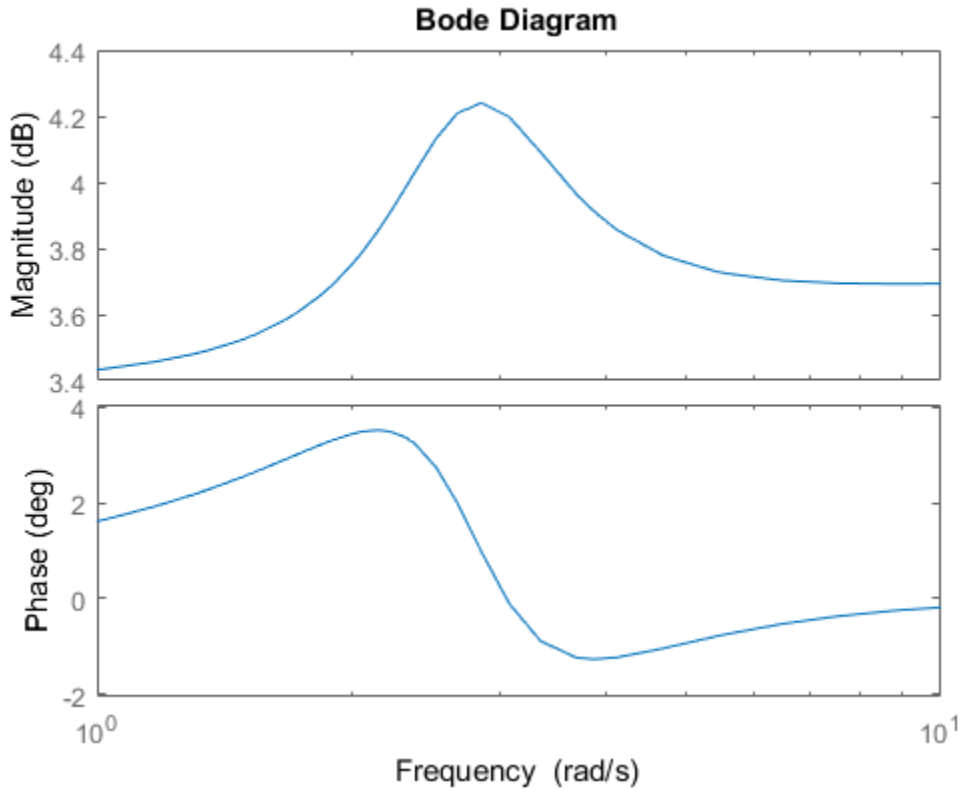
See `logspace` to generate logarithmically spaced frequency vectors.

## Examples

### Change Bode Plot Options with Plot Handle

Generate a Bode plot.

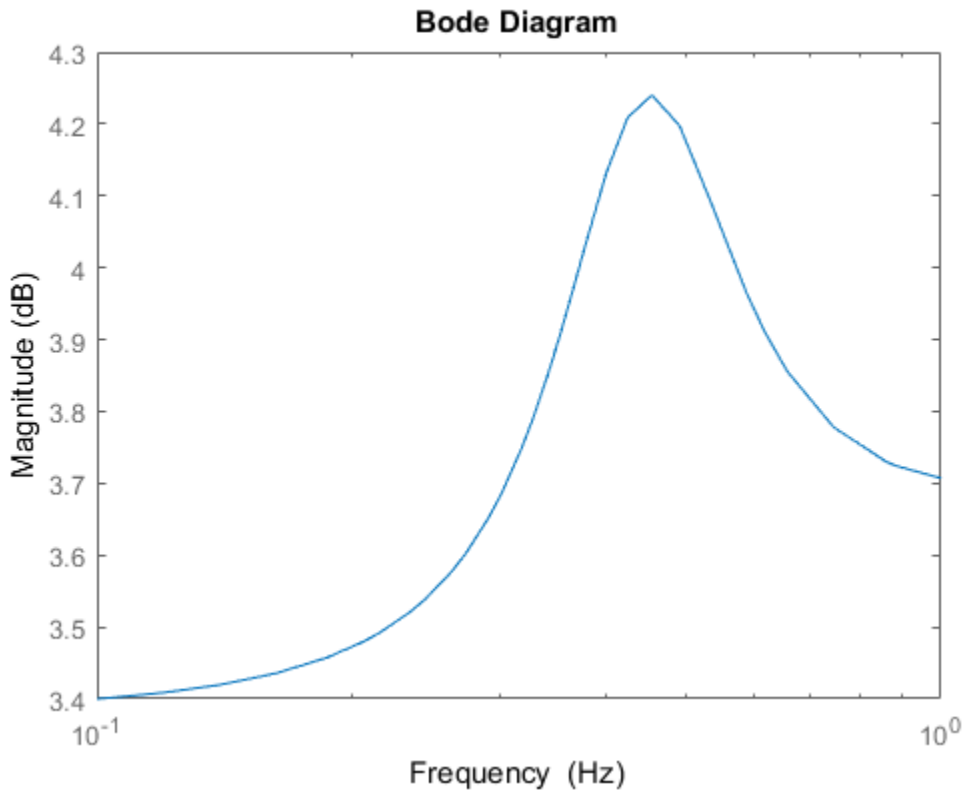
```
sys = rss(5);  
h = bodeplot(sys);
```



Change the units to Hz and suppress the phase plot. To do so, edit properties of the plot handle, `h`.

```
setoptions(h, 'FreqUnits', 'Hz', 'PhaseVisible', 'off');
```



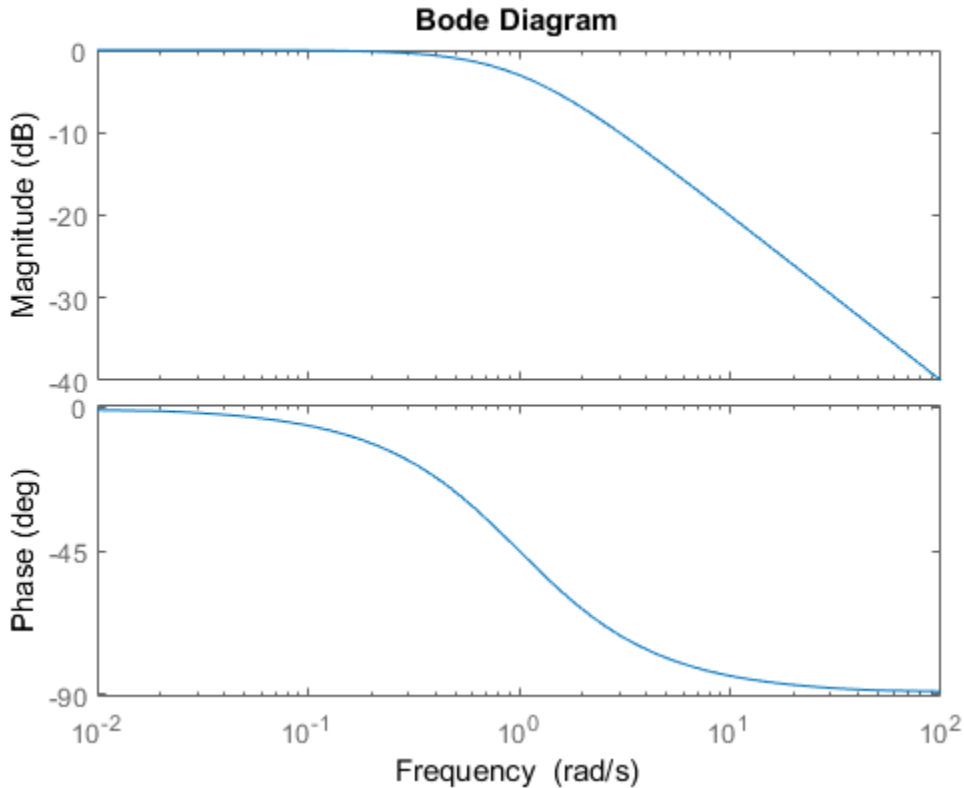


The plot automatically updates when you call `setoptions`.

### Match Phase at Specified Frequency

Create a Bode plot of a dynamic system.

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

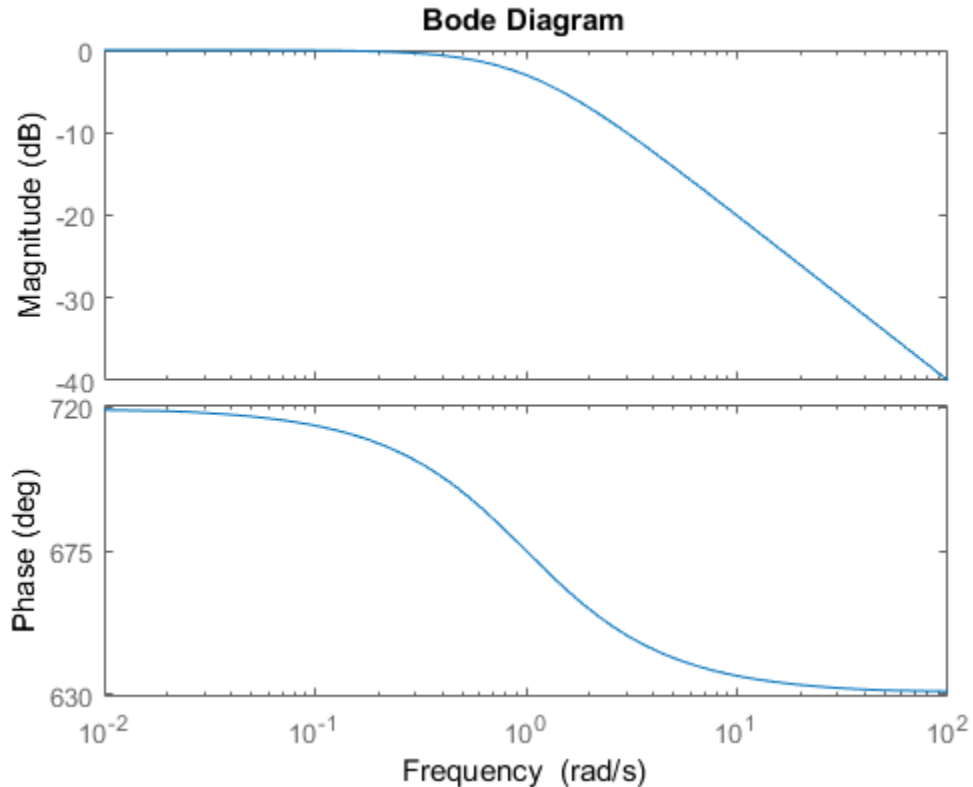


Fix the phase at 1 rad/s to 750 degrees. To do so, get the plot properties. Then alter the properties `PhaseMatchingFreq` and `PhaseMatchingValue` to match a phase to a specified frequency.

```
p = getoptions(h);  
p.PhaseMatching = 'on';  
p.PhaseMatchingFreq = 1;  
p.PhaseMatchingValue = 750;
```

Update the plot.

```
setoptions(h,p);
```



The first bode plot has a phase of -45 degrees at a frequency of 1 rad/s. Setting the phase matching options so that at 1 rad/s the phase is near 750 degrees yields the second Bode plot. Note that, however, the phase can only be  $-45 + N \cdot 360$ , where  $N$  is an integer, and so the plot is set to the nearest allowable phase, namely 675 degrees (or  $2 \cdot 360 - 45 = 675$ ).

### Display Confidence Regions of Identified Models

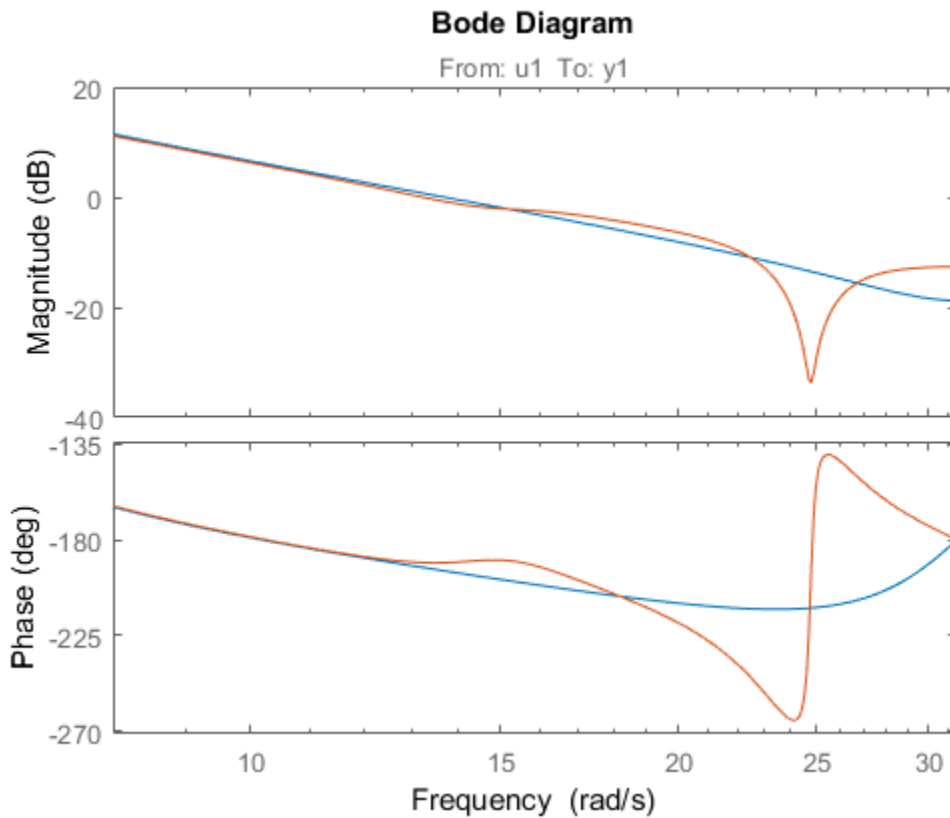
Compare the frequency responses of identified state-space models of order 2 and 6 along with their  $2\sigma$  confidence regions.

```
load iddata1
sys1 = n4sid(z1, 2);
```

```
sys2 = n4sid(z1, 6);
```

Both models produce about 70% fit to data. However, `sys2` shows higher uncertainty in its frequency response, especially close to Nyquist frequency as shown by the plot:

```
w = linspace(8,10*pi,256);  
h = bodeplot(sys1,sys2,w);  
setoptions(h, 'PhaseMatching', 'on', 'ConfidenceRegionNumberSD', 2);
```



Right-click the plot and select **Characteristics > Confidence Region** to turn on the confidence region characteristic. Alternatively, type `showConfidence(h)` to plot the confidence region.

### Frequency Response of Identified Parametric and Nonparametric Models

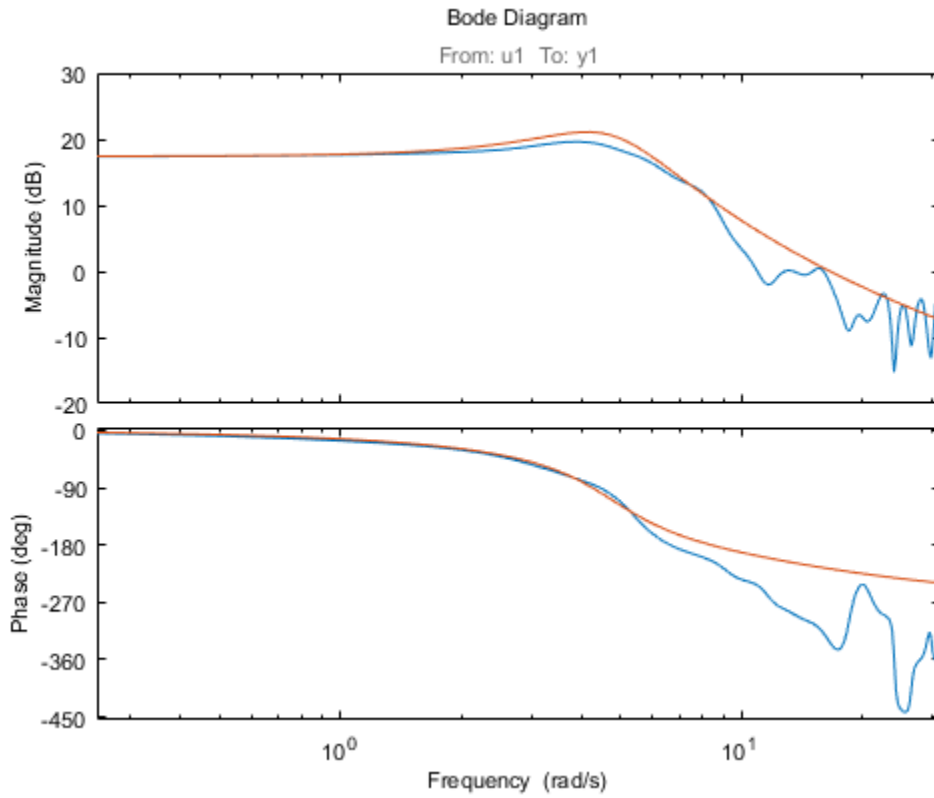
Compare the frequency response of a parametric model, identified from input/output data, to a nonparametric model identified using the same data. Identify parametric and non-parametric models based on data.

```
load iddata2 z2;  
w = linspace(0,10*pi,128);  
sys_np = spa(z2,[],w);  
sys_p = tfest(z2,2);
```

`spa` and `tfest` require System Identification Toolbox™ software. `sys_np` is a nonparametric identified model. `sys_p` is a parametric identified model.

Create a Bode plot that includes both systems.

```
opt = bodeoptions;  
opt.PhaseMatching = 'on';  
bodeplot(sys_np,sys_p,w,opt);
```



## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`bodeoptions` | `getoptions` | `setoptions` | `bode`

Introduced before R2006a

# c2d

Convert model from continuous to discrete time

## Syntax

```
sysd = c2d(sys,Ts)
sysd = c2d(sys,Ts,method)
sysd = c2d(sys,Ts,opts)
[sysd,G] = c2d(sys,Ts,method)
[sysd,G] = c2d(sys,Ts,opts)
```

## Description

`sysd = c2d(sys,Ts)` discretizes the continuous-time dynamic system model `sys` using zero-order hold on the inputs and a sample time of `Ts` seconds.

`sysd = c2d(sys,Ts,method)` discretizes `sys` using the specified discretization method `method`.

`sysd = c2d(sys,Ts,opts)` discretizes `sys` using the option set `opts`, specified using the `c2dOptions` command.

`[sysd,G] = c2d(sys,Ts,method)` returns a matrix, `G` that maps the continuous initial conditions  $x_0$  and  $u_0$  of the state-space model `sys` to the discrete-time initial state vector  $x[0]$ . `method` is optional. To specify additional discretization options, use `[sysd,G] = c2d(sys,Ts,opts)`.

## Input Arguments

### **sys**

Continuous-time dynamic system model (except frequency response data models). `sys` can represent a SISO or MIMO system, except that the 'matched' discretization method supports SISO systems only.

`sys` can have input/output or internal time delays; however, the `'matched'` and `'impulse'` methods do not support state-space models with internal time delays.

The following identified linear systems cannot be discretized directly:

- `idgrey` models whose `FunctionType` is `'c'`. Convert to `idss` model first.
- `idproc` models. Convert to `idtf` or `idpoly` model first.

For the syntax `[sysd,G] = c2d(sys,Ts,opts)`, `sys` must be a state-space model.

### **Ts**

Sample time.

### **method**

Discretization method, specified as one of the following values:

- `'zoh'` — Zero-order hold (default). Assumes the control inputs are piecewise constant over the sample time `Ts`.
- `'foh'` — Triangle approximation (modified first-order hold). Assumes the control inputs are piecewise linear over the sample time `Ts`.
- `'impulse'` — Impulse invariant discretization.
- `'tustin'` — Bilinear (Tustin) method.
- `'matched'` — Zero-pole matching method.

For more information about discretization methods, see “Continuous-Discrete Conversion Methods”.

### **opts**

Discretization options. Create `opts` using `c2dOptions`.

## **Output Arguments**

### **sysd**

Discrete-time model of the same type as the input system `sys`.



When `sys` is an identified (IDLTI) model, `sysd`:

- Includes both measured and noise components of `sys`. The innovations variance  $\lambda$  of the continuous-time identified model `sys`, stored in its `NoiseVariance` property, is interpreted as the intensity of the spectral density of the noise spectrum. The noise variance in `sysd` is thus  $\lambda/T_s$ .
- Does not include the estimated parameter covariance of `sys`. If you want to translate the covariance while discretizing the model, use `translatecov`.

## G

Matrix relating continuous-time initial conditions  $x_0$  and  $u_0$  of the state-space model `sys` to the discrete-time initial state vector  $x[0]$ , as follows:

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

For state-space models with time delays, `c2d` pads the matrix `G` with zeroes to account for additional states introduced by discretizing those delays. See “Continuous-Discrete Conversion Methods” for a discussion of modeling time delays in discretized systems.

## Examples

### Discretize a Transfer Function

Discretize the following continuous-time transfer function:

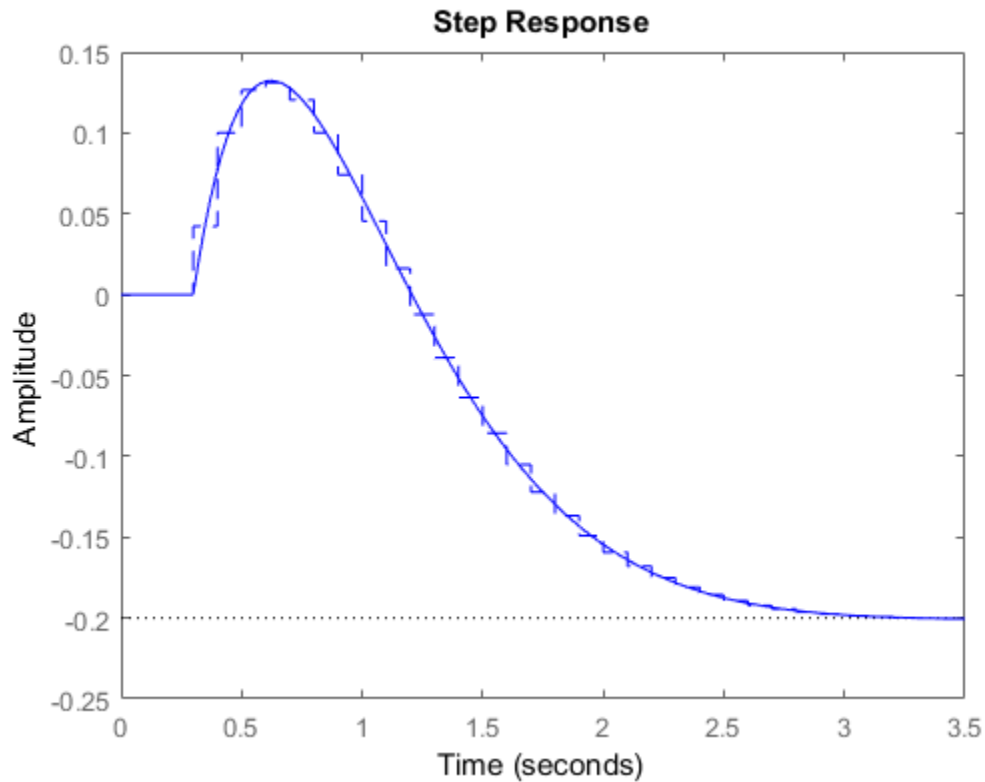
$$H(s) = e^{-0.3s} \frac{s - 1}{s^2 + 4s + 5}$$

This system has an input delay of 0.3 s. Discretize the system using the triangle (first-order-hold) approximation with sample time  $T_s = 0.1$  s.

```
H = tf([1 -1],[1 4 5], 'InputDelay', 0.3);
Hd = c2d(H,0.1, 'foh');
```

Compare the step responses of the continuous-time and discretized systems.

```
step(H, '-', Hd, '-.-')
```



### Discretize Model with Fractional Delay Absorbed into Coefficients

Discretize the following delayed transfer function using zero-order hold on the input, and a 10-Hz sampling rate.

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

```
h = tf(10,[1 3 10], 'IODelay',0.25);
```

```
hd = c2d(h,0.1)
```

```
hd =
```

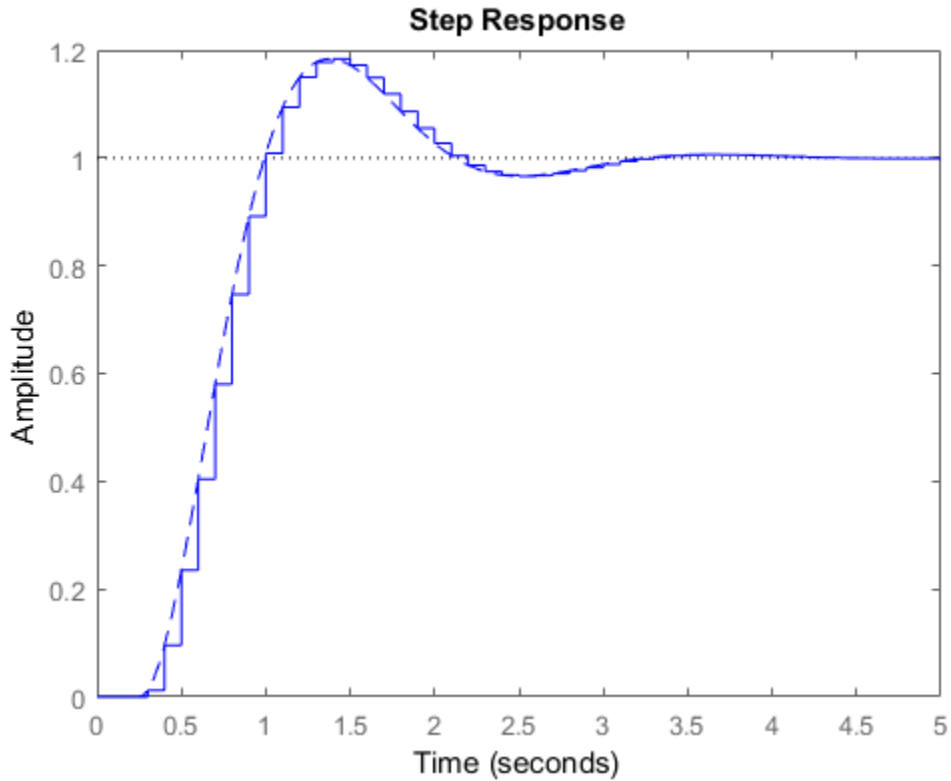
$$z^{-3} * \frac{0.01187 z^2 + 0.06408 z + 0.009721}{z^2 - 1.655 z + 0.7408}$$

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

In this example, the discretized model `hd` has a delay of three sampling periods. The discretization algorithm absorbs the residual half-period delay into the coefficients of `hd`.

Compare the step responses of the continuous-time and discretized models.

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



### Discretize Model With Approximated Fractional Delay

Create a continuous-time state-space model with two states and an input delay.

```
sys = ss(tf([1,2],[1,4,2]));
sys.InputDelay = 2.7
```

```
sys =
```

```
A =
      x1  x2
x1  -4  -2
x2   1   0
```

```

B =
      u1
x1    2
x2    0

C =
      x1    x2
y1  0.5    1

D =
      u1
y1    0

```

Input delays (seconds): 2.7

Continuous-time state-space model.

Discretize the model using the Tustin discretization method and a Thiran filter to model fractional delays. The sample time  $T_s = 1$  second.

```

opt = c2dOptions('Method','tustin','FractDelayApproxOrder',3);
sysd1 = c2d(sys,1,opt)

```

```

sysd1 =

A =
      x1      x2      x3      x4      x5
x1  -0.4286  -0.5714  -0.00265  0.06954  2.286
x2   0.2857   0.7143  -0.001325  0.03477  1.143
x3    0         0      -0.2432   0.1449  -0.1153
x4    0         0         0.25     0         0
x5    0         0         0         0.125     0

B =
      u1
x1  0.002058
x2  0.001029
x3    8
x4    0
x5    0

C =

```

```
          x1      x2      x3      x4      x5
y1      0.2857   0.7143  -0.001325  0.03477  1.143

D =
          u1
y1  0.001029
```

```
Sample time: 1 seconds
Discrete-time state-space model.
```

The discretized model now contains three additional states **x3**, **x4**, and **x5** corresponding to a third-order Thiran filter. Since the time delay divided by the sample time is 2.7, the third-order Thiran filter (`'FractDelayApproxOrder' = 3`) can approximate the entire time delay.

### Discretize Identified Model

Estimate a continuous-time transfer function, and discretize it.

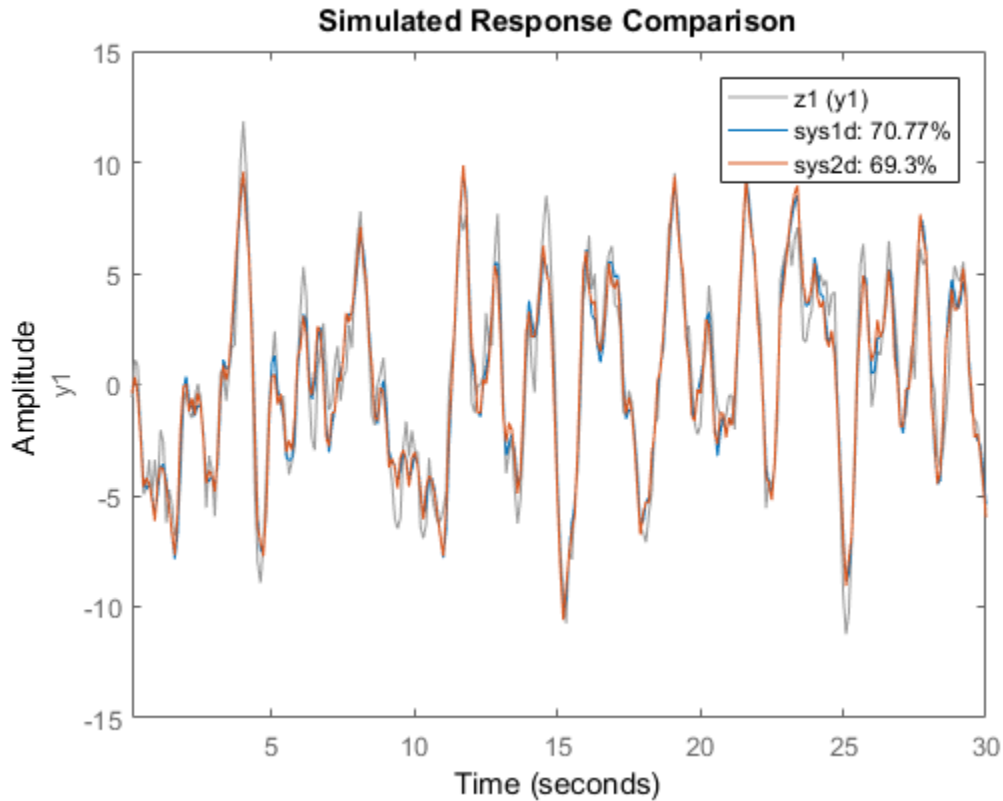
```
load iddata1
sys1c = tfest(z1,2);
sys1d = c2d(sys1c,0.1,'zoh');
```

Estimate a second order discrete-time transfer function.

```
sys2d = tfest(z1,2,'Ts',0.1);
```

Compare the response of the discretized continuous-time transfer function model, **sys1d**, and the directly estimated discrete-time model, **sys2d**.

```
compare(z1,sys1d,sys2d)
```



The two systems are almost identical.

### Build Predictor Model

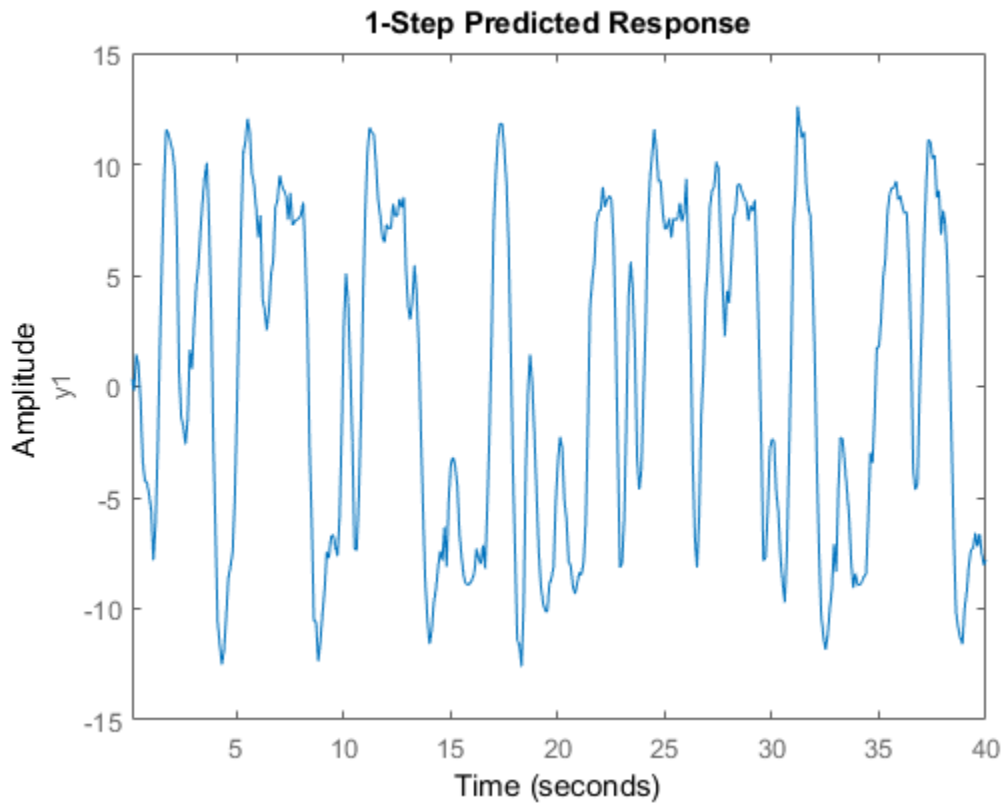
Discretize an identified state-space model to build a one-step ahead predictor of its response.

Create a continuous-time identified state-space model using estimation data.

```
load iddata2
sysc = sstest(z2,4);
```

Predict the 1-step ahead predicted response of `sysc`.

```
predict(sysc,z2)
```



Discretize the model.

```
sysd = c2d(sysc,0.1,'zoh');
```

Build a predictor model from the discretized model, `sysd`.

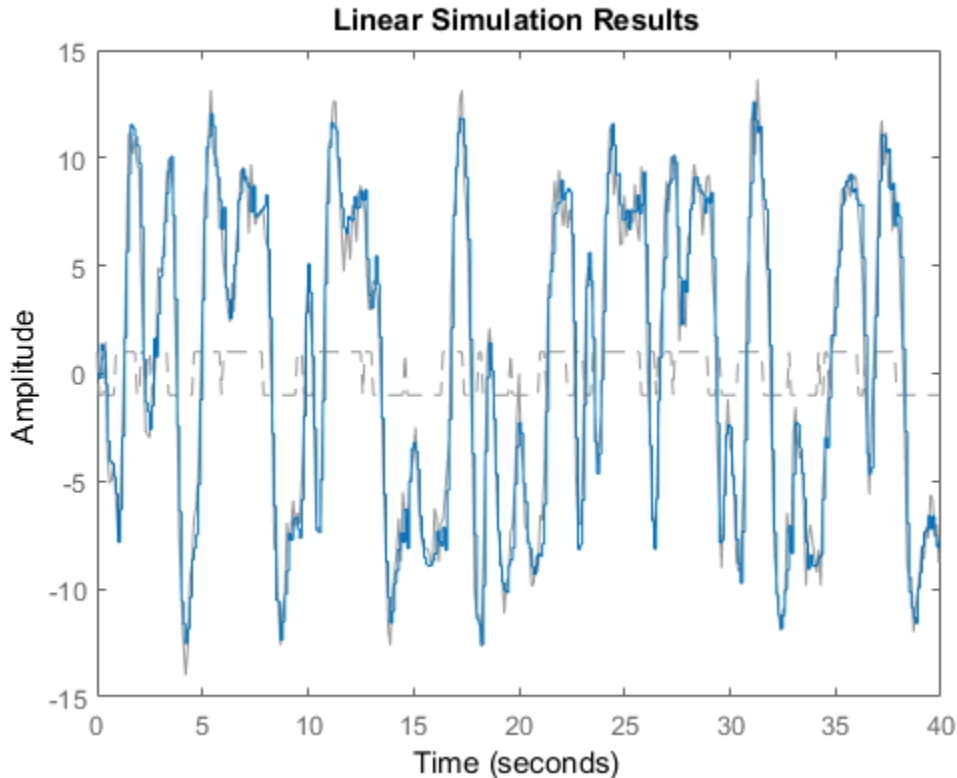
```
[A,B,C,D,K] = idssdata(sysd);
Predictor = ss(A-K*C,[K B-K*D],C,[0 D],0.1);
```

`Predictor` is a two-input model which uses the measured output and input signals (`[z1.y z1.u]`) to compute the 1-step predicted response of `sysc`.

Simulate the predictor model to get the same response as the `predict` command.

```
lsim(Predictor,[z2.y,z2.u])
```





The simulation of the predictor model gives the same response as `predict(sysc, z2)`.

## More About

### Tips

- Use the syntax `sysd = c2d(sys, Ts, method)` to discretize `sys` using the default options for `method`. To specify additional discretization options, use the syntax `sysd = c2d(sys, Ts, opts)`.
- To specify the `tustin` method with frequency prewarping (formerly known as the 'prewarp' method), use the `PrewarpFrequency` option of `c2dOptions`.

### Algorithms

For information about the algorithms for each `c2d` conversion method, see “Continuous-Discrete Conversion Methods”.

- “Dynamic System Models”
- “Discretize a Compensator”
- “Continuous-Discrete Conversion Methods”

### See Also

`d2c` | `d2d` | `c2dOptions` | `thiran` | `translatecov`

**Introduced before R2006a**

# c2dOptions

Create option set for continuous- to discrete-time conversions

## Syntax

```
opts = c2dOptions  
opts = c2dOptions('OptionName', OptionValue)
```

## Description

`opts = c2dOptions` returns the default options for `c2d`.

`opts = c2dOptions('OptionName', OptionValue)` accepts one or more comma-separated name/value pairs that specify options for the `c2d` command. Specify *OptionName* inside single quotes.

## Input Arguments

### Name-Value Pair Arguments

#### 'Method'

Discretization method, specified as one of the following values:

'zoh'	Zero-order hold, where <code>c2d</code> assumes the control inputs are piecewise constant over the sample time <code>Ts</code> .
'foh'	Triangle approximation (modified first-order hold), where <code>c2d</code> assumes the control inputs are piecewise linear over the sample time <code>Ts</code> . (See [1], p. 228.)
'impulse'	Impulse-invariant discretization.
'tustin'	Bilinear (Tustin) approximation. By default, <code>c2d</code> discretizes with no prewarp and rounds any fractional time delays to the nearest multiple of the sample time. To include prewarp, use the <code>PrewarpFrequency</code> option. To approximate fractional time delays, use the <code>FractDelayApproxOrder</code> option.

'matched' Zero-pole matching method. (See [1], p. 224.) By default, `c2d` rounds any fractional time delays to the nearest multiple of the sample time. To approximate fractional time delays, use the `FractDelayApproxOrder` option.

**Default:** 'zoh'

### 'PrewarpFrequency'

Prewarp frequency for 'tustin' method, specified in `rad/TimeUnit`, where `TimeUnit` is the time units, specified in the `TimeUnit` property, of the discretized system. Takes positive scalar values. A value of 0 corresponds to the standard 'tustin' method without prewarp.

**Default:** 0

### 'FractDelayApproxOrder'

Maximum order of the Thiran filter used to approximate fractional delays in the 'tustin' and 'matched' methods. Takes integer values. A value of 0 means that `c2d` rounds fractional delays to the nearest integer multiple of the sample time.

**Default:** 0

## Examples

### Discretize Two Models Using Tustin Discretization Method

Generate two random continuous-time state-space models.

```
sys1 = rss(3,2,2);  
sys2 = rss(4,4,1);
```

Create an option set for `c2d` to use the Tustin discretization method and 3.4 rad/s prewarp frequency.

```
opt = c2dOptions('Method','tustin','PrewarpFrequency',3.4);
```

Discretize the models, `sys1` and `sys2`, using the same option set, but different sample times.

```
dsys1 = c2d(sys1,0.1,opt);
```

```
dsys2 = c2d(sys2,0.2,opt);
```

## References

- [1] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

## See Also

c2d

**Introduced in R2010a**

# canon

State-space canonical realization

## Syntax

```
csys = canon(sys,type)
[csys,T]= canon(sys,type)
csys = canon(sys,'modal',condt)
```

## Description

`csys = canon(sys,type)` transforms the linear model `sys` into a canonical state-space model `csys`. The argument `type` specifies whether `csys` is in modal or companion form.

`[csys,T]= canon(sys,type)` also returns the state-coordinate transformation `T` that relates the states of the state-space model `sys` to the states of `csys`.

`csys = canon(sys,'modal',condt)` specifies an upper bound `condt` on the condition number of the block-diagonalizing transformation.

## Input Arguments

### **sys**

Any linear dynamic system model, except for `frd` models.

### **type**

Canonical form of `csys`, specified as one of the following values:

- 'modal' — convert `sys` to modal form.
- 'companion' — convert `sys` to companion form.

**condt**

Positive scalar value specifying an upper bound on the condition number of the block-diagonalizing transformation that converts `sys` to `csys`. This argument is available only when `type` is 'modal'.

Increase `condt` to reduce the size of the eigenvalue clusters in the  $A$  matrix of `csys`. Setting `condt = Inf` diagonalizes  $A$ .

**Default:** 1e8

## Output Arguments

**csys**

State-space (ss) model. `csys` is a state-space realization of `sys` in the canonical form specified by `type`.

**T**

Matrix specifying the transformation between the state vector  $x$  of the state-space model `sys` and the state vector  $x_c$  of `csys`:

$$x_c = Tx$$

This argument is available only when `sys` is state-space model.

## Examples

### Convert System To Modal Canonical Form

Consider a system with doubled poles and clusters of close poles:

$$G(s) = 100 \frac{(s - 1)(s + 1)}{s(s + 10)(s + 10.0001)(s - (1 + i))^2(s - (1 - i))^2}$$

Create a linear model of this system, and convert it to modal canonical form.

```
G = zpk([1 -1],[0 -10 -10.0001 1+1i 1-1i 1+1i 1-1i],100);
Gc = canon(G, 'modal');
```

The system,  $G$ , has a pair of nearby poles at  $s = -10$  and  $s = -10.0001$ .  $G$  also has two complex poles of multiplicity 2 at  $s = 1 + i$  and  $s = 1 - i$ . As a result, the modal form has a block of size 2 for the two poles near  $s = -10$ , and a block of size 4 for the complex eigenvalues.

Gc.A

ans =

```

0      0      0      0      0      0      0
0      1.0000  1.0000      0      0      0      0
0     -1.0000  1.0000  2.0548      0      0      0
0      0      0      1.0000  1.0000      0      0
0      0      0     -1.0000  1.0000      0      0
0      0      0      0      0     -10.0000  8.0573
0      0      0      0      0      0     -10.0001
```

Separate the two poles near  $s = -10$  by increasing the value of the condition number of the block-diagonalizing transformation. The default value of the condition number is  $1e8$ .

```
Gc2 = canon(G, 'modal', 1e10);
Gc2.A
```

ans =

```

0      0      0      0      0      0      0
0      1.0000  1.0000      0      0      0      0
0     -1.0000  1.0000  2.0548      0      0      0
0      0      0      1.0000  1.0000      0      0
0      0      0     -1.0000  1.0000      0      0
0      0      0      0      0     -10.0000  8.0573
0      0      0      0      0      0     -10.0001
```

The A matrix of  $Gc2$  includes separate diagonal elements for the poles near  $s = -10$ . The cost of increasing the condition number of  $A$  is that the B matrix includes some large values.



```
format shortE
Gc2.B
```

```
ans =
    3.2000e-01
   -6.5691e-03
    5.4046e-02
   -1.9502e-01
    1.0637e+00
    3.2533e+05
    3.2533e+05
```

### Convert System to Companion Canonical Form

Estimate a state-space model that is freely parameterized.

```
load icEngine.mat
z = iddata(y,u,0.04);
FreeModel = n4sid(z,4, 'InputDelay',2);
```

Convert the estimated model to companion canonical form.

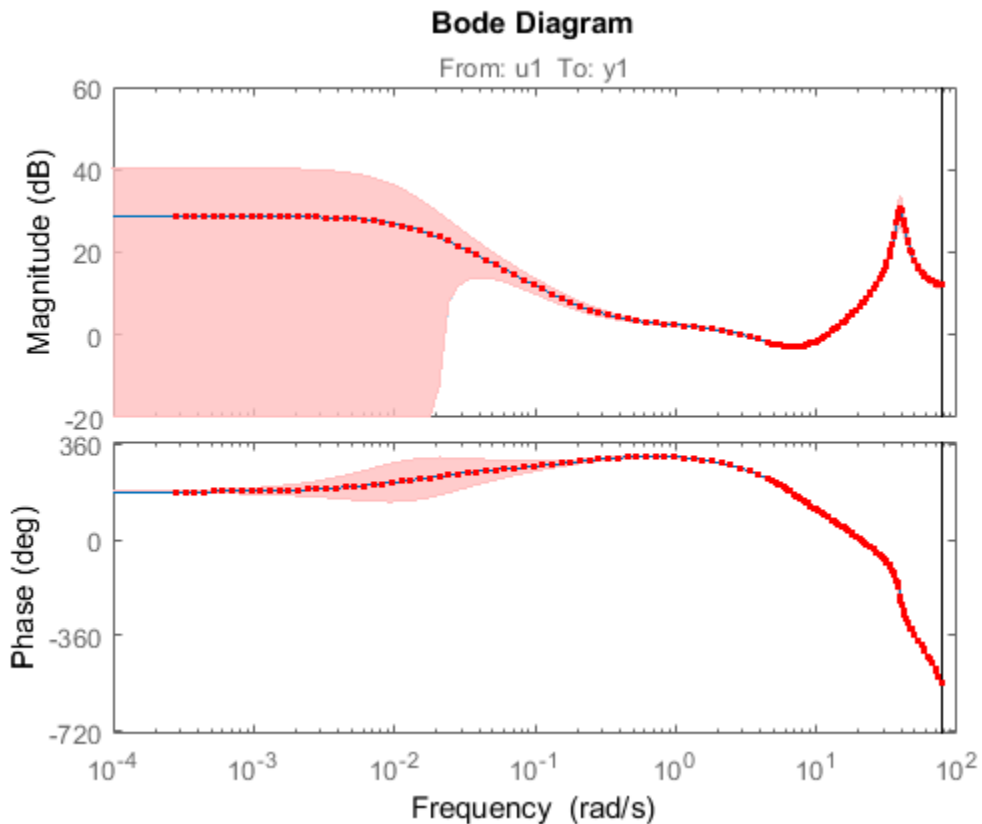
```
CanonicalModel = canon(FreeModel, 'companion');
```

Obtain the covariance of the resulting form by running a zero-iteration update to model parameters.

```
opt = ssestOptions;
opt.SearchOption.MaxIter = 0;
CanonicalModel = ssest(z,CanonicalModel,opt);
```

Compare frequency response confidence bounds of `FreeModel` to `CanonicalModel`.

```
h = bodeplot(FreeModel,CanonicalModel, 'r. ');
showConfidence(h)
```



The frequency response confidence bounds are identical.

## More About

### Modal Form

In modal form,  $A$  is a block-diagonal matrix. The block size is typically 1-by-1 for real eigenvalues and 2-by-2 for complex eigenvalues. However, if there are repeated eigenvalues or clusters of nearby eigenvalues, the block size can be larger.

For example, 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

In the companion realization, the characteristic polynomial of the system appears explicitly in the rightmost column of the  $A$  matrix. For a system with characteristic polynomial

$$p(s) = s^n + \alpha_1 s^{n-1} + \dots + \alpha_{n-1} s + \alpha_n$$

the corresponding companion  $A$  matrix is

$$A = \begin{bmatrix} 0 & 0 & \dots & \dots & 0 & -\alpha_n \\ 1 & 0 & 0 & \dots & 0 & -\alpha_{n-1} \\ 0 & 1 & 0 & \dots & \vdots & \vdots \\ \vdots & 0 & \dots & \dots & \vdots & \vdots \\ 0 & \dots & \dots & 1 & 0 & -\alpha_2 \\ 0 & \dots & \dots & 0 & 1 & -\alpha_1 \end{bmatrix}$$

The companion transformation requires that the system be controllable from the first input. The companion form is poorly conditioned for most state-space computations; avoid using it when possible.

### Algorithms

The `canon` command uses the `bdschur` command to convert `sys` into modal form and to compute the transformation `T`. If `sys` is not a state-space model, the algorithm first converts it to state space using `ss`.

The reduction to companion form uses a state similarity transformation based on the controllability matrix [1].

## References

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

### See Also

ctrb | ctrbf | ss2ss

**Introduced before R2006a**

## care

Continuous-time algebraic Riccati equation solution

### Syntax

```
[X,L,G] = care(A,B,Q)
[X,L,G] = care(A,B,Q,R,S,E)
[X,L,G,report] = care(A,B,Q,...)
[X1,X2,D,L] = care(A,B,Q,...,'factor')
```

### Description

`[X,L,G] = care(A,B,Q)` computes the unique solution  $X$  of the continuous-time algebraic Riccati equation

$$A^T X + XA - XBB^T X + Q = 0$$

The `care` function also returns the gain matrix,  $G = R^{-1}B^T XE$ .

`[X,L,G] = care(A,B,Q,R,S,E)` solves the more general Riccati equation

$$A^T XE + E^T XA - (E^T XB + S)R^{-1}(B^T XE + S^T) + Q = 0$$

When omitted,  $R$ ,  $S$ , and  $E$  are set to the default values  $R=I$ ,  $S=0$ , and  $E=I$ . Along with the solution  $X$ , `care` returns the gain matrix  $G = R^{-1}(B^T XE + S^T)$  and a vector  $L$  of closed-loop eigenvalues, where

$$L = \text{eig}(A - B * G, E)$$

`[X,L,G,report] = care(A,B,Q,...)` returns a diagnosis `report` with:

- -1 when the associated Hamiltonian pencil has eigenvalues on or very near the imaginary axis (failure)
- -2 when there is no finite stabilizing solution  $X$

- The Frobenius norm of the relative residual if X exists and is finite.

This syntax does not issue any error message when X fails to exist.

`[X1,X2,D,L] = care(A,B,Q,...,'factor')` returns two matrices X1, X2 and a diagonal scaling matrix D such that  $X = D*(X2/X1)*D$ .

The vector L contains the closed-loop eigenvalues. All outputs are empty when the associated Hamiltonian matrix has eigenvalues on the imaginary axis.

## Examples

### Example 1

#### Solve Algebraic Riccati Equation

Given

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

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  $\text{eig}(a-b*g)$ .

```
l
```

```
l =
```

```
-3.5026
-1.4370
```

## Example 2

### Solve H-infinity ( $H_\infty$ )-like Riccati Equation

To solve the  $H_\infty$ -like Riccati equation

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

rewrite it in the `care` format as

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

You can now compute the stabilizing solution  $X$  by

```
B = [B1 , B2]
```

```
m1 = size(B1,2)
```

```
m2 = size(B2,2)
```

```
R = [-g^2*eye(m1) zeros(m1,m2) ; zeros(m2,m1) eye(m2)]
```

`X = care(A,B,C'*C,R)`

### 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

$$\begin{bmatrix} Q & S \\ S^T & R \end{bmatrix} > 0$$

### More About

#### Algorithms

`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.

### 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

### See Also

`dare` | `lyap`

Introduced before R2006a



# chgFreqUnit

Change frequency units of frequency-response data model

## Syntax

```
sys_new = chgFreqUnit(sys,newfrequnits)
```

## Description

`sys_new = chgFreqUnit(sys,newfrequnits)` changes units of the frequency points in `sys` to `newfrequnits`. Both `Frequency` and `FrequencyUnit` properties of `sys` adjust so that the frequency responses of `sys` and `sys_new` match.

## Input Arguments

### **sys**

Frequency-response data (`frd`, `idfrd`, or `genfrd`) model

### **newfrequnits**

New units of frequency points, specified as one of the following values:

- 'rad/TimeUnit'
- 'cycles/TimeUnit'
- 'rad/s'
- 'Hz'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rpm'

`rad/TimeUnit` and `cycles/TimeUnit` express frequency units relative to the system time units specified in the `TimeUnit` property.

**Default:** `'rad/TimeUnit'`

## Output Arguments

### `sys_new`

Frequency-response data model of the same type as `sys` with new units of frequency points. The frequency response of `sys_new` is same as `sys`.

## Examples

### Change Frequency Units of Frequency-Response Data Model

Create a frequency-response data model.

```
load(fullfile(matlabroot, 'examples', 'controls_id', 'AnalyzerData'));  
sys = frd(resp, freq);
```

The data file `AnalyzerData` has column vectors `freq` and `resp`. These vectors contain 256 test frequencies and corresponding complex-valued frequency response points, respectively. The default frequency units of `sys` is `rad/TimeUnit`, where `TimeUnit` is the system time units.

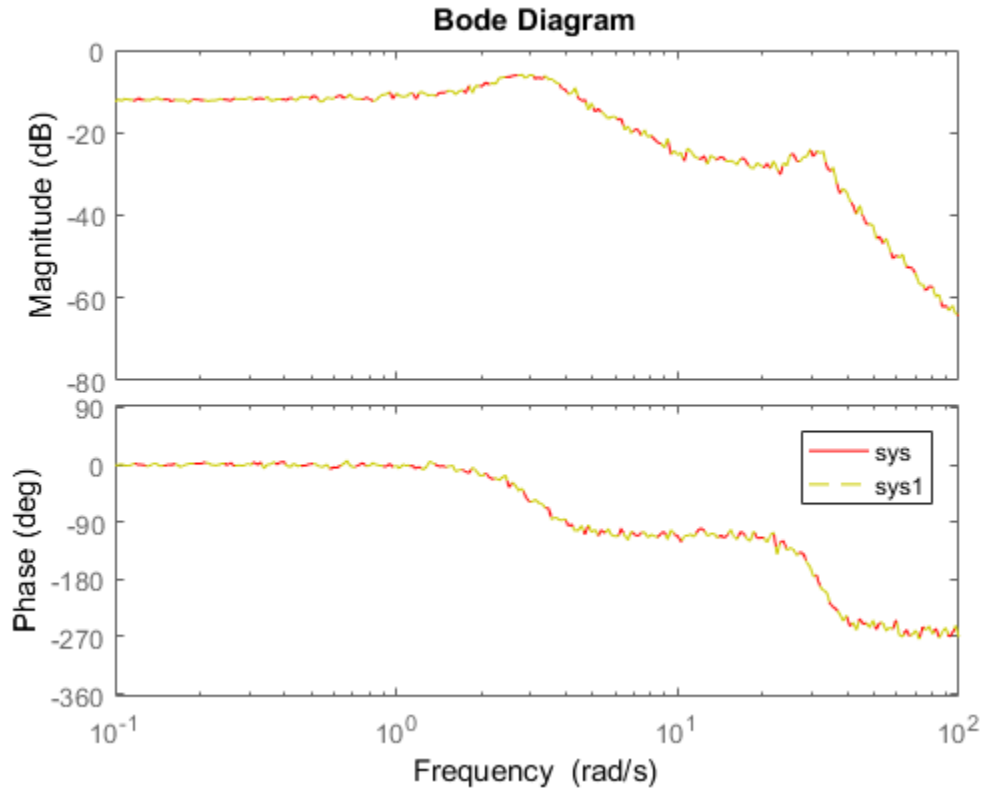
Change the frequency units.

```
sys1 = chgFreqUnit(sys, 'rpm');
```

The `FrequencyUnit` property of `sys1` is `rpm`.

Compare the Bode responses of `sys` and `sys1`.

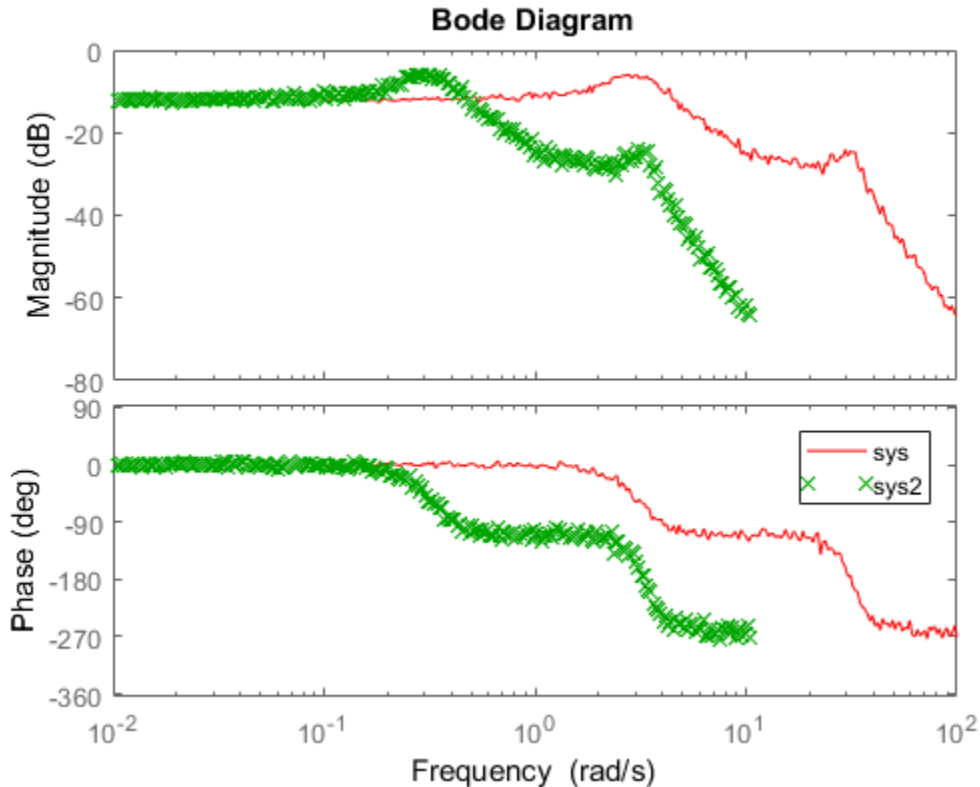
```
bodeplot(sys, 'r', sys1, 'y--');  
legend('sys', 'sys1')
```



The magnitude and phase of `sys` and `sys1` match because `chgFreqUnit` command changes the units of frequency points in `sys` without modifying system behavior.

Change the `FrequencyUnit` property of `sys` to compare the Bode response with the original system.

```
sys2 = sys;
sys2.FrequencyUnit = 'rpm';
bodeplot(sys, 'r', sys2, 'gx');
legend('sys', 'sys2');
```



Changing the `FrequencyUnit` property changes the system behavior. Therefore, the Bode responses of `sys` and `sys2` do not match. For example, the original corner frequency at about 2 rad/s changes to approximately 2 rpm (or 0.2 rad/s).

- “Specify Frequency Units of Frequency-Response Data Model”

## More About

### Tips

- Use `chgFreqUnit` to change the units of frequency points without modifying system behavior.

## **See Also**

chgTimeUnit | frd

**Introduced in R2011a**

## chgTimeUnit

Change time units of dynamic system

### Syntax

```
sys_new = chgTimeUnit(sys,newtimeunits)
```

### Description

`sys_new = chgTimeUnit(sys,newtimeunits)` changes the time units of `sys` to `newtimeunits`. The time- and frequency-domain characteristics of `sys` and `sys_new` match.

### Input Arguments

#### **sys**

Dynamic system model

#### **newtimeunits**

New time units, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'

- 'months'
- 'years'

**Default:** 'seconds'

## Output Arguments

### **sys\_new**

Dynamic system model of the same type as **sys** with new time units. The time response of **sys\_new** is same as **sys**.

If **sys** is an identified linear model, both the model parameters as and their minimum and maximum bounds are scaled to the new time units.

## Examples

### **Change Time Units of Dynamic System Model**

Create a transfer function model.

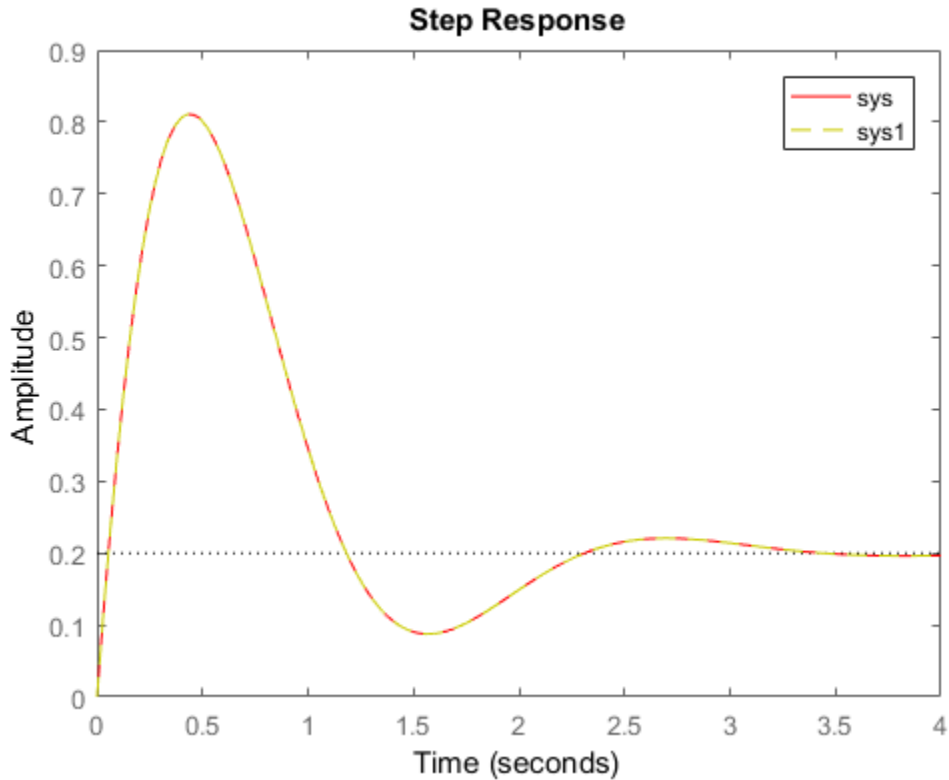
```
num = [4 2];  
den = [1 3 10];  
sys = tf(num,den);
```

By default, the time unit of **sys** is 'seconds'. Create a new model with the time units changed to minutes.

```
sys1 = chgTimeUnit(sys,'minutes');
```

This command sets the `TimeUnit` property of **sys1** to 'minutes', without changing the dynamics. To confirm that the dynamics are unchanged, compare the step responses of **sys** and **sys1**.

```
stepplot(sys,'r',sys1,'y--');  
legend('sys','sys1');
```

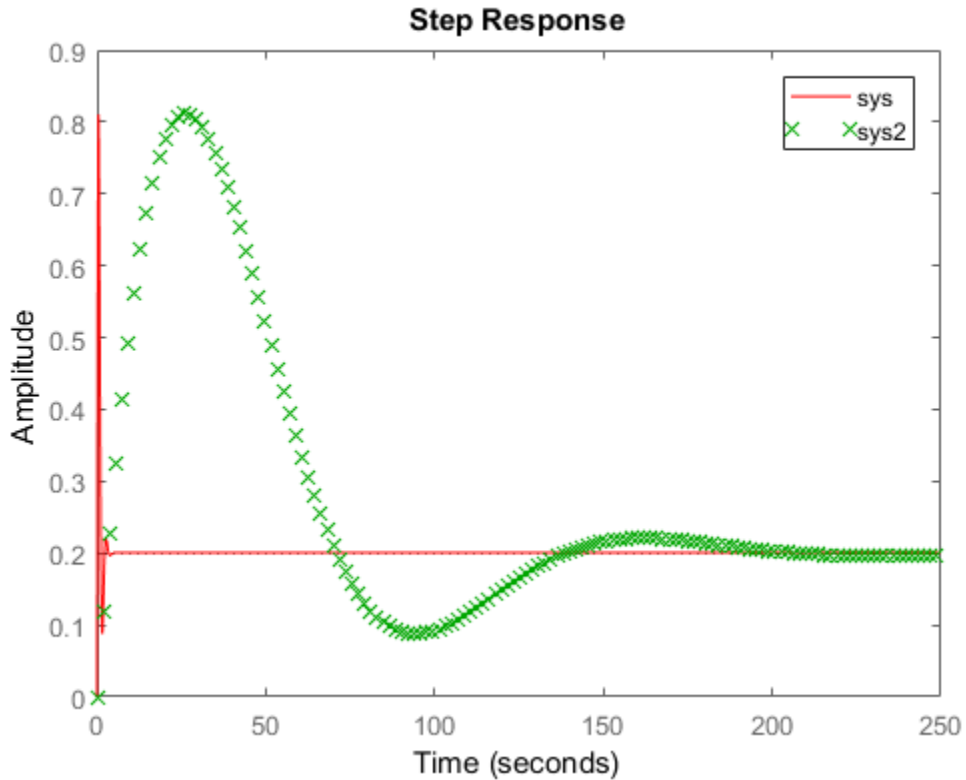


The step responses are the same.

If you change the `TimeUnit` property of the system instead of using `chgTimeUnit`, the dynamics of the system do change. To see this, change the `TimeUnit` property of a copy of `sys` and compare the step response with the original system.

```
sys2 = sys;  
sys2.TimeUnit = 'minutes';  
stepplot(sys,'r',sys2,'gx');  
legend('sys','sys2');
```





The step responses of `sys` and `sys2` do not match. For example, the original rise time of 0.04 seconds changes to 0.04 minutes.

- “Specify Model Time Units”

## More About

### Tips

- Use `chgTimeUnit` to change the time units without modifying system behavior.

**See Also**

chgFreqUnit | tf | zpk | ss | frd | pid

**Introduced in R2011a**

# clone

Copy online state estimation object

## Syntax

```
obj_clone = clone(obj)
```

## Description

`obj_clone = clone(obj)` creates a copy of the online state estimation object `obj` with the same property values.

If you want to copy an existing object and then modify properties of the copied object, use the `clone` command. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created in this way (`obj2`) also change the properties of the original object (`obj`).

## Examples

### Clone an Online State Estimation Object

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. To create the object, use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. Specify the initial state values for the two states as `[2;0]`.

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0])
```

```
obj =
```

```
    extendedKalmanFilter with properties:
```

```
    HasAdditiveProcessNoise: 1  
    StateTransitionFcn: @vdpStateFcn
```

```
HasAdditiveMeasurementNoise: 1
    MeasurementFcn: @vdpMeasurementFcn
StateTransitionJacobianFcn: []
MeasurementJacobianFcn: []
    State: [2×1 double]
    StateCovariance: [2×2 double]
    ProcessNoise: [2×2 double]
MeasurementNoise: 1
```

Use `clone` to generate an object with the same properties as the original object.

```
obj2 = clone(obj)
```

```
obj2 =
```

```
extendedKalmanFilter with properties:
```

```
    HasAdditiveProcessNoise: 1
    StateTransitionFcn: @vdpStateFcn
HasAdditiveMeasurementNoise: 1
    MeasurementFcn: @vdpMeasurementFcn
StateTransitionJacobianFcn: []
MeasurementJacobianFcn: []
    State: [2×1 double]
    StateCovariance: [2×2 double]
    ProcessNoise: [2×2 double]
MeasurementNoise: 1
```

Modify the `MeasurementNoise` property of `obj2`.

```
obj2.MeasurementNoise = 2;
```

Verify that `MeasurementNoise` property of original object `obj` remains unchanged and equals 1.

```
obj.MeasurementNoise
```

```
ans =
```

```
1
```

## Input Arguments

### **obj** — Object for online state estimation

`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Object for online state estimation of a nonlinear system, created using one of the following commands:

- `extendedKalmanFilter`
- `unscentedKalmanFilter`

## Output Arguments

### **obj\_clone** — Clone of online state estimation object

`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Clone of online state estimation object `obj`, returned as an `extendedKalmanFilter` or `unscentedKalmanFilter` object with the same properties as `obj`.

## See Also

`correct` | `extendedKalmanFilter` | `predict` | `unscentedKalmanFilter`

**Introduced in R2016b**

# conj

Form model with complex conjugate coefficients

## Syntax

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

## Description

`sysc = conj(sys)` 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.

## Examples

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` | `ss` | `tf` | `zpk`

**Introduced before R2006a**

## connect

Block diagram interconnections of dynamic systems

### Syntax

```
sysc = connect(sys1, ..., sysN, inputs, outputs)
sysc = connect(sys1, ..., sysN, inputs, outputs, APs)
sysc = connect(blksys, connections, inputs, outputs)
sysc = connect( ____, opts)
```

### Description

`sysc = connect(sys1, ..., sysN, inputs, outputs)` connects the block diagram elements `sys1, ..., sysN` based on signal names. The block diagram elements `sys1, ..., sysN` are dynamic system models. These models can include summing junctions that you create using `sumblk`. The `connect` command interconnects the block diagram elements by matching the input and output signals that you specify in the `InputName` and `OutputName` properties of `sys1, ..., sysN`. The aggregate model `sysc` is a dynamic system model having inputs and outputs specified by `inputs` and `outputs` respectively.

`sysc = connect(sys1, ..., sysN, inputs, outputs, APs)` inserts an `AnalysisPoint` at every signal location specified in `APs`. Use analysis points to mark locations of interest which are internal signals in the aggregate model. For instance, a location at which you want to extract a loop transfer function or measure the stability margins is a location of interest.

`sysc = connect(blksys, connections, inputs, outputs)` uses index-based interconnection to build `sysc` out of an aggregate, unconnected model `blksys`. The matrix `connections` specifies how the outputs and inputs of `blksys` interconnect. For index-based interconnections, `inputs` and `outputs` are index vectors that specify which inputs and outputs of `blksys` are the external inputs and outputs of `sysc`. This syntax can be convenient when you do not want to assign names to all inputs and outputs of all models to connect. However, in general, it is easier to keep track of named signals.

`sysc = connect( ____, opts)` builds the interconnected model using additional options. You can use `opts` with the input arguments of any of the previous syntaxes.

## Input Arguments

### **sys1, . . . , sysN**

Dynamic system models that correspond to the elements of your block diagram. For example, the elements of your block diagram can include one or more `tf` or `ss` models that represent plant dynamics. Block diagram elements can also include a `pid` or `tunablePID` model representing a controller. You can also include one or more summing junction that you create using `sumblk`. Provide multiple arguments `sys1, . . . , sysN` to represent all of the block diagram elements and summing junctions.

### **inputs**

For name-based interconnection, a character vector or cell array of character vectors that specify the inputs of the aggregate model `sysc`. The inputs in `inputs` must correspond to entries in the `InputName` or `OutputName` property of one or more of the block diagram elements `sys1, . . . , sysN`.

### **outputs**

For name-based interconnection, a character vector or cell array of character vectors that specify the outputs of the aggregate model `sysc`. The outputs in `outputs` must correspond to entries in the `OutputName` property of one or more of the block diagram elements `sys1, . . . , sysN`.

### **APs**

Locations (internal signals) of interest in the aggregate model, specified as a character vector or cell array of character vectors, such as `'X'` or `{'AP1', 'AP2'}`. The resulting model contains an analysis point at each such location. (See `AnalysisPoint`). Each location in `APs` must correspond to an entry in the `InputName` or `OutputName` property of one or more of the block diagram elements `sys1, . . . , sysN`.

### **blksys**

Unconnected aggregate model. To obtain `blksys`, use `append` to join dynamic system models of the elements of your block diagram. For example, if your block diagram contains dynamic system models `C`, `G`, and `S`, create `blksys` with the following command:

```
blksys = append(C,G,S)
```



## connections

Matrix that specifies the connections and summing junctions of the block diagram. Each row of **connections** specifies one connection or summing junction in terms of the input vector **u** and output vector **y** of the unconnected aggregate model **blksys**. For example, the row:

```
[3 2 0 0]
```

specifies that **y(2)** connects into **u(3)**. The row

```
[7 2 -15 6]
```

indicates that **y(2) - y(15) + y(6)** feeds into **u(7)**.

If you do not specify any connection for a particular input or output, **connect** omits that input or output from the aggregate model.

## opts

Additional options for interconnection, specified as an options set that you create with **connectOptions**.

# Output Arguments

## sysc

Interconnected system, returned as either a state-space model or frequency-response model. The type of model returned depends on the input models. For example:

- Interconnecting numeric LTI models (other than **frd** models) returns an **ss** model.
- Interconnecting a numeric LTI model with a Control Design Block returns a generalized LTI model. For instance, interconnecting a **tf** model with a **tunablePID** Control Design Block returns a **genss**.
- Interconnecting any model with frequency-response data model returns a frequency response data model.

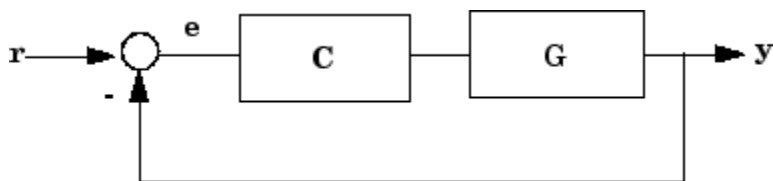
By default, **connect** automatically discards states that do not contribute to the I/O transfer function from the specified inputs to the specified outputs of the interconnected model. To retain the unconnected states, set the **Simplify** option of **connectOptions** to **false**. For example:

```
opt = connectOptions('Simplify',false);
sysc = connect(sys1,sys2,sys3,'r','y',opt);
```

## Examples

### SISO Feedback Loop

Create an aggregate model of the following block diagram from  $r$  to  $y$ .



Create  $C$  and  $G$ , and name the inputs and outputs.

```
C = pid(2,1);
C.u = 'e';
C.y = 'u';
G = zpk([], [-1, -1], 1);
G.u = 'u';
G.y = 'y';
```

The notations  $C.u$  and  $C.y$  are shorthand expressions equivalent to  $C.InputName$  and  $C.OutputName$ , respectively. For example, entering  $C.u = 'e'$  is equivalent to entering  $C.InputName = 'e'$ . The command sets the `InputName` property of  $C$  to the value `'e'`.

Create the summing junction.

```
Sum = sumblk('e = r - y');
```

Combine  $C$ ,  $G$ , and the summing junction to create the aggregate model from  $r$  to  $y$ .

```
T = connect(G,C,Sum,'r','y');
```

`connect` automatically joins inputs and outputs with matching names.

### MIMO Feedback Loop

Create the control system of the previous example where  $G$  and  $C$  are both 2-input, 2-output models.

```

C = [pid(2,1),0;0,pid(5,6)];
C.InputName = 'e';
C.OutputName = 'u';
G = ss(-1,[1,2],[1;-1],0);
G.InputName = 'u';
G.OutputName = 'y';

```

When you specify single names for vector-valued signals, the software automatically performs vector expansion of the signal names. For example, examine the names of the inputs to C.

```
C.InputName
```

```

ans =

    'e(1) '
    'e(2) '

```

Create a 2-input, 2-output summing junction.

```
Sum = sumblk('e = r-y',2);
```

sumblk also performs vector expansion of the signal names.

Interconnect the models to obtain the closed-loop system.

```
T = connect(G,C,Sum,'r','y');
```

The block diagram elements G, C, and Sum are all 2-input, 2-output models. Therefore, connect performs the same vector expansion. connect selects all entries of the two-input signals 'r' and 'y' as inputs and outputs to T, respectively. For example, examine the input names of T.

```
T.InputName
```

```

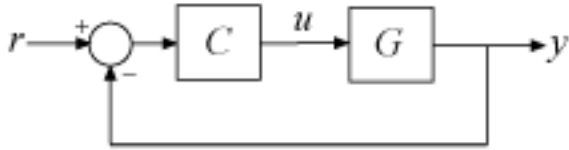
ans =

    'r(1) '
    'r(2) '

```

## Feedback Loop With Analysis Point Inserted by connect

Create a model of the following block diagram from  $r$  to  $y$ . Insert an analysis point at an internal location,  $u$ .



Create C and G, and name the inputs and outputs.

```
C = pid(2,1);
C.InputName = 'e';
C.OutputName = 'u';
G = zpk([],[-1,-1],1);
G.InputName = 'u';
G.OutputName = 'y';
```

Create the summing junction.

```
Sum = sumblk('e = r - y');
```

Combine C, G, and the summing junction to create the aggregate model, with an analysis point at  $u$ .

```
T = connect(G,C,Sum, 'r', 'y', 'u')
```

```
T =
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, and 1 analysis points.
AnalysisPoints_: Analysis point, 1 channels, 1 occurrences.
```

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

The resulting T is a `genss` model. The `connect` command creates the `AnalysisPoint` block, `AnalysisPoints_`, and inserts it into T. To see the name of the analysis point channel in `AnalysisPoints_`, use `getPoints`.

```
getPoints(T)
```

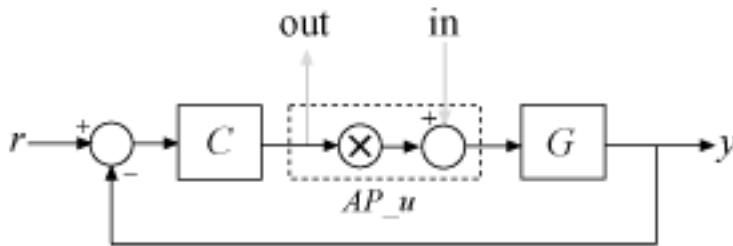
```
ans =
```

```
cell
    'u'
```

The analysis point channel is named 'u'. You can use this analysis point to extract system responses. For example, the following commands extract the open-loop transfer at  $u$  and the closed-loop response at  $y$  to a disturbance injected at  $u$ .

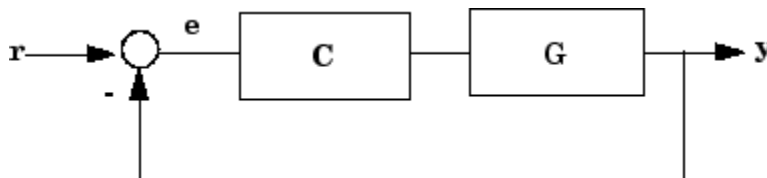
```
L = getLoopTransfer(T, 'u', -1);
Tuy = getIOTransfer(T, 'u', 'y');
```

T is equivalent to the following block diagram, where  $AP_u$  designates the AnalysisPoint block AnalysisPoints\_ with channel name  $u$ .



## Index-Based Interconnection

Create an aggregate model of the following block diagram from  $r$  to  $y$  using index-based interconnection.



Create `C`, `G`, and the unconnected aggregate model `blksys`.

```
C = pid(2,1);  
G = zpk([],[-1,-1],1);  
blksys = append(C,G);
```

The inputs `u(1)`, `u(2)` of `blksys` correspond to the inputs of `C` and `G`, respectively. The outputs `w(1)`, `w(2)` of `blksys` correspond to the outputs of `C` and `G`, respectively.

Create the matrix `connections`, which specifies which outputs of `blksys` connect to which inputs of `blksys`.

```
connections = [2 1; 1 -2];
```

The first row indicates that `w(1)` connects to `u(2)`; in other words, that the output of `C` connects to the input of `G`. The second row indicates that `-w(2)` connects to `u(1)`; in other words, that the negative of the output of `G` connects to the input of `C`.

Create the connected aggregate model from `r` to `y`.

```
T = connect(blksys,connections,1,2)
```

The last two arguments specify the external inputs and outputs in terms of the indices of `blksys`. The argument `1` specifies that the external input connects to `u(1)`. The last argument, `2`, specifies that the external output connects from `w(2)`.

## More About

- “Multi-Loop Control System”
- “MIMO Control System”
- “MIMO Feedback Loop”
- “Mark Analysis Points in Closed-Loop Models”

## See Also

| `append` | `sumblk` | `AnalysisPoint` | `feedback` | `parallel` | `series` | `lft` | `connectOptions`

**Introduced before R2006a**

# connectOptions

Options for the connect command

## Syntax

```
opt = connectOptions  
opt = connectOptions(Name,Value)
```

## Description

`opt = connectOptions` returns the default options for `connect`.

`opt = connectOptions(Name,Value)` returns an options set with the options specified by one or more `Name,Value` pair arguments.

## Examples

### Retain Unconnected States in Model Interconnection

Use `connectOptions` to cause the `connect` command to retain unconnected states in an interconnected model.

Suppose you have dynamic system models `sys1`, `sys2`, and `sys3`. Combine these dynamic system models to build an interconnected model with input `'r'` and output `'y'`. Set the option to retain states in the model that do not contribute to the dynamics in the path from `'r'` or `'y'`.

```
opt = connectOptions('Simplify',false);  
sysc = connect(sys1,sys2,sys3,'r','y',opt);
```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single

quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Simplify',false`

### 'Simplify' — Automatic elimination of unconnected states

`true` (default) | `false`

Automatic elimination of unconnected states, specified as either `true` or `false`.

- `true` — `connect` eliminates all states that do not contribute to the I/O transfer function from the specified inputs to the specified outputs of the interconnected system.
- `false` — `connect` retains unconnected states. This option can be useful, for example, when you want to compute the interconnected system response from known initial state values of the components.

Data Types: `logical`

## Output Arguments

### `opt` — Options for `connect`

`connectOptions` options set

Options for `connect`, returned as a `connectOptions` options set. Use `opt` as the last argument to `connect` when interconnecting models.

### See Also

`connect`

Introduced in R2013b



# Control System Designer

Design single-input, single-output (SISO) controllers

## Description

The **Control System Designer** app lets you design single-input, single-output (SISO) controllers for feedback systems modeled in MATLAB or Simulink (requires Simulink Control Design™ software).

Using this app, you can:

- Design controllers using:
  - Interactive Bode, root locus, and Nichols graphical editors for adding, modifying, and removing controller poles, zeros, and gains.
  - Automated PID, LQG, or IMC tuning.
  - Optimization-based tuning (requires Simulink Design Optimization™ software).
  - Automated loop shaping (requires Robust Control Toolbox™ software).
- Tune compensators for single-loop or multiloop control architectures.
- Analyze control system designs using time-domain and frequency-domain responses, such as step responses and pole-zero maps.
- Compare response plots for multiple control system designs.
- Design controllers for multimodel control applications.

## Open the Control System Designer App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `controlSystemDesigner`.
- Simulink model editor: Select **Analysis > Control Design > Control System Designer**.

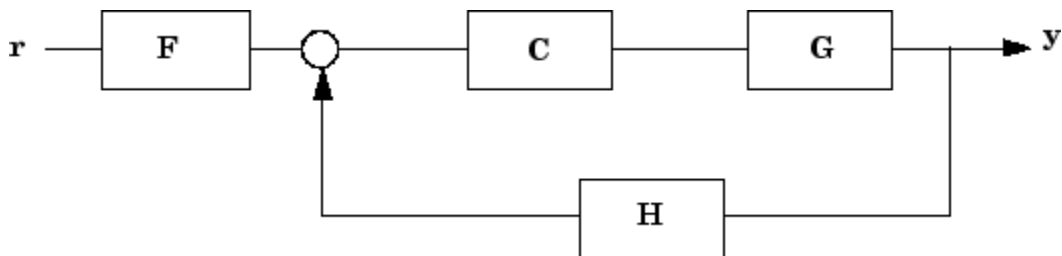
## Examples

- “Control System Designer Tuning Methods”

- “Bode Diagram Design”
- “Root Locus Design”
- “Design Compensator Using Automated Tuning Methods”
- “Design Multiloop Control System”
- “Analyze Designs Using Response Plots”
- “Compare Performance of Multiple Designs”
- “Multimodel Control Design”

### Programmatic Use

`controlSystemDesigner` opens the **Control System Designer** app using the following default control architecture:



The architecture consists of the LTI objects:

- $G$  — Plant model
- $C$  — Compensator
- $H$  — Sensor model
- $F$  — Prefilter

By default, the app configures each of these models as a unit gain.

`controlSystemDesigner(plant)` initializes the plant,  $G$ , to `plant`. `plant` can be any SISO LTI model created with `ss`, `tf`, `zpk` or `frd`, or an array of such models.

`controlSystemDesigner(plant,comp)` initializes the compensator,  $C$ , to the SISO LTI model `comp`.

`controlSystemDesigner(plant,comp,sensor)` initializes the sensor model,  $H$ , to `sensor`. `sensor` can be any SISO LTI model or an array of such models. If you specify both `plant` and `sensor` as LTI model arrays, the lengths of the arrays must match.

`controlSystemDesigner(plant,comp,sensor,prefilt)` initializes the prefilter model,  $F$ , to the SISO LTI model `prefilt`.

`controlSystemDesigner(views)` opens the app and specifies the initial graphical editor configuration. `views` can be any of the following character vectors, or a cell array of multiple character vectors.

- 'rlocus' — Root locus editor
- 'bode' — Open-loop Bode Editor
- 'nichols' — Open-loop Nichols Editor
- 'filter' — Bode Editor for the closed-loop response from prefilter input to the plant output

In addition to opening the specified graphical editors, the app plots the closed-loop, input-output step response.

`controlSystemDesigner(views,plant,comp,sensor,prefilt)` specifies the initial plot configuration and initializes the plant, compensator, sensor, and prefilter using the specified models. If a model is omitted, the app uses the default value.

`controlSystemDesigner(initData)` opens the app and initializes the system configuration using the initialization data structure `initdata`. To create `initdata`, use `sisoinit`.

`controlSystemDesigner(sessionFile)` opens the app and loads a previously saved session. `sessionFile` is the name of a session data file on the MATLAB path. This data includes the current system architecture and plot configuration, and any designs and responses saved in the **Data Browser**.

To save a session, in the **Control System Designer** app, on the **Control System** tab, click  **Save Session**.

## See Also

### Apps

Control System Tuner

**Functions**

pidTuner | sisoinit

**Introduced in R2015a**

# Control System Tuner

Tune fixed-structure control systems

## Description

The **Control System Tuner** app tunes control systems modeled in MATLAB or Simulink (requires Simulink Control Design software). This app lets you tune any control system architecture to meet your design goals. You can tune multiple fixed-order, fixed-structure control elements distributed over one or more feedback loops.

**Control System Tuner** automatically tunes the controller parameters to satisfy the must-have requirements (design constraints) and to best meet the remaining requirements (objectives). The library of tuning goals lets you capture your design requirements in a form suitable for fast automated tuning. Available tuning goals include standard control objectives such as reference tracking, disturbance rejection, loop shapes, closed-loop damping, and stability margins.

## Open the Control System Tuner App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `controlSystemTuner`.
- Simulink model editor: Select **Analysis > Control Design > Control System Tuner**.

## Examples

- “Setup for Tuning Control System Modeled in MATLAB”
- “Specify Control Architecture in Control System Tuner”
- “Tune a Control System Using Control System Tuner”

## Programmatic Use

`controlSystemTuner` opens the Control System Tuner app. When invoked without input arguments, Control System Tuner opens for tuning the default single-loop


feedback control system architecture. You can then edit the components of this default architecture as described in “Specify Control Architecture in Control System Tuner”.

`controlSystemTuner(CL)` opens the app for tuning the control architecture specified in the `genss` model `CL`. If your control architecture does not match Control System Tuner’s predefined control architecture, use this syntax with a `genss` model that has tunable components representing your controller elements. See “Specify Control Architecture in Control System Tuner”.

`controlSystemTuner mdl` opens the app for tuning blocks in a Simulink model. `mdl` is the name of a Simulink model saved in the current working directory or on the MATLAB path. (Requires Simulink Control Design software.)

`controlSystemTuner(ST)` opens the app for tuning a Simulink model associated with an `sITuner` interface, `ST`. Control System Tuner takes information such as analysis points and operating points from `ST`. (Requires Simulink Control Design software.)

`controlSystemTuner(sessionfile)` opens the app and loads a previously saved session.

When you use Control System Tuner, you can click  **Save Session** to save session data to disk such as tuning goals you have created, response I/Os you have defined, operating points, and stored designs. `sessionfile` is the name of a session data file saved in the current working directory or on the MATLAB path.

## See Also

### Functions

`genss` | `sITuner` | `systune`

**Introduced in R2014a**

## correct

Correct state and state estimation error covariance using extended or unscented Kalman filter and measurements

The `correct` command updates the state and state estimation error covariance of an `extendedKalmanFilter` or `unscentedKalmanFilter` object using measured system outputs. To implement extended or unscented Kalman filter algorithms, use the `correct` and `predict` commands together. If the current output measurement exists, you can use `correct` and `predict`. If the measurement is missing, you can only use `predict`. For information about the order in which to use the commands, see “Using predict and correct Commands” on page 2-146.

## Syntax

```
[CorrectedState,CorrectedStateCovariance] = correct(obj,y)
[CorrectedState,CorrectedStateCovariance] = correct(obj,y,
Um1,...,Umn)
```

## Description

`[CorrectedState,CorrectedStateCovariance] = correct(obj,y)` corrects the state estimate and state estimation error covariance of an extended or unscented Kalman filter object `obj` using the measured output `y`.

You create `obj` using the `extendedKalmanFilter` or `unscentedKalmanFilter` commands. You specify the state transition function and measurement function of your nonlinear system in `obj`. You also specify whether the process and measurement noise terms are additive or nonadditive in these functions. The `State` property of the object stores the latest estimated state value. Assume that at time step  $k$ , `obj.State` is  $\hat{x}[k|k-1]$ . This value is the state estimate for time  $k$ , estimated using measured outputs until time  $k-1$ . When you use the `correct` command with measured system output  $y[k]$ , the software returns the corrected state estimate  $\hat{x}[k|k]$  in the `CorrectedState` output. Where  $\hat{x}[k|k]$  is the state estimate at time  $k$ , estimated using measured outputs until time  $k$ . The command returns the state estimation error covariance of  $\hat{x}[k|k]$  in

the `CorrectedStateCovariance` output. The software also updates the `State` and `StateCovariance` properties of `obj` with these corrected values.

Use this syntax if the measurement function  $h$  that you specified in `obj.MeasurementFcn` has one of the following forms:

- $y(k) = h(x(k))$  — for additive measurement noise.
- $y(k) = h(x(k), v(k))$  — for nonadditive measurement noise.

Where  $y(k)$ ,  $x(k)$ , and  $v(k)$  are the measured output, states, and measurement noise of the system at time step  $k$ . The only inputs to  $h$  are the states and measurement noise.

`[CorrectedState, CorrectedStateCovariance] = correct(obj, y, Um1, ..., Umn)` specifies additional input arguments, if the measurement function of the system requires these inputs. You can specify multiple arguments.

Use this syntax if the measurement function  $h$  has one of the following forms:

- $y(k) = h(x(k), Um1, \dots, Umn)$  — for additive measurement noise.
- $y(k) = h(x(k), v(k), Um1, \dots, Umn)$  — for nonadditive measurement noise.

`correct` command passes these inputs to the measurement function to calculate the estimated outputs.

## Examples

### Estimate States Online Using Extended Kalman Filter

Estimate the states of a van der Pol oscillator using an extended Kalman filter algorithm and measured output data. The oscillator has two states and one output.

Create an extended Kalman filter object for the oscillator. Use previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to a van der Pol oscillator with nonlinearity parameter,  $\mu$ , equal to 1. The functions assume additive process and measurement noise in the system. Specify the initial state values for the two states as `[1;0]`. This is the guess for the state value at initial time  $k$ , using knowledge of system outputs until time  $k-1$ ,  $\hat{x}[k|k-1]$ .

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[1;0]);
```



Load the measured output data,  $y$ , from the oscillator. In this example, use simulated static data for illustration. The data is stored in the `vdp_data.mat` file.

```
load vdp_data.mat y
```

Specify the process noise and measurement noise covariances of the oscillator.

```
obj.ProcessNoise = 0.01;
obj.MeasurementNoise = 0.16;
```

Implement the extended Kalman filter algorithm to estimate the states of the oscillator by using the `correct` and `predict` commands. You first correct  $\hat{x}[k|k-1]$  using measurements at time  $k$  to get  $\hat{x}[k|k]$ . Then, you predict the state value at next time step,  $\hat{x}[k+1|k]$ , using  $\hat{x}[k|k]$ , the state estimate at time step  $k$  that is estimated using measurements until time  $k$ .

To simulate real-time data measurements, use the measured data one time step at a time.

```
for k = 1:size(y)
    [CorrectedState,CorrectedStateCovariance] = correct(obj,y(k));
    [PredictedState,PredictedStateCovariance] = predict(obj);
end
```

When you use the `correct` command, `obj.State` and `obj.StateCovariance` are updated with the corrected state and state estimation error covariance values for time step  $k$ , `CorrectedState` and `CorrectedStateCovariance`. When you use the `predict` command, `obj.State` and `obj.StateCovariance` are updated with the predicted values for time step  $k+1$ , `PredictedState` and `PredictedStateCovariance`.

In this example, you used `correct` before `predict` because the initial state value was  $\hat{x}[k|k-1]$ , a guess for the state value at initial time  $k$  using system outputs until time  $k-1$ . If your initial state value is  $\hat{x}[k-1|k-1]$ , the value at previous time  $k-1$  using measurement until  $k-1$ , then use the `predict` command first. For more information about the order of using `predict` and `correct`, see “Using `predict` and `correct` Commands”.

### Specify State Transition and Measurement Functions with Additional Inputs

Consider a nonlinear system with input  $u$  whose state  $x$  and measurement  $y$  evolve according to the following state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise  $w$  of the system is additive while the measurement noise  $v$  is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input  $u$ .

```
f = @(x,u)(sqrt(x+u));  
h = @(x,v,u)(x+2*u+v^2);
```

$f$  and  $h$  are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive,  $v$  is also specified as an input. Note that  $v$  is specified as an input before the additional input  $u$ .

Create an extended Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = extendedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of  $u$  to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement  $y[k]=0.8$  and input  $u[k]=0.2$  at time step  $k$ .

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given  $u[k]=0.2$ .

```
predict(obj,0.2)
```

- “Nonlinear State Estimation Using Unscented Kalman Filter”
- “Generate Code for Online State Estimation in MATLAB”

## Input Arguments

### **obj** — Extended or unscented Kalman filter object

`extendedKalmanFilter` object | `unscentedKalmanFilter` object

Extended or unscented Kalman filter object for online state estimation, created using one of the following commands:

- `extendedKalmanFilter` — Uses the extended Kalman filter algorithm.
- `unscentedKalmanFilter` — Uses the unscented Kalman filter algorithm.

### **y** — Measured system output

vector

Measured system output at the current time step, specified as an  $N$ -element vector, where  $N$  is the number of measurements.

### **Um1, . . . , Umn** — Additional input arguments to measurement function

input arguments of any type

Additional input arguments to the measurement function of the system, specified as input arguments of any type. The measurement function,  $h$ , is specified in the `MeasurementFcn` property of `obj`. If the function requires input arguments in addition to the state and measurement noise values, you specify these inputs in the `correct` command syntax. `correct` command passes these inputs to the measurement function to calculate estimated outputs. You can specify multiple arguments.

For example, suppose that your measurement function calculates the estimated system output  $y$  using system inputs  $u$  and current time  $k$ , in addition to the state  $x$ :

$$y(k) = h(x(k), u(k), k)$$

Then when you perform online state estimation at time step  $k$ , specify these additional inputs in the `correct` command syntax:

```
[CorrectedState,CorrectedStateCovariance] = correct(obj,y,u(k),k);
```

## Output Arguments

### **CorrectedState** — Corrected state estimate

vector

Corrected state estimate, returned as a vector of size  $M$ , where  $M$  is the number of states of the system. If you specify the initial states of `obj` as a column vector then  $M$  is returned as a column vector, otherwise  $M$  is returned as a row vector.

For information about how to specify the initial states of the object, see the `extendedKalmanFilter` and `unscentedKalmanFilter` reference pages.

### **CorrectedStateCovariance** — Corrected state estimation error covariance matrix

Corrected state estimation error covariance, returned as an  $M$ -by- $M$  matrix, where  $M$  is the number of states of the system.

## More About

### Using `predict` and `correct` Commands

After you have created an extended or unscented Kalman filter object, `obj`, to implement the extended or unscented Kalman filter algorithms, use the `correct` and `predict` commands together.

At time step  $k$ , `correct` command returns the corrected value of states and state estimation error covariance using measured system outputs  $y[k]$  at the same time step. If your measurement function has additional input arguments  $U_m$ , you specify these as inputs to the `correct` command. The command passes these values to the measurement function.

```
[CorrectedState,CorrectedCovariance] = correct(obj,y,Um)
```

The `correct` command updates the `State` and `StateCovariance` properties of the object with the estimated values, `CorrectedState` and `CorrectedCovariance`.

The `predict` command returns the prediction of state and state estimation error covariance at the next time step. If your state transition function has additional input arguments  $U_s$ , you specify these as inputs to the `predict` command. The command passes these values to the state transition function.

```
[PredictedState,PredictedCovariance] = predict(obj,Us)
```

The `predict` command updates the `State` and `StateCovariance` properties of the object with the predicted values, `PredictedState` and `PredictedCovariance`.

If the current output measurement exists at a given time step, you can use `correct` and `predict`. If the measurement is missing, you can only use `predict`. For details about how these commands implement the algorithms, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

The order in which you implement the commands depends on the availability of measured data  $y$ ,  $U_s$ , and  $U_m$  for your system:

- `correct` then `predict` — Assume that at time step  $k$ , the value of `obj.State` is  $\hat{x}[k | k - 1]$ . This value is the state of the system at time  $k$ , estimated using measured outputs until time  $k - 1$ . You also have the measured output  $y[k]$  and inputs  $U_s[k]$  and  $U_m[k]$  at the same time step.

Then you first execute the `correct` command with measured system data  $y[k]$  and additional inputs  $U_m[k]$ . The command updates the value of `obj.State` to be  $\hat{x}[k | k]$ , the state estimate for time  $k$ , estimated using measured outputs up to time  $k$ . When you then execute the `predict` command with input  $U_s[k]$ , `obj.State` now stores  $\hat{x}[k + 1 | k]$ . The algorithm uses this state value as an input to the `correct` command in the next time step.

- `predict` then `correct` — Assume that at time step  $k$ , the value of `obj.State` is  $\hat{x}[k - 1 | k - 1]$ . You also have the measured output  $y[k]$  and input  $U_m[k]$  at the same time step but you have  $U_s[k - 1]$  from the previous time step.

Then you first execute the `predict` command with input  $U_s[k - 1]$ . The command updates the value of `obj.State` to  $\hat{x}[k | k - 1]$ . When you then execute the `correct` command with input arguments  $y[k]$  and  $U_m[k]$ , `obj.State` is updated with  $\hat{x}[k | k]$ . The algorithm uses this state value as an input to the `predict` command in the next time step.

Thus, while in both cases the state estimate for time  $k$ ,  $\hat{x}[k | k]$  is the same, if at time  $k$  you do not have access to the current state transition function inputs  $U_s[k]$ , and instead have  $U_s[k - 1]$ , then use `predict` first and then `correct`.

For an example of estimating states using the `predict` and `correct` commands, see “Estimate States Online Using Extended Kalman Filter” on page 2-142.

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”

**See Also**

`clone` | `extendedKalmanFilter` | `predict` | `unscentedKalmanFilter`

**Introduced in R2016b**

## covar

Output and state covariance of system driven by white noise

### Syntax

```
P = covar(sys,W)
[P,Q] = covar(sys,W)
```

### Description

`covar` calculates the stationary covariance of the output  $y$  of an LTI model `sys` driven by Gaussian white noise inputs  $w$ . This function handles both continuous- and discrete-time cases.

`P = covar(sys,W)` returns the steady-state output response covariance

$$P = E(yy^T)$$

given the noise intensity

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

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

`[P,Q] = covar(sys,W)` also returns the steady-state state covariance

$$Q = E(xx^T)$$

when `sys` is a state-space model (otherwise `Q` is set to `[]`).

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

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

## Examples

Compute the output response covariance of the discrete SISO system

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

due to Gaussian white noise of intensity  $W = 5$ . Type

```
sys = tf([2 1],[1 0.2 0.5],0.1);  
p = covar(sys,5)
```

These commands produce the following result.

```
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.

## More About

### Algorithms

Transfer functions and zero-pole-gain models are first converted to state space with `ss`.



For continuous-time state-space models

$$\begin{aligned}\dot{x} &= Ax + Bw \\ y &= Cx + Dw,\end{aligned}$$

the steady-state state covariance  $Q$  is obtained by solving the Lyapunov equation

$$AQ + QA^T + BWB^T = 0.$$

In discrete time, the state covariance  $Q$  solves the discrete Lyapunov equation

$$AQA^T - Q + BWB^T = 0.$$

In both continuous and discrete time, the output response covariance is given by  $P = CQC^T + DWD^T$ . For unstable systems,  $P$  and  $Q$  are infinite. For continuous-time systems with nonzero feedthrough, `covar` returns `Inf` for the output covariance  $P$ .

## References

- [1] Bryson, A.E. and Y.C. Ho, *Applied Optimal Control*, Hemisphere Publishing, 1975, pp. 458-459.

## See Also

dlyap | lyap

Introduced before R2006a

## ctrb

Controllability matrix

### Syntax

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

### Description

Co = ctrb(A,B) returns the controllability matrix:

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

where  $A$  is an  $n$ -by- $n$  matrix,  $B$  is an  $n$ -by- $m$  matrix, and  $Co$  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:

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

The system is controllable if Co has full rank  $n$ .

### Examples

#### Check System Controllability

Define A and B matrices.

```
A = [1  1;
     4 -2];
B = [1 -1;
     1 -1];
```

Compute controllability matrix.

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

Determine the number of uncontrollable states.

```
unco = length(A) - rank(Co)
```

```
unco =
```

```
1
```

The uncontrollable state indicates that `Co` does not have full rank 2. Therefore the system is not controllable.

## Limitations

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

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

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

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

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

## See Also

`ctrbf` | `obsv`

Introduced before R2006a

## ctrbf

Compute controllability staircase form

### Syntax

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

### Description

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

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

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

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

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,  $\bar{A}$ ,  $\bar{B}$ , and  $\bar{C}$ , 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  $\bar{A}$ .

`ctrbf(A,B,C,tol)` uses the tolerance `tol` when calculating the controllable/uncontrollable subspaces. When the tolerance is not specified, it defaults to  $10*n*norm(A,1)*eps$ .

## Examples

Compute the controllability staircase form for

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and locate the uncontrollable mode.

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

$$Abar = \begin{bmatrix} -3.0000 & 0 \\ -3.0000 & 2.0000 \end{bmatrix}$$

$$Bbar = \begin{bmatrix} 0.0000 & 0.0000 \\ 1.4142 & -1.4142 \end{bmatrix}$$

$$Cbar = \begin{bmatrix} -0.7071 & 0.7071 \\ 0.7071 & 0.7071 \end{bmatrix}$$

$$T = \begin{bmatrix} -0.7071 & 0.7071 \\ 0.7071 & 0.7071 \end{bmatrix}$$

$$k = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

The decomposed system  $\mathbf{A}_{\text{bar}}$  shows an uncontrollable mode located at -3 and a controllable mode located at 2.

## More About

### Algorithms

`ctrbf` implements the Staircase Algorithm of [1].

## References

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

### See Also

`ctrb` | `minreal`

**Introduced before R2006a**

# ctrlpref

Set Control System Toolbox preferences

## Syntax

```
ctrlpref
```

## Description

`ctrlpref` opens a Graphical User Interface (GUI) which allows you to change the Control System Toolbox™ preferences. Preferences set in this GUI affect future plots only (existing plots are not altered).

Your preferences are stored to disk (in a system-dependent location) and will be automatically reloaded in future MATLAB sessions using the Control System Toolbox software.

## See Also

Control System Designer | Linear System Analyzer

**Introduced in R2006a**

# d2c

Convert model from discrete to continuous time

## Syntax

```
sysc = d2c(sysd)
sysc = d2c(sysd,method)
sysc = d2c(sysd,opts)
[sysc,G] = d2c(sysd,method,opts)
```

## Description

`sysc = d2c(sysd)` produces a continuous-time model `sysc` that is equivalent to the discrete-time dynamic system model `sysd` using zero-order hold on the inputs.

`sysc = d2c(sysd,method)` uses the specified conversion method `method`.

`sysc = d2c(sysd,opts)` converts `sysd` using the option set `opts`, specified using the `d2cOptions` command.

`[sysc,G] = d2c(sysd,method,opts)` returns a matrix `G` that maps the states `xd[k]` of the state-space model `sysd` to the states `xc(t)` of `sysc`.

## Input Arguments

### **sysd**

Discrete-time dynamic system model

You cannot directly use an `idgrey` model whose `FunctionType` is 'd' with `d2c`. Convert the model into `idss` form first.

### **Default:**

### **method**

Discrete-to-continuous time conversion method, specified as one of the following values:



- 'zoh' — Zero-order hold on the inputs. Assumes the control inputs are piecewise constant over the sampling period.
- 'foh' — Linear interpolation of the inputs (modified first-order hold). Assumes the control inputs are piecewise linear over the sampling period.
- 'tustin' — Bilinear (Tustin) approximation to the derivative.
- 'matched' — Zero-pole matching method of [1] (for SISO systems only).

**Default:** 'zoh'

### opts

Discrete-to-continuous time conversion options, created using `d2cOptions`.

## Output Arguments

### sysc

Continuous-time model of the same type as the input system `sysd`.

When `sysd` is an identified (IDLTI) model, `sysc`:

- Includes both the measured and noise components of `sysd`. If the noise variance is  $\lambda$  in `sysd`, then the continuous-time model `sysc` has an indicated level of noise spectral density equal to  $T_s \cdot \lambda$ .
- Does not include the estimated parameter covariance of `sysd`. If you want to translate the covariance while converting the model, use `translatecov`.

### G

Matrix mapping the states  $x_d[k]$  of the state-space model `sysd` to the states  $x_c(t)$  of `sysc`:

$$x_c(kT_s) = G \begin{bmatrix} x_d[k] \\ u[k] \end{bmatrix}.$$

Given an initial condition  $x_0$  for `sysd` and an initial input  $u_0 = u[0]$ , the corresponding initial condition for `sysc` (assuming  $u[k] = 0$  for  $k < 0$ ) is given by:

$$x_c(0) = G \begin{bmatrix} x_0 \\ u_0 \end{bmatrix}.$$

## Examples

### Convert Discrete-Time Transfer Function to Continuous Time

Create the following discrete-time transfer function:

$$H(z) = \frac{z - 1}{z^2 + z + 0.3}$$

```
H = tf([1 -1],[1 1 0.3],0.1);
```

The sample time of the model is  $T_s = 0.1s$ .

Derive a continuous-time, zero-order-hold equivalent model.

```
Hc = d2c(H)
```

```
Hc =
```

$$\frac{121.7 s + 1.405e-12}{s^2 + 12.04 s + 776.7}$$

```
Continuous-time transfer function.
```

Discretize the resulting model, Hc, with the default zero-order hold method and sample time 0.1s to return the original discrete model, H.

```
c2d(Hc,0.1)
```

```
ans =
```

$$\frac{z - 1}{z^2 + z + 0.3}$$

Sample time: 0.1 seconds  
Discrete-time transfer function.

Use the Tustin approximation method to convert H to a continuous time model.

```
Hc2 = d2c(H, 'tustin');
```

Discretize the resulting model, Hc2, to get back the original discrete-time model, H.

```
c2d(Hc2,0.1, 'tustin');
```

### Convert Identified Discrete-Time Transfer Function to Continuous Time

Estimate a discrete-time transfer function model, and convert it to a continuous-time model.

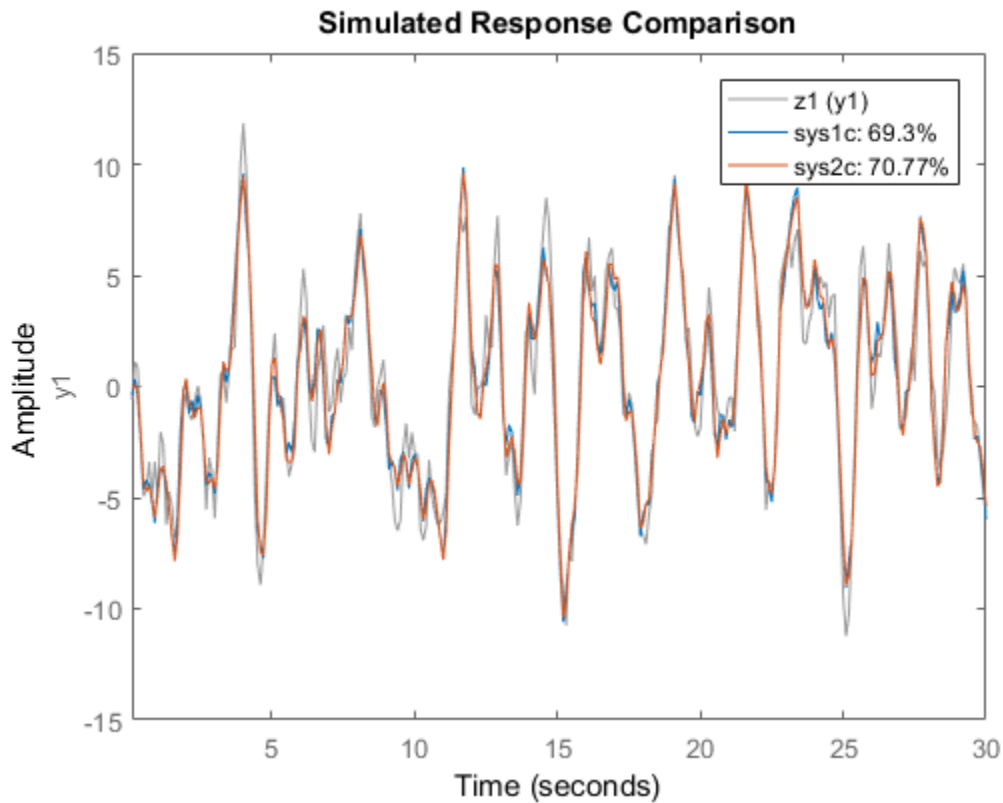
```
load iddata1
sys1d = tfest(z1,2, 'Ts',0.1);
sys1c = d2c(sys1d, 'zoh');
```

Estimate a continuous-time transfer function model.

```
sys2c = tfest(z1,2);
```

Compare the response of `sys1c` and the directly estimated continuous-time model, `sys2c`.

```
compare(z1,sys1c,sys2c)
```



The two systems are almost identical.

### Regenerate Covariance Information After Converting to Continuous-Time Model

Convert an identified discrete-time transfer function model to continuous-time.

```
load iddata1
sysd = tfest(z1,2,'Ts',0.1);
sysc = d2c(sysd,'zoh');
```

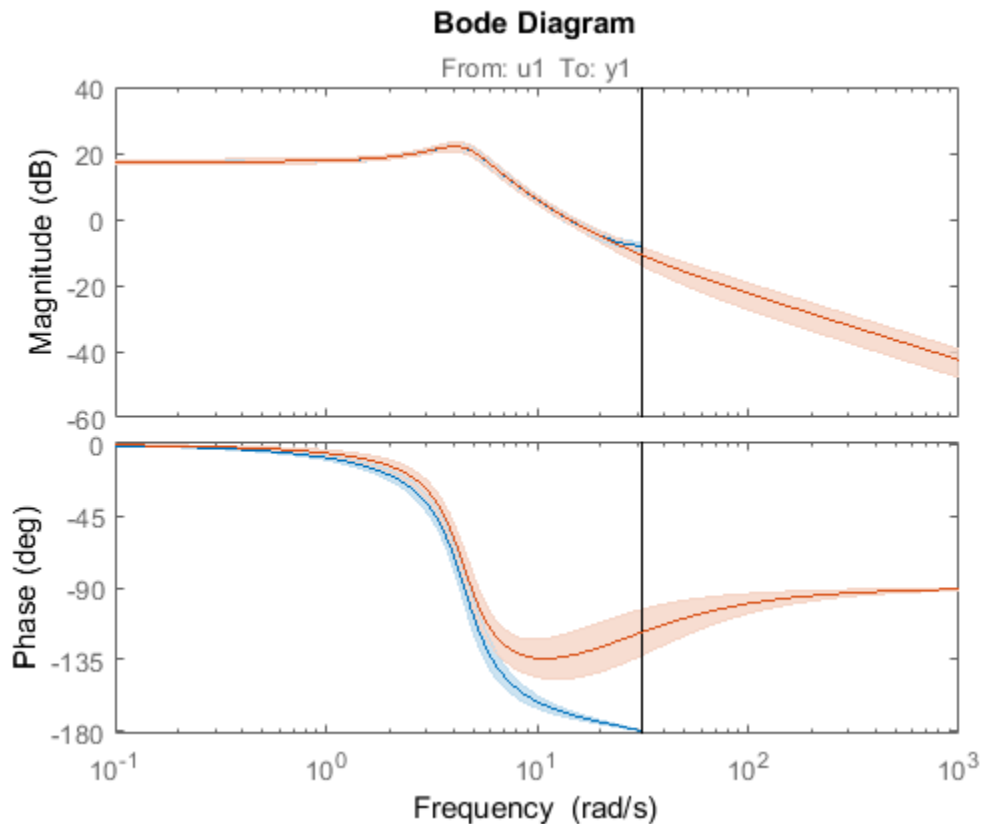
`sys1c` has no covariance information. The `d2c` operation leads to loss of covariance data of identified models.

Regenerate the covariance information using a zero iteration update with the same estimation command and estimation data.

```
opt = tfestOptions;
opt.SearchOption.MaxIter = 0;
sys1c = tfest(z1,sysc,opt);
```

Analyze the effect on frequency-response uncertainty.

```
h = bodeplot(sysd,sys1c);
showConfidence(h,3)
```



The uncertainties of `sys1c` and `sysd` are comparable up to the Nyquist frequency. However, `sys1c` exhibits large uncertainty in the frequency range for which the estimation data does not provide any information.

If you do not have access to the estimation data, use the `translatecov` command which is a Gauss-approximation formula based translation of covariance across model type conversion operations.

### 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]. The model order increases because the matrix logarithm maps real negative poles to complex poles. Single complex poles are not physically meaningful because of their complex time response.

Instead, to ensure that all complex poles of the continuous model come in conjugate pairs, `d2c` replaces negative real poles  $z = -a$  with a pair of complex conjugate poles near  $-a$ . The conversion then yields a continuous model with higher order. For example, to convert the discrete-time transfer function

$$H(z) = \frac{z + 0.2}{(z + 0.5)(z^2 + z + 0.4)}$$

type:

```
Ts = 0.1 % sample time 0.1 s
H = zpk(-0.2, -0.5, 1, Ts) * tf(1, [1 1 0.4], Ts)
Hc = d2c(H)
```

These commands produce the following result.

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)
```

To convert `Hc` back to discrete time, type:

```
c2d(Hc, Ts)
```

yielding

```
Zero/pole/gain:
      (z+0.5) (z+0.2)
-----
(z+0.5)^2 (z^2 + z + 0.4)
```

Sample time: 0.1

This discrete model coincides with  $H(z)$  after canceling the pole/zero pair at  $z = -0.5$ .

## More About

### Tips

- Use the syntax `sysc = d2c(sysd, 'method')` to convert `sysd` using the default options for 'method'. To specify `tustin` conversion with a frequency prewarp (formerly the 'prewarp' method), use the syntax `sysc = d2c(sysd, opts)`. See the `d2cOptions` reference page for more information.

### Algorithms

`d2c` performs the 'zoh' conversion in state space, and relies on the matrix logarithm (see `logm` in the MATLAB documentation).

See “Continuous-Discrete Conversion Methods” for more details on the conversion methods.

## References

- [1] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997..
- [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.

### See Also

`c2d` | `d2d` | `d2cOptions` | `translatecov` | `logm`

**Introduced before R2006a**



# d2cOptions

Create option set for discrete- to continuous-time conversions

## Syntax

```
opts = d2cOptions
opts = d2cOptions(Name,Value)
```

## Description

`opts = d2cOptions` returns the default options for `d2c`.

`opts = d2cOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

**'method'**

Discretization method, specified as one of the following values:

<b>'zoh'</b>	Zero-order hold, where <code>d2c</code> assumes the control inputs are piecewise constant over the sample time <code>Ts</code> .
<b>'foh'</b>	Linear interpolation of the inputs (modified first-order hold). Assumes the control inputs are piecewise linear over the sampling period.
<b>'tustin'</b>	Bilinear (Tustin) approximation. By default, <code>d2c</code> converts with no prewarp. To include prewarp, use the <code>PrewarpFrequency</code> option.
<b>'matched'</b>	Zero-pole matching method. (See [1], p. 224.)

**Default:** `'zoh'`

**'PrewarpFrequency'**

Prewarp frequency for 'tustin' method, specified in rad/TimeUnit, where TimeUnit is the time units, specified in the TimeUnit property, of the discrete-time system. Specify the prewarp frequency as a positive scalar value. A value of 0 corresponds to the 'tustin' method without prewarp.

**Default:** 0

For additional information about conversion methods, see “Continuous-Discrete Conversion Methods”.

## Examples

Convert a discrete-time model to continuous-time using the 'tustin' method with frequency prewarping.

Create the discrete-time transfer function

$$\frac{z+1}{z^2+z+1}$$

```
hd = tf([1 1], [1 1 1], 0.1); % 0.1s sample time
```

To convert to continuous-time, use `d2cOptions` to create the option set.

```
opts = d2cOptions('Method', 'tustin', 'PrewarpFrequency', 20);  
hc = d2c(hd, opts);
```

You can use `opts` to resample additional models using the same options.

## References

- [1] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

## See Also

`d2c`

**Introduced in R2010a**

## d2d

Resample discrete-time model

### Syntax

```
sys1 = d2d(sys, Ts)
sys1 = d2d(sys, Ts, 'method')
sys1 = d2d(sys, Ts, opts)
```

### Description

`sys1 = d2d(sys, Ts)` resamples the discrete-time dynamic system model `sys` to produce an equivalent discrete-time model `sys1` with the new sample time `Ts` (in seconds), using zero-order hold on the inputs.

`sys1 = d2d(sys, Ts, 'method')` uses the specified resampling method `'method'`:

- `'zoh'` — Zero-order hold on the inputs
- `'tustin'` — Bilinear (Tustin) approximation

`sys1 = d2d(sys, Ts, opts)` resamples `sys` using the option set with `d2dOptions`.

### Examples

#### Example 1

Consider the zero-pole-gain model

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

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

```
H = zp(0.7,0.5,1,0.1)
```

```
H2 = d2d(H,0.05)
Zero/pole/gain:
(z-0.8243)
-----
(z-0.7071)
```

Sample time: 0.05

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)
```

Sample time: 0.1

## Example 2

Suppose you estimates a discrete-time model of a sample time commensurate with the estimation data ( $T_s = 0.1$  seconds). However, your deployment application demands a faster sampling frequency ( $T_s = 0.01$  seconds).

```
load iddata1
sys = oe(z1, [2 2 1]);
sysFast = d2d(sys, 0.01, 'zoh')
```

```
bode(sys, sysFast)
```

## More About

### Tips

- Use the syntax `sys1 = d2d(sys, Ts, 'method')` to resample `sys` using the default options for 'method'. To specify `tustin` resampling with a frequency prewarp (formerly the 'prewarp' method), use the syntax `sys1 = d2d(sys, Ts, opts)`. See the `d2dOptions` reference page.
- When `sys` is an identified (IDLTI) model, `sys1` does not include the estimated parameter covariance of `sys`. If you want to translate the covariance while converting the model, use `translatecov`.

**See Also**

`c2d` | `d2c` | `d2dOptions` | `upsample` | `translatecov`

**Introduced before R2006a**

# d2dOptions

Create option set for discrete-time resampling

## Syntax

```
opts = d2dOptions
opts = d2dOptions('OptionName', OptionValue)
```

## Description

`opts = d2dOptions` returns the default options for `d2d`.

`opts = d2dOptions('OptionName', OptionValue)` accepts one or more comma-separated name/value pairs that specify options for the `d2d` command. Specify *OptionName* inside single quotes.

This table summarizes the options that the `d2d` command supports.

## Input Arguments

### Name-Value Pair Arguments

#### 'Method'

Discretization method, specified as one of the following values:

'zoh'	Zero-order hold, where <code>d2d</code> assumes the control inputs are piecewise constant over the sample time <code>Ts</code> .
'tustin'	Bilinear (Tustin) approximation. By default, <code>d2d</code> resamples with no prewarp. To include prewarp, use the <code>PrewarpFrequency</code> option.

**Default:** 'zoh'

**'PrewarpFrequency'**

Prewarp frequency for 'tustin' method, specified in `rad/TimeUnit`, where `TimeUnit` is the time units, specified in the `TimeUnit` property, of the resampled system. Takes positive scalar values. The prewarp frequency must be smaller than the Nyquist frequency before and after resampling. A value of 0 corresponds to the standard 'tustin' method without prewarp.

**Default:** 0

## Examples

Resample a discrete-time model using the 'tustin' method with frequency prewarping.

Create the discrete-time transfer function

$$\frac{z+1}{z^2+z+1}$$

```
h1 = tf([1 1], [1 1 1], 0.1); % 0.1s sample time
```

To resample to a different sample time, use `d2dOptions` to create the option set.

```
opts = d2dOptions('Method', 'tustin', 'PrewarpFrequency', 20);  
h2 = d2d(h1, 0.05, opts);
```

You can use `opts` to resample additional models using the same options.

## See Also

`d2d`

**Introduced in R2010a**



# damp

Natural frequency and damping ratio

## Syntax

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

## Description

`damp(sys)` displays a table of the damping ratio (also called *damping factor*), natural frequency, and time constant of the poles of the linear model `sys`. For a discrete-time model, the table also includes the magnitude of each pole. Frequencies are expressed in units of the reciprocal of the `TimeUnit` property of `sys`. Time constants are expressed in the same units as the `TimeUnit` property of `sys`.

`[Wn,zeta] = damp(sys)` returns the natural frequencies, `Wn`, and damping ratios, `zeta`, of the poles of `sys`.

`[Wn,zeta,P] = damp(sys)` returns the poles of `sys`.

## Input Arguments

### `sys`

Any linear dynamic system model.

## Output Arguments

### `Wn`

Vector containing the natural frequencies of each pole of `sys`, in order of increasing frequency. Frequencies are expressed in units of the reciprocal of the `TimeUnit` property of `sys`.

If `sys` is a discrete-time model with specified sample time, `Wn` contains the natural frequencies of the equivalent continuous-time poles (see “Algorithms” on page 2-177). If `sys` has an unspecified sample time (`Ts = -1`), then the software uses `Ts = 1` and calculates `Wn` accordingly.

**zeta**

Vector containing the damping ratios of each pole of `sys`, in the same order as `Wn`.

If `sys` is a discrete-time model with specified sample time, `zeta` contains the damping ratios of the equivalent continuous-time poles (see “Algorithms” on page 2-177). If `sys` has an unspecified sample time (`Ts = -1`), then the software uses `Ts = 1` and calculates `zeta` accordingly.

**P**

Vector containing the poles of `sys`, in order of increasing natural frequency. `P` is the same as the output of `pole(sys)`, except for the order.

## Examples

### Display Natural Frequency, Damping Ratio, and Poles of Continuous-Time System

Create the following continuous-time transfer function:

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

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

Display the natural frequencies, damping ratios, time constants, and poles of `H`.

```
damp(H)
```

Pole	Damping	Frequency (rad/seconds)	Time Constant (seconds)
-1.00e+00 + 1.41e+00i	5.77e-01	1.73e+00	1.00e+00
-1.00e+00 - 1.41e+00i	5.77e-01	1.73e+00	1.00e+00

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

Calculate the associated time constants.

```
tau = 1./(zeta.*Wn);
```

### Display Natural Frequency, Damping Ratio, and Poles of Discrete-Time System

Create a discrete-time transfer function.

```
H = tf([5 3 1],[1 6 4 4],0.01);
```

Display information about the poles of  $H$ .

```
damp(H)
```

Pole	Magnitude	Damping	Frequency (rad/seconds)	Time Constant (seconds)
-3.02e-01 + 8.06e-01i	8.61e-01	7.74e-02	1.93e+02	6.68e-02
-3.02e-01 - 8.06e-01i	8.61e-01	7.74e-02	1.93e+02	6.68e-02
-5.40e+00	5.40e+00	-4.73e-01	3.57e+02	-5.93e-03

The **Magnitude** column displays the discrete-time pole magnitudes. The **Damping**, **Frequency**, and **Time Constant** columns display values calculated using the equivalent continuous-time poles.

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

Calculate the associated time constants.

```
tau = 1./(zeta.*Wn);
```

## More About

### Algorithms

The natural frequency, time constant, and damping ratio of the system poles are defined in the following table:

	Continuous Time	Discrete Time with Sample Time $T_s$
<b>Pole Location</b>	$s$	$z$
<b>Equivalent Continuous-Time Pole</b>	Not applicable	$s = \frac{\ln(z)}{T_s}$
<b>Natural Frequency</b>	$\omega_n =  s $	$\omega_n =  s  = \left  \frac{\ln(z)}{T_s} \right $
<b>Damping Ratio</b>	$\zeta = -\cos(\angle s)$	$\zeta = -\cos(\angle s) = -\cos(\angle \ln(z))$
<b>Time Constant</b>	$\tau = \frac{1}{\omega_n \zeta}$	$\tau = \frac{1}{\omega_n \zeta}$

**See Also**

eig | esort | dsort | pole | pzmap | zero

Introduced before R2006a

# dare

Solve discrete-time algebraic Riccati equations (DAREs)

## Syntax

```
[X,L,G] = dare(A,B,Q,R)
[X,L,G] = dare(A,B,Q,R,S,E)
[X,L,G,report] = dare(A,B,Q,...)
[X1,X2,L,report] = dare(A,B,Q,...,'factor')
```

## Description

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

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

The `dare` function also returns the gain matrix,  $G = (B^T X B + R)^{-1} B^T X A$ , and the vector  $L$  of closed loop eigenvalues, where

$$L = \text{eig}(A - B * G, E)$$

`[X,L,G] = dare(A,B,Q,R,S,E)` solves the more general discrete-time algebraic Riccati equation,

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

or, equivalently, if  $R$  is nonsingular,

$$E^T X E = F^T X F - F^T X B (B^T X B + R)^{-1} B^T X F + Q - S R^{-1} S^T$$

where  $F = A - B R^{-1} S^T$ . When omitted,  $R$ ,  $S$ , and  $E$  are set to the default values  $R=I$ ,  $S=0$ , and  $E=I$ .

The `dare` function returns the corresponding gain matrix

$$G = (B^T X B + R)^{-1} (B^T X A + S^T)$$

and a vector `L` of closed-loop eigenvalues, where

$$L = \text{eig}(A - B * G, E)$$

`[X, L, G, report] = dare(A, B, Q, ...)` returns a diagnosis `report` with value:

- -1 when the associated symplectic pencil has eigenvalues on or very near the unit circle
- -2 when there is no finite stabilizing solution `X`
- The Frobenius norm if `X` exists and is finite

`[X1, X2, L, report] = dare(A, B, Q, ..., 'factor')` returns two matrices, `X1` and `X2`, and a diagonal scaling matrix `D` such that  $X = D * (X2 / X1) * D$ . The vector `L` contains the closed-loop eigenvalues. All outputs are empty when the associated Symplectic matrix has eigenvalues on the unit circle.

## 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

$$\begin{bmatrix} Q & S \\ S^T & R \end{bmatrix} > 0$$

## More About

### Algorithms

`dare` implements the algorithms described in [1]. It uses the QZ algorithm to deflate the extended symplectic pencil and compute its stable invariant subspace.

## 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.

## See Also

care | dlyap | gdare

**Introduced before R2006a**

# db2mag

Convert decibels (dB) to magnitude

## Syntax

```
y = db2mag(ydb)
```

## Description

`y = db2mag(ydb)` returns the corresponding magnitude  $y$  for a given decibel (dB) value  $ydb$ . The relationship between magnitude and decibels is  $ydb = 20 * \log_{10}(y)$ .

## See Also

`mag2db`

**Introduced in R2008a**



## dcbgain

Low-frequency (DC) gain of LTI system

### Syntax

```
k = dcbgain(sys)
```

### Description

`k = dcbgain(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

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

## Examples

### Example 1

To compute the DC gain of the MIMO transfer function

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

type

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

to get the result:

```
ans =  
    1.0000    -0.3333  
    1.0000    -0.6667
```

### Example 2

To compute the DC gain of an identified process model, type;

```
load iddata1  
sys = idproc('p1d');  
syse = procest(z1, sys)
```

```
dcgain(syse)
```

The DC gain is stored same as `syse.Kp`.

## More About

### Tips

The DC gain is infinite for systems with integrators.

### See Also

`norm` | `evalfr`

Introduced before R2006a

## delay2z

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

---

**Note:** delay2z has been removed. Use absorbDelay instead.

---

**Introduced before R2006a**

## delays

Create state-space models with delayed inputs, outputs, and states

### Syntax

```
sys=delays(A,B,C,D,delayterms)
sys=delays(A,B,C,D,ts,delayterms)
```

### Description

`sys=delays(A,B,C,D,delayterms)` constructs a continuous-time state-space model of the form:

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

where  $t_j, j=1, \dots, N$  are time delays expressed in seconds. `delayterms` is a struct array with fields `delay`, `a`, `b`, `c`, `d` where the fields of `delayterms(j)` contain the values of  $t_j$ ,  $A_j$ ,  $B_j$ ,  $C_j$ , and  $D_j$ , respectively. The resulting model `sys` is a state-space (SS) model with internal delays.

`sys=delays(A,B,C,D,ts,delayterms)` constructs the discrete-time counterpart:

$$x[k+1] = Ax[k] + Bu[k] + \sum_{j=1}^N \{A_j x[k - n_j] + B_j u[k - n_j]\}$$
$$y[k] = Cx[k] + Du[k] + \sum_{j=1}^N \{C_j x[k - n_j] + D_j u[k - n_j]\}$$

where  $N_j, j=1, \dots, N$  are time delays expressed as integer multiples of the sample time `ts`.

## Examples

To create the model:

$$\frac{dx}{dt} = x(t) - x(t-1.2) + 2u(t-0.5)$$

$$y(t) = x(t-0.5) + u(t)$$

type

```
DelayT(1) = struct('delay',0.5,'a',0,'b',2,'c',1,'d',0);
DelayT(2) = struct('delay',1.2,'a',-1,'b',0,'c',0,'d',0);
sys = delayss(1,0,0,1,DelayT)
```

```
a =
      x1
x1    0
```

```
b =
      u1
x1    2
```

```
c =
      x1
y1    1
```

```
d =
      u1
y1    1
```

(values computed with all internal delays set to zero)

Internal delays: 0.5 0.5 1.2

Continuous-time model.

## See Also

getdelaymodel | ss

**Introduced in R2007a**

## dlqr

Linear-quadratic (LQ) state-feedback regulator for discrete-time state-space system

### Syntax

`[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

$$J(u) = \sum_{n=1}^{\infty} (x[n]^T Qx[n] + u[n]^T Ru[n] + 2x[n]^T Nu[n])$$

for the discrete-time state-space mode

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

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

In addition to the state-feedback gain  $K$ , `dlqr` returns the 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  $e = \text{eig}(A - B \cdot K)$ . 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 | lqgreg | lqr | lqrd | lqry

Introduced before R2006a

## dlyap

Solve discrete-time Lyapunov equations

### Syntax

```
X = dlyap(A,Q)
X = dlyap(A,B,C)
X = dlyap(A,Q, [],E)
```

### Description

$X = \text{dlyap}(A,Q)$  solves the discrete-time Lyapunov equation  $AXA^T - X + Q = 0$ ,

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.

$X = \text{dlyap}(A,B,C)$  solves the Sylvester equation  $AXB - X + C = 0$ ,

where  $A$ ,  $B$ , and  $C$  must have compatible dimensions but need not be square.

$X = \text{dlyap}(A,Q,[],E)$  solves the generalized discrete-time Lyapunov equation  $AXA^T - EXE^T + Q = 0$ ,

where  $Q$  is a symmetric matrix. The empty square brackets, `[]`, are mandatory. If you place any values inside them, the function will error out.

### Diagnostics

The discrete-time Lyapunov equation has a (unique) solution if the eigenvalues  $a_1, a_2, \dots, a_N$  of  $A$  satisfy  $a_i a_j \neq 1$  for all  $(i, j)$ .

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

Solution does not exist or is not unique.



## More About

### Algorithms

dlyap uses SLICOT routines SB03MD and SG03AD for Lyapunov equations and SB04QD (SLICOT) for Sylvester equations.

## References

- [1] Barraud, A.Y., "A numerical algorithm to solve  $A X A - X = Q$ ," *IEEE Trans. Auto. Contr.*, AC-22, pp. 883-885, 1977.
- [2] Bartels, R.H. and G.W. Stewart, "Solution of the Matrix Equation  $A X + X B = C$ ," *Comm. of the ACM*, Vol. 15, No. 9, 1972.
- [3] Hammarling, S.J., "Numerical solution of the stable, non-negative definite Lyapunov equation," *IMA J. Num. Anal.*, Vol. 2, pp. 303-325, 1982.
- [4] Higham, N.J., "FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation," *A.C.M. Trans. Math. Soft.*, Vol. 14, No. 4, pp. 381-396, 1988.
- [5] Penzl, T., "Numerical solution of generalized Lyapunov equations," *Advances in Comp. Math.*, Vol. 8, pp. 33-48, 1998.
- [6] Golub, G.H., Nash, S. and Van Loan, C.F. "A Hessenberg-Schur method for the problem  $A X + X B = C$ ," *IEEE Trans. Auto. Contr.*, AC-24, pp. 909-913, 1979.
- [7] Sima, V. C, "Algorithms for Linear-quadratic Optimization," Marcel Dekker, Inc., New York, 1996.

### See Also

covar | lyap

Introduced before R2006a

## dlyapchol

Square-root solver for discrete-time Lyapunov equations

### Syntax

```
R = dlyapchol(A,B)
X = dlyapchol(A,B,E)
```

### Description

`R = dlyapchol(A,B)` computes a Cholesky factorization  $X = R' * R$  of the solution  $X$  to the Lyapunov matrix equation:

$$A * X * A' - X + B * B' = 0$$

All eigenvalues of  $A$  matrix must lie in the open unit disk for  $R$  to exist.

`X = dlyapchol(A,B,E)` computes a Cholesky factorization  $X = R' * R$  of  $X$  solving the Sylvester equation

$$A * X * A' - E * X * E' + B * B' = 0$$

All generalized eigenvalues of  $(A,E)$  must lie in the open unit disk for  $R$  to exist.

### More About

#### Algorithms

`dlyapchol` uses SLICOT routines SB03OD and SG03BD.

### 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] Hammarling, S.J., "Numerical solution of the stable, non-negative definite Lyapunov equation," *IMA J. Num. Anal.*, Vol. 2, pp. 303-325, 1982.
- [3] Penzl, T., "Numerical solution of generalized Lyapunov equations," *Advances in Comp. Math.*, Vol. 8, pp. 33-48, 1998.

## See Also

dlyap | lyapchol

**Introduced before R2006a**

## drss

Generate random discrete test model

### Syntax

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

### Description

`sys = drss(n)` generates an  $n$ -th order model with one input and one output, and returns the model in the state-space object `sys`. The poles of `sys` are random and stable with the possible exception of poles at  $z = 1$  (integrators).

`drss(n,p)` generates an  $n$ -th order model with one input and  $p$  outputs.

`drss(n,p,m)` generates an  $n$ -th order model with  $p$  outputs and  $m$  inputs.

`drss(n,p,m,s1,...sn)` generates a  $s1$ -by- $sn$  array of  $n$ -th order models with  $m$  inputs and  $p$  outputs.

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

### Examples

Generate a discrete LTI system with three states, four outputs, and two inputs.

```
sys = drss(3,4,2)
```

```
a =
      x1      x2      x3
x1  0.4766  0.1102 -0.7222
```

```
x2  0.1102  0.9115  0.1628
x3 -0.7222  0.1628 -0.202
```

b =

```
      u1      u2
x1 -0.4326  0.2877
x2      -0      -0
x3      0      1.191
```

c =

```
      x1      x2      x3
y1  1.189 -0.1867 -0
y2 -0.03763  0.7258  0.1139
y3  0.3273 -0.5883  1.067
y4  0.1746  2.183  0
```

d =

```
      u1      u2
y1 -0.09565  0
y2 -0.8323  1.624
y3  0.2944 -0.6918
y4      -0  0.858
```

Sample time: unspecified  
Discrete-time model.

## See Also

[rss](#) | [tf](#) | [zpk](#)

Introduced before R2006a

## dsort

Sort discrete-time poles by magnitude

### Syntax

```
dsort  
[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.

### Examples

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  $\mathbf{p}$  must appear in complex conjugate pairs.

## See Also

eig | esort | sort | pole | pzmap | zero

**Introduced before R2006a**

## **dss**

Create 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)
```

### **Description**

`sys = dss(A,B,C,D,E)` creates the continuous-time descriptor state-space model

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

The output `sys` is an SS model storing the model data (see “State-Space Models”). Note that `ss` produces the same type of object. If the matrix  $\mathbf{D} = \mathbf{0}$ , you 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 properties inherited from the LTI model `ltisys` (including the sample time).

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.

## Examples

The command

```
sys = dss(1,2,3,4,5,'inputdelay',0.1,'inputname','voltage',...  
          'notes','Just an example');
```

creates the model

$$\begin{aligned}5\dot{x} &= x + 2u \\ y &= 3x + 4u\end{aligned}$$

with a 0.1 second input delay. The input is labeled 'voltage', and a note is attached to tell you that this is just an example.

## See Also

`dssdata` | `get` | `set` | `ss`

**Introduced before R2006a**

## dssdata

Extract 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)` returns the values of the A, B, C, D, and E matrices for the descriptor state-space model `sys` (see `dss`). `dssdata` equals `ssdata` for regular state-space models (i.e., when  $E=I$ ).

If `sys` has internal delays, A, B, C, D are obtained by first setting all internal delays to zero (creating a zero-order Padé approximation). For some systems, setting delays to zero creates singular algebraic loops, which result in either improper or ill-defined, zero-delay approximations. For these systems, `dssdata` cannot display the matrices and returns an error. This error does not imply a problem with the model `sys` itself.

`[A,B,C,D,E,Ts] = dssdata(sys)` also returns the sample time `Ts`.

You can access other properties of `sys` using `get` or direct structure-like referencing (e.g., `sys.Ts`).

For arrays of SS models with variable order, use the syntax

```
[A,B,C,D,E] = dssdata(sys, 'cell')
```

to extract the state-space matrices of each model as separate cells in the cell arrays A, B, C, D, and E.

### See Also

`dss` | `get` | `getdelaymodel` | `ssdata`

**Introduced before R2006a**

## esort

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.

### Examples

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` | `eig` | `pole` | `pzmap` | `zero`

**Introduced before R2006a**

## estim

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

$$\begin{aligned}\dot{x} &= Ax + Bw \\ y &= Cx + Dw\end{aligned}$$

`estim` uses the following equations to generate a plant output estimate  $\hat{y}$  and a state estimate  $\hat{x}$ , which are estimates of  $y(t)=C$  and  $x(t)$ , respectively:

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

For a discrete-time plant `sys` with the following equations:

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

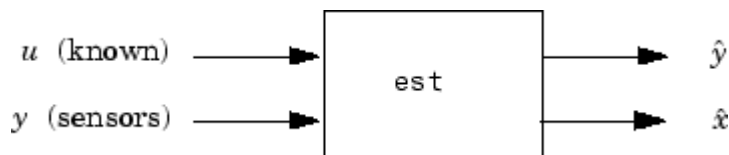
`estim` uses estimator equations similar to those for continuous-time to generate a plant output estimate  $\hat{y}[n | n - 1]$  and a state estimate  $\hat{x}[n | n - 1]$ , which are estimates of  $y[n]$  and  $x[n]$ , respectively. These estimates are based on past measurements up to  $y[n-1]$ .

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

$$\begin{aligned} \dot{x} &= Ax + B_1w + B_2u \\ \begin{bmatrix} z \\ y \end{bmatrix} &= \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} x + \begin{bmatrix} D_{11} \\ D_{21} \end{bmatrix} w + \begin{bmatrix} D_{12} \\ D_{22} \end{bmatrix} u \end{aligned}$$

The index vectors `sensors` and `known` specify which outputs of `sys` are measured ( $y$ ), and which inputs of `sys` are known ( $u$ ). The resulting estimator `est`, found using the following equations, uses both  $u$  and  $y$  to produce the output and state estimates.

$$\begin{aligned} \dot{\hat{x}} &= A\hat{x} + B_2u + L(y - C_2\hat{x} - D_{22}u) \\ \begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} &= \begin{bmatrix} C_2 \\ I \end{bmatrix} \hat{x} + \begin{bmatrix} D_{22} \\ 0 \end{bmatrix} u \end{aligned}$$



## Examples

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.

## More About

### Tips

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.

### See Also

`kalman` | `ss` | `ssest` | `predict` | `place` | `reg` | `kalmd` | `lqgreg`

**Introduced before R2006a**

## evalfr

Evaluate frequency response at given 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.

## Examples

### Example 1

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)
```

to get the result:

```
ans =
```



---

2.3077e-01 + 1.5385e-01i

## Example 2

To evaluate the frequency response of a continuous-time IDTF model at frequency  $w = 0.1$  rad/s, type:

```
sys = idtf(1,[1 2 1]);  
w = 0.1;  
s = 1j*w;  
evalfr(sys, s)
```

The result is same as `freqresp(sys, w)`.

## Limitations

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

## See Also

`freqresp` | `bode` | `sigma`

**Introduced before R2006a**

## evalSpec

Evaluate tuning requirements for tuned control system

### Syntax

```
[Hspec, fval] = evalSpec(Req, T)  
[Hspec, fval] = evalSpec(Req, T, Info)
```

### Description

`[Hspec, fval] = evalSpec(Req, T)` returns the normalized value, `fval`, of a tuning requirement evaluated for a tuned control system `T`. The `evalSpec` command also returns the transfer function, `Hspec`, used to compute this value. `evalSpec` applies the solver's loop scaling when evaluating MIMO open-loop requirements such as loop shapes or stability margins. This application ensures consistency with the tuning goal value computed by `systune` or `looptune`.

`[Hspec, fval] = evalSpec(Req, T, Info)` uses the `Info` structure returned by `systune` to maintain consistency after modifying `T` with `usample`, `usubs`, or `setBlockValue`.

### Examples

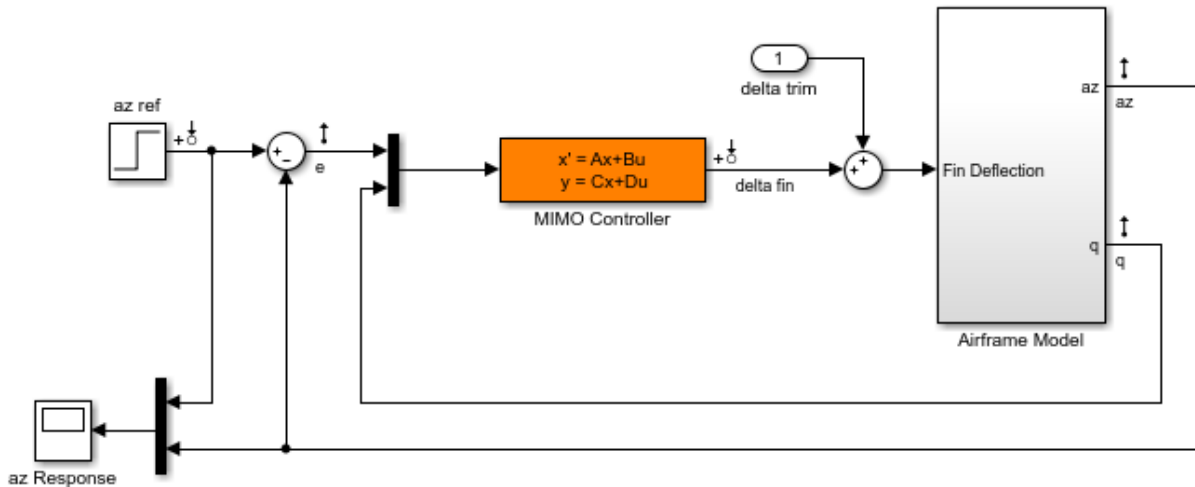
#### Evaluate Requirements for Tuned System

Tune a control system with `systune`, and evaluate the tuning requirements with `evalSpec`.

Open the Simulink® model `rct_airframe2`.

```
open_system('rct_airframe2')
```

### Two-loop autopilot for controlling the vertical acceleration of an airframe



Create tracking, roll-off, stability margin, and disturbance rejection requirements for tuning the control system.

```
Req1 = TuningGoal.Tracking('az ref','az',1);
Req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0]));
Req3 = TuningGoal.Margins('delta fin',7,45);
MaxGain = frd([2 200 200],[0.02 2 200]);
Req4 = TuningGoal.Gain('delta fin','az',MaxGain);
```

Create an `sITuner` interface and tune the model using these tuning requirements.

```
ST0 = sITuner('rct_airframe2','MIMO Controller');
rng default
[ST1,fSoft] = systune(ST0,[Req1,Req2,Req3,Req4]);
```

```
Final: Soft = 1.15, Hard = -Inf, Iterations = 80
```

ST1 is a tuned version of the `sITuner` interface to the control system. ST1 contains the tuned values of the tunable parameters of the MIMO controller in the model.

Evaluate the margin requirement for the tuned system.

```
[hspec,fval] = evalSpec(Req3,ST1);
fval
```

```
fval =  
    0.5434
```

The normalized value of the requirement is less than 1, indicating that the tuned system satisfies the margin requirement. For more information about how the normalized value of this requirement is calculated, see the `TuningGoal.Margins` reference page.

Evaluate the tracking requirement for the tuned system.

```
[hspec,fval] = evalSpec(Req1,ST1);  
fval
```

```
fval =  
    1.1460
```

The tracking requirement is nearly met, but the value exceeds 1, indicating a small violation. To further assess the violation, you can use `viewSpec` to examine the requirement against the tuned control system as a function of frequency.

## Input Arguments

### **Req** — Tuning requirement to evaluate

`TuningGoal` requirement object | vector of `TuningGoal` objects

Tuning requirement to evaluate, specified as a `TuningGoal` requirement object or vector of `TuningGoal` objects. `TuningGoal` requirement objects include:

- `TuningGoal.Tracking`
- `TuningGoal.Gain`
- `TuningGoal.WeightedGain`
- `TuningGoal.Variance`
- `TuningGoal.WeightedVariance`
- `TuningGoal.LoopShape`
- `TuningGoal.Margins`

- `TuningGoal.Poles`
- `TuningGoal.ControllerPoles`

### **T — Tuned control system**

generalized state-space model | `sITuner` interface object

Tuned control system, specified as a generalized state-space (`genss`) model or an `sITuner` interface to a Simulink model. `T` is typically the result of using the tuning requirement to tune control system parameters with `systune`.

Example: `[T,fSoft,gHard,Info] = systune(T0,SoftReq,HardReq)`, where `T0` is a tunable `genss` model

Example: `[T,fSoft,gHard,Info] = systune(ST0,SoftReq,HardReq)`, where `ST0` is a `sITuner` interface object

### **Info — System information**

data structure returned by `systune`

System information, specified as the data structure returned by `systune` when you use that command to tune a control system. Use `Info` to maintain consistency after modifying `T` with `usample`, `usubs`, or `setBlockValue`.

## **Output Arguments**

### **Hspec — transfer function associated with requirement**

state-space model

Transfer function associated with the tuning requirement, returned as a state-space (`ss`) model. `evalSpec` uses `Hspec` to compute the evaluated requirement, `fval`.

For example, suppose `Req` is a `TuningGoal` gain requirement that limits the gain,  $H(s)$ , between some specified input and output to the gain profile,  $w(s)$ . In that case, `Hspec` is given by:

$$H_{spec}(s) = \frac{1}{w(s)} H(s).$$

`fval` is the peak gain of `Hspec`. If  $H(s)$  satisfies the tuning requirement, `fval`  $\leq$  1.

For more information about the transfer function associated with the requirement, see the reference page for each `TuningGoal` requirement object.

### **fval** — Normalized value of tuning requirement

positive scalar

Normalized value of tuning requirement, returned as a positive scalar. The normalized value is a measure of how closely the requirement is met in the tuned system. The tuning requirement is satisfied if `fval < 1`. For information about how each type of `TuningGoal` requirement is converted into a normalized value, see the reference page for each `TuningGoal` requirement object.

## More About

- “Generalized Models”

## See Also

`TuningGoal.Tracking` | `TuningGoal.Sensitivity` | `TuningGoal.Overshoot` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain` | `TuningGoal.Gain` | `TuningGoal.Margins` | `TuningGoal.WeightedGain` | `TuningGoal.Variance` | `TuningGoal.WeightedVariance` | `TuningGoal.LoopShape` | `TuningGoal.Poles` | `TuningGoal.ControllerPoles` | `genss` | `sITuner` | `system` | `system` (for `sITuner`) | `viewSpec`

**Introduced in R2012b**

# evalSurf

Evaluate gain surfaces at specific design points

## Syntax

```
GV = evalSurf(GS,X)
GV = evalSurf(GS,X1,...,XM)
GV = evalSurf( ____,gridflag)
```

## Description

`GV = evalSurf(GS,X)` evaluates a gain surface at the list of points specified in the array `X`. A point is a combination of scheduling-variable values. Thus `X` is an  $N$ -by- $M$  array, where  $N$  is the number of points at which to evaluate the gain, and  $M$  is the number of scheduling variables in `GS`.

`GV = evalSurf(GS,X1,...,XM)` evaluates the gain surface over the rectangular grid generated by the vectors `X1`, ..., `XM`. Each vector contains values for one scheduling variable of `GS`.

`GV = evalSurf( ____,gridflag)` specifies the layout of `GV`.

## Examples

### Evaluate 1-D Gain Surface at Specified Values

Create a gain surface with one scheduling variable and evaluate the gain at a list of scheduling-variable values.

When you create a gain surface using `tunableSurface`, you specify design points at which the gain coefficients are tuned. These points are the typically the scheduling-variable values at which you have sampled or linearized the plant. However, you might want to implement the gain surface as a lookup table with breakpoints that are different from the specified design points. In this example, you create a gain surface with a set of

design points and then evaluate the surface using a different set of scheduling variable values.

Create a scalar gain that varies as a quadratic function of one scheduling variable,  $t$ . Suppose that you have linearized your plant every five seconds from  $t = 0$  to  $t = 40$ .

```
t = 0:5:40;
domain = struct('t',t);
shapefcn = @(x) [x,x^2];
GS = tunableSurface('GS',1,domain,shapefcn);
```

Typically, you would tune the coefficients as part of a control system. For this example, instead of tuning, manually set the coefficients to non-zero values.

```
GS = setData(GS,[12.1,4.2,2]);
```

Evaluate the gain surface at a different set of time values.

```
tvals = [0,4,11,18,25,32,39,42]; % eight values
GV = evalSurf(GS,tvals)
```

```
GV =
    9.9000
   10.0200
   10.6150
   11.7000
   13.2750
   15.3400
   17.8950
   19.1400
```

GV is an 8-by-1 array. You can use `tvals` and `GV` to implement the variable gain as a lookup table.

### Evaluate Gain Surface on Grid of Values

Evaluate a gain surface with two scheduling variables over a grid of values of those variables.

When you create a gain surface using `tunableSurface`, you specify design points at which the gain coefficients are tuned. These points are the typically the scheduling-



variable values at which you have sampled or linearized the plant. However, you might want to implement the gain surface as a lookup table with breakpoints that are different from the specified design points. In this example, you create a gain surface with a set of design points and then evaluate the surface using a different set of scheduling-variable values.

Create a scalar-valued gain surface that is a bilinear function of two independent variables,  $\alpha$  and  $V$ .

```
[alpha,V] = ndgrid(0:1.5:15,300:30:600);
domain = struct('alpha',alpha,'V',V);
shapefcn = @(x,y) [x,y,x*y];
GS = tunableSurface('GS',1,domain,shapefcn);
```

Typically, you would tune the coefficients as part of a control system. For this example, instead of tuning, manually set the coefficients to non-zero values.

```
GS = setData(GS,[100,28,40,10]);
```

Evaluate the gain at selected values of  $\alpha$  and  $V$ .

```
alpha_vec = [7:1:13]; % N1 = 7 points
V_vec = [400:25:625]; % N2 = 10 points
GV = evalSurf(GS,alpha_vec,V_vec);
```

The breakpoints at which you evaluate the gain surface need not fall within the range specified by `domain`. However, if you attempt to evaluate the gain too far outside the range used for tuning, the software issues a warning.

The breakpoints also need not be regularly spaced. `evalSurf` evaluates the gain surface over the grid formed by `ndgrid(alpha_vec,V_vec)`. Examine the dimensions of the resulting array.

```
size(GV)
```

```
ans =
```

```
7 10
```

By default, the grid dimensions  $N1$ -by- $N2$  are first in the array, followed by the gain dimensions. `GS` is scalar-valued gain, so the dimensions of `GV` are `[7,10,1,1]`, or equivalently `[7,10]`.

The value in each location of `GV` is the gain evaluated at the corresponding `(alpha_vec, V_vec)` pair in the grid. For example, `GV(2,3)` is the gain evaluated at `(alpha_vec(2), V_vec(3))` or `(8,450)`.

### Evaluate Array-Valued Gain Surface

Evaluate an array-valued gain surface with two scheduling variables over a grid of values of those variables.

Create a vector-valued gain that has two scheduling variables.

```
[alpha,V] = ndgrid(0:1.5:15,300:30:600);  
domain = struct('alpha',alpha,'V',V);  
shapefcn = @(x,y) [x,y,x*y];  
GS = tunableSurface('GS',ones(2,2),domain,shapefcn);
```

Setting the initial constant coefficient to `ones(2,2)` causes `tunableSurface` to generate a 2-by-2 gain matrix. Each entry in that matrix is an independently tunable gain surface that is a bilinear function of two scheduling variables. In other words, the gain surface is given by:

$$GS = K_0 + K_1\alpha + K_2V + K_3\alpha V,$$

where each of the coefficients  $K_0, \dots, K_3$  is itself a 2-by-2 matrix.

Typically, you would tune the coefficients of those gain surfaces as part of a control system. For this example, instead of tuning, manually set the coefficients to non-zero values.

```
K0 = 10*rand(2);  
K1 = 10*rand(2);  
K2 = 10*rand(2);  
K3 = 10*rand(2);
```

The `tunableSurface` object stores array-valued coefficients by concatenating them into a 2-by-8 array (see the `tunableSurface` reference page). Therefore, concatenate these values of  $K_0, \dots, K_3$  to change the coefficients of `GS`.

```
GS = setData(GS,[K0 K1 K2 K3]);
```

Now evaluate the gain surface at selected values of the scheduling variables.

```
alpha_vec = [7:1:13]; % N1 = 7 points
```

```
V_vec = [400:25:625]; % N2 = 10 points
GV = evalSurf(GS,alpha_vec,V_vec,'gridlast');
```

The 'gridlast' orders the array GV such that the dimensions of the grid of gain values, 7-by-10, are last. The dimensions of the gain array itself, 2-by-2, are first.

```
size(GV)
```

```
ans =
```

```
     2     2     7    10
```

## Input Arguments

### GS — Gain surface

tunableSurface object

Gain surface to evaluate, specified as a `tunableSurface` object. GS can have any number of scheduling variables, and can be scalar-valued or array-valued.

### X — Points

array

Points at which to evaluate the gain surface, specified as an array. A point is a combination of scheduling-variable values. X has dimensions  $N$ -by- $M$ , where  $M$  is the number of scheduling variables in GS and  $N$  is the number of points at which to evaluate GS. Thus, X is a list of scheduling-variable-value combinations at which to evaluate the gain. For example, suppose GS has two scheduling variables, **a** and **b**, and you want to evaluate GS at 10 (a,b) pairs. In that case, X is a 10-by-2 array that lists the (a,b). The points in X need not match the design points in `GS.SamplingGrid`.

### X1, ..., XM — Scheduling-variable values

arrays

Scheduling-variable values at which to evaluate the gain surface, specified as  $M$  arrays, where  $M$  is the number of scheduling variables in GS. For example, if GS has two scheduling variables, **a** and **b**, then X1 and X2 are vectors of **a** and **b** values, respectively. The gain surface is evaluated over the grid `ndgrid(X1,X2)`. The values in that grid need not match the design points in `GS.SamplingGrid`.

**gridflag — Layout of output array**

'gridfirst' (default) | 'gridlast'

Layout of output array, specified as either 'gridfirst' or 'gridlast'.

- 'gridfirst' — GV is of size  $[N1, \dots, NM, Ny, Nu]$  with the grid dimensions first and the gain dimensions last. This layout is the natural format for a scalar gain, where  $Ny = Nu = 1$ .
- 'gridlast' — GV is of size  $[Ny, Nu, N1, \dots, NM]$  with the gain dimensions first. This format is more readable for matrix-valued gains.

## Output Arguments

**GV — Gain values**

array

Gain values, returned as an array. GV contains the gain evaluated at the points (scheduling-variable values) specified by X or X1, . . . , XM. The size of GV depends on the number of scheduling variables in GS, the I/O dimensions of the gain defined by GS, and the value of gridflag.

If you compute the gain at a list of N points specified in an array X, then the size of GV is  $[N, Ny, Nu]$ . Here,  $[Ny, Nu]$  are the I/O dimensions of the gain. For example, suppose GS is a scalar gain surface with two scheduling variables, a and b, and X is a 10-by-2 array containing 10 (a, b) pairs. Then GV is a column vector of ten values.

If you compute the gain over a grid specified by vectors X1, . . . , XM, then the dimensions of GV depend on the value of gridflag.

- gridflag = 'gridfirst' (default) — The size of GV is  $[N1, \dots, NM, Ny, Nu]$ . Each  $Ni$  is the length of  $Xi$ , the number of values of the i-th scheduling variable. For example, suppose GS is a scalar gain surface with two scheduling variables, a and b, and X1 and X2 are vectors of 4 a values and 5 b values, respectively. Then, the size of GV is  $[4, 5, 1, 1]$  or equivalently,  $[4, 5]$ . Or, if GS is a three-output, two-input vector-valued gain, then the size of GV is  $[4, 5, 3, 2]$ .
- gridflag = 'gridlast' — The size of GV is  $[Ny, Nu, N1, \dots, NM]$ . For example, suppose GS is a scalar gain surface with two scheduling variables, a and b, and X1 and X2 are vectors of 4 a values and 5 b values, respectively. Then, the size of GV is  $[1, 1, 4, 5]$ . Or, if GS is a three-output, two-input vector-valued gain, then the size of GV is  $[3, 2, 4, 5]$ .

## More About

### Tips

- Use `evalSurf` to turn tuned gain surfaces into lookup tables. Set  $X_1, \dots, X_M$  to the desired table breakpoints and use `GV` as table data. The table breakpoints do not need to match the design points used for tuning `GS`.

### See Also

`getData` | `setData` | `tunableSurface` | `viewSurf`

**Introduced in R2015b**

## lti/exp

Create pure continuous-time delays

### Syntax

```
d = exp(tau,s)
```

### Description

`d = exp(tau,s)` creates pure continuous-time delays. The transfer function of a pure delay `tau` is:

$$d(s) = \exp(-\tau*s)$$

You can specify this transfer function using `exp`.

```
s = zpk('s')  
d = exp(-tau*s)
```

More generally, given a 2D array `M`,

```
s = zpk('s')  
D = exp(-M*s)
```

creates an array `D` of pure delays where  $D(i,j) = \exp(-M(i,j)s)$ .

All entries of `M` should be non negative for causality.

### See Also

`zpk` | `tf`

**Introduced in R2006a**

# extendedKalmanFilter

Create extended Kalman filter object for online state estimation

## Syntax

```
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,  
InitialState)  
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,  
InitialState,Name,Value)  
obj = extendedKalmanFilter(Name,Value)
```

## Description

`obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState)` creates an extended Kalman filter object for online state estimation of a discrete-time nonlinear system. `StateTransitionFcn` is a function that calculates the state of the system at time  $k$ , given the state vector at time  $k-1$ . `MeasurementFcn` is a function that calculates the output measurement of the system at time  $k$ , given the state at time  $k$ . `InitialState` specifies the initial value of the state estimates.

After creating the object, use the `correct` and `predict` commands to update state estimates and state estimation error covariance values using a first-order discrete-time extended Kalman filter algorithm and real-time data.

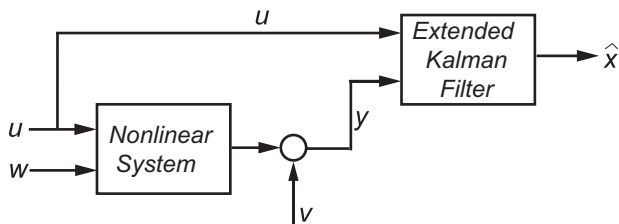
`obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState,Name,Value)` specifies additional attributes of the extended Kalman filter object using one or more `Name,Value` pair arguments.

`obj = extendedKalmanFilter(Name,Value)` creates an extended Kalman filter object with properties specified using one or more `Name,Value` pair arguments. The properties that you do not specify retain their default value.

## Object Description

`extendedKalmanFilter` creates an object for online state estimation of a discrete-time nonlinear system using the first-order discrete-time extended Kalman filter algorithm.

Consider a plant with states  $x$ , input  $u$ , output  $y$ , process noise  $w$ , and measurement noise  $v$ . Assume that you can represent the plant as a nonlinear system.



The algorithm computes the state estimates  $\hat{x}$  of the nonlinear system using state transition and measurement functions specified by you. The software lets you specify the noise in these functions as additive or nonadditive:

- **Additive Noise Terms** — The state transition and measurements equations have the following form:

$$\begin{aligned}x[k] &= f(x[k-1], u_s[k-1]) + w[k-1] \\y[k] &= h(x[k], u_m[k]) + v[k]\end{aligned}$$

Here  $f$  is a nonlinear state transition function that describes the evolution of states  $x$  from one time step to the next. The nonlinear measurement function  $h$  relates  $x$  to the measurements  $y$  at time step  $k$ .  $w$  and  $v$  are the zero-mean, uncorrelated process and measurement noises, respectively. These functions can also have additional input arguments that are denoted by  $u_s$  and  $u_m$  in the equations. For example, the additional arguments could be time step  $k$  or the inputs  $u$  to the nonlinear system. There can be multiple such arguments.

Note that the noise terms in both equations are additive. That is,  $x(k)$  is linearly related to the process noise  $w(k-1)$ , and  $y(k)$  is linearly related to the measurement noise  $v(k)$ .

- **Nonadditive Noise Terms** — The software also supports more complex state transition and measurement functions where the state  $x[k]$  and measurement  $y[k]$  are nonlinear functions of the process noise and measurement noise, respectively. When the noise terms are nonadditive, the state transition and measurements equation have the following form:



$$x[k] = f(x[k-1], w[k-1], u_s[k-1])$$

$$y[k] = h(x[k], v[k], u_m[k])$$

When you perform online state estimation, you first create the nonlinear state transition function  $f$  and measurement function  $h$ . You then construct the `extendedKalmanFilter` object using these nonlinear functions, and specify whether the noise terms are additive or nonadditive. You can also specify the Jacobians of the state transition and measurement functions. If you do not specify them, the software numerically computes the Jacobians.

After you create the object, you use the `predict` command to predict state estimate at the next time step, and `correct` to correct state estimates using the algorithm and real-time data. For information about the algorithm, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

You can use the following commands with `extendedKalmanFilter` objects:

Command	Description
<code>correct</code>	Correct the state and state estimation error covariance at time step $k$ using measured data at time step $k$ .
<code>predict</code>	Predict the state and state estimation error covariance at time the next time step.
<code>clone</code>	Create another object with the same object property values.  Do not create additional objects using syntax <code>obj2 = obj</code> . Any changes made to the properties of the new object created in this way ( <code>obj2</code> ) also change the properties of the original object ( <code>obj</code> ).

For `extendedKalmanFilter` object properties, see “Properties” on page 2-230.

## Examples

### Create Extended Kalman Filter Object for Online State Estimation

To define an extended Kalman filter object for estimating the states of your system, you first write and save the state transition function and measurement function for the system.

In this example, use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to a van der Pol oscillator with nonlinearity parameter,  $\mu$ , equal to 1. The oscillator has two states.

Specify an initial guess for the two states. You specify the guess as an  $M$ -element row or column vector, where  $M$  is the number of states.

```
initialStateGuess = [1;0];
```

Create the extended Kalman filter object. Use function handles to provide the state transition and measurement functions to the object.

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,initialStateGuess);
```

The object has a default structure where the process and measurement noise are additive.

To estimate the states and state estimation error covariance from the constructed object, use the `correct` and `predict` commands and real-time data.

### Specify Process and Measurement Noise Covariances in Extended Kalman Filter Object

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. Use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions are written for additive process and measurement noise terms. Specify the initial state values for the two states as `[2;0]`.

Since the system has two states and the process noise is additive, the process noise is a 2-element vector and the process noise covariance is a 2-by-2 matrix. Assume there is no cross-correlation between process noise terms, and both the terms have the same variance 0.01. You can specify the process noise covariance as a scalar. The software uses the scalar value to create a 2-by-2 diagonal matrix with 0.01 on the diagonals.

Specify the process noise covariance during object construction.

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0],...
    'ProcessNoise',0.01);
```

Alternatively, you can specify noise covariances after object construction using dot notation. For example, specify the measurement noise covariance as 0.2.

```
obj.MeasurementNoise = 0.2;
```

Since the system has only one output, the measurement noise is a 1-element vector and the `MeasurementNoise` property denotes the variance of the measurement noise.

### Specify Jacobians for State and Measurement Functions

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. Use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. Specify the initial state values for the two states as `[2;0]`.

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0]);
```

The extended Kalman filter algorithm uses Jacobians of the state transition and measurement functions for state estimation. You write and save the Jacobian functions and provide them as function handles to the object. In this example, use the previously written and saved functions `vdpStateJacobianFcn.m` and `vdpMeasurementJacobianFcn.m`.

```
obj.StateTransitionJacobianFcn = @vdpStateJacobianFcn.m;
obj.MeasurementJacobianFcn = @vdpMeasurementJacobianFcn;
```

Note that if you do not specify the Jacobians of the functions, the software numerically computes the Jacobians. This numerical computation may result in increased processing time and numerical inaccuracy of the state estimation.

### Specify Nonadditive Measurement Noise in Extended Kalman Filter Object

Create an extended Kalman filter object for a van der Pol oscillator with two states and one output. Assume that the process noise terms in the state transition function are additive. That is, there is a linear relation between the state and process noise. Also assume that the measurement noise terms are nonadditive. That is, there is a nonlinear relation between the measurement and measurement noise.

```
obj = extendedKalmanFilter('HasAdditiveMeasurementNoise',false);
```

Specify the state transition function and measurement functions. Use the previously written and saved functions, `vdpStateFcn.m` and `vdpMeasurementNonAdditiveNoiseFcn.m`.

The state transition function is written assuming the process noise is additive. The measurement function is written assuming the measurement noise is nonadditive.

```
obj.StateTransitionFcn = @vdpStateFcn;  
obj.StateTransitionFcn = @vdpMeasurementNonAdditiveNoiseFcn;
```

Specify the initial state values for the two states as `[2;0]`.

```
obj.State = [2;0];
```

You can now use the `correct` and `predict` commands to estimate the state and state estimation error covariance values from the constructed object.

### Specify State Transition and Measurement Functions with Additional Inputs

Consider a nonlinear system with input  $u$  whose state  $x$  and measurement  $y$  evolve according to the following state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise  $w$  of the system is additive while the measurement noise  $v$  is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input  $u$ .

```
f = @(x,u)(sqrt(x+u));  
h = @(x,v,u)(x+2*u+v^2);
```

`f` and `h` are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive,  $v$  is also specified as an input. Note that  $v$  is specified as an input before the additional input  $u$ .

Create an extended Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = extendedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of `u` to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement  $y[k]=0.8$  and input  $u[k]=0.2$  at time step  $k$ .

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given  $u[k]=0.2$ .

```
predict(obj,0.2)
```

- “Nonlinear State Estimation Using Unscented Kalman Filter”
- “Generate Code for Online State Estimation in MATLAB”

## Input Arguments

### **StateTransitionFcn** — State transition function

function handle

State transition function  $f$ , specified as a function handle. The function calculates the  $M$ -element state vector of the system at time step  $k$ , given the state vector at time step  $k-1$ .  $M$  is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system, and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise  $w$  is additive, and the state transition function specifies how the states evolve as a function of state values at the previous time step:

$$x(k) = f(x(k-1), Us1, \dots, Usn)$$

Where  $x(k)$  is the estimated state at time  $k$ , and  $Us1, \dots, Usn$  are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.

- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

$$x(k) = f(x(k-1), w(k-1), Us1, \dots, Usn)$$

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.

### **MeasurementFcn — Measurement function**

function handle

Measurement function  $h$ , specified as a function handle. The function calculates the  $N$ -element output measurement vector of the nonlinear system at time step  $k$ , given the state vector at time step  $k$ .  $N$  is the number of measurements of the system. You write and save the measurement function, and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.

The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is true — The measurement noise  $v$  is additive, and the measurement function specifies how the measurements evolve as a function of state values:

$$y(k) = h(x(k), Um1, \dots, Umn)$$

Where  $y(k)$  and  $x(k)$  are the estimated output and estimated state at time  $k$ , and  $Um1, \dots, Umn$  are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command, which in turn passes them to the measurement function.

- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

$$y(k) = h(x(k), v(k), Um1, \dots, Umn)$$

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

### **InitialState** — Initial state estimate value

vector

Initial state estimate value, specified as an  $M$ -element vector, where  $M$  is the number of states in the system. Specify the initial state values based on your knowledge of the system.

The specified value is stored in the `State` property of the object. If you specify `InitialState` as a column vector, then `State` is also a column vector, and the `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned.

If you want a filter with single-precision floating-point variables, specify `InitialState` as a single-precision vector variable. For example, for a two-state system with state transition and measurement functions `vdpStateFcn.m` and `vdpMeasurementFcn.m`, create the extended Kalman filter object with initial state estimates `[1;2]` as follows:

```
obj = extendedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,single([1;2]))
```

Data Types: `double` | `single`

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Use `Name`, `Value` arguments to specify properties of `extendedKalmanFilter` object during object creation. For example, to create an extended Kalman filter object and specify the process noise covariance as 0.01:

```
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState,'ProcessNoise
```

## Properties

`extendedKalmanFilter` object properties are of three types:

- Tunable properties that you can specify multiple times, either during object construction using `Name`, `Value` arguments, or any time afterward during state estimation. After object creation, use dot notation to modify the tunable properties.

```
obj = extendedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState);  
obj.ProcessNoise = 0.01;
```

The tunable properties are `State`, `StateCovariance`, `ProcessNoise`, and `MeasurementNoise`.

- Nontunable properties that you can specify once, either during object construction or afterward using dot notation. Specify these properties before state estimation using `correct` and `predict`. The `StateTransitionFcn`, `MeasurementFcn`, `StateTransitionJacobianFcn`, and `MeasurementJacobianFcn` properties belong to this category.
- Nontunable properties that you must specify during object construction. The `HasAdditiveProcessNoise` and `HasAdditiveMeasurementNoise` properties belong to this category.

### **HasAdditiveMeasurementNoise** — Measurement noise characteristics

`true` (default) | `false`

Measurement noise characteristics, specified as one of the following values:

- `true` — Measurement noise  $v$  is additive. The measurement function  $h$  that is specified in `MeasurementFcn` has the following form:

$$y(k) = h(x(k), Um1, \dots, Umn)$$

Where  $y(k)$  and  $x(k)$  are the estimated output and estimated state at time  $k$ , and  $Um1, \dots, Umn$  are any optional input arguments required by your measurement function.

- `false` — Measurement noise is nonadditive. The measurement function specifies how the output measurement evolves as a function of the state *and* measurement noise:



$$y(k) = h(x(k), v(k), Um1, \dots, Umn)$$

`HasAdditiveMeasurementNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

### **HasAdditiveProcessNoise — Process noise characteristics**

`true` (default) | `false`

Process noise characteristics, specified as one of the following values:

- **true** — Process noise  $w$  is additive. The state transition function  $f$  specified in `StateTransitionFcn` has the following form:

$$x(k) = f(x(k-1), Us1, \dots, Usn)$$

Where  $x(k)$  is the estimated state at time  $k$ , and  $Us1, \dots, Usn$  are any additional input arguments required by your state transition function.

- **false** — Process noise is nonadditive. The state transition function specifies how the states evolve as a function of the state *and* process noise at the previous time step:

$$x(k) = f(x(k-1), w(k-1), Us1, \dots, Usn)$$

`HasAdditiveProcessNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

### **MeasurementFcn — Measurement function**

`[]` (default) | function handle

Measurement function  $h$ , specified as a function handle. The function calculates the  $N$ -element output measurement vector of the nonlinear system at time step  $k$ , given the state vector at time step  $k$ .  $N$  is the number of measurements of the system. You write and save the measurement function and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.

The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is `true` — The measurement noise  $v$  is additive, and the measurement function specifies how the measurements evolve as a function of state values:

$$y(k) = h(x(k), Um1, \dots, Umn)$$

Where  $y(k)$  and  $x(k)$  are the estimated output and estimated state at time  $k$ , and  $Um1, \dots, Umn$  are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command which in turn passes them to the measurement function.

- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

$$y(k) = h(x(k), v(k), Um1, \dots, Umn)$$

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

`MeasurementFcn` is a nontunable property. You can specify it once before using the `correct` command either during object construction or using dot notation after object construction. You cannot change it after using the `correct` command.

### **MeasurementJacobianFcn — Jacobian of measurement function**

[ ] (default) | function handle

Jacobian of measurement function  $h$ , specified as one of the following:

- [ ] — The Jacobian is numerically computed at every call to the `correct` command. This may increase processing time and numerical inaccuracy of the state estimation.
- function handle — You write and save the Jacobian function and specify the handle to the function. For example, if `vdpMeasurementJacobianFcn.m` is the Jacobian function, specify `MeasurementJacobianFcn` as `@vdpMeasurementJacobianFcn`.

The function calculates the partial derivatives of the measurement function with respect to the states and measurement noise. The number of inputs to the Jacobian function must equal the number of inputs to the measurement function and must be specified in the same order in both functions. The number of outputs of the Jacobian function depends on the `HasAdditiveMeasurementNoise` property:

- `HasAdditiveMeasurementNoise` is true — The function calculates the partial derivatives of the measurement function with respect to the states ( $\partial h / \partial x$ ). The output is as an  $N$ -by- $M$  Jacobian matrix, where  $N$  is the number of measurements of the system and  $M$  is the number of states.
- `HasAdditiveMeasurementNoise` is false — The function also returns a second output that is the partial derivative of the measurement function with respect to the measurement noise terms ( $\partial h / \partial v$ ). The second output is returned as an  $N$ -by- $V$  Jacobian matrix, where  $V$  is the number of measurement noise terms.

To see an example of a Jacobian function for additive measurement noise, type `edit vdpMeasurementJacobianFcn` at the command line.

`MeasurementJacobianFcn` is a nontunable property. You can specify it once before using the `correct` command either during object construction or using dot notation after object construction. You cannot change it after using the `correct` command.

### **MeasurementNoise — Measurement noise covariance**

1 (default) | scalar | matrix

Measurement noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveMeasurementNoise` property:

- `HasAdditiveMeasurementNoise` is true — Specify the covariance as a scalar or an  $N$ -by- $N$  matrix, where  $N$  is the number of measurements of the system. Specify a scalar if there is no cross-correlation between measurement noise terms and all the terms have the same variance. The software uses the scalar value to create an  $N$ -by- $N$  diagonal matrix.
- `HasAdditiveMeasurementNoise` is false — Specify the covariance as a  $V$ -by- $V$  matrix, where  $V$  is the number of measurement noise terms. `MeasurementNoise` must be specified before using `correct`. After you specify `MeasurementNoise` as a matrix for the first time, to then change `MeasurementNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the measurement noise terms and all the terms have the same variance. The software extends the scalar to a  $V$ -by- $V$  diagonal matrix with the scalar on the diagonals.

`MeasurementNoise` is a tunable property. You can change it using dot notation.

### **ProcessNoise — Process noise covariance**

1 (default) | scalar | matrix

Process noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveProcessNoise` property:

- `HasAdditiveProcessNoise` is true — Specify the covariance as a scalar or an  $M$ -by- $M$  matrix, where  $M$  is the number of states of the system. Specify a scalar if there is no cross-correlation between process noise terms, and all the terms have the same variance. The software uses the scalar value to create an  $M$ -by- $M$  diagonal matrix.
- `HasAdditiveProcessNoise` is false — Specify the covariance as a  $W$ -by- $W$  matrix, where  $W$  is the number of process noise terms. `ProcessNoise` must be specified before using `predict`. After you specify `ProcessNoise` as a matrix for the first time, to then change `ProcessNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the process noise terms and all the terms have the same variance. The software extends the scalar to a  $W$ -by- $W$  diagonal matrix.

`ProcessNoise` is a tunable property. You can change it using dot notation.

### **State — State of nonlinear system**

[ ] (default) | vector

State of the nonlinear system, specified as a vector of size  $M$ , where  $M$  is the number of states of the system.

When you use the `predict` command, `State` is updated with the predicted value at time step  $k$  using the state value at time step  $k-1$ . When you use the `correct` command, `State` is updated with the estimated value at time step  $k$  using measured data at time step  $k$ .

The initial value of `State` is the value you specify in the `InitialState` input argument during object creation. If you specify `InitialState` as a column vector, then `State` is also a column vector, and the `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned. If you want a filter with single-precision floating-point variables, you must specify `State` as a single-precision variable during object construction using the `InitialState` input argument.

`State` is a tunable property. You can change it using dot notation.

### **StateCovariance — State estimation error covariance**

1 (default) | scalar | matrix

State estimation error covariance, specified as a scalar or an  $M$ -by- $M$  matrix, where  $M$  is the number of states of the system. If you specify a scalar, the software uses the scalar value to create an  $M$ -by- $M$  diagonal matrix.

Specify a high value for the covariance when you do not have confidence in the initial state values that you specify in the `InitialState` input argument.

When you use the `predict` command, `StateCovariance` is updated with the predicted value at time step  $k$  using the state value at time step  $k-1$ . When you use the `correct` command, `StateCovariance` is updated with the estimated value at time step  $k$  using measured data at time step  $k$ .

`StateCovariance` is a tunable property. You can change it using dot notation after using the `correct` or `predict` commands.

### **StateTransitionFcn** — State transition function

[ ] (default) | function handle

State transition function  $f$ , specified as a function handle. The function calculates the  $M$ -element state vector of the system at time step  $k$ , given the state vector at time step  $k-1$ .  $M$  is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise  $w$  is additive, and the state transition function specifies how the states evolve as a function of state values at previous time step:

$$x(k) = f(x(k-1), Us1, \dots, Usn)$$

Where  $x(k)$  is the estimated state at time  $k$ , and  $Us1, \dots, Usn$  are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.

- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

$$x(k) = f(x(k-1), w(k-1), Us1, \dots, Usn)$$

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.

`StateTransitionFcn` is a nontunable property. You can specify it once before using the `predict` command either during object construction or using dot notation after object construction. You cannot change it after using the `predict` command.

### **StateTransitionJacobianFcn — Jacobian of state transition function**

[ ] (default) | function handle

Jacobian of state transition function  $f$ , specified as one of the following:

- [ ] — The Jacobian is numerically computed at every call to the `predict` command. This may increase processing time and numerical inaccuracy of the state estimation.
- function handle — You write and save the Jacobian function and specify the handle to the function. For example, if `vdpStateJacobianFcn.m` is the Jacobian function, specify `StateTransitionJacobianFcn` as `@vdpStateJacobianFcn`.

The function calculates the partial derivatives of the state transition function with respect to the states and process noise. The number of inputs to the Jacobian function must equal the number of inputs of the measurement function and must be specified in the same order in both functions. The number of outputs of the function depends on the `HasAdditiveProcessNoise` property:

- `HasAdditiveProcessNoise` is true — The function calculates the partial derivative of the state transition function with respect to the states ( $\partial f / \partial x$ ). The output is an  $M$ -by- $M$  Jacobian matrix, where  $M$  is the number of states.
- `HasAdditiveProcessNoise` is false — The function must also return a second output that is the partial derivative of the state transition function with respect to the measurement noise terms ( $\partial f / \partial w$ ). The second output is returned as an  $M$ -by- $W$  Jacobian matrix, where  $W$  is the number of process noise terms.

The extended Kalman filter algorithm uses the Jacobian to compute the state estimation error covariance.

To see an example of a Jacobian function for additive process noise, type `edit vdpStateJacobianFcn` at the command line.

`StateTransitionJacobianFcn` is a nontunable property. You can specify it once before using the `predict` command either during object construction or using dot

notation after object construction. You cannot change it after using the `predict` command.

## Output Arguments

### **obj** — Extended Kalman filter object for online state estimation

`extendedKalmanFilter` object

Extended Kalman filter object for online state estimation, returned as an `extendedKalmanFilter` object. This object is created using the specified properties. Use the `correct` and `predict` commands to estimate the state and state estimation error covariance using the extended Kalman filter algorithm.

When you use `predict`, `obj.State` and `obj.StateCovariance` are updated with the predicted value at time step  $k$  using the state value at time step  $k-1$ . When you use `correct`, `obj.State` and `obj.StateCovariance` are updated with the estimated values at time step  $k$  using measured data at time step  $k$ .

## More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”
- “Validate Online State Estimation at the Command Line”
- “Troubleshoot Online State Estimation”

## See Also

### Functions

`clone` | `correct` | `kalman` | `kalmd` | `predict` | `unscentedKalmanFilter`

### Blocks

Kalman Filter

Introduced in R2016b

# fcats

Concatenate FRD models along frequency dimension

## Syntax

```
sys = fcats(sys1,sys2,...)
```

## Description

`sys = fcats(sys1,sys2,...)` takes two or more `frd` models and merges their frequency responses into a single `frd` model `sys`. The resulting frequency vector is sorted by increasing frequency. The frequency vectors of `sys1`, `sys2`, ... should not intersect. If the frequency vectors do intersect, use `fdel` to remove intersecting data from one or more of the models.

## See Also

`fselect` | `interp` | `fdel` | `frd`

**Introduced in R2006a**



# fdel

Delete specified data from frequency response data (FRD) models

## Syntax

```
sysout = fdel(sys, freq)
```

## Description

*sysout* = fdel(*sys*, *freq*) removes from the frd model *sys* the data nearest to the frequency values specified in the vector *freq*.

## Input Arguments

### **sys**

frd model.

### **freq**

Vector of frequency values.

## Output Arguments

### **sysout**

frd model containing the data remaining in *sys* after removing the frequency points closest to the entries of *freq*.

## Examples

Remove selected data from a frd model. In this example, first obtain an frd model:

```
sys = frd(tf([1],[1 1]), logspace(0,1,10))
```

Frequency(rad/s)	Response
-----	-----
1.0000	0.5000 - 0.5000i
1.2915	0.3748 - 0.4841i
1.6681	0.2644 - 0.4410i
2.1544	0.1773 - 0.3819i
2.7826	0.1144 - 0.3183i
3.5938	0.0719 - 0.2583i
4.6416	0.0444 - 0.2059i
5.9948	0.0271 - 0.1623i
7.7426	0.0164 - 0.1270i
10.0000	0.0099 - 0.0990i

Continuous-time frequency response.

The following commands remove the data nearest 2, 3.5, and 6 rad/s from `sys`.

```
freq = [2, 3.5, 6];  
sysout = fdel(sys, freq)
```

Frequency(rad/s)	Response
-----	-----
1.0000	0.5000 - 0.5000i
1.2915	0.3748 - 0.4841i
1.6681	0.2644 - 0.4410i
2.7826	0.1144 - 0.3183i
4.6416	0.0444 - 0.2059i
7.7426	0.0164 - 0.1270i
10.0000	0.0099 - 0.0990i

Continuous-time frequency response.

You do not have to specify the exact frequency of the data to remove. `fdel` removes the data nearest to the specified frequencies.

## More About

### Tips

- Use `fdel` to remove unwanted data (for example, outlier points) at specified frequencies.

- Use `fdel` to remove data at intersecting frequencies from `frd` models before merging them with `fcats`. `fcats` produces an error when you attempt to merge `frd` models that have intersecting frequency data.
- To remove data from an `frd` model within a range of frequencies, use `fselect`.

## See Also

`fcats` | `fselect` | `frd`

**Introduced in R2010a**

## feedback

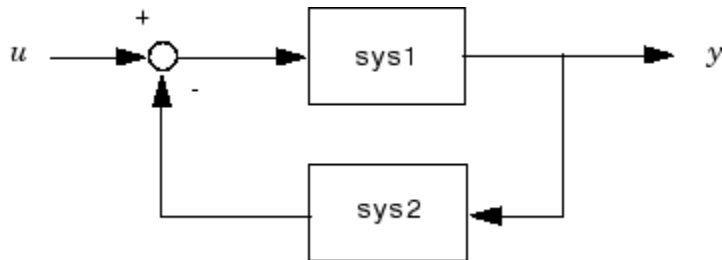
Feedback connection of two models

### Syntax

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

### Description

`sys = feedback(sys1,sys2)` returns a model object `sys` for the negative feedback interconnection of model objects `sys1` and `sys2`.



The closed-loop model `sys` has `u` as input vector and `y` as output vector. The 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 “Rules That Determine Model Type”).

To apply positive feedback, use the syntax

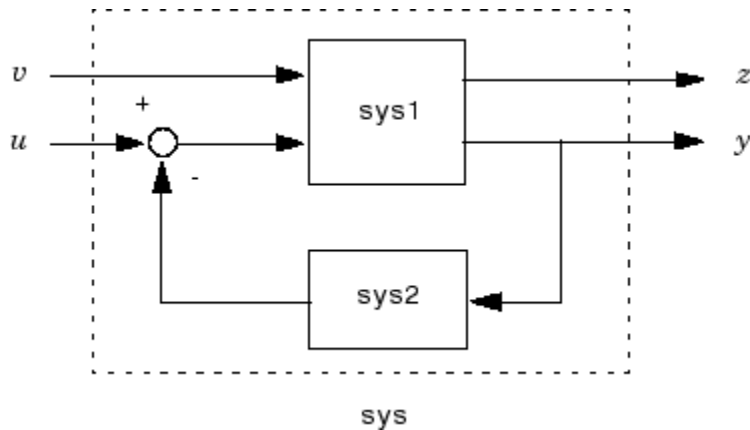
```
sys = feedback(sys1,sys2,+1)
```

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.



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 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

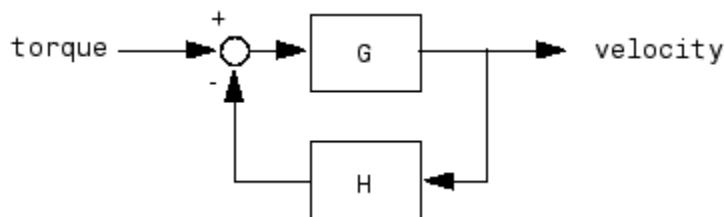
```
sys = feedback(sys1,sys2,feedin,feedout,+1)
```

to apply positive feedback.

For more complicated feedback structures, use `append` and `connect`.

## 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)
```

These commands produce the following result.

```
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.

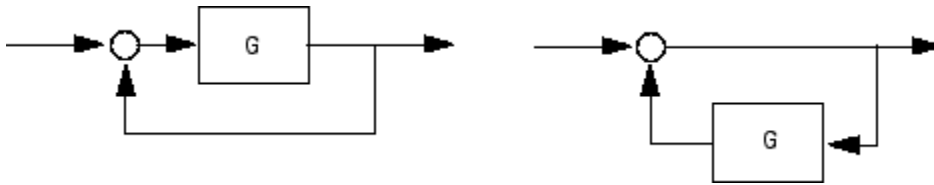
## Example 2

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` | `parallel` | `connect`

Introduced before R2006a

## **filt**

Specify discrete transfer functions in DSP format

### **Syntax**

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

### **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 property of the model, for example, the input names or the transfer function variable. For information about the available properties and their values, see the `tf` reference page.

## 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.

## Examples

Create a two-input digital filter with input names 'channel1' and 'channel2':

```
num = {1 , [1 0.3]};
den = {[1 1 2] , [5 2]};
H = filt(num,den,'inputname',{'channel1' 'channel2'})
```

This syntax returns:

Transfer function from input "channel1" to output:

```
      1
-----
1 + z^-1 + 2 z^-2
```

Transfer function from input "channel2" to output:

```
1 + 0.3 z^-1
-----
```

$5 + 2 z^{-1}$

Sample time: unspecified

## More About

### Tips

`filt` behaves as `tf` with the `Variable` property set to `'z^-1'`. See `tf` entry below for details.

### See Also

`tf` | `zpk` | `ss`

**Introduced before R2006a**

# fnorm

Pointwise peak gain of FRD model

## Syntax

```
fnrm = fnorm(sys)  
fnrm = fnorm(sys, ntype)
```

## Description

`fnrm = fnorm(sys)` computes the pointwise 2-norm of the frequency response contained in the FRD model `sys`, that is, the peak gain at each frequency point. The output `fnrm` is an FRD object containing the peak gain across frequencies.

`fnrm = fnorm(sys, ntype)` computes the frequency response gains using the matrix norm specified by `ntype`. See `norm` for valid matrix norms and corresponding `NTYPE` values.

## See Also

`norm` | `abs`

**Introduced in R2006a**

## fourierBasis

Fourier basis functions for tunable gain surface

You use basis function expansions to parameterize gain surfaces for tuning gain-scheduled controllers. `fourierBasis` generates periodic Fourier series expansions for parameterizing gain surfaces that depend periodically on the scheduling variables, such as a gain that varies with angular position, in any number of scheduling variables. Use the output of `fourierBasis` to create tunable gain surfaces with `tunableSurface`.

## Syntax

```
shapefcn = fourierBasis(N)
shapefcn = fourierBasis(N,nvars)
```

## Description

`shapefcn = fourierBasis(N)` generates a function that evaluates the first  $N$  harmonics of  $e^{inx}$ :

$$F(x) = [\cos(\pi x), \sin(\pi x), \cos(2\pi x), \sin(2\pi x), \dots, \cos(N\pi x), \sin(N\pi x)].$$

$F$  is the function represented by `shapefcn`. The term of  $F$  are the first  $2*N$  basis functions in the Fourier series expansion of a periodically varying gain,  $K(x)$ , with  $K(-1) = K(1)$ . That expansion is given by:

$$K(x) = \frac{a_0}{2} + \sum_k \{a_k \cos(k\pi x) + b_k \sin(k\pi x)\}.$$

`shapefcn = fourierBasis(N,nvars)` generates an `nvars`-dimensional Fourier basis for periodic functions on the region  $[-1,1]^{nvars}$ . This basis is the outer product of `nvars` Fourier bases with  $N$  harmonics along each dimension. The resulting function `shapefcn` takes `nvars` input arguments and returns a vector with  $(2*N+1)^{(nvars-1)}$  entries.

To specify basis functions of multiple scheduling variables where the expansions are different for each variable, use `ndBasis`.

## Examples

### Fourier Basis Functions of One Scheduling Variable

Create basis functions for a gain that varies as a periodic function of one scheduling variable.

```
shapefcn = fourierBasis(2);
```

shapefcn is a handle to a function of one variable that returns an array of four values corresponding to the first two harmonics of a periodic function on  $x = [-1,1]$ :

$$F(x) = [\cos(\pi x), \sin(\pi x), \cos(2\pi x), \sin(2\pi x)].$$

Use shapefcn as an input argument to tunableSurface to define a gain surface of the form:

$$K(x) = K_0 + K_1 \cos(\pi x) + K_2 \sin(\pi x) + K_3 \cos(2\pi x) + K_4 \sin(2\pi x).$$

The variable  $x$  is a normalized version of the scheduling variable for your tunable surface. Because the basis functions created by fourierBasis act on normalized variables, your gain-scheduled system must use design points whose endpoint values delineate exactly one period. For example, suppose you use the following design points:

```
alpha = [-7, -4, -1, 2, 5];
domain = struct('alpha', alpha);
K = tunableSurface('K', 0, domain, shapefcn);
```

In normalizing the domain, the software assumes that the gain surface,  $K$ , is periodic in  $\alpha$  such that  $K(-7) = K(5)$ .

### Fourier Basis Functions in Higher Dimensions

Create a two-dimensional Fourier basis for periodic functions of  $x$  and  $y$  on the domain  $[-1, 1]^N$ . The basis functions should go up to the third harmonic in both the  $x$  and  $y$  dimensions.

```
F2D = fourierBasis(3, 2);
```

This function is the outer product of two vectors:

```
x = fourierBasis(3);
y = fourierBasis(3);
```

Equivalently, you can obtain the outer product using `ndBasis`.

```
F = fourierBasis(3);  
F2D = ndBasis(F,F);
```

The values in the vector returned by `F` include cross-terms such as  $\sin(\pi x) \cos(\pi y)$  and  $\sin(3\pi x) \cos(2\pi y)$ .

## Input Arguments

### **N** — Number of harmonics of Fourier expansion

positive integer

Number of harmonics of Fourier expansion, specified as a positive integer.

### **nvars** — Number of variables

1 (default) | positive integer

Number of scheduling variables, specified as a positive integer.

## Output Arguments

### **shapefcn** — Fourier expansion

function handle

Fourier expansion, specified as a function handle. `shapefcn` takes as input arguments the number of variables specified by `nvars`. It returns a vector of polynomials in those variables, defined on the interval  $[-1,1]$  for each input variable. When you use `shapefcn` to create a gain surface, `tunableSurface` automatically generates tunable coefficients for each polynomial term in the vector.

## More About

### Tips

- Suppose the tunable gain  $K$  must be a periodic function of the scheduling variable  $x$ . When you create a model of  $K$  using `tunableSurface`, you specify a set of values for

the scheduling variable  $x$ , the design points. The software normalizes  $x$  to the range  $[-1,1]$  by mapping the smallest value in the set,  $x_{min}$ , to  $-1$ , and the largest value,  $x_{max}$  to  $+1$ . If you use `fourierBasis` to generate a basis function expansion for  $K$ , each function satisfies  $f(-1) = f(1)$  so the gain surface will satisfy  $K(x_{min}) = K(x_{max})$ . For this periodicity to match the desired periodicity of  $K(x)$ ,  $x_{max} - x_{min}$  must be equal to exactly one period of  $K(x)$ . In other words, the design points must span exactly one period of the gain  $K(x)$ . For example, if the periodic variable is an angle that ranges from 0 to 360 degrees, then the corresponding values in domain can range from 0 to 360 or from  $-180$  to  $180$ , but not from 10 to 350.

### See Also

`ndBasis` | `polyBasis` | `tunableSurface`

Introduced in R2015b

## frd

Create frequency-response data model, convert to frequency-response data model

### Syntax

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

### Description

`sys = frd(response, frequency)` creates a frequency-response data (`frd`) model object `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 [Data Format for the Argument Response in FRD Models](#) for a list of response data formats.

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

`sys = frd` creates an empty `frd` model object.

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 the model. For more information about available properties of `frd` models, see “Properties” on page 2-255.

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 dynamic system model `sys` to frequency response data form. The frequency response is computed at the frequencies provided by the vector `frequency`, in rad/TimeUnit, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

`sysfrd = frd(sys, frequency, units)` converts a dynamic system model to an `frd` model and interprets frequencies in the `frequency` vector to have the units specified by `units`. For a list of values that `units` can take, see the `FrequencyUnit` property in “Properties” on page 2-255.

## 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.

### Data Format for the Argument Response in FRD Models

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

## Properties

`frd` objects have the following properties:

### Frequency

Frequency points of the frequency response data. Specify `Frequency` values in the units specified by the `FrequencyUnit` property.

### **FrequencyUnit**

Frequency units of the model.

`FrequencyUnit` specifies the units of the frequency vector in the `Frequency` property. Set `FrequencyUnit` to one of the following values:

- `'rad/TimeUnit'`
- `'cycles/TimeUnit'`
- `'rad/s'`
- `'Hz'`
- `'kHz'`
- `'MHz'`
- `'GHz'`
- `'rpm'`

The units `'rad/TimeUnit'` and `'cycles/TimeUnit'` are relative to the time units specified in the `TimeUnit` property.

Changing this property changes the overall system behavior. Use `chgFreqUnit` to convert between frequency units without modifying system behavior.

**Default:** `'rad/TimeUnit'`

### **ResponseData**

Frequency response data.

The `'ResponseData'` property stores the frequency response data as a 3-D array of complex numbers. For SISO systems, `'ResponseData'` is a vector of frequency response values at the frequency points specified in the `'Frequency'` property. For MIMO systems with `Nu` inputs and `Ny` outputs, `'ResponseData'` is an array of size `[Ny Nu Nw]`, where `Nw` is the number of frequency points.

### **IODelay**

Transport delays. `IODelay` is a numeric array specifying a separate transport delay for each input/output pair.

For continuous-time systems, specify transport delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify transport delays in integer multiples of the sample time, `Ts`.

For a MIMO system with `Ny` outputs and `Nu` inputs, set `IODelay` to a `Ny`-by-`Nu` array. Each entry of this array is a numerical value that represents the transport delay for the corresponding input/output pair. You can also set `IODelay` to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

### **InputDelay**

Input delay for each input channel, specified as a scalar value or numeric vector. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0

### **OutputDelay**

Output delays. `OutputDelay` is a numeric vector specifying a time delay for each output channel. For continuous-time systems, specify output delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify output delays in integer multiples of the sample time `Ts`. For example, `OutputDelay = 3` means a delay of three sampling periods.

For a system with `Ny` outputs, set `OutputDelay` to an `Ny`-by-1 vector, where each entry is a numerical value representing the output delay for the corresponding output channel. You can also set `OutputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all output channels

### **Ts**

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit

specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** `''` for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### OutputName

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, 'measurements'.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all output channels

### OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, 'seconds'.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

### OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### Name

System name, specified as a character vector. For example, `'system_1'`.

**Default:** `''`

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** `{}`

### UserData

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** `[]`

### SamplingGrid

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the `(zeta,w)` values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:,:,1,1) [zeta=0.3, w=5] =
```

```

          25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```

          25
-----
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sLinearizer` populate `SamplingGrid` in this way.

**Default:** `[]`

## Examples

### Create Frequency-Response Model



Create a SISO FRD model from a frequency vector and response data:

```
% generate a frequency vector and response data
freq = logspace(1,2);
resp = .05*(freq).*exp(i*2*freq);
% Create a FRD model
sys = frd(resp,freq);
```

## More About

- “What Are Model Objects?”
- “Frequency Response Data (FRD) Models”

## See Also

[chgTimeUnit](#) | [chgFreqUnit](#) | [frdata](#) | [set](#) | [ss](#) | [tf](#) | [zpk](#) | [idfrd](#)

**Introduced before R2006a**

## frdata

Access data for frequency response data (FRD) object

### Syntax

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

### 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 the `frd` reference page for more information on the data format for FRD response data.

`[response,freq,covresp] = frdata(sys)` also returns the covariance `covresp` of the response data `resp` for `idfrd` model `sys`. (Using `idfrd` models requires System Identification Toolbox software.) The covariance `covresp` is a 5D-array where `covH(i,j,k,:,:) contains the 2-by-2 covariance matrix of the response resp(i,j,k). The (1,1) element is the variance of the real part, the (2,2) element the variance of the imaginary part and the (1,2) and (2,1) elements the covariance between the real and imaginary parts.`

For SISO FRD models, the syntax

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

forces `frdata` to return the response data as a column vector rather than a 3-dimensional array (see example below). Similarly

`[response, freq, Ts, covresp] = frdata(sys, 'v')` for an IDFRD model `sys` returns `covresp` as a 3-dimensional rather than a 5-dimensional array.

`[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.Frequency`).

## Arguments

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

## Examples

### Extract Data from Frequency Response Data Model

Create a frequency response data model by computing the response of a transfer function on a grid of frequencies.

```
H = tf([-1.2, -2.4, -1.5], [1, 20, 9.1]);  
w = logspace(-2, 3, 101);  
sys = frd(H, w);
```

`sys` is a SISO frequency response data (`frd`) model containing the frequency response at 101 frequencies.

Extract the frequency response data from `sys`.

```
[response, freq] = frdata(sys);
```

`response` is a 1-by-1-by-101 array. `response(1, 1, k)` is the complex frequency response at the frequency `freq(k)`.

## See Also

`frd` | `get` | `set` | `freqresp`

Introduced before R2006a

## freqresp

Frequency response over grid

### Syntax

```
[H,wout] = freqresp(sys)
H = freqresp(sys,w)
H = freqresp(sys,w,units)
[H,wout,covH] = freqresp(idsys,...)
```

### Description

`[H,wout] = freqresp(sys)` returns the frequency response of the dynamic system model `sys` at frequencies `wout`. The `freqresp` command automatically determines the frequencies based on the dynamics of `sys`.

`H = freqresp(sys,w)` returns the frequency response on the real frequency grid specified by the vector `w`.

`H = freqresp(sys,w,units)` explicitly specifies the frequency units of `w` with `units`.

`[H,wout,covH] = freqresp(idsys,...)` also returns the covariance `covH` of the frequency response of the identified model `idsys`.

### Input Arguments

#### **sys**

Any dynamic system model or model array.

#### **w**

Vector of real frequencies at which to evaluate the frequency response. Specify frequencies in units of `rad/TimeUnit`, where `TimeUnit` is the time units specified in the `TimeUnit` property of `sys`.

**units**

Units of the frequencies in the input frequency vector  $w$ , specified as one of the following values:

- 'rad/TimeUnit' — radians per the time unit specified in the TimeUnit property of `sys`
- 'cycles/TimeUnit' — cycles per the time unit specified in the TimeUnit property of `sys`
- 'rad/s'
- 'Hz'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rpm'

**Default:** 'rad/TimeUnit'

**idsys**

Any identified model.

## Output Arguments

**H**

Array containing the frequency response values.

If `sys` is an individual dynamic system model having  $N_y$  outputs and  $N_u$  inputs, `H` is a 3D array with dimensions  $N_y$ -by- $N_u$ -by- $N_w$ , where  $N_w$  is the number of frequency points. Thus,  $H(:, :, k)$  is the response at the frequency  $w(k)$  or  $wout(k)$ .

If `sys` is a model array of size  $[N_y N_u S_1 \dots S_n]$ , `H` is an array with dimensions  $N_y$ -by- $N_u$ -by- $N_w$ -by- $S_1$ -by-...-by- $S_n$  array.

If `sys` is a frequency response data model (such as `frd`, `genfrd`, or `idfrd`), `freqresp(sys, w)` evaluates to NaN for values of  $w$  falling outside the frequency

interval defined by `sys.frequency`. The `freqresp` command can interpolate between frequencies in `sys.frequency`. However, `freqresp` cannot extrapolate beyond the frequency interval defined by `sys.frequency`.

**wout**

Vector of frequencies corresponding to the frequency response values in `H`. If you omit `w` from the inputs to `freqresp`, the command automatically determines the frequencies of `wout` based on the system dynamics. If you specify `w`, then `wout = w`

**covH**

Covariance of the response `H`. The covariance is a 5D array where `covH(i,j,k,:,:)`  contains the 2-by-2 covariance matrix of the response from the `i`th input to the `j`th output at frequency `w(k)`. The (1,1) element of this 2-by-2 matrix is the variance of the real part of the response. The (2,2) element is the variance of the imaginary part. The (1,2) and (2,1) elements are the covariance between the real and imaginary parts of the response.

## Examples

### Compute Frequency Response of System

Create the following 2-input, 2-output system:

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

```
sys11 = 0;  
sys22 = 1;  
sys12 = tf(1,[1 1]);  
sys21 = tf([1 -1],[1 2]);  
sys = [sys11,sys12;sys21,sys22];
```

Compute the frequency response of the system.

```
[H,wout] = freqresp(sys);
```

H is a 2-by-2-by-45 array. Each entry  $H(:, :, k)$  in H is a 2-by-2 matrix giving the complex frequency response of all input-output pairs of `sys` at the corresponding frequency  $wout(k)$ . The 45 frequencies in `wout` are automatically selected based on the dynamics of `sys`.

## Compute Frequency Response on Specified Frequency Grid

Create the following 2-input, 2-output system:

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

```
sys11 = 0;
sys22 = 1;
sys12 = tf(1,[1 1]);
sys21 = tf([1 -1],[1 2]);
sys = [sys11,sys12;sys21,sys22];
```

Create a logarithmically-spaced grid of 200 frequency points between 10 and 100 radians per second.

```
w = logspace(1,2,200);
```

Compute the frequency response of the system on the specified frequency grid.

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

H is a 2-by-2-by-200 array. Each entry  $H(:, :, k)$  in H is a 2-by-2 matrix giving the complex frequency response of all input-output pairs of `sys` at the corresponding frequency  $w(k)$ .

## Frequency Response and Associated Covariance

Compute the frequency response and associated covariance for an identified model at its peak response frequency.

```
load iddata1 z1
model = procest(z1,'P2UZ');
w = 4.26;
```

```
[H,~,covH] = freqresp(model,w);
```

## Alternatives

Use `evalfr` to evaluate the frequency response at individual frequencies or small numbers of frequencies. `freqresp` is optimized for medium-to-large vectors of frequencies.

## More About

### Frequency Response

In continuous time, the *frequency response* at a frequency  $\omega$  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 frequency response is the transfer function evaluated at points on the unit circle that correspond to the real frequencies. `freqresp` maps the real frequencies  $w(1), \dots, w(N)$  to points on the unit circle using the transformation  $z = e^{j\omega T_s}$ .  $T_s$  is the sample time. The function returns the values of the transfer function at the resulting  $z$  values. For models with unspecified sample time, `freqresp` uses  $T_s = 1$ .

### Algorithms

For transfer functions or zero-pole-gain models, `freqresp` evaluates the numerator(s) and denominator(s) at the specified frequency points. For continuous-time state-space models  $(A, B, C, D)$ , the frequency response is

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

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.



## References

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

## See Also

bode | nyquist | evalfr | nichols | sigma | interp | spectrum

**Introduced before R2006a**

# freqsep

Slow-fast decomposition

## Syntax

```
[Gs,Gf] = freqsep(G,fcut)
[Gs,Gf] = freqsep(G,fcut,options)
```

## Description

`[Gs,Gf] = freqsep(G,fcut)` decomposes a linear dynamic system into slow and fast components around the specified cutoff frequency. The decomposition is such that  $G = G_s + G_f$ .

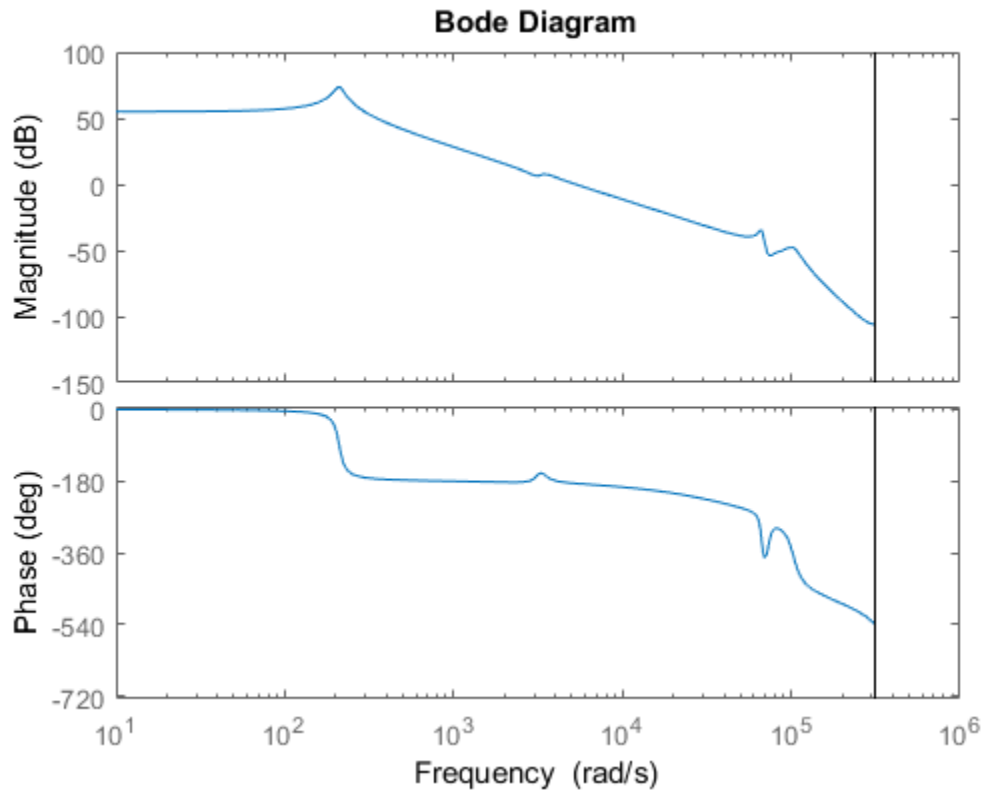
`[Gs,Gf] = freqsep(G,fcut,options)` specifies additional options for the decomposition.

## Examples

### Decompose Model into Fast and Slow Dynamics

Load a dynamic system model.

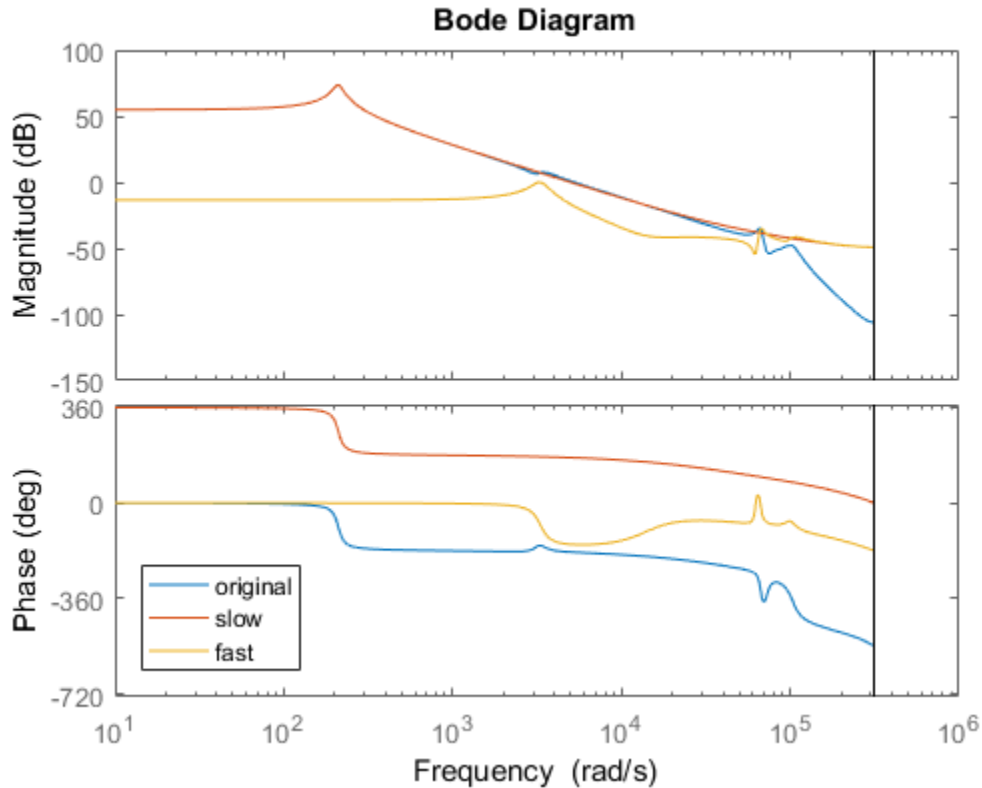
```
load numdemo Pd
bode(Pd)
```



Pd has four complex poles and one real pole. The Bode plot shows a resonance around 210 rad/s and a higher-frequency resonance below 10,000 rad/s.

Decompose this model around 1000 rad/s to separate these two resonances.

```
[Gs,Gf] = freqsep(Pd,10^3);
bode(Pd,Gs,Gf)
legend('original','slow','fast','Location','Southwest')
```



The Bode plot shows that the slow component,  $G_S$ , contains only the lower-frequency resonance. This component also matches the DC gain of the original model. The fast component,  $G_f$ , contains the higher-frequency resonances and matches the response of the original model at high frequencies. The sum of the two components  $G_S+G_f$  yields the original model.

### Separate Nearby Modes by Adjusting Tolerance

Decompose a model into slow and fast components between poles that are closely spaced.

The following system includes a real pole and a complex pair of poles that are all close to  $s = -2$ .

```
G = zpk(-.5, [-1.9999 -2+1e-4i -2-1e-4i], 10);
```

Try to decompose the model about 2 rad/s, so that the slow component contains the real pole and the fast component contains the complex pair.

```
[Gs,Gf] = freqsep(G,2);
```

Warning: One or more fast modes could not be separated from the slow modes. To force separation, increase the absolute or relative tolerances ("AbsTol" and "RelTol" options). Type "help freqsepOptions" for more information.

These poles are too close together for `freqsep` to separate. Increase the relative tolerance to allow the separation.

```
options = freqsepOptions('RelTol',1e-4);
[Gs,Gf] = freqsep(G,2,options);
```

Now `freqsep` successfully separates the dynamics about 2 rad/s.

```
slowpole = pole(Gs)
fastpole = pole(Gf)
```

```
slowpole =
```

```
-1.9999
```

```
fastpole =
```

```
-2.0000 + 0.0001i
-2.0000 - 0.0001i
```

## Input Arguments

### **G** — Dynamic system to decompose

numeric LTI model

Dynamic system to decompose, specified as a numeric LTI model, such as a `ss` or `tf` model.

### **fcut** — Cutoff frequency

positive scalar

Cutoff frequency for fast-slow decomposition, specified as a positive scalar. The output **G<sub>s</sub>** contains all poles with natural frequency less than **f<sub>cut</sub>**. The output **G<sub>f</sub>** contains all poles with natural frequency greater than or equal to **f<sub>cut</sub>**.

### **options** — Options for decomposition

`freqsepOptions` options set

Options for the decomposition, specified as an options set you create with `freqsepOptions`. Available options include absolute and relative tolerance for accuracy of the decomposed systems.

## Output Arguments

### **G<sub>s</sub>** — Slow dynamics

numeric LTI model

Slow dynamics of the decomposed system, returned as a numeric LTI model of the same type as **G**. **G<sub>s</sub>** contains all poles of **G** with natural frequency less than **f<sub>cut</sub>**, and is such that  $G = G_s + G_f$ .

### **G<sub>f</sub>** — Fast dynamics

numeric LTI model

Fast dynamics of the decomposed system, returned as a numeric LTI model of the same type as **G**. **G<sub>f</sub>** contains all poles of **G** with natural frequency greater than or equal to **f<sub>cut</sub>**, and is such that  $G = G_s + G_f$ .

### See Also

`freqsepOptions`

Introduced in R2014a

# freqsepOptions

Options for slow-fast decomposition

## Syntax

```
opt = freqsepOptions
opt = freqsepOptions(Name,Value)
```

## Description

`opt = freqsepOptions` returns the default options for `freqsep`.

`opt = freqsepOptions(Name,Value)` returns an options set with the options specified by one or more `Name,Value` pair arguments.

## Examples

### Separate Nearby Modes by Adjusting Tolerance

Decompose a model into slow and fast components between poles that are closely spaced.

The following system includes a real pole and a complex pair of poles that are all close to  $s = -2$ .

```
G = zpk(-.5, [-1.9999 -2+1e-4i -2-1e-4i], 10);
```

Try to decompose the model about 2 rad/s, so that the slow component contains the real pole and the fast component contains the complex pair.

```
[Gs,Gf] = freqsep(G,2);
```

Warning: One or more fast modes could not be separated from the slow modes. To force separation, increase the absolute or relative tolerances ("AbsTol" and "RelTol" options). Type "help freqsepOptions" for more information.

These poles are too close together for `freqsep` to separate. Increase the relative tolerance to allow the separation.

```
options = freqsepOptions('RelTol',1e-4);  
[Gs,Gf] = freqsep(G,2,options);
```

Now `freqsep` successfully separates the dynamics about 2 rad/s.

```
slowpole = pole(Gs)  
fastpole = pole(Gf)
```

```
slowpole =  
    -1.9999
```

```
fastpole =  
    -2.0000 + 0.0001i  
    -2.0000 - 0.0001i
```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'AbsTol', 1e-4`

#### 'AbsTol' — Absolute tolerance for decomposition

0 (default) | nonnegative scalar

Absolute tolerance for slow-fast decomposition, specified as a nonnegative scalar value. `freqresp` ensures that the frequency responses of the original system, `G`, and the sum of the decomposed systems `Gs+Gf`, differ by no more than `AbsTol + RelTol*abs(G)`. Increase `AbsTol` to help separate nearby modes, at the expense of the accuracy of the decomposition.

#### 'RelTol' — Relative tolerance for decomposition

1e-8 (default) | nonnegative scalar



Relative tolerance for slow-fast decomposition, specified as a nonnegative scalar value. `freqresp` ensures that the frequency responses of the original system,  $G$ , and the sum of the decomposed systems  $G_s+G_f$ , differ by no more than  $\text{AbsTol} + \text{RelTol}*\text{abs}(G)$ . Increase `RelTol` to help separate nearby modes, at the expense of the accuracy of the decomposition.

## Output Arguments

### **opt** — Options for `freqsep`

`freqsepOptions` options set

Options for `freqsep`, returned as a `freqsepOptions` options set. Use `opt` as the last argument to `freqsep` when computing slow-fast decomposition.

### **See Also**

`freqsep`

**Introduced in R2014a**

## fselect

Select frequency points or range in FRD model

### Syntax

```
subsys = fselect(sys,fmin,fmax)
subsys = fselect(sys,index)
```

### Description

`subsys = fselect(sys,fmin,fmax)` takes an FRD model `sys` and selects the portion of the frequency response between the frequencies `fmin` and `fmax`. The selected range `[fmin,fmax]` should be expressed in the FRD model units. For an IDFRD model (requires System Identification Toolbox software), the `SpectrumData`, `CovarianceData` and `NoiseCovariance` values, if non-empty, are also selected in the chosen range.

`subsys = fselect(sys,index)` selects the frequency points specified by the vector of indices `index`. The resulting frequency grid is

```
sys.Frequency(index)
```

### See Also

`fcats` | `fdel` | `interp` | `frd`

**Introduced before R2006a**

## gcare

Generalized solver for continuous-time algebraic Riccati equation

### Syntax

```
[X,L,report] = gcare(H,J,ns)
[X1,X2,D,L] = gcare(H,...,'factor')
```

### Description

`[X,L,report] = gcare(H,J,ns)` computes the unique stabilizing solution  $X$  of the continuous-time algebraic Riccati equation associated with a Hamiltonian pencil of the form

$$H - tJ = \begin{bmatrix} A & F & S1 \\ G & -A' & -S2 \\ S2' & S1' & R \end{bmatrix} - \begin{bmatrix} E & 0 & 0 \\ 0 & E' & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The optional input `ns` is the row size of the  $A$  matrix. Default values for  $J$  and  $ns$  correspond to  $E = I$  and  $R = []$ .

Optionally, `gcare` returns the vector  $L$  of closed-loop eigenvalues and a diagnosis `report` with value:

- -1 if the Hamiltonian pencil has  $iw$ -axis eigenvalues
- -2 if there is no finite stabilizing solution  $X$
- 0 if a finite stabilizing solution  $X$  exists

This syntax does not issue any error message when  $X$  fails to exist.

`[X1,X2,D,L] = gcare(H,...,'factor')` returns two matrices  $X1$ ,  $X2$  and a diagonal scaling matrix  $D$  such that  $X = D*(X2/X1)*D$ . The vector  $L$  contains the closed-loop eigenvalues. All outputs are empty when the associated Hamiltonian matrix has eigenvalues on the imaginary axis.

**See Also**

care | gdare

**Introduced before R2006a**

## gdare

Generalized solver for discrete-time algebraic Riccati equation

### Syntax

```
[X,L,report] = gdare(H,J,ns)
[X1,X2,D,L] = gdare(H,J,NS,'factor')
```

### Description

`[X,L,report] = gdare(H,J,ns)` computes the unique stabilizing solution  $X$  of the discrete-time algebraic Riccati equation associated with a Symplectic pencil of the form

$$H - tJ = \begin{bmatrix} A & F & B \\ -Q & E' & -S \\ S' & 0 & R \end{bmatrix} - \begin{bmatrix} E & 0 & 0 \\ 0 & A' & 0 \\ 0 & B' & 0 \end{bmatrix}$$

The third input `ns` is the row size of the  $A$  matrix.

Optionally, `gdare` returns the vector `L` of closed-loop eigenvalues and a diagnosis `report` with value:

- -1 if the Symplectic pencil has eigenvalues on the unit circle
- -2 if there is no finite stabilizing solution  $X$
- 0 if a finite stabilizing solution  $X$  exists

This syntax does not issue any error message when  $X$  fails to exist.

`[X1,X2,D,L] = gdare(H,J,NS,'factor')` returns two matrices  $X1$ ,  $X2$  and a diagonal scaling matrix  $D$  such that  $X = D*(X2/X1)*D$ . The vector `L` contains the closed-loop eigenvalues. All outputs are empty when the Symplectic pencil has eigenvalues on the unit circle.

### See Also

dare | gcare

**Introduced before R2006a**

# genfrd

Generalized frequency response data (FRD) model

## Description

Generalized FRD (`genfrd`) models arise when you combine numeric FRD models with models containing tunable components (Control Design Blocks). `genfrd` models keep track of how the tunable blocks interact with the tunable components. For more information about Control Design Blocks, see “Generalized Models”.

## Construction

To construct a `genfrd` model, use `series`, `parallel`, `lft`, or `connect`, or the arithmetic operators `+`, `-`, `*`, `/`, `\`, and `^`, to combine a numeric FRD model with control design blocks.

You can also convert any numeric LTI model or control design block `sys` to `genfrd` form.

`frdsys = genfrd(sys, freqs, frequits)` converts any static model or dynamic system `sys` to a generalized FRD model. If `sys` is not an `frd` model object, `genfrd` computes the frequency response of each frequency point in the vector `freqs`. The frequencies `freqs` are in the units specified by the optional argument `frequits`. If `frequits` is omitted, the units of `freqs` are `'rad/TimeUnit'`.

`frdsys = genfrd(sys, freqs, frequits, timeunits)` further specifies the time units for converting `sys` to `genfrd` form.

For more information about time and frequency units of `genfrd` models, see “Properties” on page 2-287.

## Input Arguments

### `sys`

A static model or dynamic system model object.

### **freqs**

Vector of frequency points. Express frequencies in the unit specified in `frequnits`.

### **frequnits**

Frequency units of the `genfrd` model, specified as one of the following values:

- 'rad/TimeUnit'
- 'cycles/TimeUnit'
- 'rad/s'
- 'Hz'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rpm'

**Default:** 'rad/TimeUnit'

### **timeunits**

Time units of the `genfrd` model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

**Default:** 'seconds'



## Properties

### Blocks

Structure containing the control design blocks included in the generalized LTI model or generalized matrix. The field names of **Blocks** are the **Name** property of each control design block.

You can change some attributes of these control design blocks using dot notation. For example, if the generalized LTI model or generalized matrix **M** contains a `realp` tunable parameter **a**, you can change the current value of **a** using:

```
M.Blocks.a.Value = -1;
```

### Frequency

Frequency points of the frequency response data. Specify **Frequency** values in the units specified by the **FrequencyUnit** property.

### FrequencyUnit

Frequency units of the model.

**FrequencyUnit** specifies the units of the frequency vector in the **Frequency** property. Set **FrequencyUnit** to one of the following values:

- 'rad/TimeUnit'
- 'cycles/TimeUnit'
- 'rad/s'
- 'Hz'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rpm'

The units 'rad/TimeUnit' and 'cycles/TimeUnit' are relative to the time units specified in the **TimeUnit** property.

Changing this property changes the overall system behavior. Use **chgFreqUnit** to convert between frequency units without modifying system behavior.

**Default:** 'rad/TimeUnit'

### **InputDelay**

Input delay for each input channel, specified as a scalar value or numeric vector. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0

### **OutputDelay**

Output delays. `OutputDelay` is a numeric vector specifying a time delay for each output channel. For continuous-time systems, specify output delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify output delays in integer multiples of the sample time `Ts`. For example, `OutputDelay = 3` means a delay of three sampling periods.

For a system with `Ny` outputs, set `OutputDelay` to an `Ny`-by-1 vector, where each entry is a numerical value representing the output delay for the corresponding output channel. You can also set `OutputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all output channels

### **Ts**

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model.

**Default:** 0 (continuous time)

## TimeUnit

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## InputName

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, 'seconds'.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** '' for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### **OutputName**

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, 'measurements'.

- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all output channels

### OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, 'seconds'.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

### OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### Name

System name, specified as a character vector. For example, `'system_1'`.

**Default:** `''`

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** `{}`

### UserData

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** `[]`

### SamplingGrid

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta`,`w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:, :, 1, 1) [zeta=0.3, w=5] =
```

```
      25
-----
s^2 + 3 s + 25
```

```
M(:, :, 2, 1) [zeta=0.35, w=5] =
```

```
      25
-----
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sILinearizer` populate `SamplingGrid` in this way.

**Default:** []

## More About

### Tips

- You can manipulate `genfrd` models as ordinary `frd` models. Frequency-domain analysis commands such as `bode` evaluate the model by replacing each tunable parameter with its current value.

- “Models with Tunable Coefficients”
- “Generalized Models”

### **See Also**

frd | genss | getValue | chgFreqUnit

**Introduced in R2011a**



# genmat

Generalized matrix with tunable parameters

## Description

Generalized matrices (**genmat**) are matrices that depend on tunable parameters (see **realp**). You can use generalized matrices for parameter studies. You can also use generalized matrices for building generalized LTI models (see **genss**) that represent control systems having a mixture of fixed and tunable components.

## Construction

Generalized matrices arise when you combine numeric values with static blocks such as **realp** objects. You create such combinations using any of the arithmetic operators  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\backslash$ , and  $^$ . For example, if **a** and **b** are tunable parameters, the expression  $M = a + b$  is represented as a generalized matrix.

The internal data structure of the **genmat** object **M** keeps track of how **M** depends on the parameters **a** and **b**. The **Blocks** property of **M** lists the parameters **a** and **b**.

$M = \text{genmat}(A)$  converts the numeric array or tunable parameter **A** into a **genmat** object.

## Input Arguments

### **A**

Static control design block, such as a **realp** object.

If **A** is a numeric array, **M** is a generalized matrix of the same dimensions as **A**, with no tunable parameters.

If **A** is a static control design block, **M** is a generalized matrix whose **Blocks** property lists **A** as the only block.

## Properties

### Blocks

Structure containing the control design blocks included in the generalized LTI model or generalized matrix. The field names of **Blocks** are the **Name** property of each control design block.

You can change some attributes of these control design blocks using dot notation. For example, if the generalized LTI model or generalized matrix **M** contains a `realp` tunable parameter **a**, you can change the current value of **a** using:

```
M.Blocks.a.Value = -1;
```

### SamplingGrid

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, **sysarr**, by taking snapshots of a linear time-varying system at times  $t = 0:10$ . The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, **M**, by independently sampling two variables, **zeta** and **w**. The following code attaches the (**zeta**,**w**) values to **M**.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)  
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display **M**, each entry in the array includes the corresponding **zeta** and **w** values.

M

M(:, :, 1, 1) [zeta=0.3, w=5] =

$$\frac{25}{s^2 + 3s + 25}$$

M(:, :, 2, 1) [zeta=0.35, w=5] =

$$\frac{25}{s^2 + 3.5s + 25}$$

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sILinearizer` populate `SamplingGrid` in this way.

**Default:** []

## Examples

### Generalized Matrix With Two Tunable Parameters

This example shows how to use algebraic combinations of tunable parameters to create the generalized matrix:

$$M = \begin{bmatrix} 1 & a + b \\ 0 & ab \end{bmatrix},$$

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

- 1 Create the tunable parameters using `realp`.

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

- 2 Define the generalized matrix using algebraic expressions of **a** and **b**.

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

**M** is a generalized matrix whose **Blocks** property contains **a** and **b**. The initial value of **M** is  $M = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix}$ , from the initial values of **a** and **b**.

- 3 (Optional) Change the initial value of the parameter **a**.

```
M.Blocks.a.Value = -3;
```

- 4 (Optional) Use **double** to display the new value of **M**.

```
double(M)
```

The new value of **M** is  $M = \begin{bmatrix} 1 & 0 \\ 0 & -9 \end{bmatrix}$ .

## More About

- “Models with Tunable Coefficients”
- “Dynamic System Models”

## See Also

`realp` | `genss` | `getValue`

**Introduced in R2011a**

## gensig

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`.

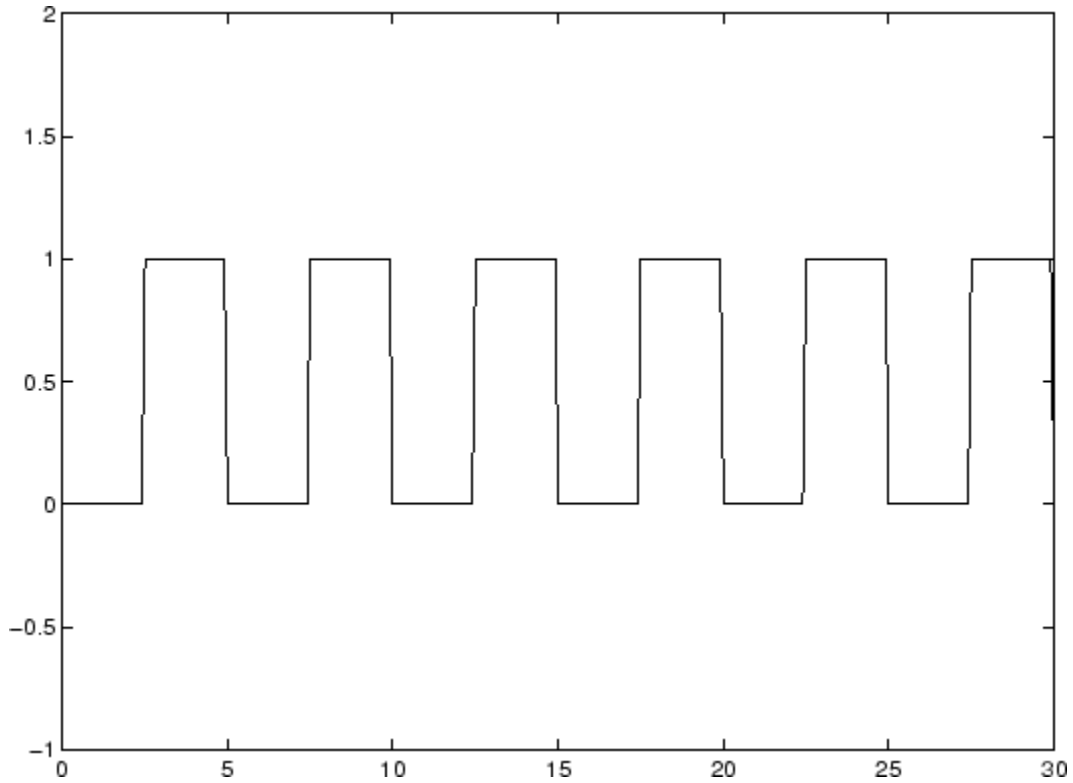
### Examples

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`

Introduced before R2006a

## genss

Generalized state-space model

### Description

Generalized state-space (**genss**) models are state-space models that include tunable parameters or components. **genss** models arise when you combine numeric LTI models with models containing tunable components (control design blocks). For more information about numeric LTI models and control design blocks, see “Models with Tunable Coefficients”.

You can use generalized state-space models to represent control systems having a mixture of fixed and tunable components. Use generalized state-space models for control design tasks such as parameter studies and parameter tuning with commands such as `sys tune` and `looptune`.

### Construction

To construct a **genss** model:

- Use `series`, `parallel`, `lft`, or `connect`, or the arithmetic operators `+`, `-`, `*`, `/`, `\`, and `^`, to combine numeric LTI models with control design blocks.
- Use `tf` or `ss` with one or more input arguments that is a generalized matrix (`genmat`) instead of a numeric array
- Convert any numeric LTI model, control design block, or `sITuner` interface (requires Simulink Control Design), for example, `sys`, to **genss** form using:

```
genssys = genss(sys)
```

When `sys` is an `sITuner` interface, `genssys` contains all the tunable blocks and analysis points specified in this interface. To compute a tunable model of a particular I/O transfer function, call `getIOTransfer(genssys, in, out)`. Here, `in` and `out` are the analysis points of interest. (Use `getPoints(sys)` to get the full list of analysis points.) Similarly, to compute a tunable model of a particular open-loop transfer function, use `getLoopTransfer(genssys, loc)`. Here, `loc` is the analysis point of interest.

## Properties

### Blocks

Structure containing the control design blocks included in the generalized LTI model or generalized matrix. The field names of **Blocks** are the **Name** property of each control design block.

You can change some attributes of these control design blocks using dot notation. For example, if the generalized LTI model or generalized matrix **M** contains a **realp** tunable parameter **a**, you can change the current value of **a** using:

```
M.Blocks.a.Value = -1;
```

### InternalDelay

Vector storing internal delays.

Internal delays arise, for example, when closing feedback loops on systems with delays, or when connecting delayed systems in series or parallel. For more information about internal delays, see “Closing Feedback Loops with Time Delays” in the *Control System Toolbox User's Guide*.

For continuous-time models, internal delays are expressed in the time unit specified by the **TimeUnit** property of the model. For discrete-time models, internal delays are expressed as integer multiples of the sample time **Ts**. For example, **InternalDelay = 3** means a delay of three sampling periods.

You can modify the values of internal delays. However, the number of entries in **sys.InternalDelay** cannot change, because it is a structural property of the model.

### InputDelay

Input delay for each input channel, specified as a scalar value or numeric vector. For continuous-time systems, specify input delays in the time unit stored in the **TimeUnit** property. For discrete-time systems, specify input delays in integer multiples of the sample time **Ts**. For example, **InputDelay = 3** means a delay of three sample times.

For a system with **Nu** inputs, set **InputDelay** to an **Nu**-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.



You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0

### **OutputDelay**

Output delays. `OutputDelay` is a numeric vector specifying a time delay for each output channel. For continuous-time systems, specify output delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify output delays in integer multiples of the sample time `Ts`. For example, `OutputDelay = 3` means a delay of three sampling periods.

For a system with `Ny` outputs, set `OutputDelay` to an `Ny`-by-1 vector, where each entry is a numerical value representing the output delay for the corresponding output channel. You can also set `OutputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all output channels

### **Ts**

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'

- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to {'controls(1)'; 'controls(2)'}

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, 'seconds'.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** '' for all input channels

### InputGroup

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### OutputName

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, 'measurements'.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all output channels

### **OutputUnit**

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, 'seconds'.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

### **OutputGroup**

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### **Name**

System name, specified as a character vector. For example, 'system\_1'.

**Default:** ''

**Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, 'System is MIMO'.

**Default:** {}

**UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

**SamplingGrid**

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times  $t = 0:10$ . The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta,w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:, :, 1, 1) [zeta=0.3, w=5] =
```

$$\frac{25}{s^2 + 3s + 25}$$

`M(:, :, 2, 1) [zeta=0.35, w=5] =`

$$\frac{25}{s^2 + 3.5s + 25}$$

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `s1Linearizer` populate `SamplingGrid` in this way.

**Default:** `[]`

## Examples

### Create Tunable Low-Pass Filter

This example shows how to create a low-pass filter with one tunable parameter  $a$ :

$$F = \frac{a}{s + a}$$

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

Create a tunable real parameter with an initial value of 10.

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

Use `tf` to create the tunable filter  $F$ .

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

F is a genss object which has the tunable parameter **a** in its **Blocks** property. You can connect F with other tunable or numeric models to create more complex control system models. For example, see “Control System with Tunable Components”.

### Create State-Space Model with Both Fixed and Tunable Parameters

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

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

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

Create the tunable parameters using `realp`.

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

Define a generalized matrix using algebraic expressions of **a** and **b**.

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

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

Create the fixed-value state-space matrices.

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

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

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

```
sys =
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 2 states, and 0 direct terms.
a: Scalar parameter, 2 occurrences.
b: Scalar parameter, 2 occurrences.
```

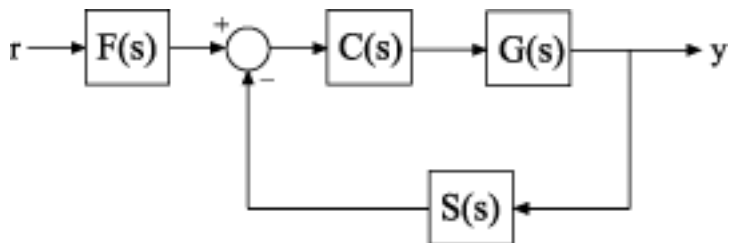
Type "ss(sys)" to see the current value, "get(sys)" to see all properties, and "sys.Blocks"

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

### Control System Model With Both Numeric and Tunable Components

This example shows how to create a tunable model of a control system that has both fixed plant and sensor dynamics and tunable control components.

Consider the the control system of the following illustration.



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

Create models representing the plant and sensor dynamics. Because the plant and sensor dynamics are fixed, represent them using numeric LTI models.

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

To model the tunable components, use Control Design Blocks. Create a tunable representation of the controller  $C$ .

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

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

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

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

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



Interconnect the models to construct a model of the complete closed-loop response from  $r$  to  $y$ .

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

```
T =
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 5 states, and 1 input.
C: Parametric PID controller, 1 occurrences.
a: Scalar parameter, 2 occurrences.
```

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

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

```
T.Blocks
```

```
ans =
```

```
struct with fields:
    C: [1×1 tunablePID]
    a: [1×1 realp]
```

When you create a **genss** model of a control system that has tunable components, you can use tuning commands such as **systemtune** to tune the free parameters to meet design requirements you specify.

## More About

### Tips

- You can manipulate **genss** models as ordinary **ss** models. Analysis commands such as **bode** and **step** evaluate the model by replacing each tunable parameter with its current value.

- “Models with Tunable Coefficients”
- “Dynamic System Models”
- “Control Design Blocks”

### **See Also**

`connect` | `feedback` | `genfrd` | `genmat` | `getValue` | `realp` | `ss` | `tf` | `tunablePID`

**Introduced in R2011a**

## get

Access model property values

### Syntax

```
Value = get(sys, 'PropertyName')  
Struct = get(sys)
```

### Description

`Value = get(sys, 'PropertyName')` returns the current value of the property `PropertyName` of the model object `sys`. `'PropertyName'` can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`). See reference pages for the individual model object types for a list of properties available for that model.

`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.

### Examples

Consider the discrete-time SISO transfer function defined by

```
h = tf(1,[1 2],0.1,'inputname','voltage','user','hello')
```

You can display all properties of `h` with

```
get(h)  
    Numerator: {[0 1]}  
  Denominator: {[1 2]}
```

```
Variable: 'z'  
IODelay: 0  
InputDelay: 0  
OutputDelay: 0  
Ts: 0.1000  
TimeUnit: 'seconds'  
InputName: {'voltage'}  
InputUnit: {''}  
InputGroup: [1x1 struct]  
OutputName: {''}  
OutputUnit: {''}  
OutputGroup: [1x1 struct]  
Name: ''  
Notes: {}  
UserData: 'hello'  
SamplingGrid: [1x1 struct]
```

or query only about the numerator and sample time values by

```
get(h, 'Numerator')
```

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

and

```
get(h, 'Ts')
```

```
ans =  
    0.1000
```

Because the numerator data (`NUMerator` property) is always stored as a cell array, the first command evaluates to a cell array containing the row vector `[0 1]`.

## More About

### Tips

An alternative to the syntax

```
Value = get(sys, 'PropertyName')
```

is the structure-like referencing

Value = sys.PropertyName

For example,

```
sys.Ts  
sys.A  
sys.user
```

return the values of the sample time, *A* matrix, and `UserData` property of the (state-space) model `sys`.

### **See Also**

set | sdata | tfdata | zpkdata | frdata | idssdata | polydata

**Introduced before R2006a**

## getBlockValue

Current value of Control Design Block in Generalized Model

### Syntax

```
val = getBlockValue(M,blockname)
[val1,val2,...] = getBlockValue(M,blockname1,blockname2,...)
S = getBlockValue(M)
```

### Description

`val = getBlockValue(M,blockname)` returns the current value of the Control Design Block `blockname` in the Generalized Model `M`. (For uncertain blocks, the “current value” is the nominal value of the block.)

`[val1,val2,...] = getBlockValue(M,blockname1,blockname2,...)` returns the values of the specified Control Design Blocks.

`S = getBlockValue(M)` returns the values of all Control Design Blocks of the generalized model in a structure. This syntax lets you transfer the block values from one generalized model to another model that uses the same Control Design Blocks, as follows:

```
S = getBlockValue(M1);
setBlockValue(M2,S);
```

### Input Arguments

**M**

Generalized LTI (`genss`) model or generalized matrix (`genmat`).

**blockname**

Name of the Control Design Block in the model `M` whose current value is evaluated.

To get a list of the Control Design Blocks in `M`, enter `M.Blocks`.

## Output Arguments

### val

Numerical LTI model or numerical value, equal to the current value of the Control Design Block `blockname`.

### S

Current values of all Control Design Blocks in `M`, returned as a structure. The names of the fields in `S` are the names of the blocks in `M`. The values of the fields are numerical LTI models or numerical values equal to the current values of the corresponding Control Design Blocks.

## Examples

### Get Current Values of Single Blocks

Create a tunable `genss` model, and evaluate the current value of the Control Design Blocks of the model.

Typically, you use `getBlockValue` to retrieve the tuned values of control design blocks after tuning the `genss` model using a tuning command such as `sys tune`. For this example, create the model and retrieve the initial block values.

```
G = zpk([], [-1, -1], 1);
C = tunablePID('C', 'PID');
a = realp('a', 10);
F = tf(a, [1 a]);
T = feedback(G*C, 1)*F;
```

```
Cval = getBlockValue(T, 'C')
```

Continuous-time I-only controller:

$$K_i * \frac{1}{s}$$

With `Ki = 0.001`

`Cval` is a numeric pid controller object.

```
aval = getBlockValue(T, 'a')  
aval =  
    10
```

`aval` is a numeric scalar, because `a` is a real scalar parameter.

### Get All Current Values as Structure

Using the `genss` model of the previous example, get the current values of all blocks in the model.

```
G = zpk([], [-1, -1], 1);  
C = tunablePID('C', 'PID');  
a = realp('a', 10);  
F = tf(a, [1 a]);  
T = feedback(G*C, 1)*F;  
  
S = getBlockValue(T)
```

```
S =  
  
    C: [1x1 pid]  
    a: 10
```

### More About

- Generalized Model
- Control Design Block

### See Also

[getValue](#) | [setBlockValue](#) | [showBlockValue](#)

**Introduced in R2011b**



# getCompSensitivity

Complementary sensitivity function from generalized model of control system

## Syntax

```
T = getCompSensitivity(CL,location)
T = getSensitivity(CL,location,opening)
```

## Description

`T = getCompSensitivity(CL,location)` returns the complementary sensitivity measured at the specified location for a generalized model of a control system.

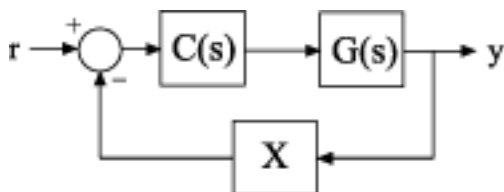
`T = getSensitivity(CL,location,opening)` specifies additional loop openings for the complementary sensitivity function calculation. Use an opening, for example, to calculate the complementary sensitivity function of an inner loop, with the outer loop open.

If `opening` and `location` list the same point, the software opens the loop after adding the disturbance signal at the point.

## Examples

### Complementary Sensitivity Function at a Location

Compute the complementary sensitivity at the plant output,  $X$ , of the control system of the following illustration.



Create a model of the system by specifying and connecting a numeric LTI plant model `G`, a tunable controller `C`, and the `AnalysisPoint` block `X`. Use the `AnalysisPoint` block to mark the location where you assess the complementary sensitivity, which in this example is the plant output.

```
G = tf([1],[1 5]);
C = tunablePID('C','p');
C.Kp.Value = 3;
X = AnalysisPoint('X');
CL = feedback(G*C,X);
```

`CL` is a `genss` model that represents the closed-loop response of the control system from  $r$  to  $y$ . Examine the Control Design Blocks of the model.

`CL.Blocks`

```
ans =
```

```
struct with fields:
    C: [1×1 tunablePID]
    X: [1×1 AnalysisPoint]
```

The model's blocks include the `AnalysisPoint` block, `X`, that identifies the analysis-point location.

Calculate the complementary sensitivity,  $T$ , at `X`.

```
T = getCompSensitivity(CL,'X')
```

```
T =
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 1 states, and 1
C: Parametric PID controller, 1 occurrences.
X: Analysis point, 1 channels, 1 occurrences.
```

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

`getCompSensitivity` preserves the Control Design Blocks of `CL`, and returns a `genss` model. To get a numeric model, you can convert `T` to transfer-function form, using the current value of the tunable block.

```
Tnum = tf(T)
```

```
Tnum =
```

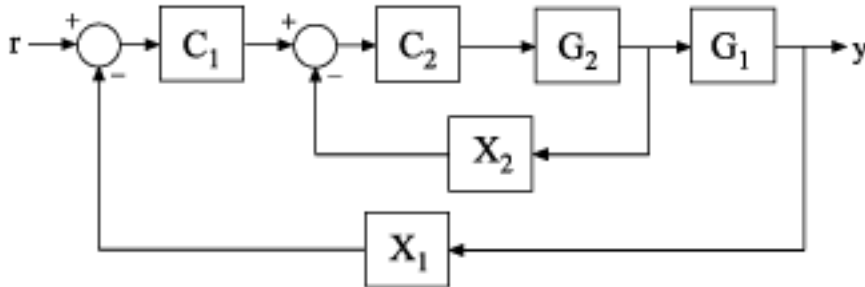
```
From input "X" to output "X":
```

```
-3
-----
s + 8
```

```
Continuous-time transfer function.
```

### Specify Additional Loop Opening for Complementary Sensitivity Function Calculation

In the multiloop system of the following illustration, calculate the inner-loop sensitivity at the output of  $G_2$ , with the outer loop open.



Create a model of the system by specifying and connecting the numeric plant models, tunable controllers, and `AnalysisPoint` blocks.  $G_1$  and  $G_2$  are plant models,  $C_1$  and  $C_2$  are tunable controllers, and  $X_1$  and  $X_2$  are `AnalysisPoint` blocks that mark potential loop-opening locations.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
```

```
CL = feedback(G1*feedback(G2*C2,X2)*C1,X1);
```

Calculate the complementary sensitivity,  $T$ , at X2, with the outer loop open at X1. Specifying X1 as the third input argument tells `getCompSensitivity` to open the loop at that location.

```
T = getCompSensitivity(CL,'X2','X1');  
tf(T)
```

```
ans =
```

```
From input "X2" to output "X2":  
    -s - 2  
-----  
s^2 + 1.2 s + 12
```

Continuous-time transfer function.

## Input Arguments

### **CL** — Model of control system

generalized state-space model

Model of a control system, specified as a generalized state-space model (`genss`).

Locations at which you can perform sensitivity analysis or open loops are marked by `AnalysisPoint` blocks in `CL`. Use `getPoints(CL)` to get the list of such locations.

### **location** — Location

character vector | cell array of character vectors

Location at which you calculate the complementary sensitivity function, specified as a character vector or cell array of character vectors. To extract the complementary sensitivity function at multiple locations, use a cell array of character vectors.

Each specified location must match an analysis point in `CL`. Analysis points are marked using `AnalysisPoint` blocks. To get the list of available analysis points in `CL`, use `getPoints(CL)`.

Example: `'u'` or `{'u','y'}`

**opening — Additional loop opening**

character vector | cell array of character vectors

Additional loop opening used to calculate the complementary sensitivity function, specified as a character vector or cell array of character vectors. To open the loop at multiple locations, use a cell array of character vectors.

Each specified opening must match an analysis point in `CL`. Analysis points are marked using `AnalysisPoint` blocks. To get the list of available analysis points in `CL`, use `getPoints(CL)`.

Use an opening, for example, to calculate the complementary sensitivity function of an inner loop, with the outer loop open.

If `opening` and `location` list the same point, the software opens the loop after adding the disturbance signal at the point.

Example: `'y_outer'` or `{'y_outer', 'y_outer2'}`

## Output Arguments

**T — Complementary sensitivity function**

generalized state-space model

Complementary sensitivity function of the control system, `T`, measured at `location`, returned as a generalized state-space model (`genss`).

- If `location` specifies a single analysis point, then `T` is a SISO `genss` model.
- If `location` is a vector signal, or specifies multiple analysis points, then `T` is a MIMO `genss` model.

## More About

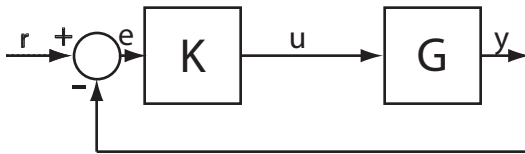
**Complementary Sensitivity**

The *complementary sensitivity function*,  $T$ , at a point is the closed-loop transfer function around the feedback loop measured at the specified location. It is related to the open-loop transfer function,  $L$ , and the sensitivity function,  $S$ , at the same point as follows:

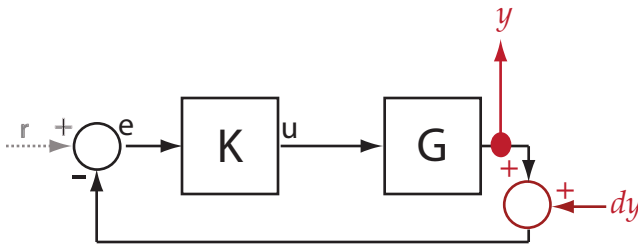
$$T = \frac{L}{1-L} = S - 1.$$

Use `getLoopTransfer` and `getSensitivity` to compute  $L$  and  $S$ .

Consider the following model:



The complementary sensitivity,  $T$ , at  $y$  is defined as the transfer function from  $dy$  to  $y$ .

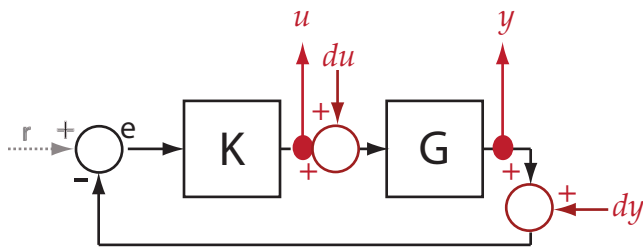


Observe that, in contrast to the sensitivity function, the disturbance,  $dy$ , is added *after* the measurement,  $y$ .

$$\begin{aligned} y &= -GK(y + dy) \\ \rightarrow y &= -GKy - GKdy \\ \rightarrow (I + GK)y &= -GKdy \\ \rightarrow y &= \underbrace{-(I + GK)^{-1}GK}_{\hat{T}} dy. \end{aligned}$$

Here,  $I$  is an identity matrix of the same size as  $GK$ . The complementary sensitivity transfer function at  $y$  is equal to -1 times the closed-loop transfer function from  $r$  to  $y$ .

Complementary sensitivity at multiple locations, for example,  $u$  and  $y$ , is defined as the MIMO transfer function from the disturbances to measurements:



$$T = \begin{bmatrix} T_{du \rightarrow u} & T_{dy \rightarrow u} \\ T_{du \rightarrow y} & T_{dy \rightarrow y} \end{bmatrix}.$$

## See Also

AnalysisPoint | genss | getCompSensitivity | getIOTransfer |  
getLoopTransfer | getPoints | getSensitivity | getValue | systune

Introduced in R2014a

## getComponents

Extract SISO control components from a 2-DOF PID controller

### Syntax

```
[C,X] = getComponents(C2,looptype)
```

### Description

`[C,X] = getComponents(C2,looptype)` decomposes the 2-DOF PID controller `C2` into two SISO control components. One of the control components, `C`, is a 1-DOF PID controller. The other, `X`, is a SISO dynamic system. When `C` and `X` are connected in the loop structure specified by `looptype`, the resulting closed-loop system is equivalent to the 2-DOF control loop.

For more information about 2-DOF PID control architectures, see “Two-Degree-of-Freedom PID Controllers”.

### Examples

#### Extract SISO Components from 2-DOF PID Controller

Decompose a 2-DOF PID controller into SISO control components, using each of the feedforward, feedback, and filter configurations.

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

```
G = tf(1,[1 0.5 0.1]);  
C2 = pidtune(G,'pidf2',1.5)
```

`C2 =`

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$



with  $K_p = 1.12$ ,  $K_i = 0.23$ ,  $K_d = 1.3$ ,  $T_f = 0.122$ ,  $b = 0.664$ ,  $c = 0.0136$

Continuous-time 2-DOF PIDF controller in parallel form.

**C2** is a `pid2` controller object, with two inputs and one output. Decompose **C2** into SISO control components using the feedforward configuration.

```
[Cff,Xff] = getComponents(C2, 'feedforward')
```

Cff =

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f s + 1}$$

with  $K_p = 1.12$ ,  $K_i = 0.23$ ,  $K_d = 1.3$ ,  $T_f = 0.122$

Continuous-time PIDF controller in parallel form.

Xff =

$$\frac{-10.898 (s+0.2838)}{(s+8.181)}$$

Continuous-time zero/pole/gain model.

As the display shows, this command returns the SISO PID controller **Cff** as a `pid` object. The feedforward compensator **X** is returned as a `zpk` object.

Decompose **C2** using the feedback configuration. In this case as well, **Cfb** is a `pid` controller object, and the feedback compensator **X** is a `zpk` model.

```
[Cfb,Xfb] = getComponents(C2, 'feedback');
```

Decompose **C2** using the filter configuration. Again, the components are a SISO `pid` controller and a `zpk` model representing the prefilter.

```
[Cfr,Xfr] = getComponents(C2, 'filter');
```

- “Decompose a 2-DOF PID Controller into SISO Components”

## Input Arguments

### C2 — 2-DOF PID controller

pid2 object | pidstd2 object

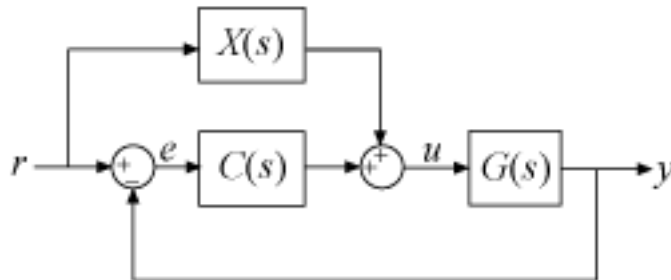
2-DOF PID controller to decompose, specified as a `pid2` or `pidstd2` controller object.

### looptype — Loop structure

'feedforward' (default) | 'feedback' | 'filter'

Loop structure for decomposing the 2-DOF controller, specified as 'feedforward', 'feedback', or 'filter'. These correspond to the following control decompositions and architectures:

- 'feedforward' —  $C$  is a conventional SISO PID controller that takes the error signal as its input.  $X$  is a feedforward controller, as shown:



If **C2** is a continuous-time, parallel-form controller, then the components are given by:

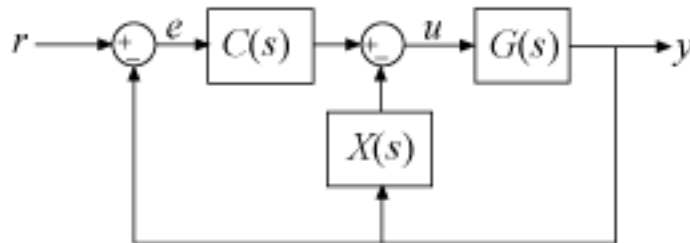
$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1},$$

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

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

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

- 'feedback' — C is a conventional SISO PID controller that takes the error signal as its input. X is a feedback controller from  $y$  to  $u$ , as shown:



If C2 is a continuous-time, parallel-form controller, then the components are given by:

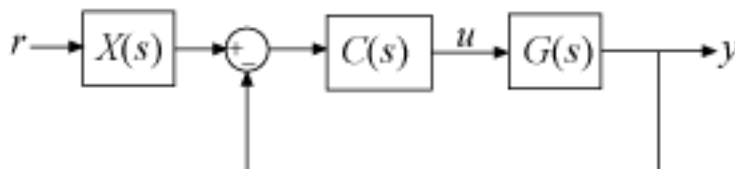
$$C(s) = bK_p + \frac{K_i}{s} + \frac{cK_d s}{T_f s + 1},$$

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

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

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

- 'filter' — X is a prefilter on the reference signal. C is a conventional SISO PID controller that takes as its input the difference between the filtered reference and the output, as shown:



If **C2** is a continuous-time, parallel-form controller, then the components are given by:

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

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

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

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

The formulas shown above pertain to continuous-time, parallel-form controllers. Standard-form controllers and controllers in discrete time can be decomposed into analogous configurations. The `getComponents` command works on all 2-DOF PID controller objects.

## Output Arguments

### **C** — SISO PID controller

`pid` object | `pidstd` object

SISO PID controller, returned as a `pid` or `pidstd` controller object. The form of **C** corresponds to the form of the input controller **C2**. For example, if **C2** is a standard-form `pidstd2` controller, then **C** is a `pidstd` object.

The precise functional form of **C** depends on the loop structure you specify with the `looptype` argument, as described in “Input Arguments” on page 2-328.

### **X** — SISO control component

`zpk` model

SISO control component, specified as a zero-pole-gain (`zpk`) model. The precise functional form of **X** depends on the loop structure you specify with the `looptype` argument, as described in “Input Arguments” on page 2-328.

## More About

- “Two-Degree-of-Freedom PID Controllers”

## See Also

[make1DOF](#) | [make2DOF](#) | [pid2](#) | [pidstd2](#)

**Introduced in R2015b**

## getData

Get current values of tunable-surface coefficients

### Syntax

```
Kco = getData(K)  
KcoJ = getData(K,J)
```

### Description

`Kco = getData(K)` extracts the current values of the tunable surface `K`. `K` is a `tunableSurface` object that represents the parametric gain surface:

$$K(n(\sigma)) = K_0 + K_1 F_1(n(\sigma)) + \dots + K_M F_M(n(\sigma)).$$

$F_1, \dots, F_M$  are basis functions, and  $n(\sigma)$  is a normalization function that maps the range of each scheduling-variable  $\sigma$  onto  $[-1, 1]$ . `KCO` is the array  $[K_0, \dots, K_M]$ .

`KcoJ = getData(K,J)` extracts the current value of the coefficient of the  $J$ th basis function  $F_J$ . Use `J = 0` to get the constant coefficient  $K_0$ .

### Input Arguments

#### **K** — Gain surface

`tunableSurface` object

Gain surface, specified as a `tunableSurface` object.

#### **J** — Index of basis function

nonnegative integer

Index of basis function, specified as a nonnegative integer. To extract the constant coefficient  $K_0$ , use `J = 0`.

## Output Arguments

### **Kco** — Current coefficient values

array

Current coefficient values of the tunable surface, returned as an array.

If the tunable surface  $K$  is a scalar-valued gain, then the length of  $K$  is  $(M+1)$ , where  $M$  is the number of basis functions in the parameterization. For example, if  $K$  represents the tunable gain surface:

$$K(\alpha, V) = K_0 + K_1\alpha + K_2V + K_3\alpha V,$$

then  $KCO$  is the 1-by-4 vector  $[K_0, K_1, K_2, K_3]$ .

For array-valued gains, each coefficient expands to the I/O dimensions of the gain. These expanded coefficients are concatenated horizontally in  $KCO$ . (See `tunableSurface`.) For example, for a two-input, two-output gain surface,  $KCO$  has dimensions  $[2, 2(M+1)]$ .

### **KcoJ** — Coefficient of $J$ th basis function

scalar | array

Coefficient of the  $J$ th basis function in the tunable surface parameterization, returned as a scalar or an array.

If the tunable surface  $K$  is a scalar-valued gain, then  $KCOJ$  is a scalar. If  $K$  is an array-valued gain, then  $KCOJ$  is an array that matches the I/O dimensions of the gain.

## See Also

`evalSurf` | `setData` | `tunableSurface` | `viewSurf`

**Introduced in R2015b**

## getDelayModel

State-space representation of internal delays

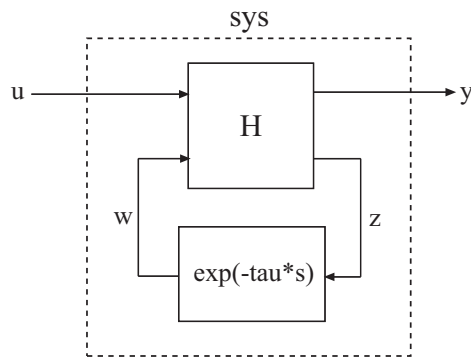
### Syntax

```
[H,tau] = getDelayModel(sys)
```

```
[A,B1,B2,C1,C2,D11,D12,D21,D22,E,tau] = getDelayModel(sys)
```

### Description

`[H,tau] = getDelayModel(sys)` decomposes a state-space model `sys` with internal delays into a delay-free state-space model, `H`, and a vector of internal delays, `tau`. The relationship among `sys`, `H`, and `tau` is shown in the following diagram.



`[A,B1,B2,C1,C2,D11,D12,D21,D22,E,tau] = getDelayModel(sys)` returns the set of state-space matrices and internal delay vector, `tau`, that explicitly describe the state-space model `sys`. These state-space matrices are defined by the state-space equations:

- Continuous-time `sys`:



$$\begin{aligned}
 E \frac{dx(t)}{dt} &= Ax(t) + B_1u(t) + B_2w(t) \\
 y(t) &= C_1x(t) + D_{11}u(t) + D_{12}w(t) \\
 z(t) &= C_2x(t) + D_{21}u(t) + D_{22}w(t) \\
 w(t) &= z(t - \tau)
 \end{aligned}$$

- Discrete-time **sys**:

$$\begin{aligned}
 Ex[k+1] &= Ax[k] + B_1u[k] + B_2w[k] \\
 y[k] &= C_1x[k] + D_{11}u[k] + D_{12}w[k] \\
 z[k] &= C_2x[k] + D_{21}u[k] + D_{22}w[k] \\
 w[k] &= z[k - \tau]
 \end{aligned}$$

## Input Arguments

### **sys**

Any state-space (ss) model.

## Output Arguments

### **H**

Delay-free state-space model (ss). **H** results from decomposing **sys** into a delay-free component and a component  $\exp(-\tau*s)$  that represents all internal delays.

If **sys** has no internal delays, **H** is equal to **sys**.

### **tau**

Vector of internal delays of **sys**, expressed in the time units of **sys**. The vector **tau** results from decomposing **sys** into a delay-free state-space model **H** and a component  $\exp(-\tau*s)$  that represents all internal delays.

If **sys** has no internal delays, **tau** is empty.

**A, B1, B2, C1, C2, D11, D12, D21, D22, E**

Set of state-space matrices that, with the internal delay vector `tau`, explicitly describe the state-space model `sys`.

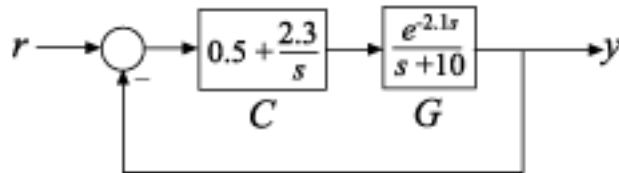
For explicit state-space models ( $E = I$ , or `sys.e = []`), the output  $E = []$ .

If `sys` has no internal delays, the outputs `B2`, `C2`, `D12`, `D21`, and `D22` are all empty (`[]`).

## Examples

### Get Delay-Free State-Space Model and Internal Delay

Decompose the following closed-loop system with internal delay into a delay-free component and a component representing the internal delay.



Create the closed-loop model `sys` from `r` to `y`.

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

`sys` is a state-space (SS) model with an internal delay that arises from closing the feedback loop on a plant with an input delay.

Decompose `sys` into a delay-free state-space model and the value of the internal delay.

```
[H,tau] = getDelayModel(sys);
```

Confirm that the internal delay matches the original input delay on the plant.

```
tau
```

```
tau =  
    2.1000
```

## More About

- “Internal Delays”

## See Also

setDelayModel

**Introduced in R2006a**

## getGainCrossover

Crossover frequencies for specified gain

### Syntax

```
wc = getGainCrossover(sys,gain)
```

### Description

`wc = getGainCrossover(sys,gain)` returns the vector `wc` of frequencies at which the frequency response of the dynamic system model, `sys`, has principal gain of `gain`. For SISO systems, the principal gain is the frequency response. For MIMO models, the principal gain is the largest singular value of `sys`.

### Examples

#### Unity Gain Crossover

Find the 0dB crossover frequencies of a single-loop control system with plant given by:

$$G(s) = \frac{1}{(s+1)^3},$$

and PI controller given by:

$$C(s) = 1.14 + \frac{0.454}{s}.$$

```
G = zpk([],[-1,-1,-1],1);  
C = pid(1.14,0.454);  
sys = G*C;  
wc = getGainCrossover(sys,1)
```

```
wc =
```

0.5214

The 0 dB crossover frequencies are the frequencies at which the open-loop response  $\text{sys} = G \cdot C$  has unity gain. Because this system only crosses unity gain once, `getGainCrossover` returns a single value.

### Notch Filter Stopband

Find the 20 dB stopband of

$$\text{sys} = \frac{s^2 + 0.05s + 100}{s^2 + 5s + 100}.$$

`sys` is a notch filter centered at 10 rad/s.

```
sys = tf([1 0.05 100],[1 5 100]);
gain = db2mag(-20);
wc = getGainCrossover(sys,gain)
```

`wc =`

```
9.7531
10.2531
```

The `db2mag` command converts the gain value of -20 dB to absolute units. The `getGainCrossover` command returns the two frequencies that define the stopband.

## Input Arguments

### **sys** — Input dynamic system

dynamic system model

Input dynamic system, specified as any SISO or MIMO dynamic system model.

### **gain** — Input gain

positive real scalar

Input gain in absolute units, specified as a positive real scalar.

- If `sys` is a SISO model, the gain is the frequency response magnitude of `sys`.
- If `sys` is a MIMO model, gain means the largest singular value of `sys`.

## Output Arguments

### **wc** — Crossover frequencies

column vector

Crossover frequencies, returned as a column vector. This vector lists the frequencies at which the gain or largest singular value of `sys` is `gain`.

## More About

### Algorithms

`getGainCrossover` computes gain crossover frequencies using structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see <http://slicot.org>.

- “Dynamic System Models”

### See Also

`bandwidth` | `bode` | `freqresp` | `getPeakGain` | `sigma`

**Introduced in R2012a**

# getIOTransfer

Closed-loop transfer function from generalized model of control system

## Syntax

```
H = getIOTransfer(T,in,out)
H = getIOTransfer(T,in,out,openings)
```

## Description

`H = getIOTransfer(T,in,out)` returns the transfer function from specified inputs to specified outputs of a control system, computed from a closed-loop generalized model of the control system.

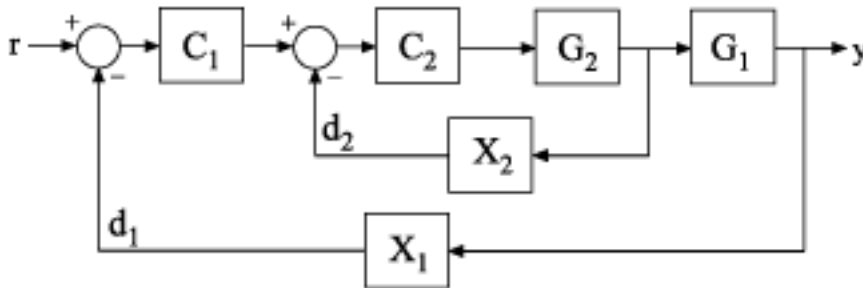
`H = getIOTransfer(T,in,out,openings)` returns the transfer function calculated with one or more loops open.

## Examples

### Closed-Loop Responses of Control System Model

Analyze responses of a control system by using `getIOTransfer` to compute responses between various inputs and outputs of a closed-loop model of the system.

Consider the following control system.



Create a `genss` model of the system by specifying and connecting the numeric plant models `G1` and `G2`, the tunable controllers `C1` and `C2`, and the `AnalysisPoint` blocks `X1` and `X2` that mark potential loop-opening or signal injection sites.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
T = feedback(G1*feedback(G2*C2,X2)*C1,X1);
T.InputName = 'r';
T.OutputName = 'y';
```

If you tuned the free parameters of this model (for example, using the tuning command `systemtune`), you might want to analyze the tuned system performance by examining various system responses.

For example, examine the response at the output, `y`, to a disturbance injected at the point `d1`.

```
H1 = getIOTransfer(T,'X1','y');
```

`H1` represents the closed-loop response of the control system to a disturbance injected at the implicit input associated with the `AnalysisPoint` block `X1`, which is the location of `d1`:





H1 is a `genss` model that includes the tunable blocks of T. If you have tuned the free parameters of T, H1 allows you to validate the disturbance response of your tuned system. For example, you can use analysis commands such as `bodeplot` or `stepplot` to examine the responses of H1. You can also use `getValue` to obtain the current value of H1, in which all the tunable blocks are evaluated to their current numeric values.

Similarly, examine the response at the output to a disturbance injected at the point  $d_2$ .

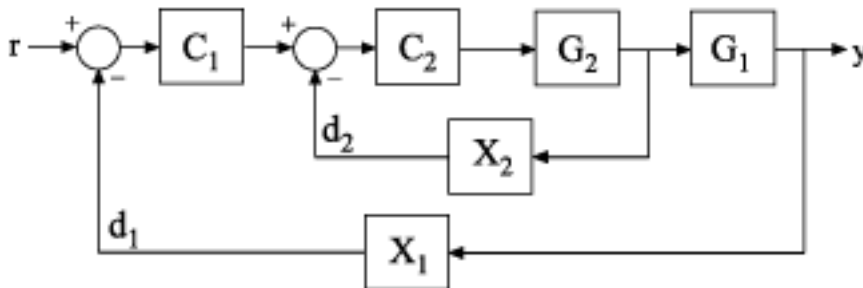
```
H2 = getIOTransfer(T, 'X2', 'y');
```

You can also generate a two-input, one-output model representing the response of the control system to simultaneous disturbances at both  $d_1$  and  $d_2$ . To do so, provide `getIOTransfer` with a cell array that specifies the multiple input locations.

```
H = getIOTransfer(T, {'X1', 'X2'}, 'y');
```

### Responses with Some Loops Open and Others Closed

Compute the response from  $r$  to  $y$  of the following cascaded control system, with the inner loop open, and the outer loop closed.



Create a `genss` model of the system by specifying and connecting the numeric plant models `G1` and `G2`, the tunable controllers `C1` and `C2`, and the `AnalysisPoint` blocks `X1` and `X2` that mark potential loop-opening or signal injection sites.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
T = feedback(G1*feedback(G2*C2,X2)*C1,X1);
T.InputName = 'r';
T.OutputName = 'y';
```

If you tuned the free parameters of this model (for example, using the tuning command `systemtune`), you might want to analyze the tuned system performance by examining various system responses.

For example, compute the response of the system with the inner loop open, and the outer loop closed.

```
H = getIOTransfer(T,'r','y','X2');
```

By default, the loops are closed at the analysis points `X1` and `X2`. Specifying `'X2'` for the `openings` argument causes `getIOTransfer` to open the loop at `X2` for the purposes of computing the requested transfer from `r` to `y`. The switch at `X1` remains closed for this computation.

## Input Arguments

### **T** — Model of control system

generalized state-space model

Model of a control system, specified as a generalized state-space model (`genss`).

### **in** — Input to extracted transfer function

character vector | cell array of character vectors

Input to extracted transfer function, specified as a character vector or cell array of character vectors. To extract a multiple-input transfer function from the control system, use a cell array of character vectors. Each specified input must match either:

- An input of the control system model `T`; that is, a channel name from `T.InputName`.

- An analysis point in  $T$ , corresponding to a channel of an `AnalysisPoint` block in  $T$ . To get the list of available analysis points in  $T$ , use `getPoints(T)`.

When you specify an analysis point as an input `in`, `getIOTransfer` uses the input implicitly associated with the `AnalysisPoint` channel, arranged as follows.



This input signal models a disturbance entering at the output of the switch.

If an analysis point has the same name as an input of  $T$ , then `getIOTransfer` uses the input of  $T$ .

Example: `{ 'r', 'X1' }`

#### **out** — Output of extracted transfer function

character vector | cell array of character vectors

Output of extracted transfer function, specified as a character vector or cell array of character vectors. To extract a multiple-output transfer function from the control system, use a cell array of character vectors. Each specified output must match either:

- An output of the control system model  $T$ ; that is, a channel name from `T.OutputName`.
- An analysis point in  $T$ , corresponding to a channel of an `AnalysisPoint` block in  $T$ . To get the list of available analysis points in  $T$ , use `getPoints(T)`.

When you specify an analysis point as an output `out`, `getIOTransfer` uses the output implicitly associated with the `AnalysisPoint` channel, arranged as follows.



If an analysis point has the same name as an output of  $T$ , then `getIOTransfer` uses the output of  $T$ .

Example: { 'y', 'X2' }

### **openings** — Locations for opening feedback loops

character vector | cell array of character vectors

Locations for opening feedback loops for computation of the response from **in** to **out**, specified as a character vector or cell array of character vectors that identify analysis points in **T**. Analysis points are marked by **AnalysisPoint** blocks in **T**. To get the list of available analysis points in **T**, use **getPoints(T)**.

Use **openings** when you want to compute the response from **in** to **out** with some loops in the control system open. For example, in a cascaded loop configuration, you can calculate the response from the system input to the system output with the inner loop open.

## Output Arguments

### **H** — Closed-loop transfer function

generalized state-space model

Closed-loop transfer function of the control system **T** from **in** to **out**, returned as a generalized state-space model (**genss**).

- If both **in** and **out** specify a single signal, then **T** is a SISO **genss** model.
- If **in** or **out** specifies multiple signals, then **T** is a MIMO **genss** model.

## More About

### Tips

- You can use **getIOTransfer** to extract various subsystem responses, given a generalized model of the overall control system. This is useful for validating responses of a control system that you tune with tuning commands such as **systune**.

For example, in addition to evaluating the overall response of a tuned control system from inputs to outputs, you can use **getIOTransfer** to extract the transfer function from a disturbance input to a system output. Evaluate the responses of that transfer function (such as with **step** or **bode**) to confirm that the tuned system meets your disturbance rejection requirements.

- `getIOTransfer` is the `genss` equivalent to the Simulink Control Design `getIOTransfer` command, which works with the `sITuner` and `sLLinearizer` interfaces. Use the Simulink Control Design command when your control system is modeled in Simulink.

### **See Also**

`AnalysisPoint` | `genss` | `getIOTransfer` | `getLoopTransfer` | `getPoints` | `system`

**Introduced in R2012b**

## getLFTModel

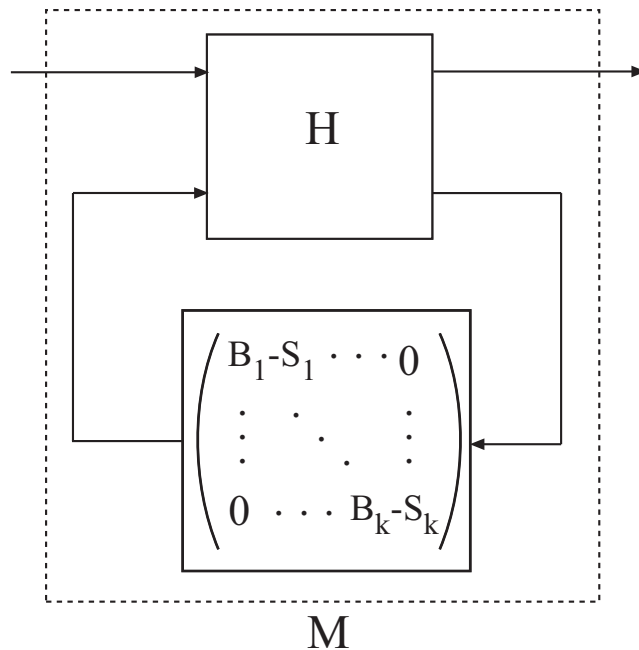
Decompose generalized LTI model

### Syntax

`[H,B,S] = getLFTModel(M)`

### Description

`[H,B,S] = getLFTModel(M)` extracts the components  $H$ ,  $B$ , and  $S$  that make up the Generalized matrix or Generalized LTI model  $M$ . The model  $M$  decomposes into  $H$ ,  $B$ , and  $S$ . These components are related to  $M$  as shown in the following illustration.



The cell array  $B$  contains the Control Design Blocks of  $M$ . The component  $H$  is a numeric matrix, `ss` model, or `frd` model that describes the fixed portion of  $M$  and the

interconnections between the blocks of **B**. The matrix  $S = \text{blkdiag}(S_1, \dots, S_k)$  contains numerical offsets that ensure that the interconnection is well-defined when the current (nominal) value of **M** is finite.

You can recombine **H**, **B**, and **S** into **M** using `lft`, as follows:

```
M = lft(H,blkdiag(B{:})-S);
```

## Input Arguments

### **M**

Generalized LTI model (`genss` or `genfrd`) or Generalized matrix (`genmat`).

## Output Arguments

### **H**

Matrix, `ss` model, or `frd` model describing the numeric portion of **M** and how it the numeric portion is connected to the Control Design Blocks of **M**.

### **B**

Cell array of Control Design Blocks (for example, `realp` or `tunableSS`) of **M**.

### **S**

Matrix of offset values. The software might introduce offsets when you build a Generalized model to ensure that **H** is finite when the current (nominal) value of **M** is finite.

## More About

### Tips

- `getLFTModel` gives you access to the internal representation of Generalized LTI models and Generalized Matrices. For more information about this representation, see “Internal Structure of Generalized Models”.

- “Generalized Matrices”
- “Generalized and Uncertain LTI Models”
- “Models with Tunable Coefficients”
- “Internal Structure of Generalized Models”

### **See Also**

genfrd | genss | genmat | lft | getValue | nblocks

**Introduced in R2011a**



# getLoopTransfer

Open-loop transfer function of control system

## Syntax

```
L = getLoopTransfer(T,Locations)
L = getLoopTransfer(T,Locations,sign)
L = getLoopTransfer(T,Locations,sign,openings)
```

## Description

`L = getLoopTransfer(T,Locations)` returns the point-to-point open-loop transfer function of a control system measured at specified analysis points. The point-to-point open-loop transfer function is the open-loop response obtained by injecting signals at the specified locations and measuring the return signals at the same locations.

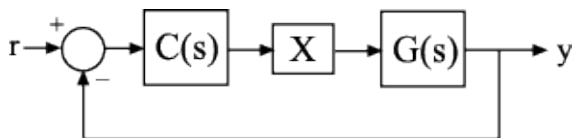
`L = getLoopTransfer(T,Locations,sign)` specifies the feedback sign for calculating the open-loop response. The relationship between the closed-loop response `T` and the open-loop response `L` is `T = feedback(L,1,sign)`.

`L = getLoopTransfer(T,Locations,sign,openings)` specifies additional loop-opening locations to open for computing the open-loop response at `Locations`.

## Examples

### Open-Loop Transfer Function at Analysis Point

Compute the open-loop response of the following control system model at an analysis point specified by an `AnalysisPoint` block, `X`.



Create a model of the system by specifying and connecting a numeric LTI plant model, **G**, a tunable controller, **C**, and the **AnalysisPoint**, **X**.

```
G = tf([1 2],[1 0.2 10]);
C = tunablePID('C','pi');
X = AnalysisPoint('X');
T = feedback(G*X*C,1);
```

**T** is a **genss** model that represents the closed-loop response of the control system from **r** to **y**. The model contains **AnalysisPoint** block **X**, which identifies the potential loop-opening location.

Calculate the open-loop point-to-point loop transfer at location **X**.

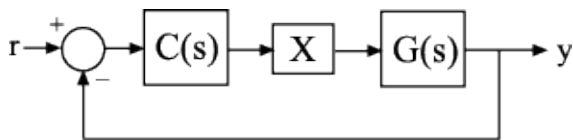
```
L = getLoopTransfer(T,'X');
```

This command computes the positive-feedback transfer function you would obtain by opening the loop at **X**, injecting a signal into **G**, and measuring the resulting response at the output of **C**. By default, **getLoopTransfer** computes the positive feedback transfer function. In this example, the positive feedback transfer function is  $L(s) = -G(s)C(s)$

The output **L** is a **genss** model that includes the tunable block **C**. You can use **getValue** to obtain the current value of **L**, in which all the tunable blocks of **L** are evaluated to their current numeric value.

### Negative-Feedback Open-Loop Transfer Function

Compute the negative-feedback open-loop transfer of the following control system model at an analysis point specified by an **AnalysisPoint** block, **X**.



Create a model of the system by specifying and connecting a numeric LTI plant model **G**, a tunable controller **C**, and the **AnalysisPoint** block **X**.

```
G = tf([1 2],[1 0.2 10]);
C = tunablePID('C','pi');
X = AnalysisPoint('X');
T = feedback(G*X*C,1);
```

T is a `genss` model that represents the closed-loop response of the control system from  $r$  to  $y$ . The model contains the `AnalysisPoint` block X that identifies the potential loop-opening location.

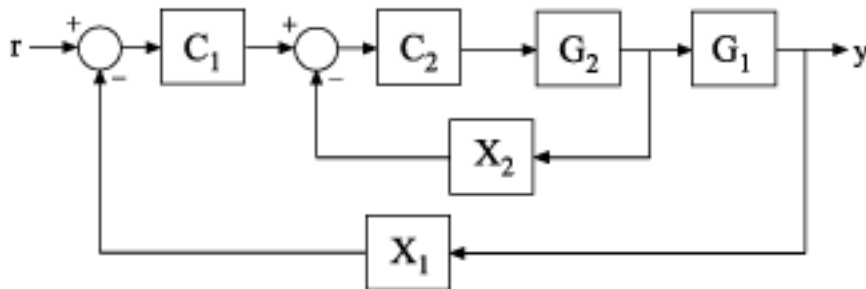
Calculate the open-loop point-to-point loop transfer at the location X.

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

This command computes the open-loop transfer function from the input of G to the output of C, assuming that the loop is closed with negative feedback. That is, the relationships between L and T is given by  $T = \text{feedback}(L, 1)$ . In this example, the positive feedback transfer function is  $L(s) = G(s)C(s)$

### Transfer Function with Additional Loop Openings

Compute the open-loop response of the inner loop of the following cascaded control system, with the outer loop open.



Create a model of the system by specifying and connecting the numeric plant models G1 and G2, the tunable controllers C1, and the `AnalysisPoint` blocks X1 and X2 that mark potential loop-opening locations.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
```

```
T = feedback(G1*feedback(G2*C2,X2)*C1,X1);
```

Compute the negative-feedback open-loop response of the inner loop, at the location X2, with the outer loop opened at X1.

```
L = getLoopTransfer(T, 'X2', -1, 'X1');
```

By default, the loop is closed at the analysis-point location marked by the `AnalysisPoint` block X1. Specifying 'X1' for the `openings` argument causes `getLoopTransfer` to open the loop at X1 for the purposes of computing the requested loop transfer at X2. In this example, the negative-feedback open-loop response  $L(s) = G_2(s)C_2(s)$ .

## Input Arguments

### **T** — Model of control system

generalized state-space model

Model of a control system, specified as a Generalized State-Space (`genss`) Model. Locations at which you can open loops and perform open-loop analysis are marked by `AnalysisPoint` blocks in T.

### **Locations** — Analysis-point locations

character vector | cell array of character vectors

Analysis-point locations in the control system model at which to compute the open-loop point-to-point response, specified as a character vector or a cell array of character vectors that identify analysis-point locations in T.

Analysis-point locations are marked by `AnalysisPoint` blocks in T. An `AnalysisPoint` block can have single or multiple channels. The `Location` property of an `AnalysisPoint` block gives names to these feedback channels.

The name of any channel in an `AnalysisPoint` block in T is a valid entry for the `Locations` argument to `getLoopTransfer`. To get a full list of available analysis points in T, use `getPoints(T)`.

`getLoopTransfer` computes the open-loop response you would obtain by injecting a signal at the implicit input associated with an `AnalysisPoint` channel, and measuring the response at the implicit output associated with the channel. These implicit inputs and outputs are arranged as follows.



$L$  is the open-loop transfer function from `in` to `out`.

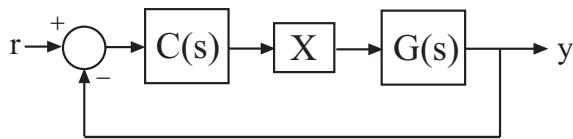
### sign — Feedback sign

+1 (default) | -1

Feedback sign, specified as +1 or -1. The feedback sign determines the sign of the open-loop transfer function.

- +1 — Compute the positive-feedback loop transfer. In this case, the relationship between the closed-loop response  $T$  and the open-loop response  $L$  is  $T = \text{feedback}(L, 1, +1)$ .
- -1 — Compute the negative-feedback loop transfer. In this case, the relationship between the closed-loop response  $T$  and the open-loop response  $L$  is  $T = \text{feedback}(L, 1)$ .

Choose a feedback sign that is consistent with the conventions of the analysis you intend to perform with the loop transfer function. For example, consider the following system, where  $T$  is the closed-loop transfer function from  $r$  to  $y$ .



To compute the stability margins of this system with the `margin` command, which assumes negative feedback, you need to use the negative-feedback open-loop response. Therefore, you can use `L = getLoopTransfer(T, 'X', -1)` to obtain the negative-feedback transfer function  $L = GC$ .

### openings — Additional locations for opening feedback loops

character vector | cell array of character vectors

Additional locations for opening feedback loops for computation of the open-loop response, specified as character vector or cell array of character vectors that

identify analysis-point locations in `T`. Analysis-point locations are marked by `AnalysisPoint` blocks in `T`. Any channel name contained in the `Location` property of an `AnalysisPoint` block in `T` is a valid entry for `openings`.

Use `openings` when you want to compute the open-loop response at one analysis-point location with other loops also open at other locations. For example, in a cascaded loop configuration, you can calculate the inner loop open-loop response with the outer loop also open. Use `getPoints(T)` to get a full list of available analysis-point locations in `T`.

## Output Arguments

### **L** — Point-to-point open-loop response

generalized state-space model

Point-to-point open-loop response of the control system `T` measured at the analysis points specified by `Locations`, returned as a generalized state-space (`genss`) model.

- If `Locations` specifies a single analysis point, then `L` is a SISO `genss` model. In this case, `L` represents the response obtained by opening the loop at `Locations`, injecting signals and measuring the return signals at the same location.
- If `Locations` is a vector signal, or specifies multiple analysis points, then `L` is a MIMO `genss` model. In this case, `L` represents the open-loop MIMO response obtained by opening loops at all locations listed in `Locations`, injecting signals and measuring the return signals at those locations.

## More About

### Tips

- You can use `getLoopTransfer` to extract open-loop responses given a generalized model of the overall control system. This is useful, for example, for validating open-loop responses of a control system that you tune with the a tuning command such as  `systune`.
- `getLoopTransfer` is the `genss` equivalent to the Simulink Control Design command `getLoopTransfer`, which works with the `sITuner` and `sLLinearizer` interfaces. Use the Simulink Control Design command when your control system is modeled in Simulink.

## **See Also**

AnalysisPoint | genss | getIOTransfer | getLoopTransfer | getPoints | systune

**Introduced in R2012b**

## getoptions

Return @PlotOptions handle or plot options property

### Syntax

```
p = getoptions(h)
p = getoptions(h,propertyname)
```

### Description

`p = getoptions(h)` returns the plot options handle associated with plot handle `h`. `p` contains all the settable options for a given response plot.

`p = getoptions(h,propertyname)` returns the specified options property, `propertyname`, for the plot with handle `h`. You can use this to interrogate a plot handle. For example,

```
p = getoptions(h,'Grid')
```

returns 'on' if a grid is visible, and 'off' when it is not.

For a list of the properties and values available for each plot type, see “Properties and Values Reference”.

### See Also

setoptions

**Introduced before R2006a**



# getPassiveIndex

Compute passivity index of linear system

`getPassiveIndex` computes various measures of the excess or shortage of passivity for a given system.

A linear system  $G(s)$  is *passive* if all its I/O trajectories  $(u(t),y(t))$  satisfy:

$$\int_0^T y(t)^\top u(t) dt > 0,$$

for all  $T > 0$ . Equivalently, a system is passive if its frequency response is positive real, such that for all  $\omega > 0$ ,

$$G(j\omega) + G(j\omega)^H > 0$$

(or the discrete-time equivalent).

## Syntax

```
R = getPassiveIndex(G)
nu = getPassiveIndex(G,'input')
rho = getPassiveIndex(G,'output')
tau = getPassiveIndex(G,'io')
DX = getPassiveIndex(G,dQ)

index = getPassiveIndex( ____,tol)
index = getPassiveIndex( ____,tol,fband)
[index,FI] = getPassiveIndex( ____)
[index,FI,Qout,dQout] = getPassiveIndex( ____)
```

## Description

`R = getPassiveIndex(G)` computes the relative passivity index.  $G$  is passive when  $R$  is less than one.  $R$  measures the relative excess ( $R < 1$ ) or shortage ( $R > 1$ ) of passivity.

For more information about the notion of passivity indices, see “About Passivity and Passivity Indices”.

`nu = getPassiveIndex(G, 'input')` computes the input passivity index. The system is *input strictly passive* when `nu > 0`. `nu` is also called the input feedforward passivity (IFP) index. The value of `nu` is the minimum feedforward action such that the resulting system is passive.

For more information about the notion of passivity indices, see “About Passivity and Passivity Indices”.

`rho = getPassiveIndex(G, 'output')` computes the output passivity index. The system is *output strictly passive* when `rho > 0`. `rho` is also called the output feedback passivity (OFP) index. The value of `rho` is the minimum feedback action such that the resulting system is passive.

For more information about the notion of passivity indices, see “About Passivity and Passivity Indices”.

`tau = getPassiveIndex(G, 'io')` computes the combined I/O passivity index. The system is *very strictly passive* when `tau > 0`.

For more information about the notion of passivity indices, see “About Passivity and Passivity Indices”.

`DX = getPassiveIndex(G, dQ)` computes the directional passivity index in the direction specified by the matrix `dQ`.

`index = getPassiveIndex( ____, tol)` computes the passivity index with relative accuracy specified by `tol`. Use this syntax with any of the previous combinations of input arguments. `index` is the corresponding passivity index `R`, `nu`, `rho`, `tau`, or `DX`.

`index = getPassiveIndex( ____, tol, fband)` computes passivity indices restricted to a specific frequency interval.

`[index, FI] = getPassiveIndex( ____)` also returns the frequency at which the returned index value is achieved.

`[index, FI, Qout, dQout] = getPassiveIndex( ____)` also returns the sector matrix `Qout` for passivity and the directional index matrix `dQout`.

## Examples

### Relative, Input, and Output Passivity Indices

Compute passivity indices for the following dynamic system:

$$G(s) = \frac{s^2 + s + 5s + 0.1}{s^3 + 2s^2 + 3s + 4}$$

```
G = tf([1,1,5,.1],[1,2,3,4]);
```

Compute the relative passivity index.

```
R = getPassiveIndex(G)
```

```
R =
```

```
0.9512
```

The system is passive, but with a relatively small excess of passivity.

Compute the input and output passivity indices.

```
nu = getPassiveIndex(G, 'input')
rho = getPassiveIndex(G, 'output')
```

```
nu =
```

```
0.0250
```

```
rho =
```

```
0.2591
```

This system is both input strictly passive and output strictly passive.

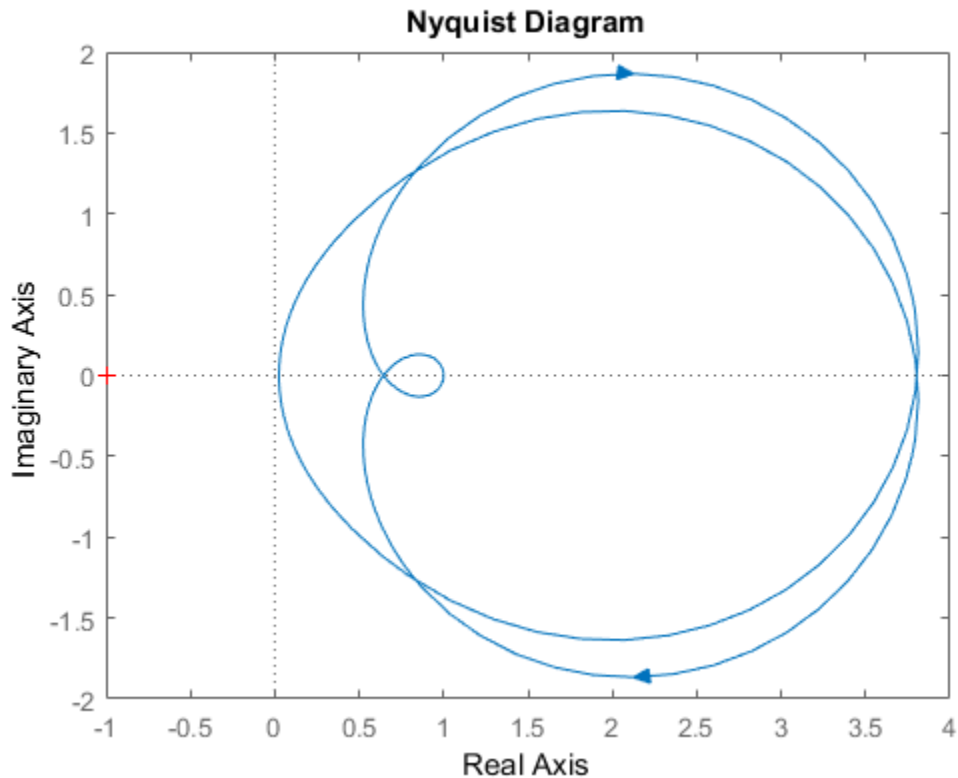
Compute the combined I/O passivity index.

```
tau = getPassiveIndex(G, 'io')
```

```
tau =  
    0.0250
```

The system is very strictly passive as well. A system that is very strictly passive is also strictly positive real. Examining the Nyquist plot confirms this, showing that the frequency response lies entirely within the right half-plane.

```
nyquistplot(G)
```



The relatively small `tau` value is reflected in how close the frequency response comes to the imaginary axis.

- “Passivity Indices”

## Input Arguments

### **G** — Model to analyze

dynamic system model | model array

Model to analyze for passivity, specified as a dynamic system model such as a `tf`, `ss`, or `genss` model. `G` can be MIMO, if the number of inputs equals the number of outputs. `G` can be continuous or discrete. If `G` is a generalized model with tunable or uncertain blocks, `getPassiveIndex` evaluates passivity of the current, nominal value of `G`.

If `G` is a model array, then `getPassiveIndex` returns the passivity index as an array of the same size, where:

```
index(k) = getPassivityIndex(G(:, :, k), ___)
```

Here, `index` is any of `R`, `nu`, `rho`, `tau`, or `DX`, depending on which input arguments you use.

### **dQ** — Custom direction

matrix

Custom direction in which to compute passivity, specified as a symmetric square matrix that is  $2 \times n_y$  on a side, where  $n_y$  is the number of outputs of `G`.

The `rho`, `nu`, and `tau` indices each correspond to a particular direction in the  $y/u$  space of the system, with a corresponding `dQ` value. (See `dQout` for these values.) Use this argument to specify your own value for this direction.

### **tol** — Relative accuracy

0.01 (default) | positive real value

Relative accuracy for the calculated passivity index. By default, the tolerance is 1%, meaning that the returned passivity index is within 1% of the actual passivity index.

### **fband** — Frequency interval

1-by-2 array

Frequency interval for determining passivity index, specified as an array of the form `[fmin, fmax]`. When you provide `fband`, then `getPassiveIndex` restricts the frequency-domain computation of the passivity index to that frequency range. For example, the relative passivity index `R` is the peak gain of the bilinear-transformed system  $(I - G)(I + G)^{-1}$  (for minimum-phase  $(I + G)$ ). When you provide `fband`, then `R` is the peak gain within the frequency band.

Specify frequencies in units of `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the dynamic system model `G`.

## Output Arguments

### **R** — Relative passivity index

scalar | array

Relative passivity index, returned as a scalar, or an array if `G` is an array.

The system `G` is passive when `R` is less than one.

- $R < 1$  indicates a relative excess of passivity.
- $R > 1$  indicates a relative shortage of passivity.

When  $I + G$  is minimum phase, `R` is the peak gain of the bilinear-transformed system  $(I - G)(I + G)^{-1}$ .

For more information about the notion of passivity indices, see “About Passivity and Passivity Indices”.

### **nu** — Input passivity index

scalar | array

Input passivity index, returned as a scalar, or an array if `G` is an array. `nu` is defined as the largest value of  $\nu$  for which:

$$\int_0^T y(t)^T u(t) dt > \nu \int_0^T u(t)^T u(t) dt,$$

for all  $T > 0$ . Equivalently, `nu` is the largest  $\nu$  for which:

$$G(j\omega) + G(j\omega)^H > 2\nu I$$

(or the discrete-time equivalent). The system is *input strictly passive* when  $\text{nu} > 0$ .  $\text{nu}$  is also called the input feedforward passivity (IFP) index. The value of  $\text{nu}$  is the minimum feedforward action such that the resulting system is passive.

### **rho — Output passivity index**

scalar | array

Output passivity index, returned as a scalar, or an array if  $\mathbf{G}$  is an array.  $\text{rho}$  is defined as the largest value of  $\rho$  for which:

$$\int_0^T y(t)^\top u(t) dt > \rho \int_0^T y(t)^\top y(t) dt,$$

for all  $T > 0$ . The system is *output strictly passive* when  $\text{rho} > 0$ .  $\text{rho}$  is also called the output feedback passivity (OFP) index. The value of  $\text{rho}$  is the minimum feedback action such that the resulting system is passive.

### **tau — Combined I/O passivity index**

scalar | array

Combined I/O passivity index, returned as a scalar, or an array if  $\mathbf{G}$  is an array.  $\text{tau}$  is defined as the largest value of  $\tau$  for which:

$$\int_0^T y(t)^\top u(t) dt > \tau \int_0^T \left( u(t)^\top u(t) + y(t)^\top y(t) \right) dt,$$

for all  $T > 0$ . The system is *very strictly passive* when  $\text{tau} > 0$ .

### **DX — Directional passivity index**

scalar | array

Directional passivity index in the direction specified by  $d\mathbf{Q}$ , returned as a scalar, or an array if  $\mathbf{G}$  is an array. The directional passivity index is the largest value of  $D$  for which:

$$\int_0^T y(t)^\top u(t) dt > D \int_0^T \left( \begin{array}{c} y(t) \\ u(t) \end{array} \right)^\top d\mathbf{Q} \left( \begin{array}{c} y(t) \\ u(t) \end{array} \right) dt,$$

for all  $T > 0$ . The  $\text{rho}$ ,  $\text{nu}$ , and  $\text{tau}$  indices correspond to particular choices of  $d\mathbf{Q}$  (see the output argument `dQout`). To compute  $\text{DX}$ , the software uses the custom  $d\mathbf{Q}$  value you supply, `dQ`.

**FI — Frequency at which index is achieved**

nonnegative scalar | array

Frequency at which the returned passivity index is achieved, returned as a nonnegative scalar, or an array if **G** is an array. In general, the passivity indices vary with frequency (see `passiveplot`). For each index type, the returned value is the largest value over all frequencies. **FI** is the frequency at which this value occurs, returned in units of `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of **G**.

**Qout — Sector geometry**

matrix

Sector geometry used for computing the passivity index, returned as a matrix. For passivity indices, **Qout** is given by:

`Qout = [zeros(ny), -1/2*eye(ny); -1/2*eye(ny), zeros(ny)];`

where `ny` is the number of outputs of **G**. For example, for a SISO **G**,

`Qout = [ 0, -0.5;  
-0.5, 0 ];`

For more information about sector geometry, see `getSectorIndex`.

**dQout — Direction**

matrix

Direction in which passivity is computed, returned as a square matrix that is `2*ny` on a side, where `ny` is the number of outputs of **G**. The value returned for **dQout** depends on what kind of passivity index you calculate:

- `nu` — For the input passivity index, **dQout** is given by:

`dQout = [zeros(ny), zeros(ny); zeros(ny), eye(ny)];`

For instance, for a SISO system, `dQout = [0,0;0,1]`.

- `rho` — For the output passivity index, **dQout** is given by:

`dQout = [eye(ny), zeros(ny); zeros(ny), zeros(ny)];`

For instance, for a SISO system, `dQout = [1,0;0,0]`.

- `tau` — For the combined I/O passivity index, **dQout** is given by:



```
dQout = eye(2*ny);
```

For instance, for a SISO system, `dQout = [1,0;0,1]`.

- **DX** — `dQout` is the custom value you provide in the `dQ` input argument.
- **R** — The relative passivity index does not involve a direction, so in this case the function returns `dQout = []`.

For more information about directional indices, see `getSectorIndex`.

## More About

- “About Passivity and Passivity Indices”

## See Also

`getSectorCrossover` | `getSectorIndex` | `isPassive` | `nyquist` | `passiveplot` | `sectorplot`

**Introduced in R2016a**

## getPeakGain

Peak gain of dynamic system frequency response

### Syntax

```
gpeak = getPeakGain(sys)
gpeak = getPeakGain(sys,tol)
gpeak = getPeakGain(sys,tol,fband)
[gpeak,fpeak] = getPeakGain( ___ )
```

### Description

`gpeak = getPeakGain(sys)` returns the peak input/output gain in absolute units of the dynamic system model, `sys`.

- If `sys` is a SISO model, then the peak gain is the largest value of the frequency response magnitude.
- If `sys` is a MIMO model, then the peak gain is the largest value of the frequency response 2-norm (the largest singular value across frequency) of `sys`. This quantity is also called the  $L_\infty$  norm of `sys`, and coincides with the  $H_\infty$  norm for stable systems.
- If `sys` is a model that has tunable or uncertain parameters, `getPeakGain` evaluates the peak gain at the current or nominal value of `sys`.
- If `sys` is a model array, `getPeakGain` returns an array of the same size as `sys`, where `gpeak(k) = getPeakGain(sys(:, :, k))`.

`gpeak = getPeakGain(sys,tol)` returns the peak gain of `sys` with relative accuracy `tol`.

`gpeak = getPeakGain(sys,tol,fband)` returns the peak gain in the frequency interval `fband`.

`[gpeak,fpeak] = getPeakGain( ___ )` also returns the frequency `fpeak` at which the gain achieves the peak value `gpeak`, and can include any of the input arguments in previous syntaxes.

## Examples

### Peak Gain of Transfer Function

Compute the peak gain of the resonance in the following transfer function:

$$sys = \frac{90}{s^2 + 1.5s + 90}$$

```
sys = tf(90,[1,1.5,90]);
gpeak = getPeakGain(sys)
```

```
gpeak =
    6.3246
```

The `getPeakGain` command returns the peak gain in absolute units.

### Peak Gain with Specified Accuracy

Compute the peak gain of the resonance in the transfer function with a relative accuracy of 0.01%.

$$sys = \frac{90}{s^2 + 1.5s + 90}$$

```
sys = tf(90,[1,1.5,90]);
gpeak = getPeakGain(sys,0.0001)
```

```
gpeak =
    6.3444
```

The second argument specifies a relative accuracy of 0.0001. The `getPeakGain` command returns a value that is within 0.0001 (0.01%) of the true peak gain of the transfer function. By default, the relative accuracy is 0.01 (1%).

### Peak Gain Within Specified Band

Compute the peak gain of the higher-frequency resonance in the transfer function

$$sys = \left( \frac{1}{s^2 + 0.2s + 1} \right) \left( \frac{100}{s^2 + s + 100} \right).$$

`sys` is the product of resonances at 1 rad/s and 10 rad/s.

```
sys = tf(1,[1,.2,1])*tf(100,[1,1,100]);  
fband = [8,12];  
gpeak = getPeakGain(sys,0.01,fband);
```

The `fband` argument causes `getPeakGain` to return the local peak gain between 8 and 12 rad/s.

### Frequency of Peak Gain

Identify which of the two resonances has higher gain in the transfer function

$$sys = \left( \frac{1}{s^2 + 0.2s + 1} \right) \left( \frac{100}{s^2 + s + 100} \right).$$

`sys` is the product of resonances at 1 rad/s and 10 rad/s.

```
sys = tf(1,[1,.2,1])*tf(100,[1,1,100]);  
[gpeak,fpeak] = getPeakGain(sys)
```

```
gpeak =  
  
5.0502
```

```
fpeak =  
  
1.0000
```

`fpeak` is the frequency corresponding to the peak gain `gpeak`. The peak at 1 rad/s is the overall peak gain of `sys`.

## Input Arguments

### `sys` — Input dynamic system

dynamic system model | model array

Input dynamic system, specified as any dynamic system model or model array. `sys` can be SISO or MIMO.

### **tol** — Relative accuracy

0.01 (default) | positive real scalar

Relative accuracy of the peak gain, specified as a positive real scalar value.

`getPeakGain` calculates `gpeak` such that the fractional difference between `gpeak` and the true peak gain of `sys` is no greater than `tol`. The default value is 0.01, meaning that `gpeak` is within 1% of the true peak gain.

### **fband** — Frequency interval

[0, Inf] (default) | 1-by-2 vector of positive real values

Frequency interval in which to calculate the peak gain, specified as a 1-by-2 vector of positive real values. Specify `fband` as a row vector of the form [ `fmin`, `fmax` ].

## Output Arguments

### **gpeak** — Peak gain of dynamic system

scalar | array

Peak gain of the dynamic system model or model array `sys`, returned as a scalar value or an array.

- If `sys` is a single model, then `gpeak` is a scalar value.
- If `sys` is a model array, then `gpeak` is an array of the same size as `sys`, where `gpeak(k) = getPeakGain(sys(:, :, k))`.

### **fpeak** — Frequency of peak gain

nonnegative real scalar | array of nonnegative real values

Frequency at which the gain achieves the peak value `gpeak`, returned as a nonnegative real scalar value or an array of nonnegative real values. The frequency is expressed in units of `rad/TimeUnit`, relative to the `TimeUnit` property of `sys`.

- If `sys` is a single model, then `fpeak` is a scalar.
- If `sys` is a model array, then `fpeak` is an array of the same size as `sys`, where `fpeak(k)` is the peak gain frequency of the  $k$ th model in the array.

# More About

## Algorithms

`getPeakGain` uses the algorithm of [1]. All eigenvalue computations are performed using structure-preserving algorithms from the SLICOT library. For more information about the SLICOT library, see <http://slicot.org>.

- “Dynamic System Models”

## References

[1] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the  $H_\infty$ -Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

## See Also

`bode` | `freqresp` | `getGainCrossover` | `sigma`

**Introduced in R2012a**

# getPoints

Get list of analysis points in generalized model of control system

## Syntax

```
points = getPoints(T)
```

## Description

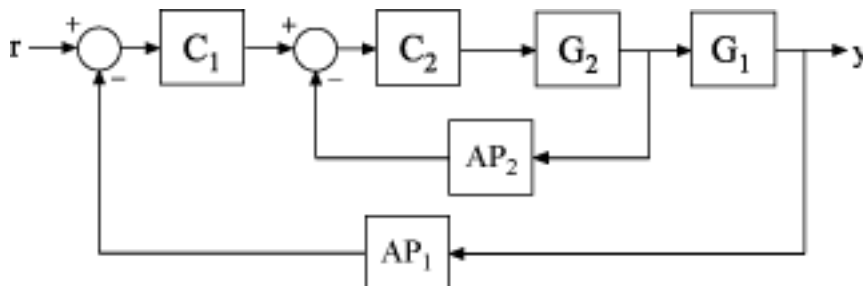
`points = getPoints(T)` returns the names of all analysis-point locations in a generalized state-space model of a control system. Use this function to query the list of available analysis points in the model for control system analysis or tuning. You can refer to the analysis-point locations by name to create design goals control system tuning or to compute open-loop and closed-loop responses using analysis commands such as `getLoopTransfer` and `getIOTransfer`.

## Examples

### Analysis-Point Locations in Control System Model

Build a closed-loop model of a cascaded feedback loop system, and get a list of analysis point locations in the model.

Create a model of the following cascaded feedback loop.  $C_1$  and  $C_2$  are tunable controllers.  $AP_1$  and  $AP_2$  are points of interest for analysis, which you mark with `AnalysisPoint` blocks.



```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
AP1 = AnalysisPoint('AP1');
AP2 = AnalysisPoint('AP2');
T = feedback(G1*feedback(G2*C2,AP2)*C1,AP1);
```

T is a `genss` model whose Control Design Blocks include the tunable controllers and the switches AP1 and AP2.

Get a list of the loop-opening sites in T.

```
points = getPoints(T)
```

```
points =
    2×1 cell array
    'AP1'
    'AP2'
```

`getPoints` returns a cell array listing loop-opening sites in the model.

For more complicated closed-loop models, you can use `getPoints` to keep track of a larger number of analysis points.

## Input Arguments

### **T** — Model of control system

generalized state-space model

Model of a control system, specified as a generalized state-space (`genss`) model.

Locations in the model at which you can calculate system responses or specify design goals for tuning are marked by `AnalysisPoint` blocks in T.

## Output Arguments

### **points** — Analysis-point locations

cell array of character vectors



Analysis-point locations in the control system model, returned as a cell array of character vectors. This output is obtained by concatenating the `LOCATION` properties of all `AnalysisPoint` blocks in the control system model.

## More About

- “Generalized Models”

## See Also

`AnalysisPoint` | `genss` | `getIOTransfer` | `getLoopTransfer`

**Introduced in R2014b**

## getSectorCrossover

Crossover frequencies for sector bound

### Syntax

`wc = getSectorCrossover(H,Q)`

### Description

`wc = getSectorCrossover(H,Q)` returns the frequencies at which the following matrix  $M(\omega)$  is singular:

$$M(\omega) = H(j\omega)^H Q H(j\omega).$$

When a frequency-domain sector plot exists, these frequencies are the frequencies at which the relative sector index (R-index) for H and Q equals 1. See “About Sector Bounds and Sector Indices” for details.

### Examples

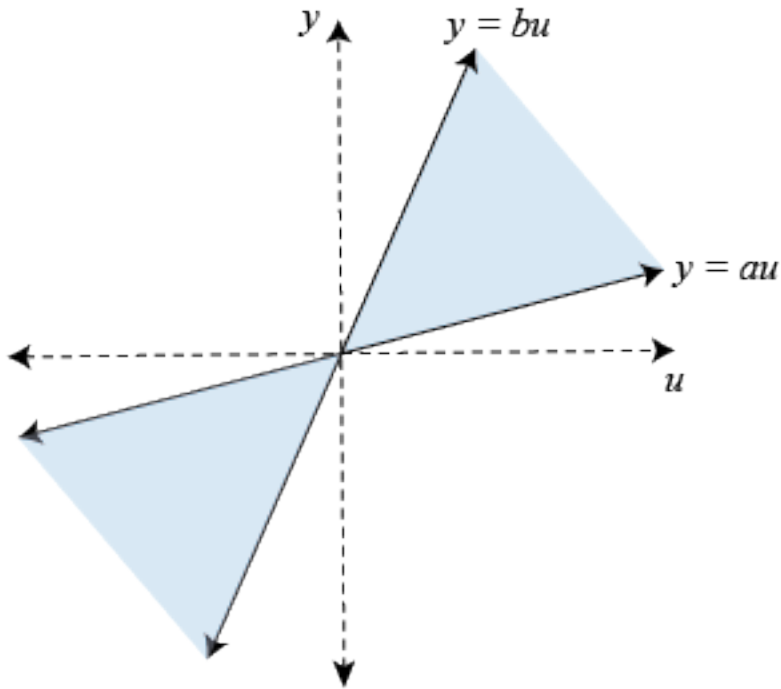
#### Find Sector Crossover Frequency

Find the crossover frequencies for the dynamic system  $G(s) = (s + 2) / (s + 1)$  and the sector defined by:

$$S = \{(y, u) : au^2 < uy < bu^2\},$$

for various values of  $a$  and  $b$ .

In U/Y space, this sector is the shaded region of the following diagram (for  $a, b > 0$ ).



The  $Q$  matrix for this sector is given by:

$$Q = \begin{pmatrix} 1 & -(a+b)/2 \\ -(a+b)/2 & ab \end{pmatrix}.$$

`getSectorCrossover` finds the frequencies at which  $H(s)^H Q H(s)$  is singular, for  $H(s) = [G(s); I]$ . For instance, find these frequencies for the sector defined by  $Q$  with  $a = 0.1$  and  $b = 10$ .

```
G = tf([1 2],[1 1]);
H = [G;1];
```

```
a = 0.1;
b = 10;
```

```
Q = [1 -(a+b)/2 ; -(a+b)/2 a*b];
```

```
w = getSectorCrossover(H,Q)
```

```
w =
```

```
0×1 empty double column vector
```

The empty result means that there are no such frequencies.

Now find the frequencies at which  $H^H Q H$  is singular for a narrower sector, with  $a = 0.5$  and  $b = 1.5$ .

```
a2 = 0.5;
```

```
b2 = 1.5;
```

```
Q2 = [1 -(a2+b2)/2 ; -(a2+b2)/2 a2*b2];
```

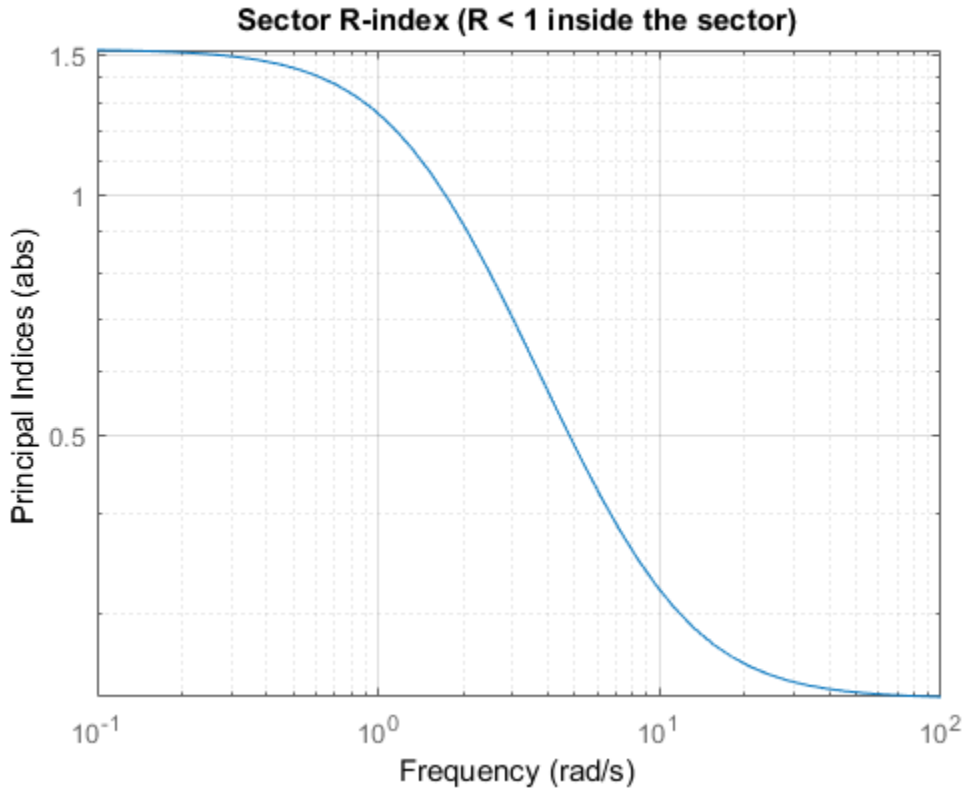
```
w2 = getSectorCrossover(H,Q2)
```

```
w2 =
```

```
1.7321
```

Here the resulting frequency is where the R-index for H and Q2 is equal to 1, as shown in the sector plot.

```
sectorplot(H,Q2)
```



Thus, when a sector plot exists for a system  $H$  and sector  $Q$ , `getSectorCrossover` finds the frequencies at which the R-index is 1.

## Input Arguments

### **H** — Model to analyze

dynamic system model

Model to analyze against sector bounds, specified as a dynamic system model such as a `tf`, `ss`, or `genss` model.  $H$  can be continuous or discrete. If  $H$  is a generalized model with tunable or uncertain blocks, `getSectorCrossover` analyzes the current, nominal value of  $H$ .

To get the frequencies at which the I/O trajectories  $(u,y)$  of a linear system  $G$  lie in a particular sector, use  $H = [G; I]$ , where  $I = \text{eyes}(nu)$ , and  $nu$  is the number of inputs of  $G$ .

### **Q** — Sector geometry

matrix | LTI model

Sector geometry, specified as:

- A matrix, for constant sector geometry.  $Q$  is a symmetric square matrix that is  $ny$  on a side, where  $ny$  is the number of outputs of  $H$ .
- An LTI model, for frequency-dependent sector geometry.  $Q$  satisfies  $Q(s)' = Q(-s)$ . In other words,  $Q(s)$  evaluates to a Hermitian matrix at each frequency.

The matrix  $Q$  must be indefinite to describe a well-defined conic sector. An indefinite matrix has both positive and negative eigenvalues.

For more information, see “About Sector Bounds and Sector Indices”.

## Output Arguments

### **wc** — Sector crossover frequencies

vector | []

Sector crossover frequencies, returned as a vector. The frequencies are expressed in  $\text{rad}/\text{TimeUnit}$ , relative to the `TimeUnit` property of  $H$ . If the trajectories of  $H$  never cross the boundary,  $wc = []$ .

## More About

- “About Sector Bounds and Sector Indices”

### See Also

`getGainCrossover` | `getSectorIndex` | `sectorplot`

Introduced in R2016a

# getSectorIndex

Compute conic-sector index of linear system

## Syntax

```

RX = getSectorIndex(H,Q)
RX = getSectorIndex(H,Q,tol)
RX = getSectorIndex(H,Q,tol,fband)
[RX,FX] = getSectorIndex(____)
[RX,FX,W1,W2,Z] = getSectorIndex(____)

DX = getSectorIndex(H,Q,dQ)
DX = getSectorIndex(H,Q,dQ,tol)

```

## Description

`getSectorIndex(H,Q)` computes the relative index `RX` for the linear system `H` and the conic sector specified by `Q`. When `RX < 1`, all output trajectories  $y(t) = Hu(t)$  lie in the sector defined by:

$$\int_0^T y(t)^T Q y(t) dt < 0,$$

for all  $T \geq 0$ .

`getSectorIndex` can also check whether all I/O trajectories  $\{u(t),y(t)\}$  of a linear system `G` lie in the sector defined by:

$$\int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt < 0,$$

for all  $T \geq 0$ . To do so, use `getSectorIndex` with `H = [G;I]`, where `I = eyes(nu)`, and `nu` is the number of inputs of `G`.

For more information about sector bounds and the relative index, see “About Sector Bounds and Sector Indices”.

$RX = \text{getSectorIndex}(H, Q, \text{tol})$  computes the index with relative accuracy specified by  $\text{tol}$ .

$RX = \text{getSectorIndex}(H, Q, \text{tol}, \text{fband})$  computes the passivity index by restricting the inequalities that define the index to a specified frequency interval. This syntax is available only when  $Q$  has as many negative eigenvalues as there are inputs in  $H$ .

$[RX, FX] = \text{getSectorIndex}(\text{___})$  also returns the frequency at which the index value  $RX$  is achieved.  $FX$  is set to  $\text{NaN}$  when the number of negative eigenvalues in  $Q$  differs from the number of inputs in  $H$ . You can use this syntax with any of the previous combinations of input arguments.

$[RX, FX, W1, W2, Z] = \text{getSectorIndex}(\text{___})$  also returns the decomposition of  $Q$  into its positive and negative parts, as well as the spectral factor  $Z$  when  $Q$  is dynamic. When  $Q$  is a matrix (constant sector bounds),  $Z = 1$ . You can use this syntax with any of the previous combinations of input arguments.

$DX = \text{getSectorIndex}(H, Q, \text{dQ})$  computes the index in the direction specified by the matrix  $\text{dQ}$ . If  $DX > 0$ , then the output trajectories of  $H$  fit in the conic sector specified by  $Q$ . For more information about the directional index, see “About Sector Bounds and Sector Indices”.

The directional index is not available if  $H$  is a frequency-response data ( $\text{frd}$ ) model.

$DX = \text{getSectorIndex}(H, Q, \text{dQ}, \text{tol})$  computes the index with relative accuracy specified by  $\text{tol}$ .

## Examples

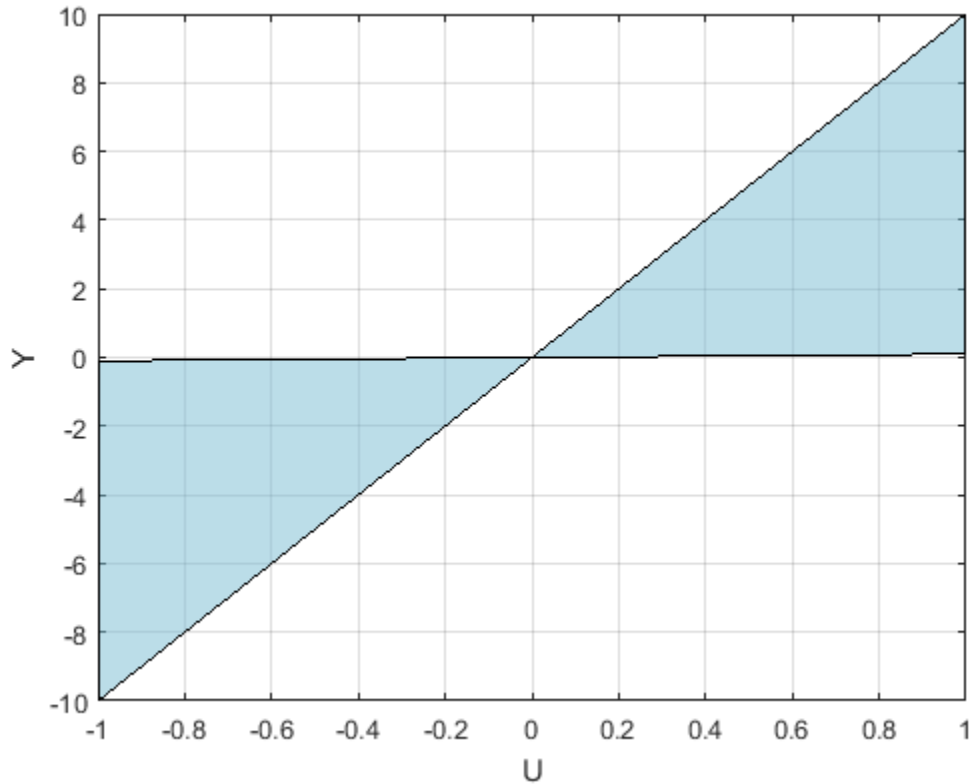
### Check Sector Bounds

Test whether, on average, the I/O trajectories of  $G(s) = (s + 2) / (s + 1)$  belong within the sector defined by:

$$S = \{(y, u) : 0.1u^2 < uy < 10u^2\}.$$

In  $U/Y$  space, this sector is the shaded region of the following diagram.





The  $Q$  matrix corresponding to this sector is given by:

$$Q = \begin{bmatrix} 1 & -(a+b)/2 \\ -(a+b)/2 & ab \end{bmatrix}; \quad a = 0.1, \quad b = 10.$$

A trajectory  $y(t) = Gu(t)$  lies within the sector  $S$  when for all  $T > 0$ ,

$$0.1 \int_0^T u(t)^2 dt < \int_0^T u(t)y(t) dt < 10 \int_0^T u(t)^2 dt.$$

To check whether trajectories of  $G$  satisfy the sector bound, represented by  $Q$ , compute the  $R$ -index for  $H = [G; 1]$ .

```
G = tf([1 2],[1 1]);  
  
a = 0.1; b = 10;  
Q = [1 -(a+b)/2 ; -(a+b)/2 a*b];  
  
R = getSectorIndex([G;1],Q)  
  
R =  
  
    0.4074
```

This resulting  $R$  is less than 1, indicating that the trajectories fit within the sector. The value of  $R$  tells you how much tightly the trajectories fit in the sector. This value,  $R = 0.41$ , means that the trajectories would fit in a narrower sector with a base  $1/0.41 = 2.4$  times smaller.

## Input Arguments

### **H** — Model to analyze

dynamic system model | model array

Model to analyze against sector bounds, specified as a dynamic system model such as a `tf`, `ss`, or `genss` model.  $H$  can be continuous or discrete. If  $H$  is a generalized model with tunable or uncertain blocks, `getSectorIndex` analyzes the current, nominal value of  $H$ .

To analyze whether all I/O trajectories  $\{u(t),y(t)\}$  of a linear system  $G$  lie in a particular sector, use  $H = [G;I]$ .

If  $H$  is a model array, then `getSectorIndex` returns the passivity index as an array of the same size, where:

```
index(k) = getSectorIndex(H(:, :, k), ___)
```

Here, `index` is either `RX`, or `DX`, depending on which input arguments you use.

### **Q** — Sector geometry

matrix | LTI model

Sector geometry, specified as:

- A matrix, for constant sector geometry.  $Q$  is a symmetric square matrix that is  $n_y$  on a side, where  $n_y$  is the number of outputs of  $H$ .
- An LTI model, for frequency-dependent sector geometry.  $Q$  satisfies  $Q(s)' = Q(-s)$ . In other words,  $Q(s)$  evaluates to a Hermitian matrix at each frequency.

The matrix  $Q$  must be indefinite to describe a well-defined conic sector. An indefinite matrix has both positive and negative eigenvalues.

For more information, see “About Sector Bounds and Sector Indices”.

#### **tol** — Relative accuracy

0.01 (default) | positive real value

Relative accuracy for the calculated sector index. By default, the tolerance is 1%, meaning that the returned index is within 1% of the actual index.

#### **fband** — Frequency interval

1-by-2 array

Frequency interval for calculating the sector index, specified as an array of the form  $[f_{min}, f_{max}]$ . When you provide **fband**, `getSectorIndex` restricts to the specified frequency interval the inequalities that define the index. Specify frequencies in units of  $\text{rad}/\text{TimeUnit}$ , where `TimeUnit` is the `TimeUnit` property of the dynamic system model  $H$ .

#### **dQ** — Direction

matrix

Direction in which to compute directional sector index, specified as a nonnegative definite matrix. The matrix  $dQ$  is a symmetric square matrix that is  $n_y$  on a side, where  $n_y$  is the number of outputs of  $H$ .

## Output Arguments

#### **RX** — Relative sector index

scalar | array

Relative index of the system  $H$  for the sector specified by  $Q$ , returned as a scalar value, or an array if  $H$  is an array. If  $RX < 1$ , then the output trajectories of  $H$  fit inside the cone of  $Q$ .

The value of  $RX$  provides a measure of how tightly the output trajectories of  $H$  fit inside the cone. Let the following be an orthogonal decomposition of the symmetric matrix  $Q$  into its positive and negative parts.

$$Q = W_1 W_1^T - W_2 W_2^T, \quad W_1^T W_2 = 0.$$

(Such a decomposition is readily obtained from the Schur decomposition of  $Q$ .) Then,  $RX$  is the smallest  $R$  that satisfies:

$$\int_0^T y(t)^T (W_1 W_1^T - R^2 W_2 W_2^T) y(t) dt < 0,$$

for all  $T \geq 0$ . Varying  $R$  is equivalent to adjusting the slant angle of the cone specified by  $Q$  until the cone fits tightly around the output trajectories of  $H$ . The cone base-to-height ratio is proportional to  $R$ .

For more information about interpretations of the relative index, see “About Sector Bounds and Sector Indices”.

**FX — Frequency at which index is achieved**

nonnegative scalar | array

Frequency at which the index  $RX$  is achieved, returned as a nonnegative scalar, or an array if  $H$  is an array. In general, the index varies with frequency (see `sectorplot`). The returned value is the largest value over all frequencies.  $FX$  is the frequency at which this value occurs, returned in units of  $\text{rad}/\text{TimeUnit}$ , where  $\text{TimeUnit}$  is the  $\text{TimeUnit}$  property of  $H$ .

**W1, W2 — Positive and negative factors of Q**

matrices

Positive and negative factors of  $Q$ , returned as matrices. For a constant  $Q$ ,  $W1$  and  $W2$  satisfy:

$$Q = W_1 W_1^T - W_2 W_2^T, \quad W_1^T W_2 = 0.$$

**Z — Bistable model**

state-space model | 1

Bistable model in the factorization of  $Q$ , returned as:

- If  $\mathbf{Q}$  is a constant matrix,  $Z = 1$ .
- If  $\mathbf{Q}$  is frequency-dependent, then  $Z$  is a state-space (SS) model such that:

$$Q(j\omega) = Z(j\omega)^H (W_1 W_1^T - W_2 W_2^T) Z(j\omega).$$

### **DX — Directional sector index**

scalar | array

Directional sector index of the system  $\mathbf{H}$  for the sector specified by  $\mathbf{Q}$  in the direction  $d\mathbf{Q}$ , returned as a scalar value, or an array if  $\mathbf{H}$  is an array. The directional index is the largest  $\tau$  which satisfies:

$$\int_0^T y(t)^T (Q + \tau dQ) y(t) dt < 0,$$

for all  $T \geq 0$ .

## **More About**

- “About Sector Bounds and Sector Indices”

## **See Also**

getPassiveIndex | getPeakGain | getSectorCrossover | nyquist | sectorplot

**Introduced in R2016a**

## getSensitivity

Sensitivity function from generalized model of control system

### Syntax

```
S = getSensitivity(T,location)
S = getSensitivity(T,location,opening)
```

### Description

`S = getSensitivity(T,location)` returns the sensitivity function at the specified location for a generalized model of a control system.

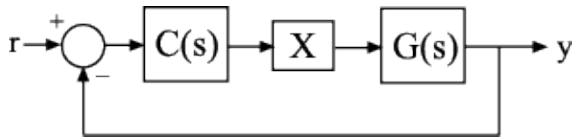
`S = getSensitivity(T,location,opening)` specifies additional loop openings for the sensitivity function calculation. Use an opening, for example, to calculate the sensitivity function of an inner loop, with the outer loop open.

If `opening` and `location` list the same point, the software opens the loop after measuring the signal at the point.

### Examples

#### Sensitivity Function at a Location

Compute the sensitivity at the plant input, marked by the analysis point X.



Create a model of the system by specifying and connecting a numeric LTI plant model  $G$ , a tunable controller  $C$ , and the `AnalysisPoint` block  $X$ . Use the `AnalysisPoint` block to mark the location where you assess the sensitivity (plant input in this example).

```
G = tf([1],[1 5]);
```

```

C = tunablePID('C','p');
C.Kp.Value = 3;
X = AnalysisPoint('X');
T = feedback(G*X*C,1);

```

T is a `genss` model that represents the closed-loop response of the control system from  $r$  to  $y$ . The model contains the `AnalysisPoint` block, X, that identifies the analysis point.

Calculate the sensitivity,  $S$ , at X.

```

S = getSensitivity(T, 'X');
tf(S)

```

ans =

```

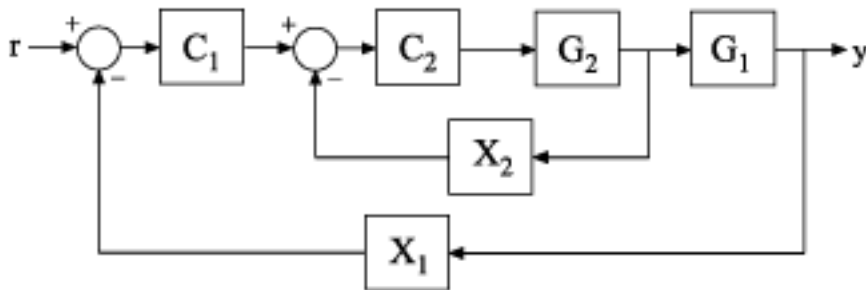
From input "X" to output "X":
s + 5
-----
s + 8

```

Continuous-time transfer function.

### Specify Additional Loop Opening for Sensitivity Function Calculation

Calculate the inner-loop sensitivity at the output of G2, with the outer loop open.



Create a model of the system by specifying and connecting the numeric plant models, tunable controllers, and `AnalysisPoint` blocks. G1 and G2 are plant models, C1 and C2

are tunable controllers, and X1 and X2 are `AnalysisPoint` blocks that mark potential loop-opening locations.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = tunablePID('C','pi');
C2 = tunableGain('G',1);
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
T = feedback(G1*feedback(G2*C2,X2)*C1,X1);
```

Calculate the sensitivity,  $S$ , at X2, with the outer loop open at X1.

```
S = getSensitivity(T,'X2','X1');
tf(S)
```

```
ans =
```

```
From input "X2" to output "X2":
  s^2 + 0.2 s + 10
  -----
  s^2 + 1.2 s + 12
```

Continuous-time transfer function.

## Input Arguments

### T — Model of control system

generalized state-space model

Model of a control system, specified as a generalized state-space model (`genss`).

Locations at which you can perform sensitivity analysis or open loops are marked by `AnalysisPoint` blocks in T. Use `getPoints(T)` to get the list of such locations.

### location — Location

character vector | cell array of character vectors

Location at which you calculate the sensitivity function, specified as a character vector or cell array of character vectors. To extract the sensitivity function at multiple locations, use a cell array of character vectors.



Each specified location must match an analysis point in `T`. Analysis points are marked using `AnalysisPoint` blocks. To get the list of available analysis points in `T`, use `getPoints(T)`.

Example: `'u'` or `{'u', 'y'}`

### **opening** — Additional loop opening

character vector | cell array of character vectors

Additional loop opening used to calculate the sensitivity function, specified as a character vector or cell array of character vectors. To open the loop at multiple locations, use a cell array of character vectors.

Each specified opening must match an analysis point in `T`. Analysis points are marked using `AnalysisPoint` blocks. To get the list of available analysis points in `T`, use `getPoints(T)`.

Use an opening, for example, to calculate the sensitivity function of an inner loop, with the outer loop open.

If `opening` and `location` list the same point, the software opens the loop after measuring the signal at the point.

Example: `'y_outer'` or `{'y_outer', 'y_outer2'}`

## **Output Arguments**

### **S** — Sensitivity function

generalized state-space model

Sensitivity function of the control system, `T`, measured at `location`, returned as a generalized state-space model (`genss`).

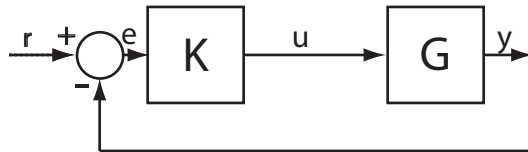
- If `location` specifies a single analysis point, then `S` is a SISO `genss` model.
- If `location` is a vector signal, or specifies multiple analysis points, then `S` is a MIMO `genss` model.

## More About

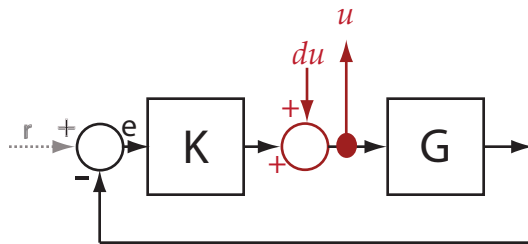
### Sensitivity Function

The *sensitivity function*, also referred to simply as *sensitivity*, measures how sensitive a signal is to an added disturbance. Feedback reduces the sensitivity in the frequency band where the open-loop gain is greater than 1.

Consider the following model:



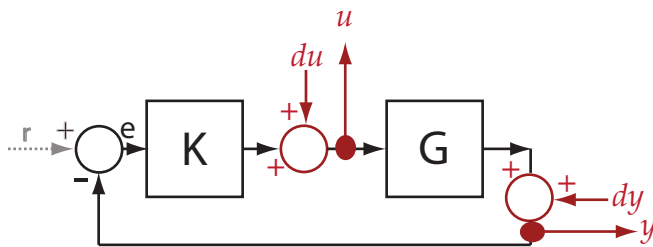
The sensitivity,  $S_u$ , at  $u$  is defined as the transfer function from  $du$  to  $u$ :



$$\begin{aligned}
 u &= du - KG u \\
 \rightarrow (I + KG)u &= du \\
 \rightarrow u &= \underbrace{(I + KG)^{-1}}_{S_u} du.
 \end{aligned}$$

Here,  $I$  is an identity matrix of the same size as  $KG$ .

Sensitivity at multiple locations, for example,  $u$  and  $y$ , is defined as the MIMO transfer function from the disturbances to sensitivity measurements:



$$S = \begin{bmatrix} S_{du \rightarrow u} & S_{dy \rightarrow u} \\ S_{du \rightarrow y} & S_{dy \rightarrow y} \end{bmatrix}.$$

## See Also

AnalysisPoint | genss | getCompSensitivity | getIOTransfer |  
getLoopTransfer | getPoints | getSensitivity | getValue | systune

Introduced in R2014a

## getValue

Current value of Generalized Model

### Syntax

```
curval = getValue(M)  
curval = getValue(M,blockvalues)  
curval = getValue(M,Mref)
```

### Description

`curval = getValue(M)` returns the current value `curval` of the Generalized LTI model or Generalized matrix `M`. The current value is obtained by replacing all Control Design Blocks in `M` by their current value. (For uncertain blocks, the “current value” is the nominal value of the block.)

`curval = getValue(M,blockvalues)` uses the block values specified in the structure `blockvalues` to compute the current value. The field names and values of `blockvalues` specify the block names and corresponding values. Blocks of `M` not specified in `blockvalues` are replaced by their current values.

`curval = getValue(M,Mref)` inherits block values from the generalized model `Mref`. This syntax is equivalent to `curval = getValue(M,Mref.Blocks)`. Use this syntax to evaluate the current value of `M` using block values computed elsewhere (for example, tuned values obtained with tuning commands such as `systemtune`, `looptune`, or the Robust Control Toolbox command `hinfstruct`).

### Input Arguments

**M**

Generalized LTI model or Generalized matrix.

**blockvalues**

Structure specifying blocks of `M` to replace and the values with which to replace those blocks.

The field names of `blockvalues` match names of Control Design Blocks of `M`. Use the field values to specify the replacement values for the corresponding blocks of `M`. The field values can be numeric values, dynamic system models, or static models. If some field values are Control Design Blocks or Generalized LTI models, the current values of those models are used to compute `curval`.

### Mref

Generalized LTI model. If you provide `Mref`, `getValue` computes `curval` using the current values of the blocks in `Mref` whose names match blocks in `M`.

## Output Arguments

### curval

Numeric array or Numeric LTI model representing the current value of `M`.

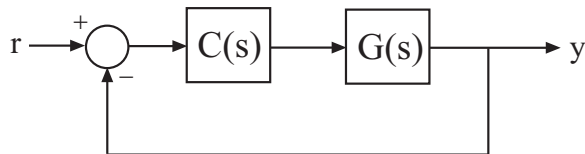
If you do not specify a replacement value for a given Control Design Block of `M`, `getValue` uses the current value of that block.

## Examples

### Evaluate Model for Specified Values of its Blocks

This example shows how to replace a Control Design Block in a Generalized LTI model with a specified replacement value using `getValue`.

Consider the following closed-loop system:



The following code creates a `genss` model of this system with  $G(s) = \frac{(s-1)}{(s+1)^3}$  and a tunable PI controller `C`.

```
G = zpk(1,[-1,-1,-1],1);  
C = tunablePID('C','pi');  
Try = feedback(G*C,1)
```

The `genss` model `Try` has one Control Design Block, `C`. The block `C` is initialized to default values, and the model `Try` has a current value that depends on the current value of `C`. Use `getValue` to evaluate `C` and `Try` to examine the current values.

- 1 Evaluate `C` to obtain its current value.

```
Cnow = getValue(C)
```

This command returns a numeric `pid` object whose coefficients reflect the current values of the tunable parameters in `C`.

- 2 Evaluate `Try` to obtain its current value.

```
Tnow = getValue(Try)
```

This command returns a numeric model that is equivalent to `feedback(G*Cnow,1)`.

### Access Values of Tuned Models and Blocks

Propagate changes in block values from one model to another using `getValue`.

This technique is useful for accessing values of models and blocks tuned with tuning commands such as `systemtune`, `looptune`, or `hinfstruct`. For example, if you have a closed-loop model of your control system `T0`, with two tunable blocks, `C1` and `C2`, you can tune it using:

```
[T,fSoft] = systemtune(T0,SoftReqs);
```

You can then access the tuned values of `C1` and `C2`, as well as any closed-loop model `H` that depends on `C1` and `C2`, using the following:

```
C1t = getValue(C1,T);  
C2t = getValue(C2,T);  
Ht = getValue(H,T);
```

### See Also

`genss` | `replaceBlock` | `systemtune` | `looptune` | `hinfstruct`

**Introduced in R2011b**

## gram

Controllability and observability Gramians

### Syntax

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

### Description

`Wc = gram(sys, 'c')` calculates the controllability Gramian of the state-space (ss) model `sys`.

`Wo = gram(sys, 'o')` calculates the observability Gramian of the ss model `sys`.

`Wc = gram( ____, opt)` calculates time-limited or frequency-limited Gramians. `opt` is an option set that specifies time or frequency intervals for the computation. Create `opt` using the `gramOptions` command.

You can use Gramians to study the controllability and observability properties of state-space models and for model reduction [1]. They have better numerical properties than the controllability and observability matrices formed by `ctrb` and `obsv`.

Given the continuous-time state-space model

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

the controllability Gramian is defined by

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

The controllability Gramian is positive definite if and only if  $(A, B)$  is controllable.

The observability Gramian is defined by

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

The observability Gramian is positive definite if and only if  $(A, C)$  is observable.

The discrete-time counterparts of the controllability and observability Gramians are

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

respectively.

Use time-limited or frequency-limited Gramians to examine the controllability or observability of states within particular time or frequency intervals. The definition of these Gramians is as described in [2].

## Examples

### Compute Frequency-Limited Gramian

Compute the controllability Gramian of the following state-space model. Focus the computation on the frequency interval with the most energy.

```
sys = ss([- .1 -1; 1 0], [1; 0], [0 1], 0);
```

The model contains a peak at 1 rad/s. Use `gramOptions` to specify an interval around that frequency.

```
opt = gramOptions('FreqIntervals', [0.8 1.2]);  
gc = gram(sys, 'c', opt)
```

```
gc =
```

```
    4.2132    -0.0000  
   -0.0000    4.2433
```



## 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).

## More About

### Algorithms

The controllability Gramian  $W_c$  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 Gramian  $W_o$  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.

The computation of time-limited and frequency-limited Gramians is as described in [2].

## References

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

- [2] Gawronski, W. and J.N. Juang. “Model Reduction in Limited Time and Frequency Intervals.” *International Journal of Systems Science*. Vol. 21, Number 2, 1990, pp. 349–376.

**See Also**

gramOptions | lyap | dlyap | hsvd | balreal

**Introduced before R2006a**

# gramOptions

Options for the gram command

## Syntax

```
opt = gramOptions
opt = gramOptions(Name,Value)
```

## Description

`opt = gramOptions` returns an option set with the default options for `gram`.

`opt = gramOptions(Name,Value)` returns an options set with the options specified by one or more `Name,Value` pair arguments.

## Examples

### Compute Frequency-Limited Gramian

Compute the controllability Gramian of the following state-space model. Focus the computation on the frequency interval with the most energy.

```
sys = ss([-0.1 -1;1 0],[1;0],[0 1],0);
```

The model contains a peak at 1 rad/s. Use `gramOptions` to specify an interval around that frequency.

```
opt = gramOptions('FreqIntervals',[0.8 1.2]);
gc = gram(sys,'c',opt)
```

```
gc =
```

```
    4.2132    -0.0000
   -0.0000    4.2433
```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'FreqIntervals', [0.8 1.2]`

#### 'FreqIntervals' — Frequency intervals for computing Gramians

`[]` (default) | two-column matrix

Frequency intervals for computing frequency-limited controllability and observability Gramians, specified as a matrix with two columns. Each row specifies a frequency interval [`fmin` `fmax`], where `fmin` and `fmax` are nonnegative frequencies, expressed in the frequency unit of the model. For example:

- To restrict the computation to the range between 3 rad/s and 15 rad/s, assuming the frequency unit of the model is rad/s, set `FreqIntervals` to `[3 15]`.
- To restrict the computation to two frequency intervals, 3-15 rad/s and 40-60 rad/s, use `[3 15; 40 60]`.
- To specify all frequencies below a cutoff frequency `fcut`, use `[0 fcut]`.
- To specify all frequencies above the cutoff, use `[fcut Inf]` in continuous time, or `[fcut pi/Ts]` in discrete time, where `Ts` is the sample time of the model.

The default value, `[]`, imposes no frequency limitation and is equivalent to `[0 Inf]` in continuous time or `[0 pi/Ts]` in discrete time. However, if you specify a `TimeIntervals` value other than `[]`, then this limit overrides `FreqIntervals = []`. If you specify both a `TimeIntervals` value and a `FreqIntervals` value, then the computation uses the union of these intervals.

#### 'TimeIntervals' — Time intervals for computing Gramians

`[]` (default) | two-column matrix

Time intervals for computing time-limited controllability and observability Gramians, specified as a matrix with two columns. Each row specifies a time interval [`tmin` `tmax`],

where `tmin` and `tmax` are nonnegative times, expressed in the time unit of the model. For example:

- To restrict the computation to the range between 3 s and 15 s, assuming the time unit of the model is seconds, set `TimeIntervals` to `[3 15]`.
- To restrict the computation to two time intervals, 3-15 s and 40-60 s, use `[3 15; 40 60]`.
- To specify all times from zero up to a cutoff time `tcut`, use `[0 tcut]`. To specify all times after the cutoff, use `[tcut Inf]`.

The default value, `[]`, imposes no time limitation and is equivalent to `[0 Inf]`. However, if you specify a `FreqIntervals` value other than `[]`, then this limit overrides `Timeintervals = []`. If you specify both a `TimeIntervals` value and a `FreqIntervals` value, then the computation uses the union of these intervals.

## Output Arguments

**opt** — Options for `gram`  
`gramOptions options set`

Options for `gram`, returned as a `gramOptions options set`. Use `opt` as the last argument to `gram` to compute time-limited or frequency-limited Gramians.

## See Also

`gram` | `hsvd`

Introduced in R2016a

## hasdelay

True for linear model with time delays

### Syntax

```
B = hasdelay(sys)
B = hasdelay(sys, 'elem')
```

### Description

`B = hasdelay(sys)` returns 1 (true) if the model `sys` has input delays, output delays, I/O delays, or internal delays, and 0 (false) otherwise. If `sys` is a model array, then `B` is true if least one model in `sys` has delays.

`B = hasdelay(sys, 'elem')` returns a logical array of the same size as the model array `sys`. The logical array indicates which models in `sys` have delays.

### See Also

`totaldelay` | `absorbDelay`

**Introduced before R2006a**

# hasInternalDelay

Determine if model has internal delays

## Syntax

```
B = hasInternalDelay(sys)
B = hasInternalDelay(sys, 'elem')
```

## Description

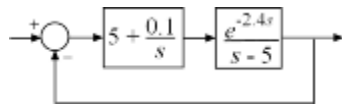
`B = hasInternalDelay(sys)` returns 1 (true) if the model `sys` has internal delays, and 0 (false) otherwise. If `sys` is a model array, then `B` is true if least one model in `sys` has delays.

`B = hasInternalDelay(sys, 'elem')` checks each model in the model array `sys` and returns a logical array of the same size as `sys`. The logical array indicates which models in `sys` have internal delays.

## Examples

### Check Model for Internal Delays

Build a dynamic system model of the following closed-loop system.



```
s = tf('s');
G = exp(-2.4*s)/(s-5);
C = pid(5,0.1);
sys = feedback(G*C,1);
```

Check if the model for internal delays.

```
B = hasInternalDelay(sys)
```

```
B =  
    logical  
    1
```

The model, `sys`, has an internal delay because of the transfer delay in the plant `G`. Therefore, `hasInternalDelay` returns 1.

## Input Arguments

**sys** — Model or array to check

dynamic system model | model array

Model or array to check for internal delays, specified as a dynamic system model or array of dynamic system models.

## Output Arguments

**B** — Flag indicating presence of internal delays

logical | logical array

Flag indicating presence of internal delays in input model or array, returned as a logical value or logical array.

## See Also

`getDelayModel` | `hasdelay`

**Introduced in R2013a**



# hsvd

Hankel singular values of dynamic system

## Syntax

```
hsv = hsvd(sys)  
hsv = hsvd(sys,opts)  
[hsv,baldata] = hsvd( ___ )  
hsvd( ___ )
```

## Description

*hsv* = `hsvd(sys)` computes the Hankel singular values *hsv* of the dynamic system *sys*. In state coordinates that equalize the input-to-state and state-to-output energy transfers, the Hankel singular values measure the contribution of each state to the input/output behavior. Hankel singular values are to model order what singular values are to matrix rank. In particular, small Hankel singular values signal states that can be discarded to simplify the model (see `balred`).

For models with unstable poles, `hsvd` only computes the Hankel singular values of the stable part and entries of *hsv* corresponding to unstable modes are set to `Inf`.

*hsv* = `hsvd(sys,opts)` computes the Hankel singular values using options that you specify using `hsvdOptions`. Options include offset and tolerance options for computing the stable-unstable decompositions. The options also allow you to limit the HSV computation to energy contributions within particular time and frequency intervals. See `hsvdOptions` for details.

[*hsv*,*baldata*] = `hsvd( ___ )` returns additional data to speed up model order reduction with `balred`. You can use this syntax with any of the previous combinations of input arguments.

`hsvd( ___ )` displays a Hankel singular values plot.

## Examples

### Compute Hankel Singular Values of System With Near-Unstable Pole

Create a system with a stable pole very near to 0, and display the Hankel singular values.

```
sys = zpk([1 2],[-1 -2 -3 -10 -1e-7],1);  
hsv = hsvd(sys)
```

```
hsv =  
  
1.0e+05 *  
  
1.6667  
0.0000  
0.0000  
0.0000  
0.0000
```

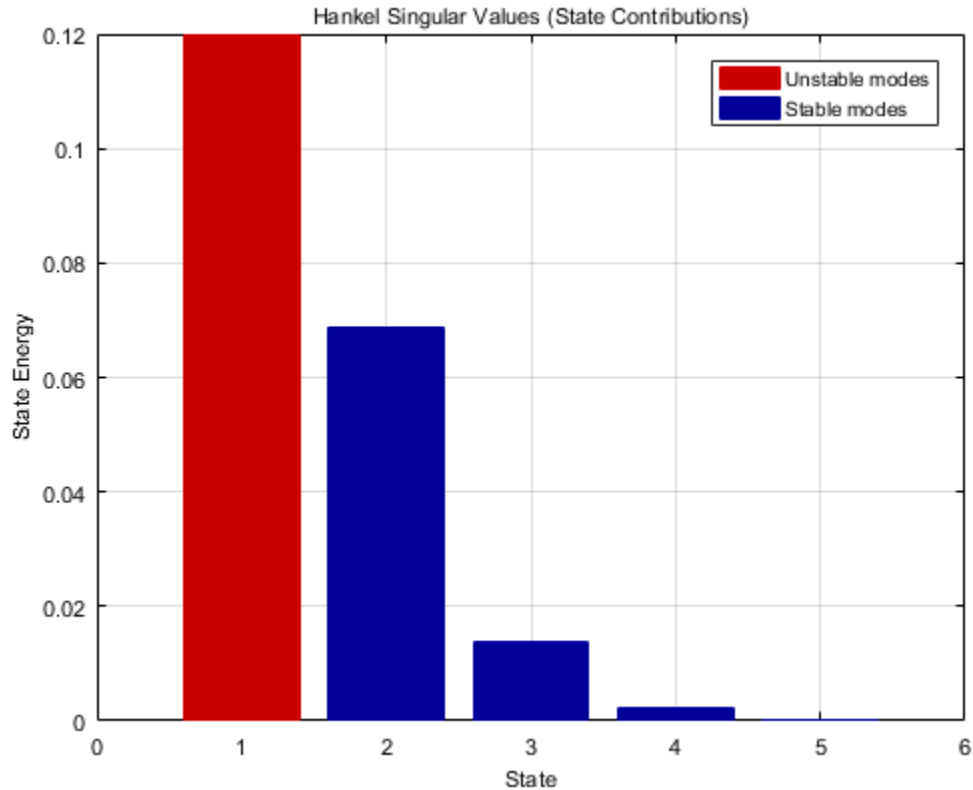
Notice the dominant Hankel singular value with magnitude  $10^5$ , which is so much larger that the significant digits of the other modes are not displayed. This value is due to the near-unstable mode at  $s = 10^{-7}$ . Use the 'Offset' option to treat this mode as unstable.

```
opts = hsvdOptions('Offset',1e-7);  
hsvu = hsvd(sys,opts)
```

```
hsvu =  
  
Inf  
0.0688  
0.0138  
0.0024  
0.0001
```

The Hankel singular value of modes that are unstable, or treated as unstable, is returned as `Inf`. Create a Hankel singular-value plot while treating this mode as unstable.

```
hsvd(sys,opts)
```



The unstable mode is shown in red on the plot.

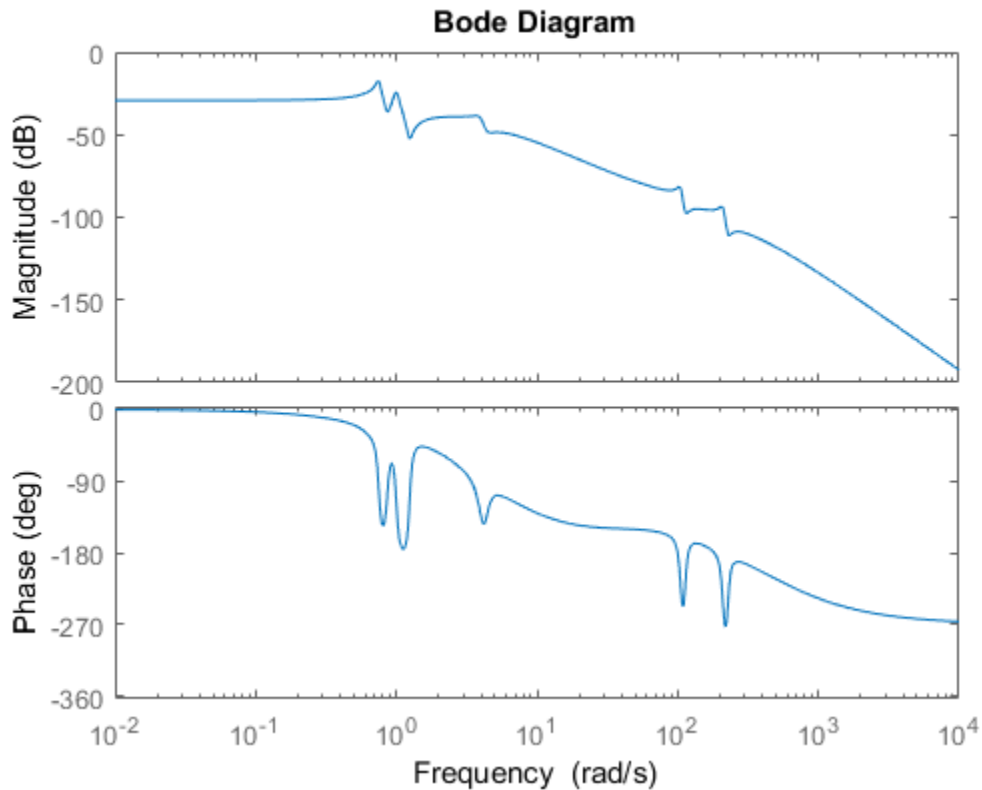
By default, `hsvd` uses a linear scale. To switch the plot to a log scale, right-click on the plot and select **Y Scale > Log**. For information about programmatically changing properties of HSV plots, see `hsvplot`.

### Frequency-Limited Hankel Singular Values

Compute the Hankel singular values of a model with low-frequency and high-frequency dynamics. Focus the calculation on the high-frequency modes.

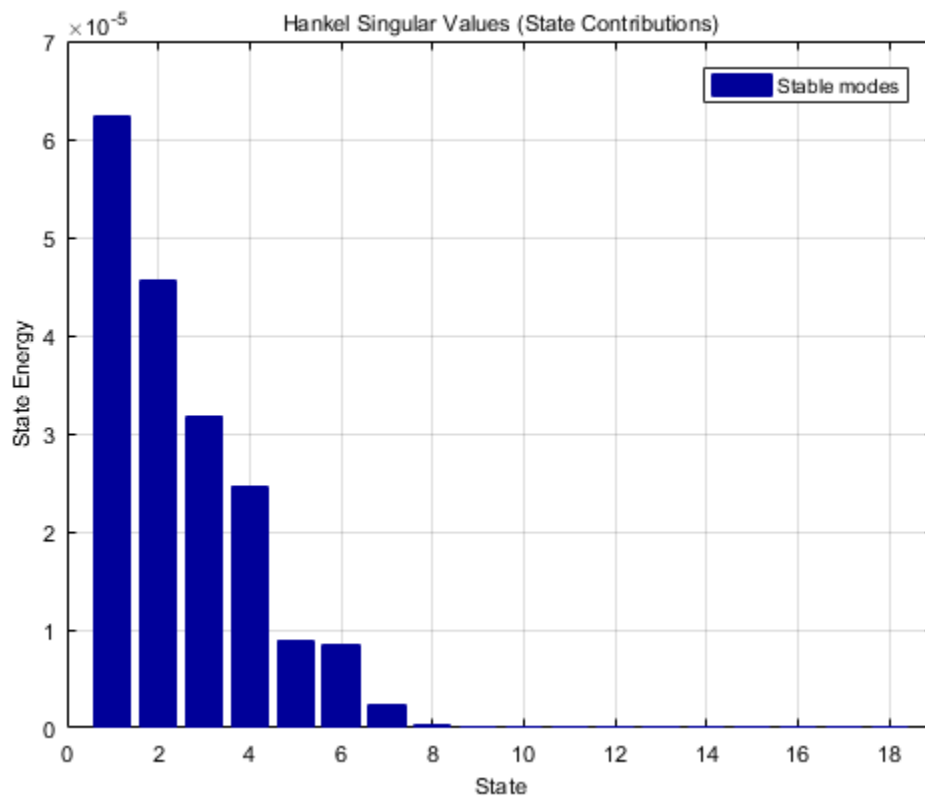
Load the model and examine its frequency response.

```
load modeselect Gms
bodeplot(Gms)
```



Gms has two sets of resonances, one at relatively low frequency and the other at relatively high frequency. Compute the Hankel singular values of the high-frequency modes, excluding the energy contributions to the low-frequency dynamics. To do so, use `hsvdOptions` to specify a frequency interval above 30 rad/s.

```
opts = hsvdOptions('FreqInterval',[30 Inf]);
hsvd(Gms,opts)
```



## More About

### Tips

To create a Hankel singular-value plot with more flexibility to programmatically customize the plot, use `hsvplot`.

### Algorithms

The `AbsTol`, `RelTol`, and `Offset` options of `hsvdOptions` are only used for models with unstable or marginally stable dynamics. Because Hankel singular values are only

meaningful for stable dynamics, `hsvd` must first split such models into the sum of their stable and unstable parts:

$$G = G_s + G_{ns}$$

This decomposition can be tricky when the model has modes close to the stability boundary (e.g., a pole at  $s = -1e-10$ ), or clusters of modes on the stability boundary (e.g., double or triple integrators). While `hsvd` is able to overcome these difficulties in most cases, it sometimes produces unexpected results such as

- 1 Large Hankel singular values for the stable part.

This happens when the stable part `G_s` contains some poles very close to the stability boundary. To force such modes into the unstable group, increase the `'Offset'` option to slightly grow the unstable region.

- 2 Too many modes are labeled "unstable." For example, you see 5 red bars in the HSV plot when your model had only 2 unstable poles.

The stable/unstable decomposition algorithm has built-in accuracy checks that reject decompositions causing a significant loss of accuracy in the frequency response. Such loss of accuracy arises, e.g., when trying to split a cluster of stable and unstable modes near  $s=0$ . Because such clusters are numerically equivalent to a multiple pole at  $s=0$ , it is actually desirable to treat the whole cluster as unstable. In some cases, however, large relative errors in low-gain frequency bands can trip the accuracy checks and lead to a rejection of valid decompositions. Additional modes are then absorbed into the unstable part `G_ns`, unduly increasing its order.

Such issues can be easily corrected by adjusting the `AbSTol` and `ReLTol` tolerances. By setting `AbSTol` to a fraction of smallest gain of interest in your model, you tell the algorithm to ignore errors below a certain gain threshold. By increasing `ReLTol`, you tell the algorithm to sacrifice some relative model accuracy in exchange for keeping more modes in the stable part `G_s`.

If you use the `TimeIntervals` or `FreqIntervals` options of `hsvdOptions`, then `hsvd` bases the computation of state energy contributions on time-limited or frequency-limited controllability and observability Gramians. For information about calculating time-limited and frequency-limited Gramians, see `gram` and [1].

## References

- [1] Gawronski, W. and J.N. Juang. “Model Reduction in Limited Time and Frequency Intervals.” *International Journal of Systems Science*. Vol. 21, Number 2, 1990, pp. 349–376.

## See Also

hsvdOptions | hsvplot | balred | balreal

**Introduced before R2006a**

## hsvdOptions

Create option set for computing Hankel singular values and input/output balancing

### Syntax

```
opts = hsvdOptions  
opts = hsvdOptions(Name,Value)
```

### Description

*opts* = hsvdOptions returns the default options for the `hsvd` and `balreal` commands.

*opts* = hsvdOptions(Name,Value) returns an options set with the options specified by one or more Name,Value pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as Name1,Value1, . . . ,NameN,ValueN.

#### 'FreqIntervals'

Frequency intervals for computing frequency-limited Hankel singular values (`hsvd`) or balanced realization (`balreal`), specified as a matrix with two columns. Each row specifies a frequency interval [fmin fmax], where fmin and fmax are nonnegative frequencies, expressed in the frequency unit of the model. For example:

- To restrict the computation to the range between 3 rad/s and 15 rad/s, assuming the frequency unit of the model is rad/s, set `FreqIntervals` to [3 15].



- To restrict the computation to two frequency intervals, 3-15 rad/s and 40-60 rad/s, use `[3 15; 40 60]`.
- To specify all frequencies below a cutoff frequency `fcut`, use `[0 fcut]`.
- To specify all frequencies above the cutoff, use `[fcut Inf]` in continuous time, or `[fcut pi/Ts]` in discrete time, where `Ts` is the sample time of the model.

The default value, `[]`, imposes no frequency limitation and is equivalent to `[0 Inf]` in continuous time or `[0 pi/Ts]` in discrete time. However, if you specify a `TimeIntervals` value other than `[]`, then this limit overrides `FreqIntervals = []`. If you specify both a `TimeIntervals` value and a `FreqIntervals` value, then the computation uses the union of these intervals.

**Default:** `[]`

### 'TimeIntervals'

Time intervals for computing time-limited Hankel singular values (`hsvd`) or balanced realization (`balreal`), specified as a matrix with two columns. Each row specifies a time interval `[tmin tmax]`, where `tmin` and `tmax` are nonnegative times, expressed in the time unit of the model. The software computes state contributions to the system's impulse response in these time intervals only. For example:

- To restrict the computation to the range between 3 s and 15 s, assuming the time unit of the model is seconds, set `TimeIntervals` to `[3 15]`.
- To restrict the computation to two time intervals, 3-15 s and 40-60 s, use `[3 15; 40 60]`.
- To specify all times from zero up to a cutoff time `tcut`, use `[0 tcut]`. To specify all times after the cutoff, use `[tcut Inf]`.

The default value, `[]`, imposes no time limitation and is equivalent to `[0 Inf]`. However, if you specify a `FreqIntervals` value other than `[]`, then this limit overrides `Timeintervals = []`. If you specify both a `TimeIntervals` value and a `FreqIntervals` value, then the computation uses the union of these intervals.

### 'AbsTol, RelTol'

Absolute and relative error tolerance for stable/unstable decomposition. Positive scalar values. For an input model  $G$  with unstable poles, `hsvd` and `balreal` first extract the stable dynamics by computing the stable/unstable decomposition  $G \rightarrow GS + GU$ .

The `AbsTol` and `RelTol` tolerances control the accuracy of this decomposition by ensuring that the frequency responses of  $G$  and  $GS + GU$  differ by no more than  $\text{AbsTol} + \text{RelTol} \cdot \text{abs}(G)$ . Increasing these tolerances helps separate nearby stable and unstable modes at the expense of accuracy. See `stabsep` for more information.

**Default:** `AbsTol = 0; RelTol = 1e-8`

**'Offset'**

Offset for the stable/unstable boundary. Positive scalar value. In the stable/unstable decomposition, the stable term includes only poles satisfying:

- $\text{Re}(s) < -\text{Offset} * \max(1, |\text{Im}(s)|)$  (Continuous time)
- $|z| < 1 - \text{Offset}$  (Discrete time)

Increase the value of `Offset` to treat poles close to the stability boundary as unstable.

**Default:** `1e-8`

For additional information on the options and how they affect the calculation, see `hsvd`. The time-limited and frequency-limited state contributions are calculated using the time-limited and frequency-limited controllability and observability Gramians, as described in `gram` and in [1].

## Examples

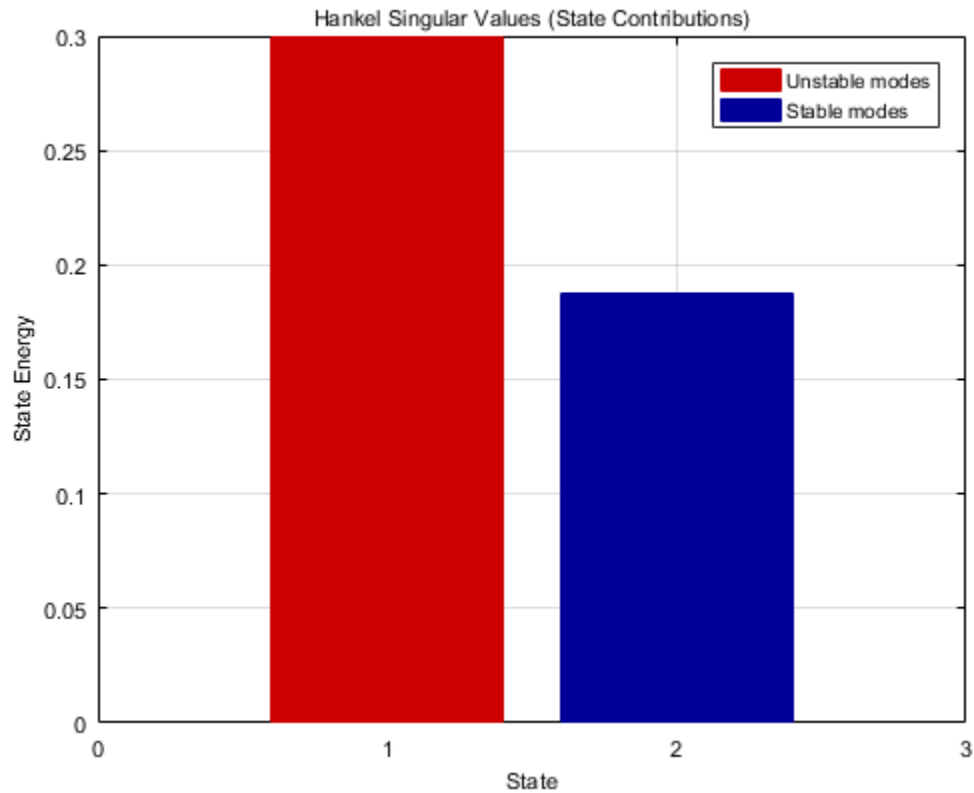
### Hankel Singular-Value Plot with Near-Unstable Pole

Compute the Hankel singular values of the system given by:

$$sys = \frac{(s + 0.5)}{(s + 10^{-6})(s + 2)}$$

Use the `Offset` option to force `hsvd` to exclude the pole at  $s = 10^{-6}$  from the stable term of the stable/unstable decomposition.

```
sys = zpk(-.5, [-1e-6 -2], 1);
opts = hsvdOptions('Offset', .001);
hsvd(sys, opts)
```



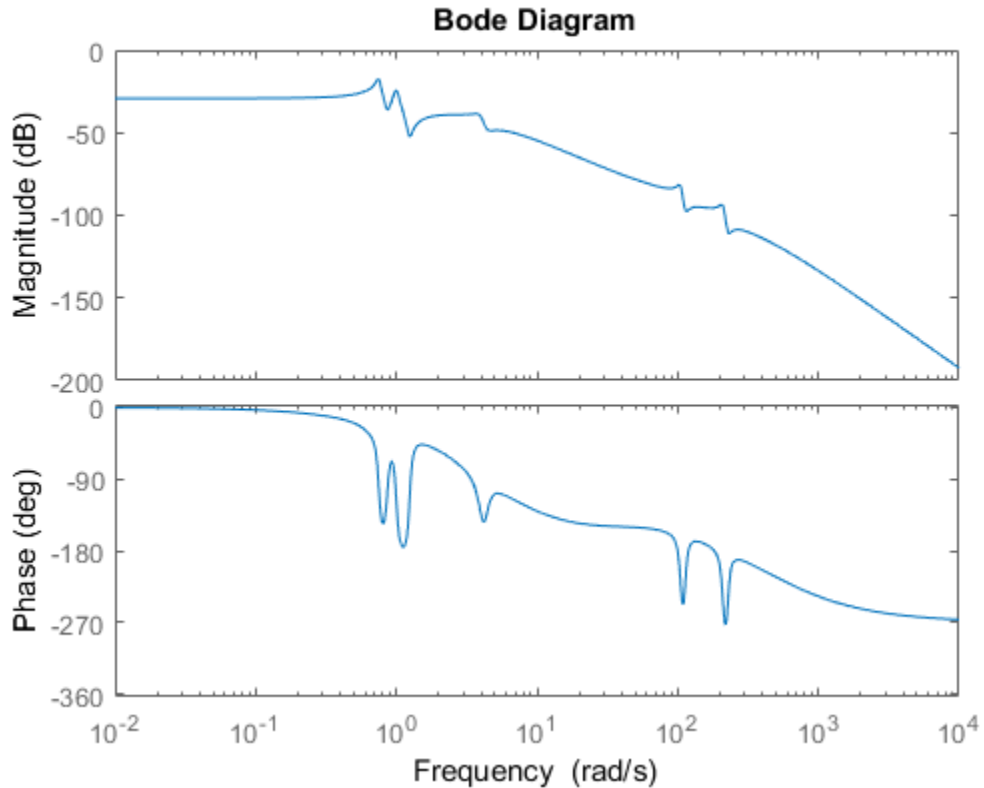
The plot shows that one state is treated as unstable. `hsvd` computes the energy contributions of the stable states only.

### Frequency-Limited Hankel Singular Values

Compute the Hankel singular values of a model with low-frequency and high-frequency dynamics. Focus the calculation on the high-frequency modes.

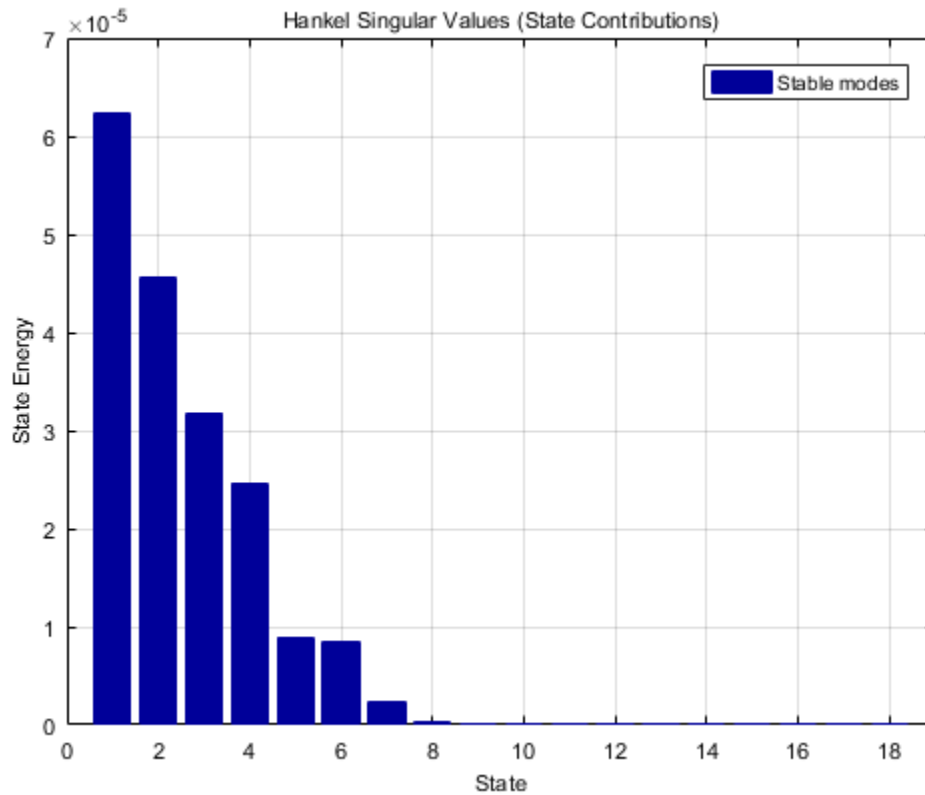
Load the model and examine its frequency response.

```
load modeselect Gms
bodeplot(Gms)
```



Gms has two sets of resonances, one at relatively low frequency and the other at relatively high frequency. Compute the Hankel singular values of the high-frequency modes, excluding the energy contributions to the low-frequency dynamics. To do so, use `hsvdOptions` to specify a frequency interval above 30 rad/s.

```
opts = hsvdOptions('FreqInterval',[30 Inf]);
hsvd(Gms,opts)
```



## References

- [1] Gawronski, W. and J.N. Juang. “Model Reduction in Limited Time and Frequency Intervals.” *International Journal of Systems Science*. Vol. 21, Number 2, 1990, pp. 349–376.

## See Also

hsvd | balreal | gram | balred

Introduced in R2010a

## hsvoptions

Plot options for `hsvplot`

### Syntax

```
P = hsvoptions  
P = hsvoptions('cstpref')
```

### Description

`P = hsvoptions` returns a list of available options for Hankel singular value (HSV) plots with default values set. Use dot notation to change the option values. You can use these options to customize the appearance of a Hankel singular value plot created with `hsvplot`.

`P = hsvoptions('cstpref')` initializes the plot options you selected in the Control System Toolbox Preferences Editor dialog box. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation.

The Hankel singular-value plot options include:

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid — [off on]	Show or hide the grid
GridColor — [Vector of RGB values in the range [0,1]   color   'none']	Color of the grid lines
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
YScale — [linear log]	Scale for Y-axis
<ul style="list-style-type: none"><li>• FreqIntervals</li><li>• TimeIntervals</li></ul>	Options for the Hankel singular value computation. See <code>hsvdOptions</code> for detailed information about these options.

Option	Description
<ul style="list-style-type: none"><li>• AbsTol</li><li>• RelTol</li><li>• Offset</li></ul>	

## Tips

- Both `hsvd` and `hsvplot` generate Hankel singular-value plots. `hsvplot` is useful when you want to customize properties of your plot such as axis limits, scale, and label styles. Use `hsvoptions` with `hsvplot` to define properties for your plot. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Examples

### Set Properties in HSV Plot

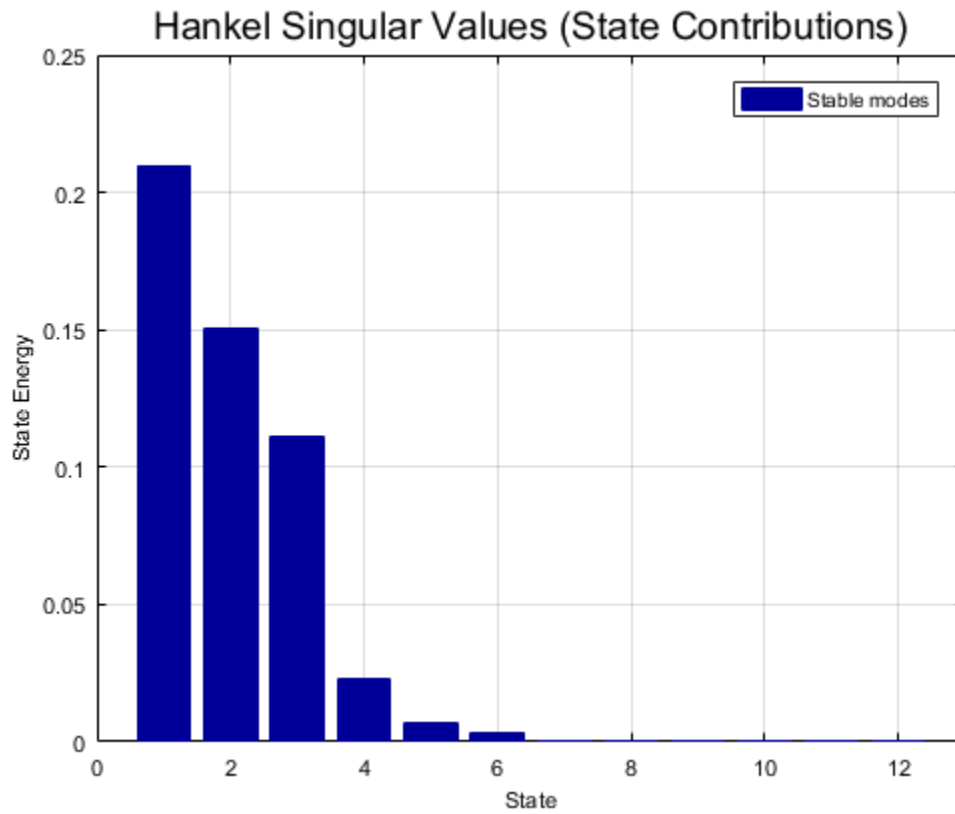
Use `hsvplot` to create a Hankel singular-value plot with and customized plot properties.

Create an options set for `hsvplot` that sets the `Yscale` property and the title font size.

```
P = hsvoptions;  
P.YScale = 'linear';  
P.Title.FontSize = 14;
```

Use the options set to generate an HSV plot. Note the linear y-axis scale in the plot.

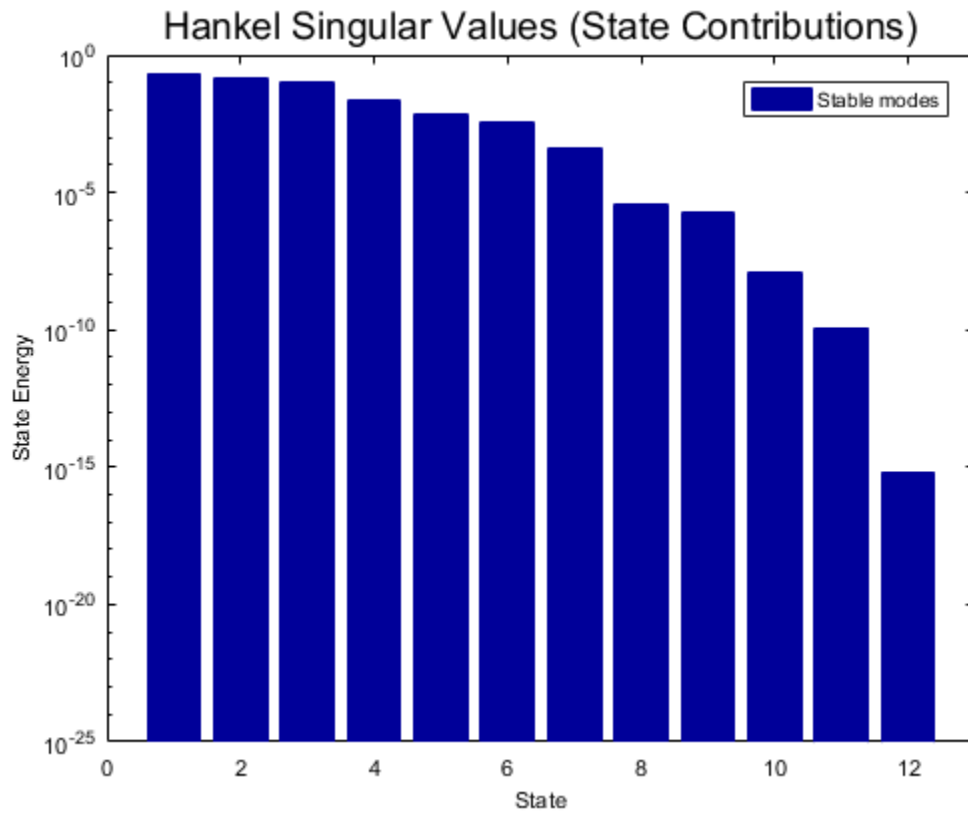
```
h = hsvplot(rss(12),P);
```



`hsvplot` returns a plot handle. You can use the plot handle to change properties of the existing plot. For example, switch to log scale and turn off the grid.

```
setoptions(h, 'Yscale', 'log', 'Grid', 'Off')
```





### See Also

`hsvd` | `hsvdOptions` | `hsvplot` | `getoptions` | `setoptions` | `stabsep`

Introduced in R2008a

## hsvplot

Plot Hankel singular values and return plot handle

### Syntax

```
h = hsvplot(sys)
hsvplot(sys)
hsvplot(sys, AbsTol',ATOL,'RelTol',RTOL,'Offset',ALPHA)
hsvplot(AX,sys,...)
```

### Description

`h = hsvplot(sys)` plots the Hankel singular values of an LTI system `sys` and returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. See `hsvoptions` for a list of some available plot options.

`hsvplot(sys)` plots the Hankel singular values of the LTI model `sys`. See `hsvd` for details on the meaning and purpose of Hankel singular values. The Hankel singular values for the stable and unstable modes of `sys` are shown in blue and red, respectively.

`hsvplot(sys, AbsTol',ATOL,'RelTol',RTOL,'Offset',ALPHA)` specifies additional options for computing the Hankel singular values.

`hsvplot(AX,sys,...)` attaches the plot to the axes with handle `AX`.

### Examples

#### Set Properties in HSV Plot

Use `hsvplot` to create a Hankel singular-value plot with and customized plot properties.

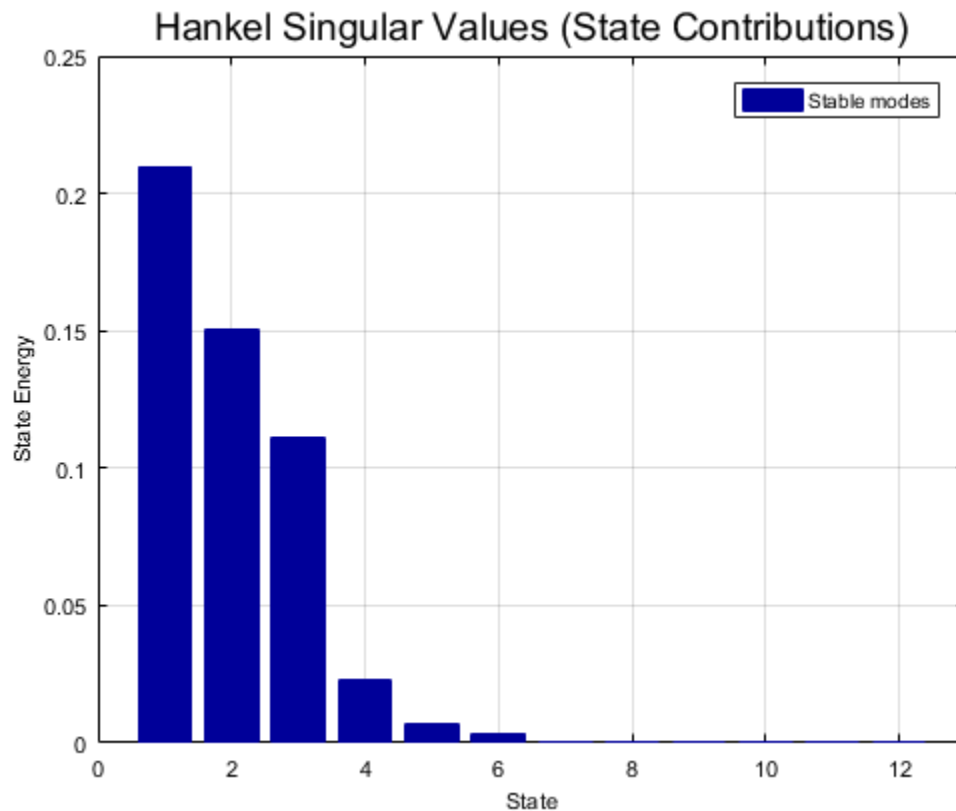
Create an options set for `hsvplot` that sets the `Yscale` property and the title font size.

```
P = hsvoptions;
```

```
P.YScale = 'linear';  
P.Title.FontSize = 14;
```

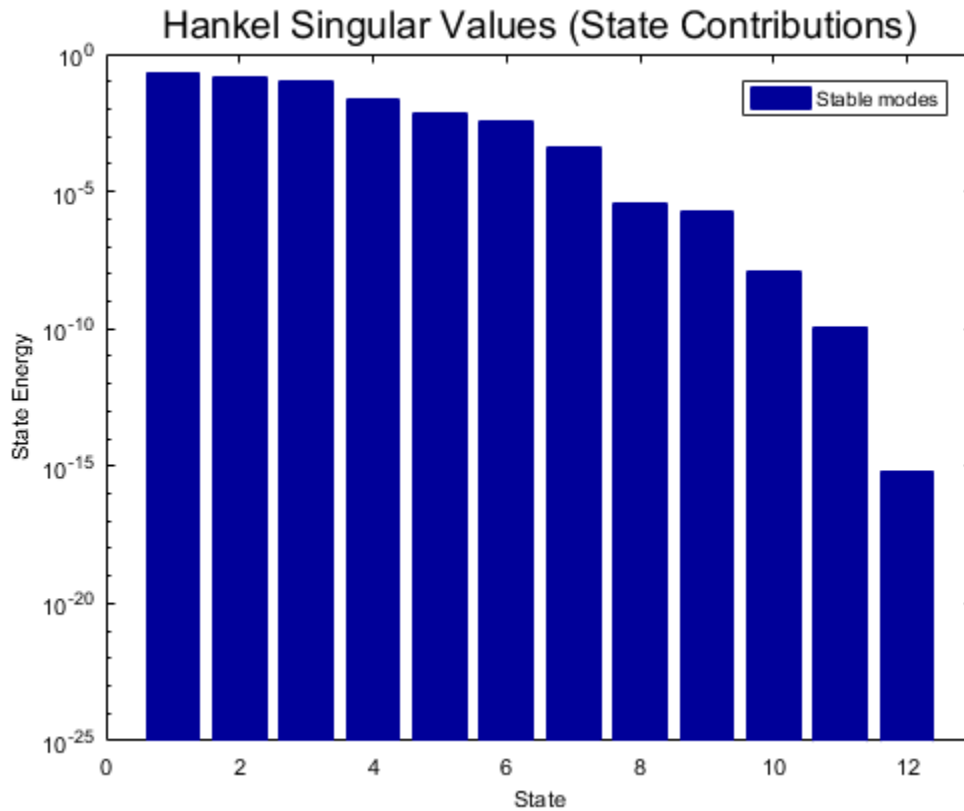
Use the options set to generate an HSV plot. Note the linear y-axis scale in the plot.

```
h = hsvplot(rss(12),P);
```



`hsvplot` returns a plot handle. You can use the plot handle to change properties of the existing plot. For example, switch to log scale and turn off the grid.

```
setoptions(h, 'Yscale', 'log', 'Grid', 'Off')
```



## More About

### Tips

- Both `hsvd` and `hsvplot` generate Hankel singular-value plots. `hsvplot` is useful when you want to customize properties of your plot such as axis limits, scale, and label styles. Use `hsvoptions` with `hsvplot` to define properties for your plot. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

**See Also**

`getoptions` | `hsvd` | `hsvoptions` | `setoptions`

**Introduced before R2006a**

## imp2exp

Convert implicit linear relationship to explicit input-output relation

### Syntax

```
B = imp2exp(A,yidx,uidx)
```

### Description

`B = imp2exp(A,yidx,uidx)` transforms a linear constraint between variables `Y` and `U` of the form  $A(:, [yidx; uidx]) * [Y; U] = 0$  into an explicit input/output relationship  $Y = B*U$ . The vectors `yidx` and `uidx` refer to the columns (inputs) of `A` as referenced by the explicit relationship for `B`.

The constraint matrix `A` can be a `double`, `ss`, `tf`, `zpk` and `frd` object as well as an uncertain object, including `umat`, `uss` and `ufrd`. The result `B` will be of the same class.

### Examples

#### Scalar Algebraic Constraint

Consider the constraint  $4y + 7u = 0$ . Solving for `y` gives  $y = -1.75u$ . You form the equation using `imp2exp`:

```
A = [4 7];  
Yidx = 1;  
Uidx = 2;
```

and then

```
B = imp2exp(A,Yidx,Uidx)  
B =  
    -1.7500
```

yields `B` equal to `-1.75`.

## Matrix Algebraic Constraint

Consider two motor/generator constraints among 4 variables  $[V; I; T; W]$ , namely  $[1 \ -1 \ 0 \ -2e-3; 0 \ -2e-3 \ 1 \ 0] * [V; I; T; W] = 0$ . You can find the 2-by-2 matrix  $B$  so that  $[V; T] = B * [W; I]$  using `imp2exp`.

```
A = [1 -1 0 -2e-3; 0 -2e-3 1 0];
Yidx = [1 3];
Uidx = [4 2];
B = imp2exp(A,Yidx,Uidx)
B =
    0.0020    1.0000
         0    0.0020
```

You can find the 2-by-2 matrix  $C$  so that  $[I; W] = C * [T; V]$

```
Yidx = [2 4];
Uidx = [3 1];
C = imp2exp(A,Yidx,Uidx)
C =
         500         0
    -250000    500
```

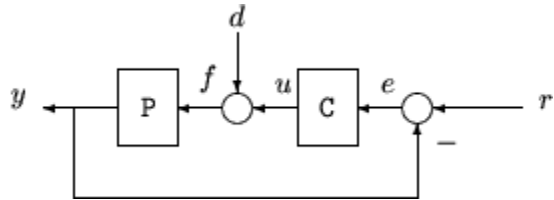
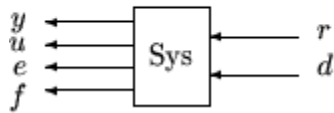
## Uncertain Matrix Algebraic Constraint

Consider two uncertain motor/generator constraints among 4 variables  $[V; I; T; W]$ , namely  $[1 \ -R \ 0 \ -K; 0 \ -K \ 1 \ 0] * [V; I; T; W] = 0$ . You can find the uncertain 2-by-2 matrix  $B$  so that  $[V; T] = B * [W; I]$ .

```
R = ureal('R',1,'Percentage',[-10 40]);
K = ureal('K',2e-3,'Percentage',[-30 30]);
A = [1 -R 0 -K; 0 -K 1 0];
Yidx = [1 3];
Uidx = [4 2];
B = imp2exp(A,Yidx,Uidx)
UMAT: 2 Rows, 2 Columns
    K: real, nominal = 0.002, variability = [-30 30]%, 2 occurrences
    R: real, nominal = 1, variability = [-10 40]%, 1 occurrence
```

## Scalar Dynamic System Constraint

Consider a standard single-loop feedback connection of controller  $C$  and an uncertain plant  $P$ , described by the equations  $e = r - y$ ;  $u = Ce$ ;  $f = d + u$ ;  $y = Pf$ .



```
P = tf([1],[1 0]);
C = tf([2*.707*1 1^2],[1 0]);
A = [1 -1 0 0 0 -1;0 -C 1 0 0 0;0 0 -1 -1 1 0;0 0 0 0 -P 1];
OutputIndex = [6;3;2;5]; % [y;u;e;f]
InputIndex = [1;4]; % [r;d]
Sys = imp2exp(A,OutputIndex,InputIndex);
Sys.InputName = {'r';'d'};
Sys.OutputName = {'y';'u';'e';'f'};
```

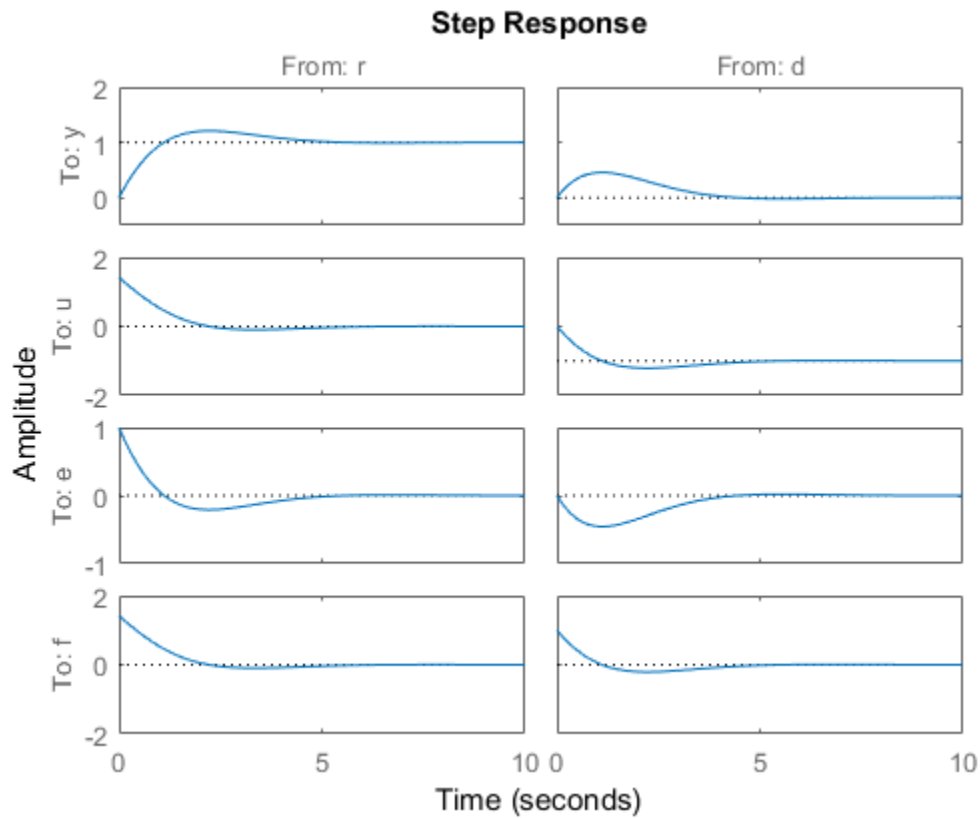
```
pole(Sys)
```

ans =

```
-0.7070 + 0.7072i
-0.7070 - 0.7072i
-0.7070 + 0.7072i
-0.7070 - 0.7072i
```

```
stepplot(Sys)
```





## More About

### Algorithms

The number of rows of  $A$  must equal the length of  $yidx$ .

### See Also

`iconnect` | `inv`

Introduced in R2011b

## impulse

Impulse response plot of dynamic system; impulse response data

### Syntax

```
impulse(sys)
impulse(sys,Tfinal)
impulse(sys,t)
impulse(sys1,sys2,...,sysN)
impulse(sys1,sys2,...,sysN,Tfinal)
impulse(sys1,sys2,...,sysN,t)
[y,t] = impulse(sys)
[y,t] = impulse(sys,Tfinal)
y = impulse(sys,t)
[y,t,x] = impulse(sys)
[y,t,x,yzd] = impulse(sys)
```

### Description

`impulse` calculates the unit impulse response of a dynamic system model. For continuous-time dynamic systems, the impulse response is the response to a Dirac input  $\delta(t)$ . For discrete-time systems, the impulse response is the response to a unit area pulse of length  $T_s$  and height  $1/T_s$ , where  $T_s$  is the sample time of the system. (This pulse approaches  $\delta(t)$  as  $T_s$  approaches zero.) For state-space models, `impulse` assumes initial state values are zero.

`impulse(sys)` plots the impulse response of the dynamic system 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,Tfinal)` simulates the impulse response from  $t = 0$  to the final time  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time ( $T_s = -1$ ), `impulse` interprets `Tfinal` as the number of sampling periods to simulate.

`impulse(sys,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time models, `t` should be of the form `Ti:Ts:Tf`, where `Ts` is the sample time. For continuous-time models, `t` should be of the form `Ti:dt:Tf`, where `dt` becomes the sample time of a discrete approximation to the continuous system (see “Algorithms” on page 2-437). The `impulse` command always applies the impulse at `t=0`, regardless of `Ti`.

To plot the impulse responses of several models `sys1`,..., `sysN` on a single figure, use:

```
impulse(sys1,sys2,...,sysN)
```

```
impulse(sys1,sys2,...,sysN,Tfinal)
```

```
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 output arguments:

```
[y,t] = impulse(sys)
```

```
[y,t] = impulse(sys,Tfinal)
```

```
y = impulse(sys,t)
```

`impulse` returns the output response `y` and the time vector `t` used for simulation (if not supplied as an argument to `impulse`). 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

For state-space models only:

```
[y,t,x] = impulse(sys)
```

(length of `t`) × (number of outputs) × (number of inputs)

and  $y(:, :, j)$  gives the response to an impulse disturbance entering the  $j$ th input channel. Similarly, the dimensions of  $x$  are (length of  $t$ )  $\times$  (number of states)  $\times$  (number of inputs)

`[y,t,x,yzd] = impulse(sys)` returns the standard deviation YSD of the response  $Y$  of an identified system  $SYS$ . YSD is empty if  $SYS$  does not contain parameter covariance information.

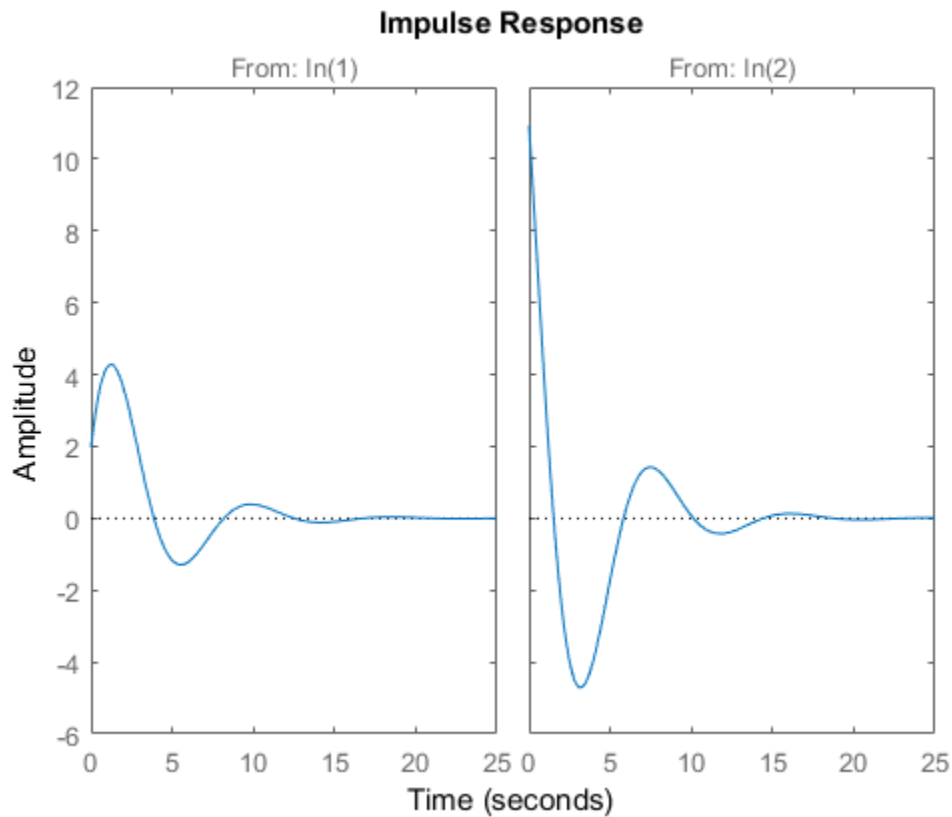
## Examples

### Impulse Response Plot of Second-Order State-Space Model

Plot the impulse response of the second-order state-space model

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$
$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

```
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 =
```

```
139    1    2
```

(the first dimension is the length of `t`). The impulse response of the first input channel is then accessed by

```
ch1 = y(:, :, 1);  
size(ch1)
```

```
ans =
```

```
139    1
```

## Impulse Data from Identified System

Fetch the impulse response and the corresponding 1 std uncertainty of an identified linear system .

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotordata'));  
z = iddata(y, u, 0.1, 'Name', 'DC-motor');  
set(z, 'InputName', 'Voltage', 'InputUnit', 'V');  
set(z, 'OutputName', {'Angular position', 'Angular velocity'});  
set(z, 'OutputUnit', {'rad', 'rad/s'});  
set(z, 'Tstart', 0, 'TimeUnit', 's');  
  
model = tfest(z, 2);  
[y, t, -, ysd] = impulse(model, 2);  
  
% Plot 3 std uncertainty  
subplot(211)  
plot(t, y(:, 1), t, y(:, 1) + 3 * ysd(:, 1), 'k:', t, y(:, 1) - 3 * ysd(:, 1), 'k:');  
subplot(212)  
plot(t, y(:, 2), t, y(:, 2) + 3 * ysd(:, 2), 'k:', t, y(:, 2) - 3 * ysd(:, 2), 'k:');
```

## Limitations

The impulse response of a continuous system with nonzero  $D$  matrix is infinite at  $t = 0$ . `impz` ignores this discontinuity and returns the lower continuity value  $Cb$  at  $t = 0$ .

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

## Algorithms

Continuous-time models are first converted to state space. The impulse response of a single-input state-space model

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

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

$$\begin{aligned}\dot{x} &= Ax, \quad x(0) = b \\ y &= Cx\end{aligned}$$

To simulate this response, the system is discretized using zero-order hold on the inputs. The sample time is chosen automatically based on the system dynamics, except when a time vector  $t = 0:dt:Tf$  is supplied ( $dt$  is then used as sample time).

## See Also

`initial` | `Linear System Analyzer` | `lsim` | `step`

**Introduced before R2006a**

# impzplot

Plot impulse response and return plot handle

## Syntax

```
impzplot(sys)
impzplot(sys,Tfinal)
impzplot(sys,t)
impzplot(sys1,sys2,...,sysN)
impzplot(sys1,sys2,...,sysN,Tfinal)
impzplot(sys1,sys2,...,sysN,t)
impzplot(AX,...)
impzplot(..., plotoptions)
h = impzplot(...)
```

## Description

`impzplot` plots the impulse response of the dynamic system model `sys`. For multi-input models, independent impulse commands are applied to each input channel. The time range and number of points are chosen automatically. For continuous systems with direct feedthrough, the infinite pulse at  $t=0$  is disregarded. `impzplot` can also return the plot handle, `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

`impzplot(sys)` plots the impulse response of the LTI model without returning the plot handle.

`impzplot(sys,Tfinal)` simulates the impulse response from  $t = 0$  to the final time  $t = Tfinal$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time ( $Ts = -1$ ), `impzplot` interprets `Tfinal` as the number of sampling intervals to simulate.

`impzplot(sys,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time



models, `t` should be of the form `Ti:Ts:Tf`, where `Ts` is the sample time. For continuous-time models, `t` should be of the form `Ti:dt:Tf`, where `dt` becomes the sample time of a discrete approximation to the continuous system (see `impz`). The `impzplot` command always applies the impulse at `t=0`, regardless of `Ti`.

To plot the impulse response of multiple LTI models `sys1,sys2,...` on a single plot, use:

```
impzplot(sys1,sys2,...,sysN)
impzplot(sys1,sys2,...,sysN,Tfinal)
impzplot(sys1,sys2,...,sysN,t)
```

You can also specify a color, line style, and marker for each system, as in

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

`impzplot(AX,...)` plots into the axes with handle `AX`.

`impzplot(..., plotoptions)` plots the impulse response with the options specified in `plotoptions`. Type

```
help timeoptions
```

for more detail.

`h = impzplot(...)` plots the impulse response and returns the plot handle `h`.

## Examples

### Example 1

Normalize the impulse response of a third-order system.

```
sys = rss(3);
h = impzplot(sys);
% Normalize responses
setoptions(h,'Normalize','on');
```

### Example 2

Plot the impulse response and the corresponding 1 std "zero interval" of an identified linear system.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotordata'));
z = iddata(y, u, 0.1, 'Name', 'DC-motor');
set(z, 'InputName', 'Voltage', 'InputUnit', 'V');
set(z, 'OutputName', {'Angular position', 'Angular velocity'});
set(z, 'OutputUnit', {'rad', 'rad/s'});
set(z, 'Tstart', 0, 'TimeUnit', 's');
model = n4sid(z,4,n4sidOptions('Focus', 'simulation'));
h = impulseplot(model,2);
showConfidence(h);
```

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`impulse` | `setoptions` | `getoptions`

**Introduced before R2006a**

# initial

Initial condition response of state-space model

## Syntax

```
initial(sys,x0)
initial(sys,x0,Tfinal)
initial(sys,x0,t)
initial(sys1,sys2,...,sysN,x0)
initial(sys1,sys2,...,sysN,x0,Tfinal)
initial(sys1,sys2,...,sysN,x0,t)
[y,t,x] = initial(sys,x0)
[y,t,x] = initial(sys,x0,Tfinal)
[y,t,x] = initial(sys,x0,t)
```

## Description

`initial(sys,x0)` calculates the unforced response of a state-space (ss) model `sys` with an initial condition on the states specified by the vector `x0`:

$$\begin{aligned}\dot{x} &= Ax, & x(0) &= x_0 \\ y &= Cx\end{aligned}$$

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

`initial(sys,x0,Tfinal)` simulates the response from  $t = 0$  to the final time  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time (`Ts = -1`), `initial` interprets `Tfinal` as the number of sampling periods to simulate.

`initial(sys,x0,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time models, `t` should be of the form `0:Ts:Tf`, where `Ts` is the sample time. For continuous-time models, `t` should be of the form `0:dt:Tf`, where `dt` becomes the sample time of a discrete approximation to the continuous system (see `impulse`).

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,Tfinal)
initial(sys1,sys2,...,sysN,x0,t)
```

(see `impulse` for details).

When invoked with output arguments,

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

## Examples

### Response of State-Space Model to Initial Condition

Plot the response of the following state-space model:

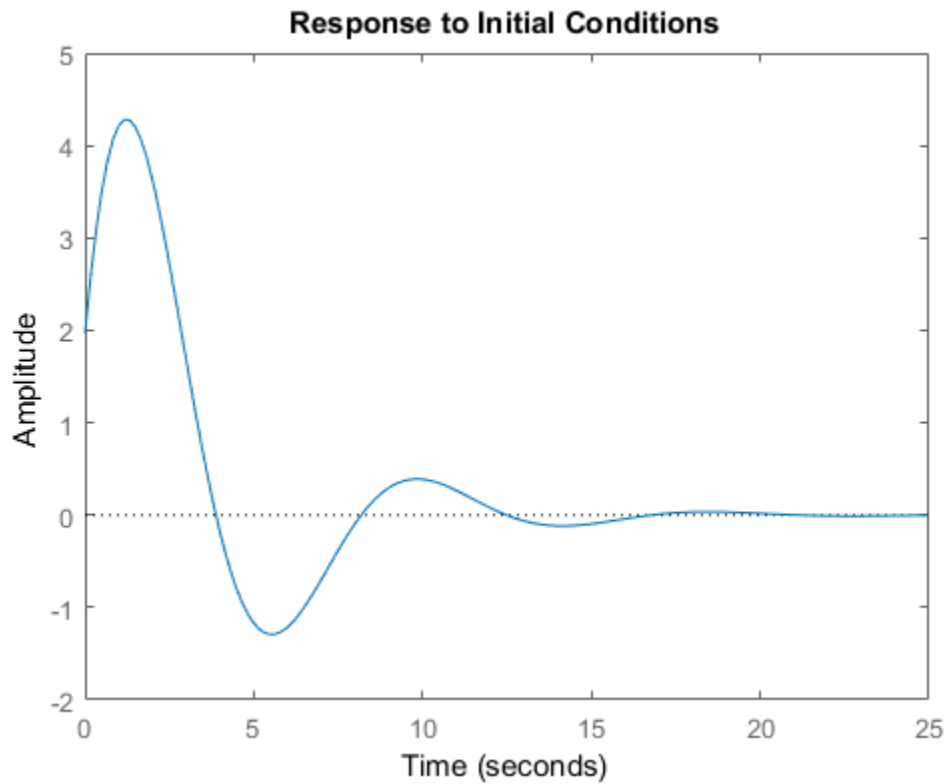
$$\begin{aligned} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} &= \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ y &= \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \end{aligned}$$

Take the following 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)
```



## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

**See Also**

impulse | Linear System Analyzer | lsim | step

**Introduced before R2006a**

# initialplot

Plot initial condition response and return plot handle

## Syntax

```
initialplot(sys,x0)
initialplot(sys,x0,Tfinal)
initialplot(sys,x0,t)
initialplot(sys1,sys2,...,sysN,x0)
initialplot(sys1,sys2,...,sysN,x0,Tfinal)
initialplot(sys1,sys2,...,sysN,x0,t)
initialplot(AX,...)
initialplot(..., plotoptions)
h = initialplot(...)
```

## Description

`initialplot(sys,x0)` plots the undriven response of the state-space (ss) model `sys` with initial condition `x0` on the states. This response is characterized by these equations:

Continuous time:  $\dot{x} = A x$ ,  $y = C x$ ,  $x(0) = x0$

Discrete time:  $x[k+1] = A x[k]$ ,  $y[k] = C x[k]$ ,  $x[0] = x0$

The time range and number of points are chosen automatically. `initialplot` also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

`initialplot(sys,x0,Tfinal)` simulates the response from  $t = 0$  to the final time  $t = Tfinal$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time ( $Ts = -1$ ), `initialplot` interprets `Tfinal` as the number of sampling periods to simulate.

`initialplot(sys,x0,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time models, `t` should be of the form `0:Ts:Tf`, where `Ts` is the sample time. For continuous-time models, `t` should be of the form `0:dt:Tf`, where `dt` becomes the sample time of a discrete approximation to the continuous system (see `impulse`).

To plot the initial condition responses of several LTI models on a single figure, use

```
initialplot(sys1,sys2,...,sysN,x0)
```

```
initialplot(sys1,sys2,...,sysN,x0,Tfinal)
```

```
initialplot(sys1,sys2,...,sysN,x0,t)
```

You can also specify a color, line style, and marker for each system, as in

```
initialplot(sys1,'r',sys2,'y--',sys3,'gx',x0).
```

```
initialplot(AX,...) plots into the axes with handle AX.
```

```
initialplot(..., plotoptions) plots the initial condition response with the options specified in plotoptions. Type
```

```
help timeoptions
```

for more detail.

```
h = initialplot(...) plots the system response and returns the plot handle h.
```

## Examples

Plot a third-order system's response to initial conditions and use the plot handle to change the plot's title.

```
sys = rss(3);  
h = initialplot(sys,[1,1,1])  
p = getoptions(h); % Get options for plot.  
p.Title.String = 'My Title'; % Change title in options.  
setoptions(h,p); % Apply options to the plot.
```



## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`getoptions` | `initial` | `setoptions`

**Introduced before R2006a**

## interp

Interpolate FRD model

### Syntax

```
isys = interp(sys,freqs)
```

### Description

`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`. If `sys` is an IDFRD model (requires System Identification Toolbox software), the noise spectrum, if non-empty, is also interpolated. The response and noise covariance data, if available, are also interpolated.

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).

### See Also

`freqresp` | `frd`

**Introduced before R2006a**

## inv

Invert models

## Syntax

inv

## Description

inv inverts the input/output relation

$$y = G(s)u$$

to produce the model 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.

## Examples

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. These commands produce the following result.

Transfer function from input 1 to output...

#1: 1

#2: 0

Transfer function from input 2 to output...

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

#2: 1

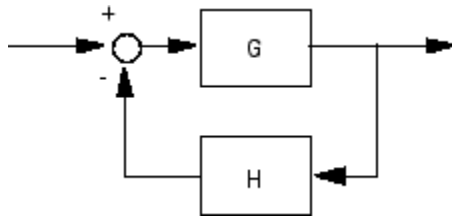
You can verify that

$H * Hi$

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:

$s$

-----  
 $(s^2 + s + 1)$

**Introduced before R2006a**

## **iopzmap**

Plot pole-zero map for I/O pairs of model

### **Syntax**

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

### **Description**

`iopzmap(sys)` computes and plots the poles and zeros of each input/output pair of the dynamic system 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 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 model arrays, `iopzmap` plots the poles and zeros of each model in the array on the same diagram.

### **Examples**

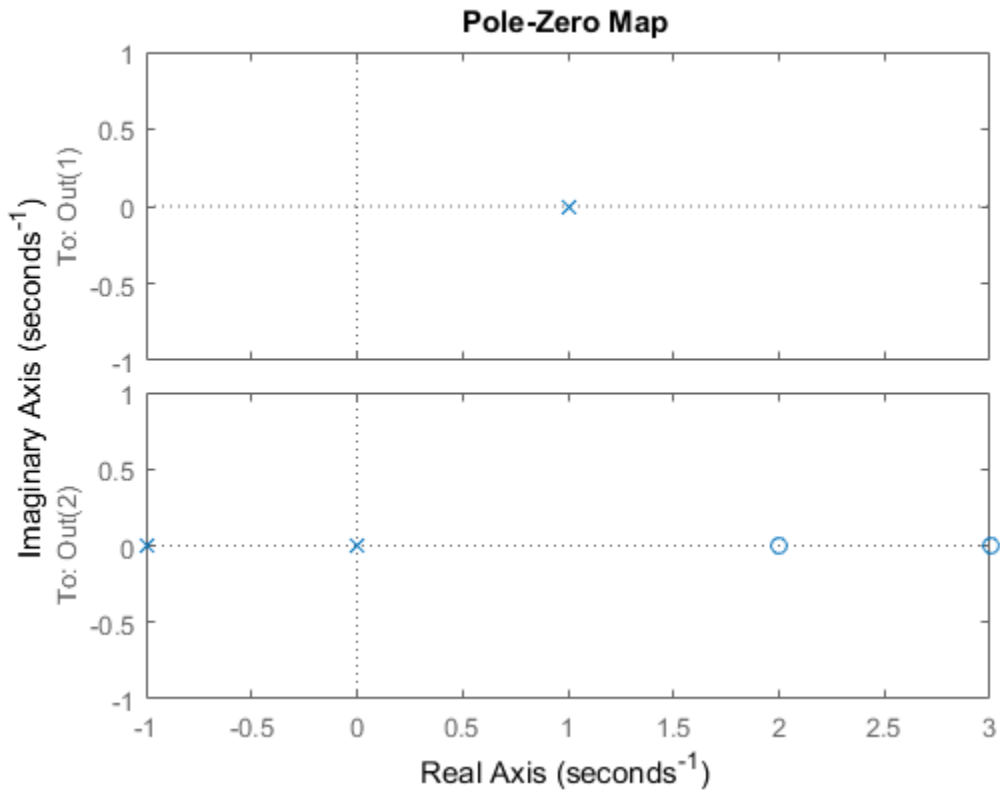
#### **Pole-Zero Map for MIMO System**

Create a one-input, two-output dynamic system.

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

Plot a pole-zero map.

```
iopzmap(H)
```

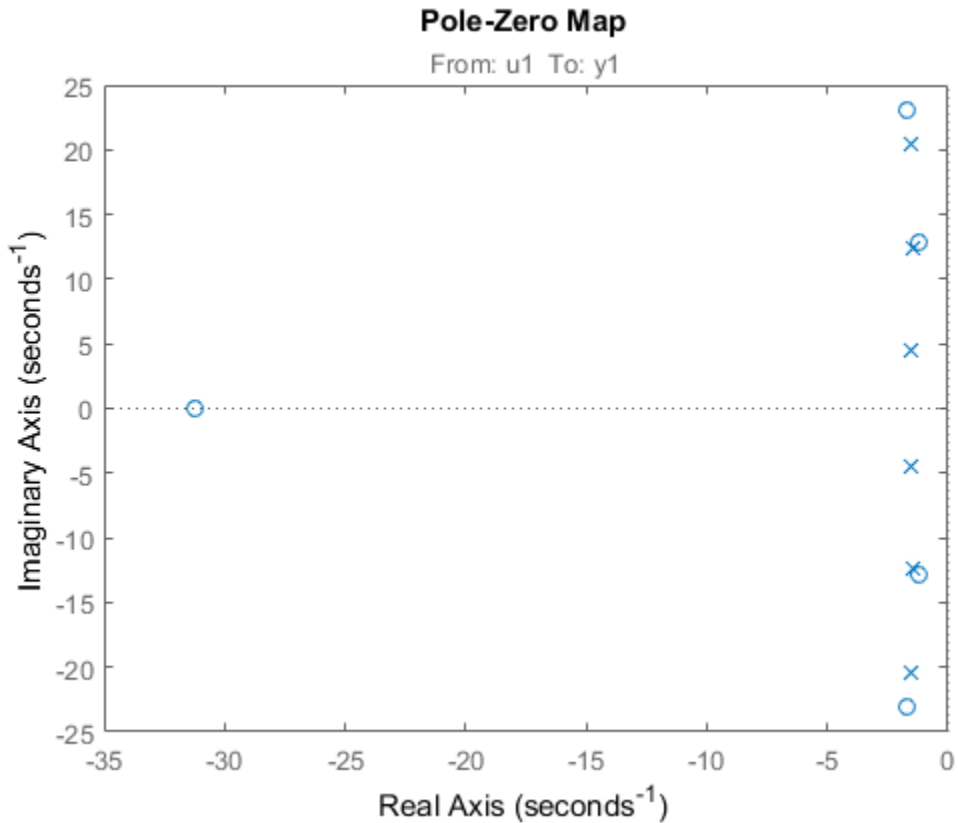


`iopzmap` generates a separate map for each I/O pair in the system.

### Pole-Zero Map of Identified Model

View the poles and zeros of an over-parameterized state-space model estimated from input-output data. (Requires System Identification Toolbox™).

```
load iddata1
sys = ssest(z1,6,ssestOptions('focus','simulation'));
iopzmap(sys)
```



The plot shows that there are two pole-zero pairs that almost overlap, which hints are their potential redundancy.

## More About

### Tips

For additional options for customizing the appearance of the pole-zero plot, use `iopzplot`.



## **See Also**

pole | zero | sgrid | zgrid | iopzplot | pzmap

**Introduced before R2006a**

## iopzplot

Plot pole-zero map for I/O pairs and return plot handle

### Syntax

```
h = iopzplot(sys)
iopzplot(sys1,sys2,...)
iopzplot(AX,...)
iopzplot(..., plotoptions)
```

### Description

`h = iopzplot(sys)` computes and plots the poles and zeros of each input/output pair of the dynamic system model `sys`. The poles are plotted as x's and the zeros are plotted as o's. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help pzoptions
```

for a list of available plot options. For more information on the ways to change properties of your plots, see “Ways to Customize Plots”.

`iopzplot(sys1,sys2,...)` shows the poles and zeros of multiple dynamic system models `sys1,sys2,...` on a single plot. You can specify distinctive colors for each model, as in

```
iopzplot(sys1,'r',sys2,'y',sys3,'g')
```

`iopzplot(AX,...)` plots into the axes with handle `AX`.

`iopzplot(..., plotoptions)` plots the poles and zeros with the options specified in `plotoptions`. Type

```
help pzoptions
```

for more detail.

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

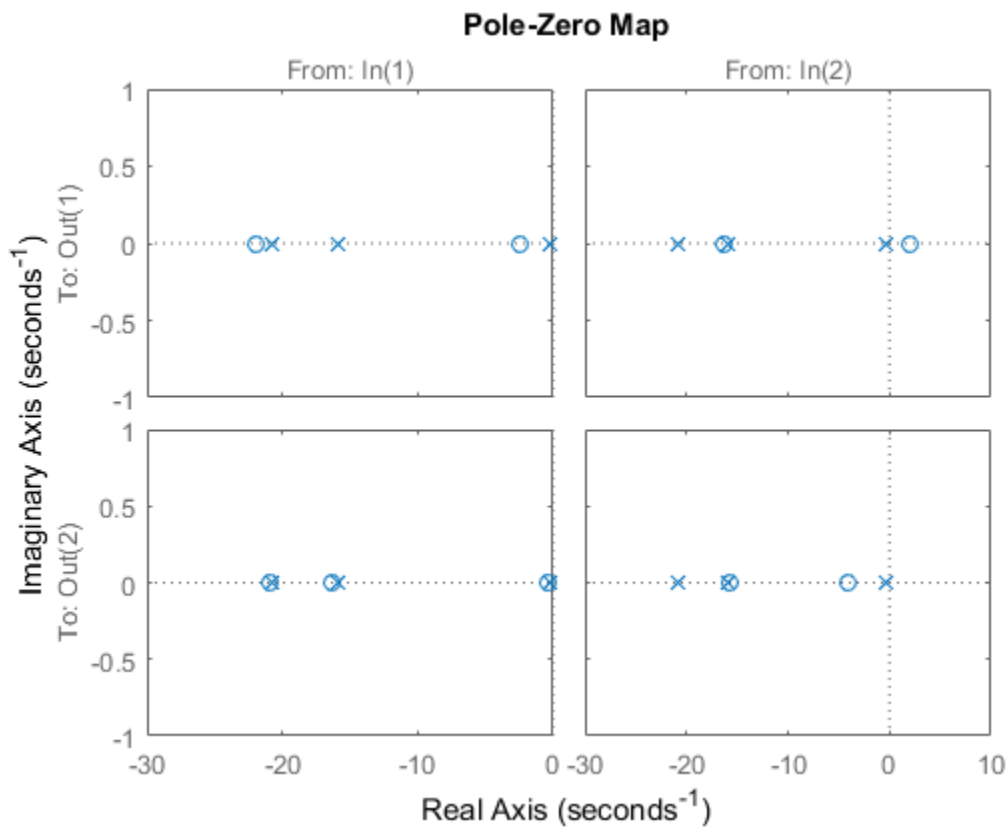
For arrays `sys` of LTI models, `iopzplot` plots the poles and zeros of each model in the array on the same diagram.

## Examples

### Change I/O Grouping on Pole/Zero Map

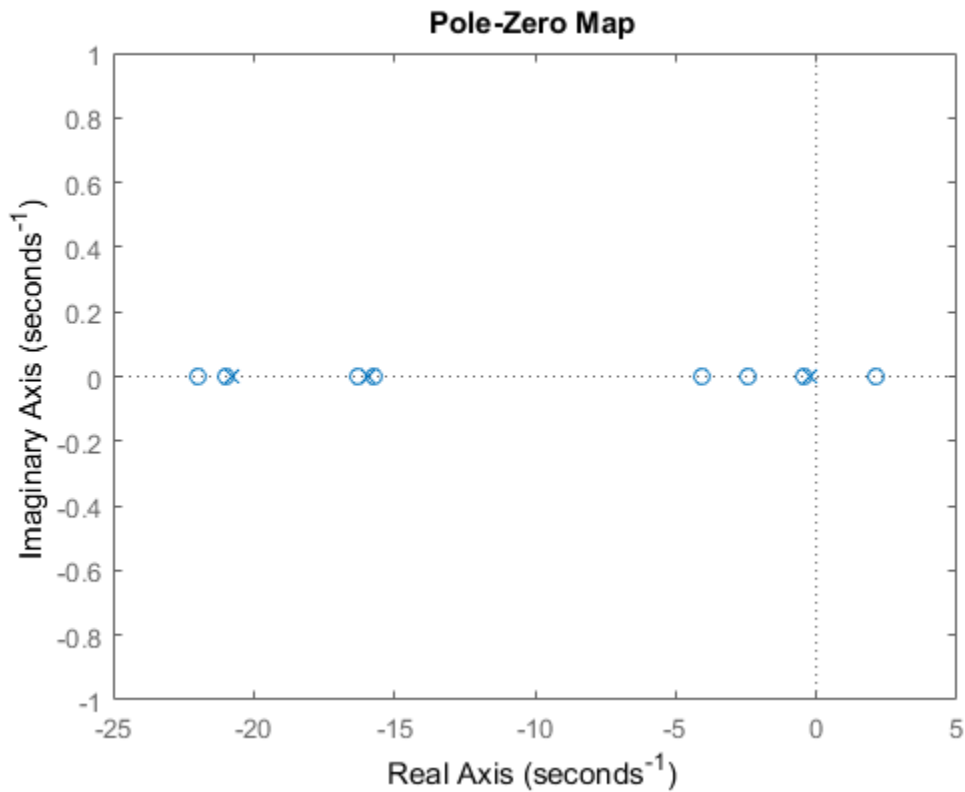
Create a pole/zero map of a two-input, two-output dynamic system.

```
sys = rss(3,2,2);
h = iopzplot(sys);
```



By default, the plot displays the poles and zeros of each I/O pair on its own axis. Use the plot handle to view all I/Os on a single axis.

```
setoptions(h, 'IOGrouping', 'all')
```

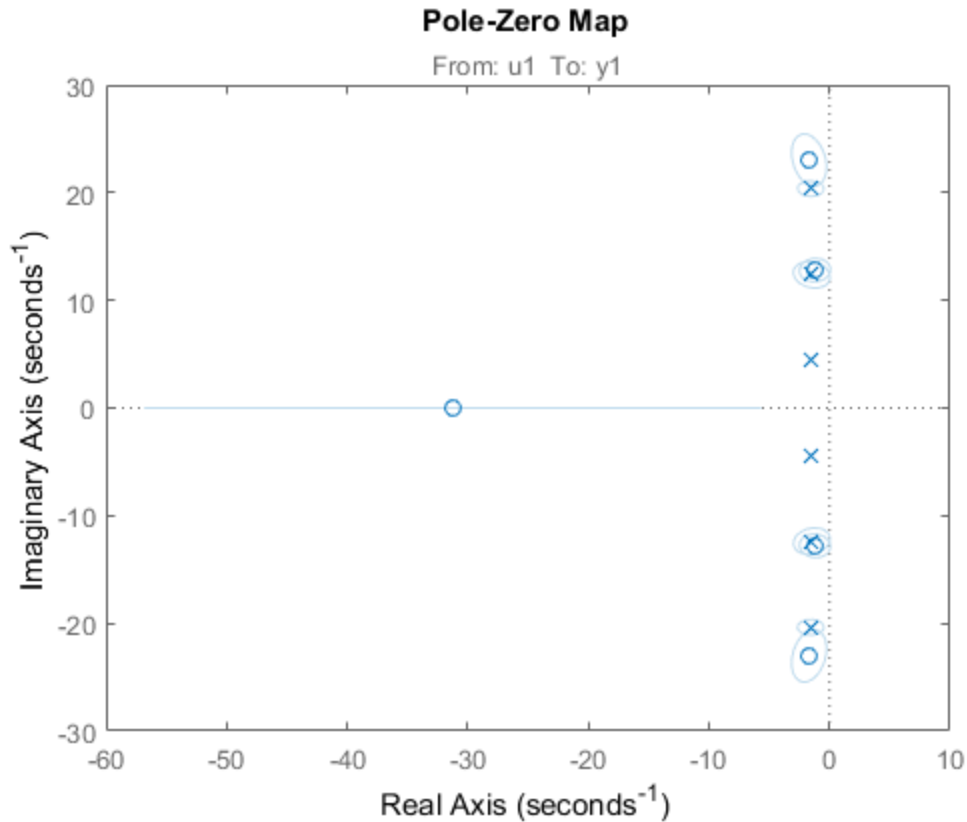


### Use Pole-Zero Map to Examine Identified Model

View the poles and zeros of a sixth-order state-space model estimated from input-output data. Use the plot handle to display the confidence intervals of the identified model's pole and zero locations.

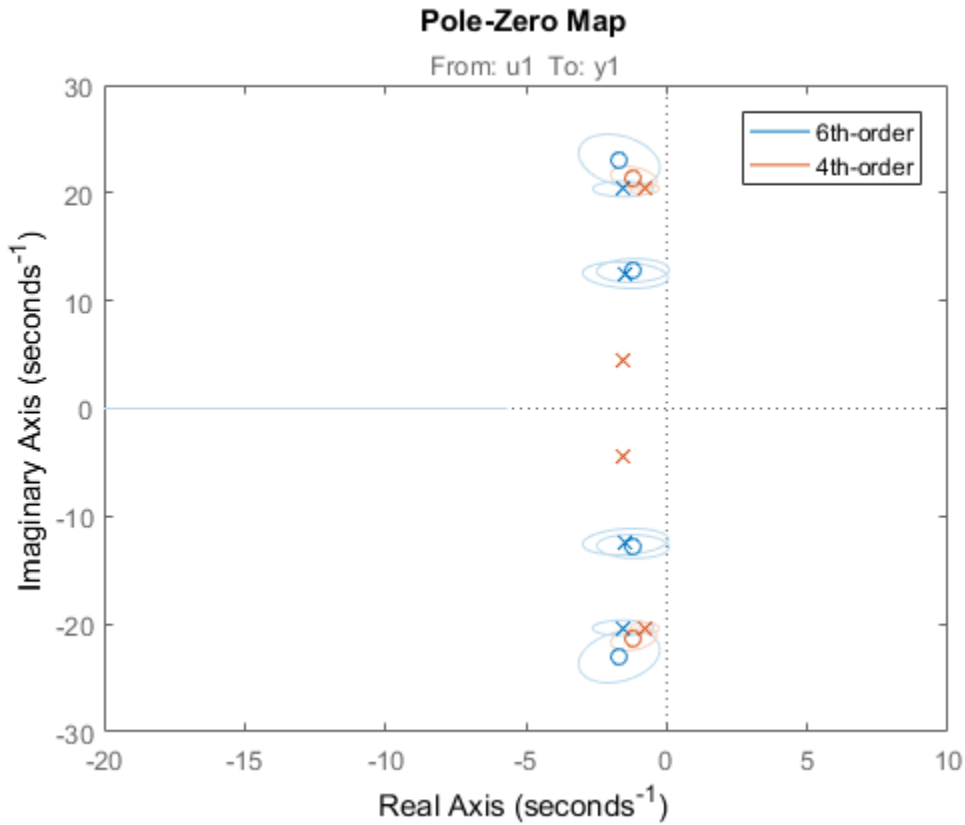
```
load iddata1  
sys = sstest(z1,6,sstestOptions('focus','simulation'));
```

```
h = iopzplot(sys);
showConfidence(h)
```



There is at least one pair of complex-conjugate poles whose locations overlap with those of a complex zero, within the 1- $\sigma$  confidence region. This suggests their redundancy. Hence, a lower (4th) order model might be more robust for the given data.

```
sys2 = ssest(z1,4,ssestOptions('focus','simulation'));
h = iopzplot(sys,sys2);
showConfidence(h)
legend('6th-order','4th-order')
axis([-20, 10 -30 30])
```



The fourth-order model `sys2` shows less variability in the pole-zero locations.

### See Also

`iopzmap` | `setoptions` | `getoptions`

Introduced before R2006a

## isct

Determine if dynamic system model is in continuous time

### Syntax

```
bool = isct(sys)
```

### Description

`bool = isct(sys)` returns a logical value of 1 (**true**) if the dynamic system model `sys` is a continuous-time model. The function returns a logical value of 0 (**false**) otherwise.

### Input Arguments

#### **sys**

Dynamic system model or array of such models.

### Output Arguments

#### **bool**

Logical value indicating whether `sys` is a continuous-time model.

`bool = 1 (true)` if `sys` is a continuous-time model (`sys.Ts = 0`). If `sys` is a discrete-time model, `bool = 0 (false)`.

For a static gain, both `isct` and `isdtd` return **true** unless you explicitly set the sample time to a nonzero value. If you do so, `isdtd` returns **true** and `isct` returns **false**.

For arrays of models, `bool` is **true** if the models in the array are continuous.

### See Also

`isdtd` | `isstable`

**Introduced in R2007a**



## isdt

Determine if dynamic system model is in discrete time

### Syntax

```
bool = isdt(sys)
```

### Description

`bool = isdt(sys)` returns a logical value of 1 (**true**) if the dynamic system model `sys` is a discrete-time model. The function returns a logical value of 0 (**false**) otherwise.

### Input Arguments

#### **sys**

Dynamic system model or array of such models.

### Output Arguments

#### **bool**

Logical value indicating whether `sys` is a discrete-time model.

`bool = 1 (true)` if `sys` is a discrete-time model (`sys.Ts ≠ 0`). If `sys` is a continuous-time model, `bool = 0 (false)`.

For a static gain, both `isct` and `isdt` return **true** unless you explicitly set the sample time to a nonzero value. If you do so, `isdt` returns **true** and `isct` returns **false**.

For arrays of models, `bool` is **true** if the models in the array are discrete.

### See Also

`isct` | `isstable`

**Introduced in R2007a**

## isempty

Determine whether dynamic system model is empty

### Syntax

```
isempty(sys)
```

### Description

`isempty(sys)` returns a logical value of 1 (**true**) if the dynamic system model `sys` has no input or no output, and a logical value of 0 (**false**) otherwise. Where `sys` is a `frd` model, `isempty(sys)` returns 1 when the frequency vector is empty. Where `sys` is a model array, `isempty(sys)` returns 1 when the array has empty dimensions or when the LTI models in the array are empty.

### Examples

Both commands

```
isempty(tf) % tf by itself returns an empty transfer function  
isempty(ss(1,2,[],[]))
```

return 1 while

```
isempty(ss(1,2,3,4))
```

returns 0.

### See Also

`size` | `issiso`

Introduced before R2006a

## isfinite

Determine if model has finite coefficients

### Syntax

```
B = isfinite(sys)
B = isfinite(sys, 'elem')
```

### Description

`B = isfinite(sys)` returns a logical value of 1 (`true`) if the model `sys` has finite coefficients, and a logical value of 0 (`false`) otherwise. If `sys` is a model array, then `B = 1` if all models in `sys` have finite coefficients.

`B = isfinite(sys, 'elem')` checks each model in the model array `sys` and returns a logical array of the same size as `sys`. The logical array indicates which models in `sys` have finite coefficients.

### Examples

#### Check Model for Finite Coefficients

Create model and check whether its coefficients are all finite.

```
sys = rss(3);
B = isfinite(sys)
```

```
B =
```

```
logical
```

```
1
```

The model, `sys`, has finite coefficients.

### Check Each Model in Array

Create a 1-by-5 array of models, and check each model for finite coefficients.

```
sys = rss(2,2,2,1,5);  
B = isfinite(sys, 'elem')
```

B =

```
1×5 logical array  
  
1 1 1 1 1
```

`isfinite` checks each model in the model array, `sys`, and returns a logical array indicating which models have all finite coefficients.

## Input Arguments

**sys** — Model or array to check

input-output model | model array

Model or array to check, specified as an input-output model or model array. Input-output models include dynamic system models such as numeric LTI models and generalized models. Input-output models also include static models such as tunable parameters or generalized matrices.

## Output Arguments

**B** — Flag indicating whether model has finite coefficients

logical | logical array

Flag indicating whether model has finite coefficients, returned as a logical value or logical array.

## See Also

`isreal`

**Introduced in R2013a**

## isParametric

Determine if model has tunable parameters

### Syntax

```
bool = isParametric(M)
```

### Description

`bool = isParametric(M)` returns a logical value of 1 (**true**) if the model `M` contains parametric (tunable) “Control Design Blocks”. The function returns a logical value of 0 (**false**) otherwise.

### Input Arguments

**M**

A Dynamic System model or Static model, or an array of such models.

### Output Arguments

**bool**

Logical value indicating whether `M` contains tunable parameters.

`bool = 1 (true)` if the model `M` contains parametric (tunable) “Control Design Blocks” such as `realp` or `tunableSS`. If `M` does not contain parametric Control Design Blocks, `bool = 0 (false)`.

### More About

- “Control Design Blocks”
- “Dynamic System Models”

- “Static Models”

### **See Also**

nblocks

**Introduced in R2011a**



# isPassive

Check passivity of linear systems

## Syntax

```
pf = isPassive(G)
pf = isPassive(G,nu,rho)
[pf,R] = isPassive(G, ___ )
```

## Description

`pf = isPassive(G)` returns a logical value of 1 (**true**) if the dynamic system model **G** is passive, and a logical value of 0 (**false**) otherwise. A system is *passive* if all its I/O trajectories  $(u(t),y(t))$  satisfy:

$$\int_0^T y(t)^\top u(t) dt > 0,$$

for all  $T > 0$ . Equivalently, a system is passive if its frequency response is positive real, which means that for all  $\omega > 0$ ,

$$G(j\omega) + G(j\omega)^H > 0$$

(or the discrete-time equivalent). If **G** is a model array, then `isPassive` returns a logical array of the same array dimensions as **G**, where each entry in the array reflects the passivity of the corresponding entry in **G**.

For more information about the notion of passivity, see “About Passivity and Passivity Indices”.

`pf = isPassive(G,nu,rho)` returns 1 (**true**) if **G** is passive with index **nu** at the inputs, and index **rho** at the outputs. Such systems satisfy:

$$\int_0^T y(t)^\top u(t) dt > \nu \int_0^T u(t)^\top u(t) dt + \rho \int_0^T y(t)^\top y(t) dt,$$

for all  $T > 0$ .

- Use `rho = 0` to check whether a system is *input passive* with index `nu` at the inputs.
- Use `nu = 0` to check whether a system is *output passive* with index `rho` at the outputs.

For more information about input and output passivity, see “About Passivity and Passivity Indices”.

`[pf,R] = isPassive(G, ___)` also returns the relative index for the corresponding passivity bound (see `getPassiveIndex`). `R` measures the amount by which the passivity property is satisfied ( $R < 1$ ) or violated ( $R > 1$ ). You can use this syntax with any of the previous combinations of input arguments.

## Examples

### Check Passivity of Dynamic System

Test whether the following transfer function is passive:

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

```
G = tf([1,1],[1,2]);  
[pf,R] = isPassive(G)
```

```
pf =
```

```
logical
```

```
1
```

```
R =
```

```
0.3333
```

`pf = 1` indicates that `G` is passive. `R = 0.3333` indicates that `R` has a relative excess of passivity.

### Check Input and Output Passivity

Test whether the transfer function `G` is input passive with index `0.25`. To do so, use `nu = 0.25` and `rho = 0`.

```
G = tf([1,1],[1,2]);  
[pfin,Rin] = isPassive(G,0.25,0)
```

```
pfin =  
    logical  
    1
```

```
Rin =  
    0.6096
```

The result shows that **G** is input passive with this  $\nu$  value and has some excess passivity.

Test whether **G** is output passive with index 2.

```
[pfout,Rout] = isPassive(G,0,2)
```

```
pfout =  
    logical  
    0
```

```
Rout =  
    2.6180
```

Here, the result  $\text{pfout} = 0$  shows that **G** is not output passive with this  $\rho$  value. The **R** value gives a relative measure of the shortage of passivity.

### Check Passivity of Models in Array

You can use `isPassive` to evaluate the passivity of multiple models in a model array simultaneously. For this example, generate a random array of transfer function models.

```
G = rss(3,1,1,1,5);
```

**G** is a 1-by-5 array of 3-state SISO models. Check the passivity of all the models in **G**.

```
[pf,R] = isPassive(G)
```

```
pf =
```

```
1×5 logical array
```

```
0 0 0 1 0
```

```
R =
```

```
35.3759      Inf      Inf    0.1130    4.3096
```

**pf** and **R** are also 1-by-5 arrays. Each **pf** entry indicates whether the corresponding model in **G** is passive. Likewise, each **R** value gives the relative excess or shortage of passivity in the corresponding model in **G**. For instance, examine the passivity of the second entry in **G**, and compare the result with the second entries in **pf** and **R**.

```
[pf2,R2] = isPassive(G(:, :, 2))
```

```
pf2 =
```

```
logical
```

```
0
```

```
R2 =
```

```
Inf
```

- “Passivity Indices”

## Input Arguments

### **G** — Model to analyze

dynamic system model | model array

Model to analyze for passivity, specified as a dynamic system model such as a `tf`, `ss`, or `genss` model. `G` can be MIMO, if the number of inputs equals the number of outputs. `G` can be continuous or discrete. If `G` is a generalized model with tunable or uncertain blocks, `isPassive` evaluates passivity of the current, nominal value of `G`.

### **nu** — Input passivity index

0 (default) | real scalar

Input passivity index, specified as a real scalar value. Use `nu` and `rho` to specify particular passivity bounds. To check whether a system is passive with a particular index at the inputs, set `nu` to that value and set `rho` = 0.

### **rho** — Output passivity index

0 (default) | real scalar

Output passivity index, specified as a real scalar value. Use `nu` and `rho` to specify particular passivity bounds. To check whether a system is passive with a particular passivity index at the outputs, set `rho` to that value and set `nu` = 0.

## Output Arguments

### **pf** — Passivity indicator

1 (true) | 0 (false) | logical array

Passivity indicator, returned as a boolean value:

- 1 (**true**) if `G` is passive.
- 0 (**false**) if `G` is not passive.

If you specify input and output passivity indices `nu` and `rho`, then `pf` indicates passivity with respect to the corresponding passivity bound.

If `G` is a model array, then `pf` is an array of the same size, where `pf(k)` indicates the passivity of the `k`th entry in `G`, `G(:, :, k)`.

### **R** — Relative passivity index

positive real scalar

Relative passivity index, returned as a positive real scalar. `R` measures the excess ( $R < 1$ ) or shortage ( $R > 1$ ) of passivity in the system.

If you specify  $\nu \neq 0$  or  $\rho \neq 0$ , then **R** measures how much the specified passivity properties are satisfied or violated.

For more information about the notion of relative passivity index, see “About Passivity and Passivity Indices”.

### More About

- “About Passivity and Passivity Indices”

### See Also

`getPassiveIndex` | `getPeakGain` | `getSectorCrossover` | `getSectorIndex` | `passiveplot` | `sectorplot`

**Introduced in R2016a**

# isproper

Determine if dynamic system model is proper

## Syntax

```
B = isproper(sys)
B = isproper(sys, 'elem')
[B,sysr] = isproper(sys)
```

## Description

`B = isproper(sys)` returns a logical value of **1 (true)** if the dynamic system model `sys` is proper and a logical value of **0 (false)** otherwise.

A proper model has relative degree  $\leq 0$  and is causal. SISO transfer functions and zero-pole-gain models are proper if the degree of their numerator is less than or equal to the degree of their denominator (in other words, if they have at least as many poles as zeroes). MIMO transfer functions are proper if all their SISO entries are proper. Regular state-space models (state-space models having no **E** matrix) are always proper. A descriptor state-space model that has an invertible **E** matrix is always proper. A descriptor state-space model having a singular (non-invertible) **E** matrix is proper if the model has at least as many poles as zeroes.

If `sys` is a model array, then **B** is **1** if all models in the array are proper.

`B = isproper(sys, 'elem')` checks each model in a model array `sys` and returns a logical array of the same size as `sys`. The logical array indicates which models in `sys` are proper.

`[B,sysr] = isproper(sys)` also returns an equivalent model `sysr` with fewer states (reduced order) and a non-singular **E** matrix, if `sys` is a proper descriptor state-space model with a non-invertible **E** matrix. If `sys` is not proper, `sysr = sys`.

## Examples

### Examine Whether Models are Proper

The following commands

```
B1 = isproper(tf([1 0],1))      % transfer function s
B2 = isproper(tf([1 0],[1 1])) % transfer function s/(s+1)
```

return 0 (false) and 1 (true), respectively.

### Compute Equivalent Lower-Order Model

Combining state-space models sometimes yields results that include more states than necessary. Use `isproper` to compute an equivalent lower-order model.

```
H1 = ss(tf([1 1],[1 2 5]));
H2 = ss(tf([1 7],[1]));
H = H1*H2;
size(H)
```

State-space model with 1 outputs, 1 inputs, and 4 states.

H is proper and reducible. `isproper` returns the reduced model.

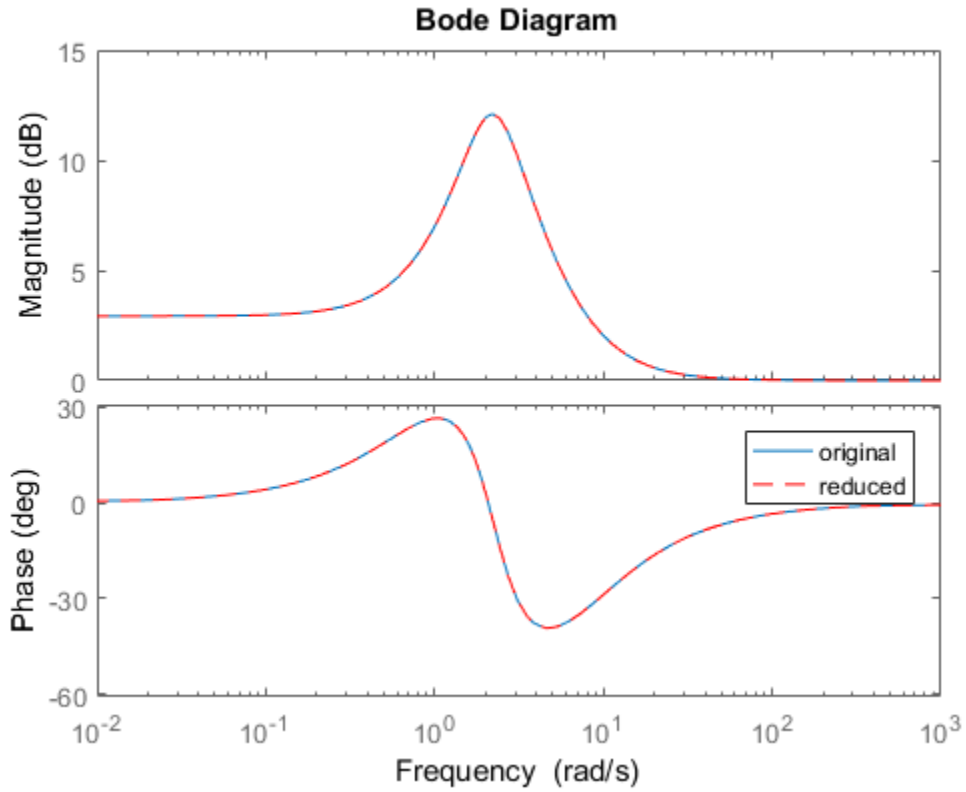
```
[isprop,Hr] = isproper(H);
size(Hr)
```

State-space model with 1 outputs, 1 inputs, and 2 states.

H and Hr are equivalent, as a Bode plot demonstrates.

```
bodeplot(H,Hr,'r--')
legend('original','reduced')
```





### See Also

ss | dss

Introduced before R2006a

# isreal

Determine if model has real-valued coefficients

## Syntax

```
B = isreal(sys)
B = isreal(sys, 'elem')
```

## Description

`B = isreal(sys)` returns a logical value of 1 (**true**) if the model `sys` has real-valued coefficients, and a logical value of 0 (**false**) otherwise. If `sys` is a model array, then `B = 1` if all models in `sys` have real-valued coefficients.

`B = isreal(sys, 'elem')` checks each model in the model array `sys` and returns a logical array of the same size as `sys`. The logical array indicates which models in `sys` have real coefficients.

## Examples

### Check Model for Real-Valued Coefficients

Create a model and check whether its coefficients are all real-valued.

```
sys = rss(3);
B = isreal(sys)
```

```
B =
    logical
     1
```

The model, `sys`, has real-valued coefficients.

### Check Model Array for Real-Valued Coefficients

Create a 1-by-5 array of models, and check each model for real-valued coefficients.

```
sys = rss(2,2,2,1,5);  
B = isreal(sys, 'elem')
```

B =

```
1×5 logical array  
  
1 1 1 1 1
```

`isreal` checks each model in the model array, `sys`, and returns a logical array indicating which models have all real-valued coefficients.

## Input Arguments

**sys** — Model or array to check

input-output model | model array

Model or array to check, specified as an input-output model or model array. Input-output models include dynamic system models such as numeric LTI models and generalized models. Input-output models also include static models such as tunable parameters or generalized matrices.

## Output Arguments

**B** — Flag indicating whether model has real-valued coefficients

logical | logical array

Flag indicating whether model has real-valued coefficients, returned as a logical value or logical array.

## See Also

`isfinite`

**Introduced in R2013a**

# isstable

Determine whether system is stable

## Syntax

```
B = isstable(sys)
B = isstable(sys, 'elem')
```

## Description

`B = isstable(sys)` returns a logical value of 1 (**true**) if the dynamic system model `sys` has stable dynamics, and a logical value of 0 (**false**) otherwise. If `sys` is a model array, then `B = 1` only if all models in `sys` are stable.

`B = isstable(sys, 'elem')` returns a logical array of the same dimensions as the model array `sys`. The logical array indicates which models in `sys` are stable.

`isstable` is only supported for analytical models with a finite number of poles.

## Examples

### Determine Stability of Models in Model Array

Create an array of SISO transfer function models with poles varying from -2 to 2. To do so, first initialize an array of dimension `[1, length(a)]` with zero-valued SISO transfer functions.

```
a = [-2:2];
sys = tf(zeros(1,1,1,length(a)));
```

Populate this array with transfer functions of the form  $1/(s-a)$ .

```
for j = 1:length(a)
    sys(1,1,1,j) = tf(1,[1 -a(j)]);
end
sys.SamplingGrid = struct('a',a);
```

Examine the stability of the model array.

```
B_all = isstable(sys)
```

```
B_all =  
logical  
0
```

By default, `isstable` returns a single Boolean value that is 1 (`true`) only if all models in the array are stable. `sys` contains some models with nonnegative poles, which are not stable. Therefore, `isstable` returns 0 (`false`) for the entire array.

Examine stability of each model in the array, element by element.

```
B_elem = isstable(sys, 'elem')
```

```
B_elem =  
1×5 logical array  
1 1 0 0 0
```

The `'elem'` flag causes `isstable` to return an array of Boolean values, which indicate the stability of the corresponding entry in the model array. For example, `B_elem(2) = 1`, which indicates that `sys(1,1,1,2)` is stable. This result is expected, because `sys(1,1,1,2)` has `a = -1`.

### See Also

`pole`

**Introduced in R2012a**

## issiso

Determine if dynamic system model is single-input/single-output (SISO)

### Syntax

```
issiso(sys)
```

### Description

`issiso(sys)` returns a logical value of 1 (`true`) if the dynamic system model `sys` is SISO and a logical value of 0 (`false`) otherwise.

### See Also

`size` | `isempty`

**Introduced before R2006a**

# isstatic

Determine if model is static or dynamic

## Syntax

```
B = isstatic(sys)
B = isstatic(sys, 'elem')
```

## Description

`B = isstatic(sys)` returns a logical value of 1 (`true`) if the model `sys` is a static model, and a logical value of 0 (`false`) if `sys` has dynamics, such as states or delays. If `sys` is a model array, then `B = 1` if all models in `sys` are static.

`B = isstatic(sys, 'elem')` checks each model in the model array `sys` and returns a logical array of the same size as `sys`. The logical array indicates which models in `sys` are static.

## Input Arguments

**sys** — Model or array to check

input-output model | model array

Model or array to check, specified as an input-output model or model array. Input-output models include dynamic system models such as numeric LTI models and generalized models. Input-output models also include static models such as tunable parameters or generalized matrices.

## Output Arguments

**B** — Flag indicating whether input model is static

logical | logical array

Flag indicating whether input model is static, returned as a logical value or logical array.



## More About

- “Types of Model Objects”

## See Also

hasdelay | pole | zero

**Introduced in R2013a**

## kalman

Kalman filter design, Kalman estimator

### Syntax

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

### Description

`kalman` designs a Kalman filter or Kalman state estimator given a state-space model of the plant and the process and measurement noise covariance data. The Kalman estimator provides the optimal solution to the following continuous or discrete estimation problems.

#### Continuous-Time Estimation

Given the continuous plant

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

with known inputs  $u$ , white process noise  $w$ , and white measurement noise  $v$  satisfying

$$E(w) = E(v) = 0, \quad E(ww^T) = Q, \quad E(vv^T) = R, \quad E(wv^T) = N$$

construct a state estimate  $\hat{x}(t)$  that minimizes the steady-state error covariance

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

The optimal solution is the Kalman filter with equations

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

$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} \hat{x} + \begin{bmatrix} D \\ 0 \end{bmatrix} u$$

The filter gain  $L$  is determined by solving an algebraic Riccati equation to be

$$L = (PC^T + \bar{N})\bar{R}^{-1}$$

where

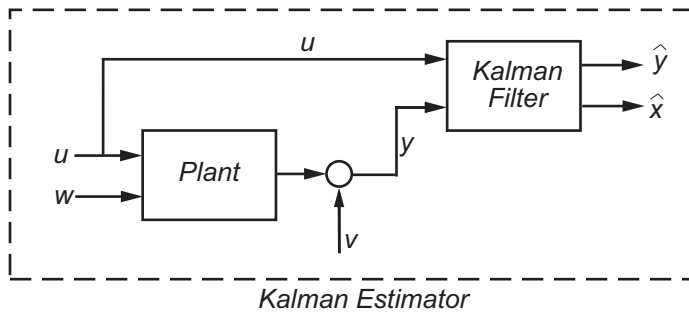
$$\bar{R} = R + HN + N^T H^T + HQH^T$$

$$\bar{N} = G(QH^T + N)$$

and  $P$  solves the corresponding algebraic Riccati equation.

The estimator uses the known inputs  $u$  and the measurements  $y$  to generate the output and state estimates  $\hat{y}$  and  $\hat{x}$ . Note that  $\hat{y}$  estimates the true plant output

$$y = Cx + Du + Hw + v$$



### Discrete-Time Estimation

Given the discrete plant

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

$$y[n] = Cx[n] + Du[n] + Hw[n] + v[n]$$

and the noise covariance data

$$E(w[n]u[n]^T) = Q, \quad E(v[n]b[n]^T) = R, \quad E(w[n]b[n]^T) = N$$

The estimator has the following state equation:

$$\hat{x}[n+1 | n] = A\hat{x}[n | n-1] + Bu[n] + L(y[n] - C\hat{x}[n | n-1] - Du[n])$$

The gain matrix  $L$  is derived by solving a discrete Riccati equation to be

$$L = (APC^T + \bar{N})(CPC^T + \bar{R})^{-1}$$

where

$$\begin{aligned} \bar{R} &= R + HN + N^T H^T + HQH^T \\ \bar{N} &= G(QH^T + N) \end{aligned}$$

There are two variants of discrete-time Kalman estimators:

- The current estimator generates output estimates  $\hat{y}[n | n]$  and state estimates  $\hat{x}[n | n]$  using all available measurements up to  $y[n]$ . This estimator has the output equation

$$\begin{bmatrix} \hat{y}[n | n] \\ \hat{x}[n | n] \end{bmatrix} = \begin{bmatrix} (I - M_y)C \\ I - M_x C \end{bmatrix} \hat{x}[n | n-1] + \begin{bmatrix} (I - M_y)D & M_y \\ -M_x D & M_x \end{bmatrix} \begin{bmatrix} u[n] \\ y[n] \end{bmatrix},$$

where the innovation gains  $M_x$  and  $M_y$  are defined as:

$$\begin{aligned} M_x &= PC^T (CPC^T + \bar{R})^{-1}, \\ M_y &= (CPC^T + HQH^T + HN)(CPC^T + \bar{R})^{-1}. \end{aligned}$$

$M_x$  updates the prediction  $\hat{x}[n | n-1]$  using the new measurement  $y[n]$ .

$$\hat{x}[n|n] = \hat{x}[n|n-1] + M_x \underbrace{(y[n] - C\hat{x}[n|n-1] - Du[n])}_{\text{innovation}}$$

When  $H = 0$ ,  $M_y = CM_x$  and  $y[n|n] = Cx[n|n] + Du[n]$ .

- The delayed estimator generates output estimates  $\hat{y}[n|n-1]$  and state estimates  $\hat{x}[n|n-1]$  using measurements only up to  $y_v[n-1]$ . This estimator is easier to implement inside control loops and has the output equation

$$\begin{bmatrix} \hat{y}[n|n-1] \\ \hat{x}[n|n-1] \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} \hat{x}[n|n-1] + \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u[n] \\ y[n] \end{bmatrix}$$

`[kest, L, P] = kalman(sys, Qn, Rn, Nn)` creates a state-space model `kest` of the Kalman estimator given the plant model `sys` and the noise covariance data `Qn`, `Rn`, `Nn` (matrices  $Q$ ,  $R$ ,  $N$  described in “Description” on page 2-488). `sys` must be a state-space model with matrices  $A, [B \ G], C, [D \ H]$ .

The resulting estimator `kest` has inputs  $[u; y]$  and outputs  $[\hat{y}; \hat{x}]$  (or their discrete-time counterparts). 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`. `P` solves the associated Riccati equation.

`[kest, L, P] = kalman(sys, Qn, Rn, Nn, sensors, known)` handles the more general situation when

- Not all outputs of `sys` are measured.
- The disturbance inputs  $w$  are not the last inputs of `sys`.

The index vectors `sensors` and `known` specify which outputs  $y$  of `sys` are measured and which inputs  $u$  are known (deterministic). All other inputs of `sys` are assumed stochastic.

`[kest, L, P, M, Z] = kalman(sys, Qn, Rn, ..., type)` specifies the estimator type for discrete-time plants `sys`. The `type` argument is either 'current' (default) or

'delayed'. For discrete-time plants, `kalman` returns the estimator and innovation gains  $L$  and  $M$  and the steady-state error covariances

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

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

## Examples

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

$$\bar{Q} = GQG^T$$

$$\bar{R} = R + HN + N^T H^T + HQH^T$$

$$\bar{N} = G(QH^T + N)$$

## References

- [1] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems*, Second Edition, Addison-Wesley, 1990.
- [2] Lewis, F., *Optimal Estimation*, John Wiley & Sons, Inc, 1986.

**See Also**

`care` | `dare` | `estim` | `extendedKalmanFilter` | `Kalman Filter` | `kalmd` | `lqg` | `lqgreg` | `ss` | `unscentedKalmanFilter`

**Introduced before R2006a**

## kalmd

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

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

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

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

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.

## 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.

## See Also

kalman | lqrd | lqgreg

**Introduced before R2006a**

## lft

Generalized feedback interconnection of two models (Redheffer star product)

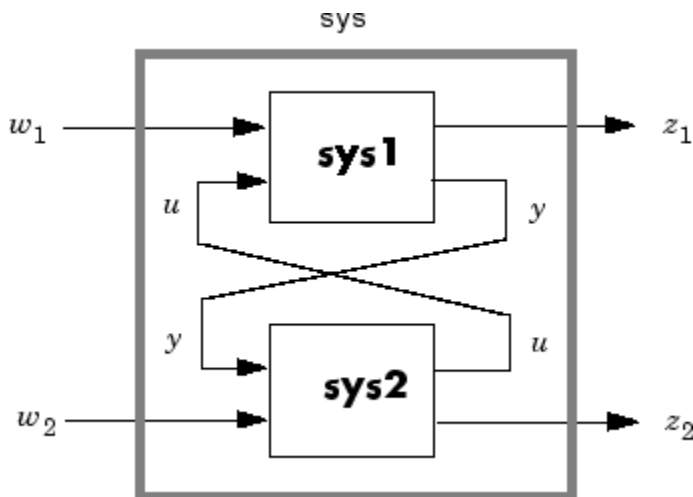
## Syntax

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

## Description

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

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



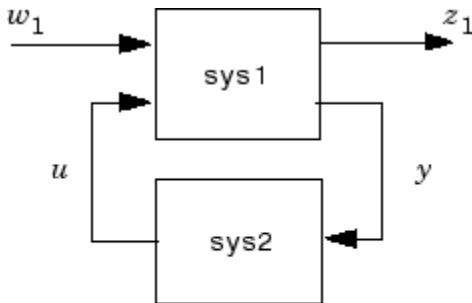
This feedback loop connects the first `nu` outputs of `sys2` to the last `nu` inputs of `sys1` (signals  $u$ ), and the last `ny` outputs of `sys1` to the first `ny` inputs of `sys2` (signals  $y$ ). The resulting system `sys` maps the input vector  $[w_1 ; w_2]$  to the output vector  $[z_1 ; z_2]$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

The abbreviated syntax

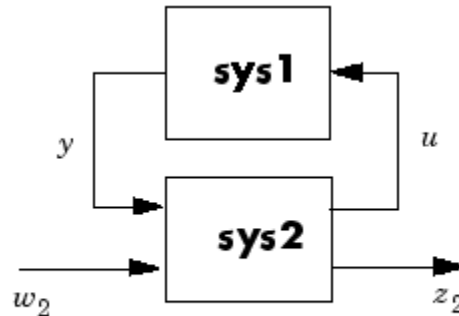
```
sys = lft(sys1,sys2)
```

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

## Limitations

There should be no algebraic loop in the feedback connection.

## More About

### Algorithms

The closed-loop model is derived by elementary state-space manipulations.

### See Also

[connect](#) | [feedback](#)

**Introduced before R2006a**

# Linear System Analyzer

Analyze time and frequency responses of linear time-invariant (LTI) systems

## Description

The **Linear System Analyzer** app lets you analyze time and frequency responses of LTI systems. Using this app, you can:

- View and compare the response plots of SISO and MIMO systems, or of several linear models at the same time.
- Generate time response plots such as step, impulse, and time response to arbitrary inputs.
- Generate frequency response plots such as Bode, Nyquist, Nichols, singular-value, and pole-zero plots.
- Inspect key response characteristics, such as rise time, maximum overshoot, and stability margins.

## Available Plots

**Linear System Analyzer** can generate the following response plots:

- Step response
- Impulse response
- Simulated time response to specified input signal
- Simulated time response from specified initial conditions (state-space models only)
- Bode diagram (magnitude and phase, or magnitude alone)
- Nyquist plot
- Nichols plot
- Singular value plot
- Pole/zero map and I/O pole/zero map

## Open the Linear System Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.

- MATLAB command prompt: Enter `linearSystemAnalyzer`.

## Examples

- “Linear Analysis Using the Linear System Analyzer”
- “Joint Time-Domain and Frequency-Domain Analysis”

## Programmatic Use

`linearSystemAnalyzer` opens the **Linear System Analyzer** app with no LTI systems to analyze. To specify a system to analyze, select **File > Import**.

`linearSystemAnalyzer(sys1,sys2,...,sysn)` opens **Linear System Analyzer** and displays the step response of one or more dynamic system models, `sys1`, `sys2`, ..., `sysn`. Such models include:

- Numeric LTI models such as `tf`, `zpk`, or `ss` models.
- Identified models such as `idtf`, `idss`, or `idproc` (requires System Identification Toolbox software).
- Generalized LTI models such as `genss` or `uss` models. For generalized LTI models without uncertainty, **Linear System Analyzer** plots the response of the nominal value of the model. For generalized models with uncertainty, the app plots the responses of 20 random samples of the uncertain system. (Uncertain models require Robust Control Toolbox software.)

`linearSystemAnalyzer(sys1,plotstyle1,sys2,plotstyle2,...,sysn,plotstylen)` specifies the line style, marker, and color of the line and marker of each response plot. Specify plot styles using one, two, or three characters. For example, the following code uses red asterisks for the response of `sys1`, and a magenta dotted line for the response of `sys2`.

```
linearSystemAnalyzer(sys1, 'r-*', sys2, 'm--');
```

For more information about configuring the `PlotStyle` argument, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

`linearSystemAnalyzer(plottype, ___)` opens **Linear System Analyzer** and displays the response types specified by `plottype`. You can use this syntax with any of

the previous input argument combinations. The `plottype` argument can be any one of the following:

- `'step'` — Step response.
- `'impulse'` — Impulse response.
- `'lsim'` — Linear simulation plot. When you use this plot type, the Linear Simulation Tool dialog box prompts you to specify an input signal for the simulation.
- `'initial'` — Initial condition plot (state-space models only). You can use the `extras` argument to specify the initial state. If you do not, the Linear Simulation Tool dialog box opens and prompts you to specify an initial state for the simulation.
- `'bode'` — Bode diagram.
- `'bodemag'` — Bode magnitude diagram.
- `'nyquist'` — Nyquist plot.
- `'nichols'` — Nichols plot.
- `'sigma'` — Singular value plot. (See `sigma`).
- `'pzmap'` — Pole/zero map.
- `'iopzmap'` — Pole/zero map of each input/output pair of the LTI system.

To open **Linear System Analyzer** with multiple response plots, use a cell array of up to six of these plot types for the `plottype` input argument. For example, the following command opens the app with a step response plot and a Nyquist plot for the system `sys`.

```
linearSystemAnalyzer({'step';'nyquist'},sys)
```

`linearSystemAnalyzer(plottype,sys1,sys2,...,sysn,extras)` specifies additional input arguments specific to the type of response plot. `extras` can be one or more of the input arguments available for the function corresponding to the plot type. For example, suppose `plottype` is `'step'`. Then, `extras` enables you to use the additional arguments that you could use with the `step` command, such as the desired final time, `Tfinal`. Thus, the following command opens the app with a step response plot of `sys`, with a final time of `Tfinal`.

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

If `plottype` is `'initial'`, you can use `extras` to supply the initial conditions `x0`, and other arguments such as `Tfinal`. For example:

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

To determine appropriate arguments for `extras`, see the reference pages of the functions corresponding to each plot type, such as `step`, `bode`, or `initial`.

`h = linearSystemAnalyzer( ___ )` returns a handle to the **Linear System Analyzer** figure. You can use this syntax with any of the previous combinations of input arguments. Use the handle to modify previously opened **Linear System Analyzer** instances, as described in the next two syntaxes.

`linearSystemAnalyzer( 'clear', h )` clears the plots and data from the **Linear System Analyzer** corresponding to handle `h`. To clear multiple app instances at once, set `h` to a vector of handles.

`linearSystemAnalyzer( 'current', sys1, sys2, ..., sysn, h )` adds the responses of the systems `sys1`, `sys2`, ..., `sysn` to the **Linear System Analyzer** corresponding to handle `h`. To update multiple app instances at once, set `h` to a vector of handles. If the new systems have different I/O dimensions from the currently displayed systems, the app clears the existing responses and displays only the new ones.

## See Also

### Apps

Control System Designer

### Functions

`bode` | `bodemag` | `impulse` | `initial` | `iopzmap` | `lsim` | `nichols` | `nyquist` | `pzmap` | `sigma` | `step`

**Introduced in R2015a**



# lqg

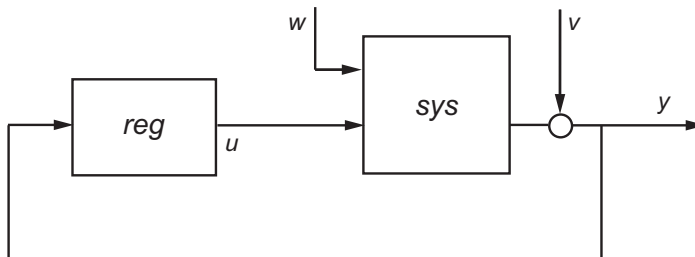
Linear-Quadratic-Gaussian (LQG) design

## Syntax

```
reg = lqg(sys, QXU, QWV)
reg = lqg(sys, QXU, QWV, QI)
reg = lqg(sys, QXU, QWV, QI, '1dof')
reg = lqg(sys, QXU, QWV, QI, '2dof')
```

## Description

`reg = lqg(sys, QXU, QWV)` computes an optimal linear-quadratic-Gaussian (LQG) regulator `reg` given a state-space model `sys` of the plant and weighting matrices `QXU` and `QWV`. The dynamic regulator `reg` uses the measurements `y` to generate a control signal `u` that regulates `y` around the zero value. Use positive feedback to connect this regulator to the plant output `y`.



The LQG regulator minimizes the cost function

$$J = E \left\{ \lim_{\tau \rightarrow \infty} \frac{1}{\tau} \int_0^{\tau} \begin{bmatrix} x^T & u^T \end{bmatrix} Q_{xu} \begin{bmatrix} x \\ u \end{bmatrix} dt \right\}$$

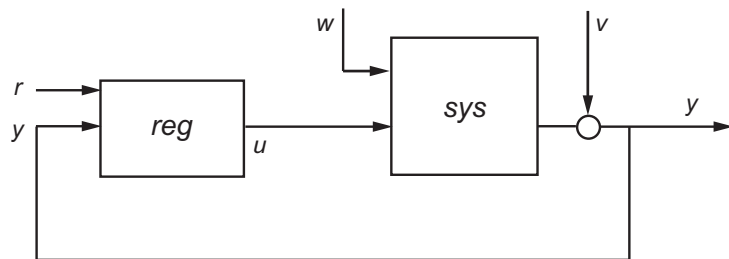
subject to the plant equations

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

where the process noise  $w$  and measurement noise  $v$  are Gaussian white noises with covariance:

$$E([w;v] * [w',v']) = QWV$$

`reg = lqg(sys, QXU, QWV, QI)` uses the setpoint command  $r$  and measurements  $y$  to generate the control signal  $u$ . `reg` has integral action to ensure that  $y$  tracks the command  $r$ .



The LQG servo-controller minimizes the cost function

$$J = E \left\{ \lim_{\tau \rightarrow \infty} \frac{1}{\tau} \int_0^{\tau} \left( \begin{bmatrix} x^T & u^T \end{bmatrix} Q_{xu} \begin{bmatrix} x \\ u \end{bmatrix} + x_i^T Q_i x_i \right) dt \right\}$$

where  $x_i$  is the integral of the tracking error  $r - y$ . For MIMO systems,  $r$ ,  $y$ , and  $x_i$  must have the same length.

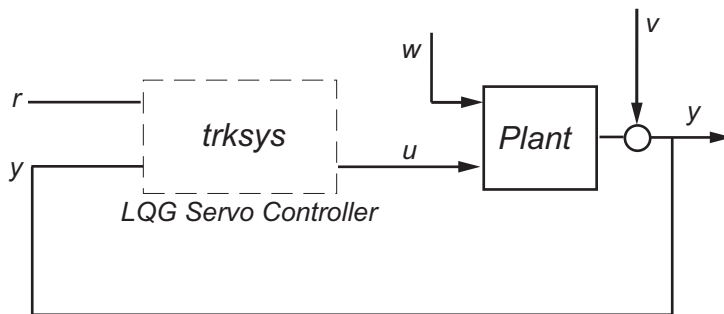
`reg = lqg(sys, QXU, QWV, QI, '1dof')` computes a one-degree-of-freedom servo controller that takes  $e = r - y$  rather than  $[r; y]$  as input.

`reg = lqg(sys, QXU, QWV, QI, '2dof')` is equivalent to `LQG(sys, QXU, QWV, QI)` and produces the two-degree-of-freedom servo-controller shown previously.

## Examples

### Linear-Quadratic-Gaussian (LQG) Regulator and Servo Controller Design

This example shows how to design an linear-quadratic-Gaussian (LQG) regulator, a one-degree-of-freedom LQG servo controller, and a two-degree-of-freedom LQG servo controller for the following system.



The plant has three states ( $x$ ), two control inputs ( $u$ ), three random inputs ( $w$ ), one output ( $y$ ), measurement noise for the output ( $v$ ), and the following state and measurement equations.

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

where

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0.3 & 1 \\ 0 & 1 \\ -0.3 & 0.9 \end{bmatrix}$$

$$C = [1.9 \quad 1.3 \quad 1] \quad D = [0.53 \quad -0.61]$$

The system has the following noise covariance data:

$$Q_n = E(\omega\omega^T) = \begin{bmatrix} 4 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_n = E(vv^T) = 0.7$$

For the regulator, use the following cost function to define the tradeoff between regulation performance and control effort:

$$J(u) = \int_0^{\infty} \left( 0.1x^T x + u^T \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} u \right) dt$$

For the servo controllers, use the following cost function to define the tradeoff between tracker performance and control effort:

$$J(u) = \int_0^{\infty} \left( 0.1x^T x + x_i^2 + u^T \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} u \right) dt$$

To design the LQG controllers for this system:

- 1 Create the state-space system by typing the following in the MATLAB Command Window:

```
A = [0 1 0; 0 0 1; 1 0 0];
B = [0.3 1; 0 1; -0.3 0.9];
C = [1.9 1.3 1];
D = [0.53 -0.61];
sys = ss(A,B,C,D);
```

- 2 Define the noise covariance data and the weighting matrices by typing the following commands:

```
nx = 3; %Number of states
ny = 1; %Number of outputs
Qn = [4 2 0; 2 1 0; 0 0 1];
Rn = 0.7;
R = [1 0; 0 2]
QXU = blkdiag(0.1*eye(nx),R);
QWV = blkdiag(Qn,Rn);
QI = eye(ny);
```

- 3 Form the LQG regulator by typing the following command:

```
KLQG = lqg(sys,QXU,QWV)
```

This command returns the following LQG regulator:

```
a =
      x1_e      x2_e      x3_e
x1_e -6.212 -3.814 -4.136
x2_e -4.038 -3.196 -1.791
x3_e -1.418 -1.973 -1.766
```

```

b =
      y1
x1_e  2.365
x2_e  1.432
x3_e  0.7684

c =
      x1_e      x2_e      x3_e
u1  -0.02904  0.0008272  0.0303
u2  -0.7147   -0.7115   -0.7132

d =
      y1
u1  0
u2  0

```

```

Input groups:
      Name      Channels
Measurement    1

```

```

Output groups:
      Name      Channels
Controls      1,2

```

Continuous-time model.

- 4** Form the one-degree-of-freedom LQG servo controller by typing the following command:

```
KLQG1 = lqg(sys,QXU,QWV,QI,'1dof')
```

This command returns the following LQG servo controller:

```

a =
      x1_e      x2_e      x3_e      xi1
x1_e  -7.626   -5.068   -4.891   0.9018
x2_e  -5.108   -4.146   -2.362   0.6762
x3_e  -2.121   -2.604   -2.141   0.4088
xi1    0         0         0         0

b =
      e1
x1_e  -2.365
x2_e  -1.432
x3_e  -0.7684
xi1    1

```

```
c =
      x1_e    x2_e    x3_e    xi1
u1 -0.5388 -0.4173 -0.2481  0.5578
u2 -1.492  -1.388  -1.131  0.5869
```

```
d =
      e1
u1    0
u2    0
```

```
Input groups:
      Name    Channels
Error        1
```

```
Output groups:
      Name    Channels
Controls    1,2
```

Continuous-time model.

- 5 Form the two-degree-of-freedom LQG servo controller by typing the following command:

```
KLQG2 = lqg(sys,QXU,QWV,QI,'2dof')
```

This command returns the following LQG servo controller:

```
a =
      x1_e    x2_e    x3_e    xi1
x1_e -7.626  -5.068  -4.891  0.9018
x2_e -5.108  -4.146  -2.362  0.6762
x3_e -2.121  -2.604  -2.141  0.4088
xi1   0        0        0        0
```

```
b =
      r1    y1
x1_e    0    2.365
x2_e    0    1.432
x3_e    0    0.7684
xi1     1    -1
```

```
c =
      x1_e    x2_e    x3_e    xi1
u1 -0.5388 -0.4173 -0.2481  0.5578
u2 -1.492  -1.388  -1.131  0.5869
```

```

d =
      r1  y1
u1   0   0
u2   0   0

Input groups:
      Name      Channels
Setpoint      1
Measurement   2

Output groups:
      Name      Channels
Controls      1,2

Continuous-time model.

```

## More About

### Tips

`lqg` can be used for both continuous- and discrete-time plants. In discrete-time, `lqg` uses  $x[n|n-1]$  as state estimate (see `kalman` for details).

To compute the LQG regulator, `lqg` uses the commands `lqr` and `kalman`. To compute the servo-controller, `lqg` uses the commands `lqi` and `kalman`.

When you want more flexibility for designing regulators you can use the `lqr`, `kalman`, and `lqgreg` commands. When you want more flexibility for designing servo controllers, you can use the `lqi`, `kalman`, and `lqgtrack` commands. For more information on using these commands and how to decide when to use them, see “Linear-Quadratic-Gaussian (LQG) Design for Regulation” and “Linear-Quadratic-Gaussian (LQG) Design of Servo Controller with Integral Action”.

### See Also

`care` | `dare` | `kalman` | `lqi` | `lqr` | `lqry` | `ss`

**Introduced before R2006a**

## lqgreg

Form linear-quadratic-Gaussian (LQG) regulator

### Syntax

```
rlqg = lqgreg(kest,k)
rlqg = lqgreg(kest,k,controls)
```

### Description

`lqgreg` forms the linear-quadratic-Gaussian (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.

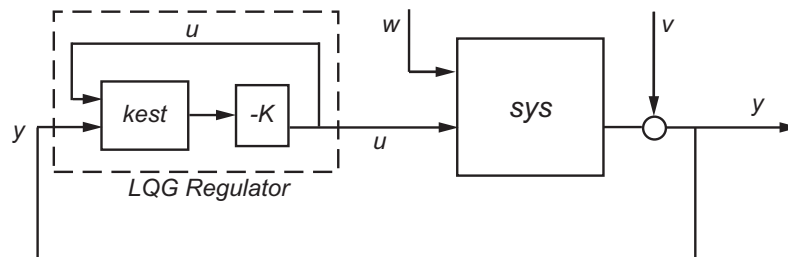
In continuous time, the LQG regulator generates the commands

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

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

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

where  $y$  is the vector of plant output measurements (see `kalman` for background and notation). The following diagram shows this dynamic regulator in relation to the plant.





In discrete time, you can form the LQG regulator using either the delayed state estimate  $\hat{x}[n | n-1]$  of  $x[n]$ , based on measurements up to  $y[n-1]$ , or the current state estimate  $\hat{x}[n | n]$ , based on all available measurements including  $y[n]$ . 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  $u[n]$  after the measurements  $y[n]$  become available (this amounts to a time delay in the feedback loop).

## Examples

See the example LQG Regulation.

## More About

### Tips

`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

- $u[n] = -K\hat{x}[n | n]$  when `kest` is the “current” Kalman estimator

- $u[n] = -K\hat{x}[n | n - 1]$  when `kest` is the “delayed” Kalman estimator

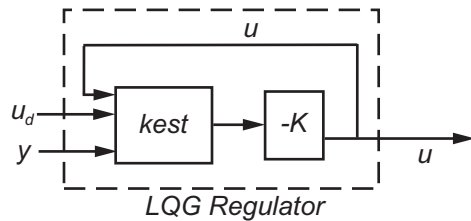
For more information on Kalman estimators, see the `kalman` reference page.

`rlqg = lqgreg(kest, k, controls)` handles estimators that have access to additional deterministic 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  $y$  as inputs (see the next figure).

---

**Note** Always use *positive* feedback to connect the LQG regulator to the plant.

---



## See Also

`kalman` | `kalmd` | `lqr` | `dlqr` | `lqrd` | `lqry` | `reg`

**Introduced before R2006a**

# lqgtrack

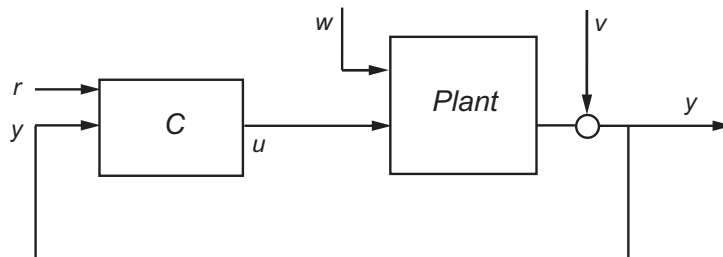
Form Linear-Quadratic-Gaussian (LQG) servo controller

## Syntax

```
C = lqgtrack(kest,k)
C = lqgtrack(kest,k,'2dof')
C = lqgtrack(kest,k,'1dof')
C = lqgtrack(kest,k,...CONTROLS)
```

## Description

`lqgtrack` forms a Linear-Quadratic-Gaussian (LQG) servo controller with integral action for the loop shown in the following figure. This compensator ensures that the output  $y$  tracks the reference command  $r$  and rejects process disturbances  $w$  and measurement noise  $v$ . `lqgtrack` assumes that  $r$  and  $y$  have the same length.

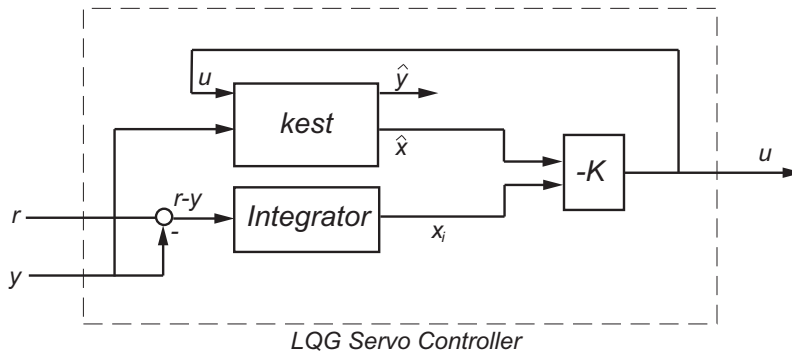



---

**Note:** Always use positive feedback to connect the LQG servo controller  $C$  to the plant output  $y$ .

---

`C = lqgtrack(kest,k)` forms a two-degree-of-freedom LQG servo controller  $C$  by connecting the Kalman estimator `kest` and the state-feedback gain `k`, as shown in the following figure.  $C$  has inputs  $[r;y]$  and generates the command  $u = -K[\hat{x};x_i]$ , where  $\hat{x}$  is the Kalman estimate of the plant state, and  $x_i$  is the integrator output.



The size of the gain matrix  $k$  determines the length of  $x_i$ .  $x_i$ ,  $y$ , and  $r$  all have the same length.

The two-degree-of-freedom LQG servo controller state-space equations are

$$\begin{bmatrix} \dot{\hat{x}} \\ \dot{x}_i \end{bmatrix} = \begin{bmatrix} A - BK_x - LC + LDK_x & -BK_i + LDK_i \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix} + \begin{bmatrix} 0 & L \\ I & -I \end{bmatrix} \begin{bmatrix} r \\ y \end{bmatrix}$$

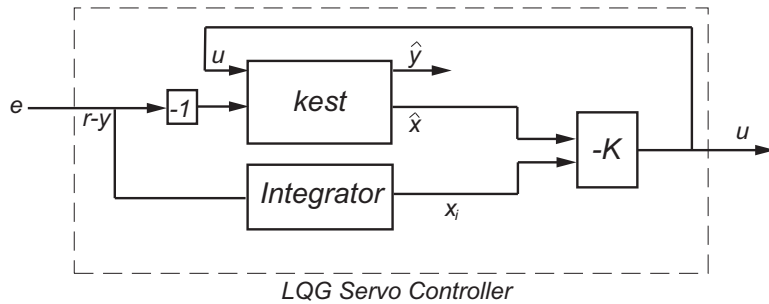
$$u = \begin{bmatrix} -K_x & -K_i \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix}$$

---

**Note:** The syntax `C = lqgtrack(kest,k,'2dof')` is equivalent to `C = lqgtrack(kest,k)`.

---

`C = lqgtrack(kest,k,'1dof')` forms a one-degree-of-freedom LQG servo controller `C` that takes the tracking error  $e = r - y$  as input instead of  $[r ; y]$ , as shown in the following figure.



The one-degree-of-freedom LQG servo controller state-space equations are

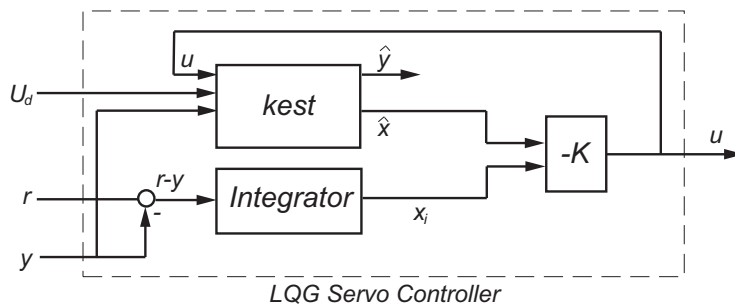
$$\begin{bmatrix} \dot{\hat{x}} \\ \dot{x}_i \end{bmatrix} = \begin{bmatrix} A - BK_x - LC + LDK_x & -BK_i + LDK_i \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix} + \begin{bmatrix} -L \\ I \end{bmatrix} e$$

$$u = \begin{bmatrix} -K_x & -K_i \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix}$$

`C = lqgtrack(kest, k, ...CONTROLS)` forms an LQG servo controller `C` when the Kalman estimator `kest` has access to additional known (deterministic) commands  $U_d$  of the plant. In the index vector `CONTROLS`, specify which inputs of `kest` are the control channels  $u$ . The resulting compensator  $C$  has inputs

- $[U_d ; r ; y]$  in the two-degree-of-freedom case
- $[U_d ; e]$  in the one-degree-of-freedom case

The corresponding compensator structure for the two-degree-of-freedom cases appears in the following figure.



### Examples

See the example “Design an LQG Servo Controller”.

### More About

#### Tips

You can use `lqgtrack` for both continuous- and discrete-time systems.

In discrete-time systems, integrators are based on forward Euler (see `lqi` for details). The state estimate  $\hat{x}$  is either  $x[n | n]$  or  $x[n | n-1]$ , depending on the type of estimator (see `kalman` for details).

#### See Also

`lqg` | `lqi` | `kalman` | `lqr` | `lqgreg`

**Introduced in R2008b**

# lqi

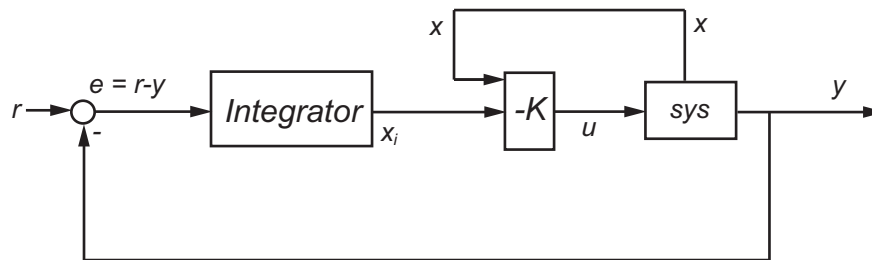
Linear-Quadratic-Integral control

## Syntax

`[K,S,e] = lqi(SYS,Q,R,N)`

## Description

`lqi` computes an optimal state-feedback control law for the tracking loop shown in the following figure.



For a plant `sys` with the state-space equations (or their discrete counterpart):

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

the state-feedback control is of the form

$$u = -K[x; x_i]$$

where  $x_i$  is the integrator output. This control law ensures that the output  $y$  tracks the reference command  $r$ . For MIMO systems, the number of integrators equals the dimension of the output  $y$ .

`[K, S, e] = lqi(SYS, Q, R, N)` calculates the optimal gain matrix  $K$ , given a state-space model  $SYS$  for the plant and weighting matrices  $Q$ ,  $R$ ,  $N$ . The control law  $u = -Kz = -K[x; x_i]$  minimizes the following cost functions (for  $r = 0$ )

- $J(u) = \int_0^{\infty} \{z^T Q z + u^T R u + 2z^T N u\} dt$  for continuous time

- $J(u) = \sum_{n=0}^{\infty} \{z^T Q z + u^T R u + 2z^T N u\}$  for discrete time

In discrete time, `lqi` computes the integrator output  $x_i$  using the forward Euler formula

$$x_i[n+1] = x_i[n] + Ts(r[n] - y[n])$$

where  $Ts$  is the sample time of  $SYS$ .

When you omit the matrix  $N$ ,  $N$  is set to 0. `lqi` also returns the solution  $S$  of the associated algebraic Riccati equation and the closed-loop eigenvalues  $e$ .

## Limitations

For the following state-space system with a plant with augmented integrator:

$$\begin{aligned} \frac{\delta z}{\delta t} &= A_a z + B_a u \\ y &= C_a z + D_a u \end{aligned}$$

The problem data must satisfy:

- The pair  $(A_a, B_a)$  is stabilizable.
- $R > 0$  and  $Q - NR^{-1}N^T \geq 0$ .
- $(Q - NR^{-1}N^T, A_a - B_a R^{-1}N^T)$  has no unobservable mode on the imaginary axis (or unit circle in discrete time).



## More About

### Tips

`lqi` supports descriptor models with nonsingular  $E$ . The output `S` of `lqi` is the solution of the Riccati equation for the equivalent explicit state-space model

$$\frac{dx}{dt} = E^{-1}Ax + E^{-1}Bu$$

## References

- [1] P. C. Young and J. C. Willems, “An approach to the linear multivariable servomechanism problem”, *International Journal of Control*, Volume 15, Issue 5, May 1972 , pages 961–979.

### See Also

`lqr` | `lqgreg` | `lqg` | `care` | `dare` | `lqgtrack`

Introduced in R2008b

## lqr

Linear-Quadratic Regulator (LQR) design

### Syntax

$$[K, S, e] = \text{lqr}(\text{SYS}, Q, R, N)$$

$$[K, S, e] = \text{LQR}(A, B, Q, R, N)$$

### Description

$[K, S, e] = \text{lqr}(\text{SYS}, Q, R, N)$  calculates the optimal gain matrix  $K$ .

For a continuous time system, the state-feedback law  $u = -Kx$  minimizes the quadratic cost function

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

subject to the system dynamics

$$\dot{x} = Ax + Bu.$$

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  $e = \text{eig}(A - B \cdot K)$ .  $K$  is derived from  $S$  using

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

For a discrete-time state-space model,  $u[n] = -Kx[n]$  minimizes

$$J = \sum_{n=0}^{\infty} \{x^T Q x + u^T R u + 2x^T N u\}$$

subject to  $x[n + 1] = Ax[n] + Bu[n]$ .

`[K, S, e] = LQR(A, B, Q, R, N)` is an equivalent syntax for continuous-time models with dynamics  $\dot{x} = Ax + Bu$ .

In all cases, when you omit the matrix N, N is set to 0.

## 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 (or unit circle in discrete time).

## More About

### Tips

`lqr` supports descriptor models with nonsingular  $E$ . The output `S` of `lqr` is the solution of the Riccati equation for the equivalent explicit state-space model:

$$\frac{dx}{dt} = E^{-1}Ax + E^{-1}Bu$$

### See Also

`care` | `dlqr` | `lqgreg` | `lqrd` | `lqry` | `lqi`

Introduced before R2006a

## lqrd

Design discrete linear-quadratic (LQ) regulator for continuous plant

### Syntax

```
lqrd
[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.

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

## Limitations

The discretized problem data should meet the requirements for dlqr.

## More About

### Algorithms

The equivalent discrete gain matrix  $K_d$  is determined by discretizing the continuous plant and weighting matrices using the sample time  $T_s$  and the zero-order hold approximation.

With the notation

$$\begin{aligned} \Phi(\tau) &= e^{A\tau}, & A_d &= \Phi(T_s) \\ \Gamma(\tau) &= \int_0^{\tau} e^{A\eta} B d\eta, & B_d &= \Gamma(T_s) \end{aligned}$$

the discretized plant has equations

$$x[n+1] = A_d x[n] + B_d u[n]$$

and the weighting matrices for the equivalent discrete cost function are

$$\begin{bmatrix} Q_d & N_d \\ N_d^T & R_d \end{bmatrix} = \int_0^{T_s} \begin{bmatrix} \Phi^T(\tau) & 0 \\ \Gamma^T(\tau) & I \end{bmatrix} \begin{bmatrix} Q & N \\ N^T & R \end{bmatrix} \begin{bmatrix} \Phi(\tau) & \Gamma(\tau) \\ 0 & I \end{bmatrix} d\tau$$

The integrals are computed using matrix exponential formulas due to Van Loan (see [2]). The plant is discretized using c2d and the gain matrix is computed from the discretized data using dlqr.

## 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-23, June 1978.

## See Also

c2d | dlqr | kalmd | lqr

**Introduced before R2006a**

# lqry

Form linear-quadratic (LQ) state-feedback regulator with output weighting

## Syntax

`[K,S,e] = lqry(sys,Q,R,N)`

## Description

Given the plant

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

or its discrete-time counterpart, `lqry` designs a state-feedback control

$$u = -Kx$$

that minimizes the quadratic cost function with output weighting

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

(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.

## Examples

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` | `dlqr` | `kalman` | `lqgreg`

**Introduced before R2006a**



# lsim

Simulate time response of dynamic system to arbitrary inputs

## Syntax

```
lsim(sys,u,t)
lsim(sys,u,t,x0)
lsim(sys,u,t,x0,method)
lsim(sys1,...,sysn,u,t)
lsim(sys1,PlotStyle1,...,sysN,PlotStyleN,u,t)
y = lsim(____)
[y,t,x] = lsim(____)
lsim(sys)
```

## 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 dynamic system model `sys` to the input history, `t,u`. The vector `t` specifies the time samples for the simulation (in system time units, specified in the `TimeUnit` property of `sys`), and consists of regularly spaced time samples:

```
t = 0:dt:Tfinal
```

The input `u` is an array having as many rows as time samples (`length(t)`) and as many columns as system inputs. For instance, if `sys` is a SISO system, then `u` is a `t`-by-1 vector. If `sys` has three inputs, then `u` is a `t`-by-3 array. Each row `u(i,:)` specifies the input value(s) at the time sample `t(i)`. The signal `u` also appears on the plot.

The model `sys` can be continuous or discrete, SISO or MIMO. In discrete time, `u` must be sampled at the same rate as the system. In this case, the input `t` is redundant and can be omitted or set to an 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 “Algorithms” on page 2-531 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 when `sys` is a state-space model. `x0` is a vector whose entries are the initial values of the corresponding states of `sys`.

`lsim(sys,u,t,x0,method)` explicitly specifies how the input values should be interpolated between samples, when `sys` is a continuous-time system. Specify `method` as one of the following values:

- 'zoh' — Use zero-order hold
- 'foh' — Use linear interpolation (first-order hold)

If you do not specify a method, `lsim` selects the interpolation method automatically based on the smoothness of the signal `u`.

`lsim(sys1,...,sysn,u,t)` simulates the responses of several dynamic system models to the same input history `t,u` and plots these responses on a single figure. You can also use the `x0` and `method` input arguments when computing the responses of multiple models.

`lsim(sys1,PlotStyle1,...,sysN,PlotStyleN,u,t)` specifies the line style, marker, and color of each of the system responses in the plot. You can also use the `x0` and `method` input arguments with this syntax. Each `PlotStyle` argument is specified as a vector of one, two, or three characters. The characters can appear in any order. For example, the following code plots the response of `sys1` as a yellow dotted line and the response of `sys2` as a green dashed line:

```
lsim(sys1,'y:',sys2,'g--',u,t,x0)
```

For more information about configuring the `PlotStyle` argument, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

`y = lsim(____)` returns the system response `y`, sampled at the same times as the input (`t`). The output `y` is an array having as many rows as time samples (`length(t)`) and as many columns as system outputs. No plot is drawn on the screen. You can use this syntax with any of the input arguments described in previous syntaxes except the `PlotStyle` arguments.

`[y,t,x] = lsim( ___ )` also returns the time vector `t` used for simulation and the state trajectories `x` (for state-space models only). The output `x` has as many rows as time samples (`length(t)`) and as many columns as system states. You can use this syntax with any of the input arguments described in previous syntaxes except the `PlotStyle` arguments.

`lsim(sys)` opens the Linear Simulation Tool GUI. For more information about working with this GUI, see [Working with the Linear Simulation Tool](#).

## Examples

### Simulate Response to Square Wave

Simulate and plot the response of the following system to a square wave with period of four seconds:

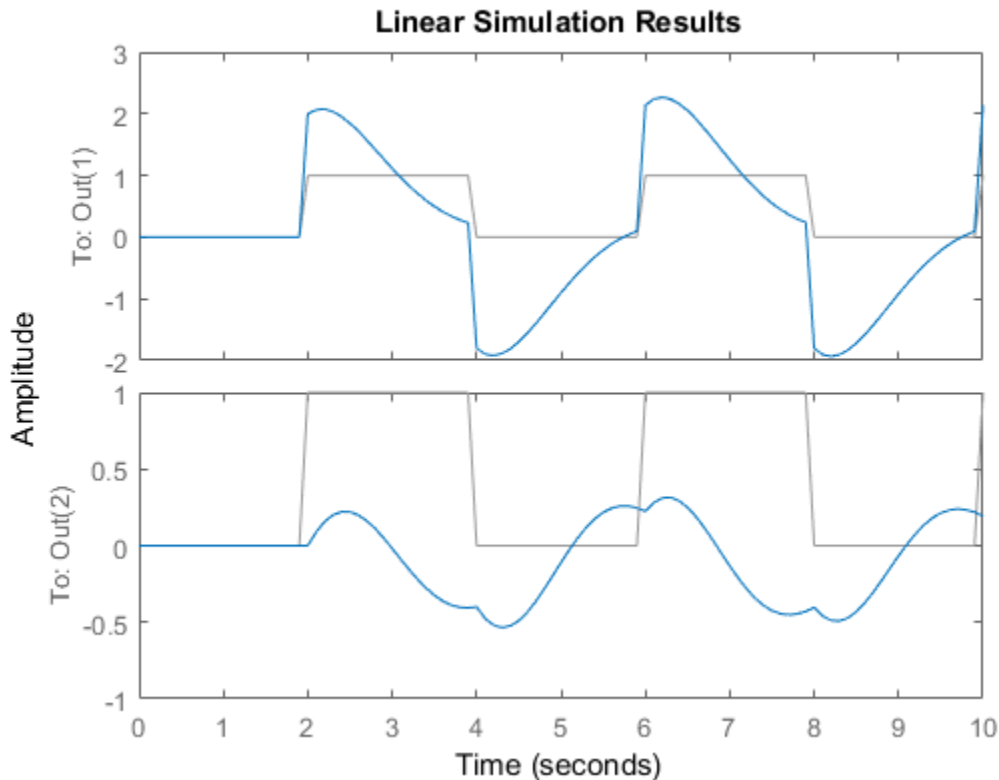
$$H(s) = \begin{bmatrix} \frac{2s^2 + 5s + 1}{s^2 + 2s + 3} \\ \frac{s - 1}{s^2 + s + 5} \end{bmatrix}.$$

Create the transfer function, and generate the square wave with `gensig`. Sample every 0.1 second during 10 seconds.

```
H = [tf([2 5 1],[1 2 3]);tf([1 -1],[1 1 5])];
[u,t] = gensig('square',4,10,0.1);
```

Then simulate with `lsim`.

```
lsim(H,u,t)
```



The plot displays both the applied signal and the response.

## Simulate Response of Identified Model

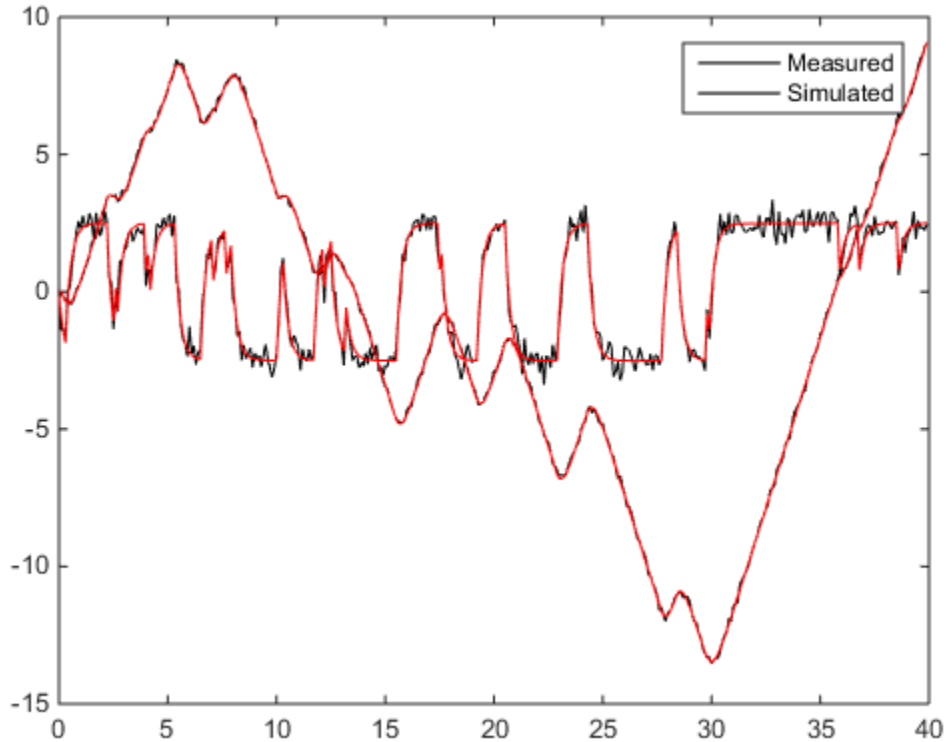
Simulate the response of an identified linear model using the same input signal as the one used for estimation and the initial states returned by the estimation command.

```
load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotordata'));
z = iddata(y,u,0.1, 'Name', 'DC-motor');
```

```
[sys,x0] = n4sid(z,4);
[y,t,x] = lsim(0.1, sys, z.InputData, [], x0);
```

Compare the simulated response `y` to measured response `z.OutputData`.

```
plot(t,z.OutputData,'k',t,y,'r')  
legend('Measured','Simulated')
```



## More About

### Algorithms

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 sample time is set to the spacing  $\Delta t$  between the user-supplied time samples  $\mathbf{t}$ .

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

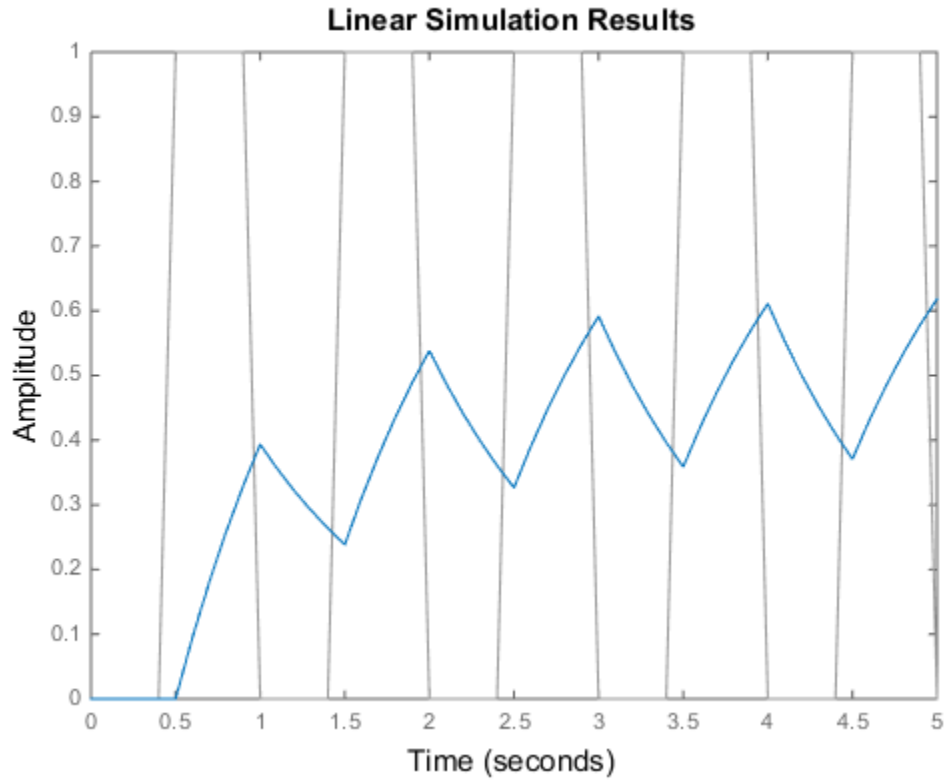
$$H(s) = \frac{\omega^2}{s^2 + 2s + \omega^2}, \quad \omega = 62.83$$

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

```
w2 = 62.83^2;
h = tf(w2,[1 2 w2]);
t = 0:0.1:5;           % vector of time samples
u = (rem(t,1) >= 0.5); % square wave values
lsim(h,u,t)
```

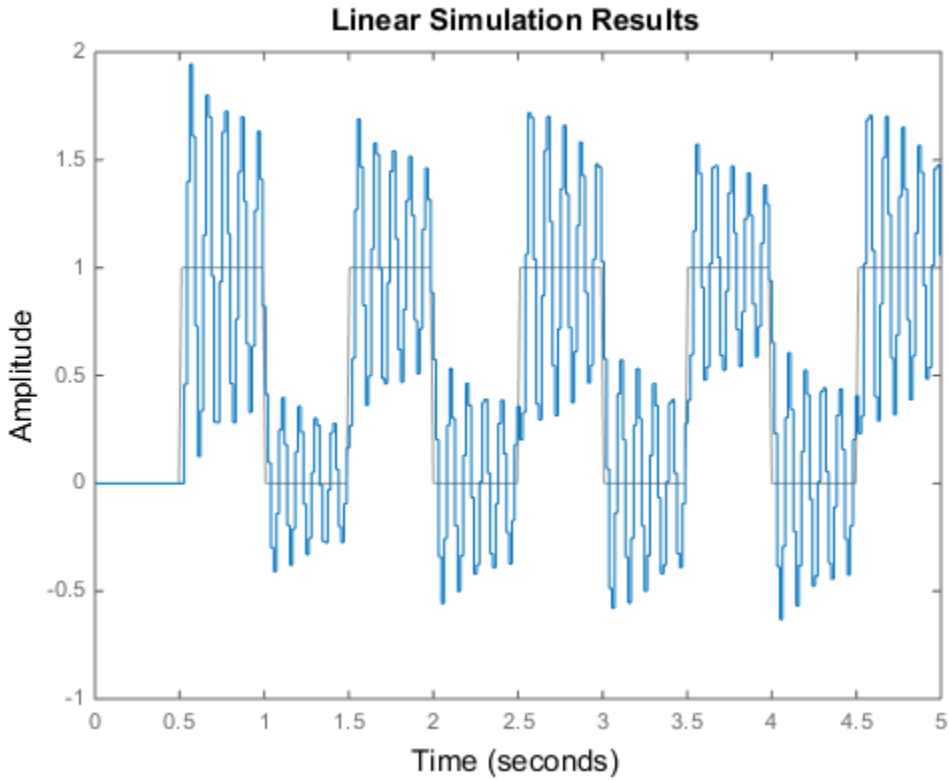
`lsim` evaluates the specified sample time, and issues a warning:

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



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

```
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 that is hidden in the undersampled version.

### See Also

[gensig](#) | [impulse](#) | [initial](#) | [Linear System Analyzer](#) | [lsiminfo](#) | [step](#)

**Introduced before R2006a**



# lsiminfo

Compute linear response characteristics

## Syntax

```
S = lsiminfo(y,t,yfinal)
S = lsiminfo(y,t)
S = lsiminfo(...,'SettlingTimeThreshold',ST)
```

## Description

`S = lsiminfo(y,t,yfinal)` takes the response data  $(t,y)$  and a steady-state value `yfinal` and returns a structure `S` containing the following performance indicators:

- `SettlingTime` — Settling time
- `Min` — Minimum value of `Y`
- `MinTime` — Time at which the min value is reached
- `Max` — Maximum value of `Y`
- `MaxTime` — Time at which the max value is reached

For SISO responses, `t` and `y` are vectors with the same length `NS`. For responses with `NY` outputs, you can specify `y` as an `NS`-by-`NY` array and `yfinal` as a `NY`-by-1 array. `lsiminfo` then returns an `NY`-by-1 structure array `S` of performance metrics for each output channel.

`S = lsiminfo(y,t)` uses the last sample value of `y` as steady-state value `yfinal`. `s = lsiminfo(y)` assumes `t = 1:NS`.

`S = lsiminfo(...,'SettlingTimeThreshold',ST)` lets you specify the threshold `ST` used in the settling time calculation. The response has settled when the error  $|y(t) - y_{final}|$  becomes smaller than a fraction `ST` of its peak value. The default value is `ST=0.02` (2%).

## Examples

Create a fourth order transfer function and ascertain the response characteristics.

```
sys = tf([1 -1],[1 2 3 4]);  
[y,t] = impulse(sys);  
s = lsiminfo(y,t,0) % final value is 0  
s =
```

```
    SettlingTime: 22.8626  
             Min: -0.4270  
    MinTime: 2.0309  
             Max: 0.2845  
    MaxTime: 4.0619
```

### See Also

[impulse](#) | [stepinfo](#) | [lsim](#) | [initial](#)

**Introduced in R2006a**

# lsimplot

Simulate response of dynamic system to arbitrary inputs and return plot handle

## Syntax

```
h = lsimplot(sys)
lsimplot(sys1,sys2,...)
lsimplot(sys,u,t)
lsimplot(sys,u,t,x0)
lsimplot(sys1,sys2,...,u,t,x0)
lsimplot(AX,...)
lsimplot(..., plotoptions)
lsimplot(sys,u,t,x0,'zoh')
lsimplot(sys,u,t,x0,'foh')
```

## Description

`h = lsimplot(sys)` opens the Linear Simulation Tool for the dynamic system model `sys`, which enables interactive specification of driving input(s), the time vector, and initial state. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

`lsimplot(sys1,sys2,...)` opens the Linear Simulation Tool for multiple models `sys1,sys2,...`. Driving inputs are common to all specified systems but initial conditions can be specified separately for each.

`lsimplot(sys,u,t)` plots the time response of the model `sys` to the input signal described by `u` and `t`. The time vector `t` consists of regularly spaced time samples (in system time units, specified in the `TimeUnit` property of `sys`). For MIMO systems, `u` is a matrix with as many columns as inputs and whose `i`th row specifies the input value at time `t(i)`. For SISO systems `u` can be specified either as a row or column vector. For example,

```
t = 0:0.01:5;
u = sin(t);
lsimplot(sys,u,t)
```

simulates the response of a single-input model `sys` to the input  $u(t)=\sin(t)$  during 5 seconds.

For discrete-time models, `u` should be sampled at the same rate as `sys` (`t` is then redundant and can be omitted or set to the empty matrix).

For continuous-time models, choose the sampling period  $t(2) - t(1)$  small enough to accurately describe the input `u`. `lsim` issues a warning when `u` is undersampled, and hidden oscillations can occur.

`lsimplot(sys,u,t,x0)` specifies the initial state vector `x0` at time  $t(1)$  (for state-space models only). `x0` is set to zero when omitted.

`lsimplot(sys1,sys2,...,u,t,x0)` simulates the responses of multiple LTI models `sys1,sys2,...` on a single plot. The initial condition `x0` is optional. You can also specify a color, line style, and marker for each system, as in

```
lsimplot(sys1,'r',sys2,'y--',sys3,'gx',u,t)
```

`lsimplot(AX,...)` plots into the axes with handle `AX`.

`lsimplot(..., plotoptions)` plots the initial condition response with the options specified in `plotoptions`. Type

```
help timeoptions
```

for more detail.

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

### See Also

`lsim` | `setoptions` | `getoptions`

**Introduced before R2006a**

# looptune

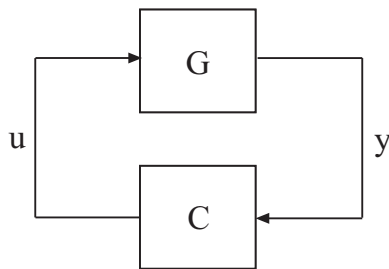
Tune fixed-structure feedback loops

## Syntax

```
[G,C,gam] = looptune(G0,C0,wc)
[G,C,gam] = looptune(G0,C0,wc,Req1,...,ReqN)
[G,C,gam] = looptune(...,options)
[G,C,gam,info] = looptune(...)
```

## Description

`[G,C,gam] = looptune(G0,C0,wc)` tunes the feedback loop



to meet the following default requirements:

- Bandwidth — Gain crossover for each loop falls in the frequency interval `wc`
- Performance — Integral action at frequencies below `wc`
- Robustness — Adequate stability margins and gain roll-off at frequencies above `wc`

The tunable `genss` model `C0` specifies the controller structure, parameters, and initial values. The model `G0` specifies the plant. `G0` can be a Numeric LTI model, or, for co-tuning the plant and controller, a tunable `genss` model. The sensor signals `y`

(measurements) and actuator signals **u** (controls) define the boundary between plant and controller.

---

**Note:** For tuning Simulink models with `looptune`, use `sITuner` to create an interface to your Simulink model. You can then tune the control system with `looptune` for `sITuner` (requires Simulink Control Design).

---

`[G,C,gam] = looptune(GO,C0,wc,Req1,...,ReqN)` tunes the feedback loop to meet additional design requirements specified in one or more tuning goal objects `Req1,...,ReqN`. Omit `wc` to use the requirements specified in `Req1,...,ReqN` instead of an explicit target crossover frequency and the default performance and robustness requirements.

`[G,C,gam] = looptune(...,options)` specifies further options, including target gain margin, target phase margin, and computational options for the tuning algorithm.

`[G,C,gam,info] = looptune(...)` returns a structure `info` with additional information about the tuned result. Use `info` with the `loopview` command to visualize tuning constraints and validate the tuned design.

## Input Arguments

### **GO**

Numeric LTI model or tunable `genss` model representing plant in control system to tune.

The plant is the portion of your control system whose outputs are sensor signals (measurements) and whose inputs are actuator signals (controls). Use `connect` to build `GO` from individual numeric or tunable components.

### **C0**

Generalized LTI model representing controller. `C0` specifies the controller structure, parameters, and initial values.

The controller is the portion of your control system that receives sensor signals (measurements) as inputs and produces actuator signals (controls) as outputs. Use Control Design Blocks and Generalized LTI models to represent tunable components of the controller. Use `connect` to build `C0` from individual numeric or tunable components.

**wc**

Vector specifying target crossover region [ $w_{\min}$ ,  $w_{\max}$ ]. The `looptune` command attempts to tune all loops in the control system so that the open-loop gain crosses 0 dB within the target crossover region.

A scalar `wc` specifies the target crossover region [ $wc/2$ ,  $2*wc$ ].

**Req1, . . . , ReqN**

One or more `TuningGoal` objects specifying design requirements, such as `TuningGoal.Tracking`, `TuningGoal.Gain`, or `TuningGoal.LoopShape`.

**options**

Set of options for `looptune` algorithm, specified using `looptuneOptions`. See `looptuneOptions` for information about the available options, including target gain margin and phase margin.

## Output Arguments

**G**

Tuned plant.

If `G0` is a Numeric LTI model, `G` is the same as `G0`.

If `G0` is a tunable `genss` model, `G` is a `genss` model with Control Design Blocks of the same number and types as `G0`. The current value of `G` is the tuned plant.

**C**

Tuned controller. `C` is a `genss` model with Control Design Blocks of the same number and types as `C0`. The current value of `C` is the tuned controller.

**gam**

Parameter indicating degree of success at meeting all tuning constraints. A value of `gam <= 1` indicates that all requirements are satisfied. `gam >> 1` indicates failure to

meet at least one requirement. Use `loopview` to visualize the tuned result and identify the unsatisfied requirement.

For best results, use the `RandomStart` option in `looptuneOptions` to obtain several minimization runs. Setting `RandomStart` to an integer  $N > 0$  causes `looptune` to run the optimization  $N$  additional times, beginning from parameter values it chooses randomly. You can examine `gam` for each run to help identify an optimization result that meets your design requirements.

### **info**

Data for validating tuning results, returned as a structure. To use the data in `info`, use the command `loopview(G,C,info)` to visualize tuning constraints and validate the tuned design.

`info` contains the following tuning data:

### **Di,Do**

Optimal input and output scalings, returned as state-space models. The scaled plant is given by  $Do \backslash G * Di$ .

### **Specs**

Design requirements that `looptune` constructs for its call to `system` for tuning (see “Algorithms” on page 2-544), returned as a vector of `TuningGoal` requirement objects.

### **Runs**

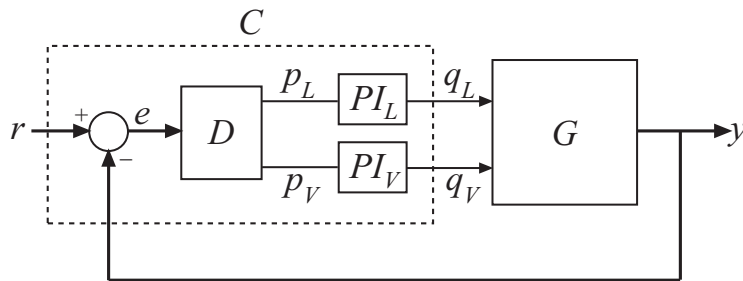
Detailed information about each optimization run performed by `system` when called by `looptune` for tuning (see “Algorithms” on page 2-544), returned as a data structure.

The contents of `Runs` are the `info` output of the call to `system`. For information about the fields of `Runs`, see the `info` output argument description on the `system` reference page.

## **Examples**

Tune the control system of the following illustration, to achieve crossover between 0.1 and 1 rad/min.





The 2-by-2 plant  $G$  is represented by:

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

The fixed-structure controller,  $C$ , includes three components: the 2-by-2 decoupling matrix  $D$  and two PI controllers  $PI\_L$  and  $PI\_V$ . The signals  $r$ ,  $y$ , and  $e$  are vector-valued signals of dimension 2.

Build a numeric model that represents the plant and a tunable model that represents the controller. Name all inputs and outputs as in the diagram, so that `looptune` knows how to interconnect the plant and controller via the control and measurement signals.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL','qV'};
G.OutputName = 'y';

D = tunableGain('Decoupler',eye(2));
D.InputName = 'e';
D.OutputName = {'pL','pV'};
PI_L = tunablePID('PI_L','pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';
PI_V = tunablePID('PI_V','pi');
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';
sum1 = sumblk('e = r - y',2);
CO = connect(PI_L,PI_V,D,sum1,{'r','y'},{'qL','qV'});

wc = [0.1,1];
[G,C,gam,info] = looptune(G,CO,wc);
```

C is the tuned controller, in this case a `genss` model with the same block types as `C0`.

You can examine the tuned result using `loopview`.

## Alternatives

For tuning Simulink models with `looptune`, see `sITuner` and `looptune` (requires Simulink Control Design).

## More About

### Algorithms

`looptune` automatically converts target bandwidth, performance requirements, and additional design requirements into weighting functions that express the requirements as an  $H_\infty$  optimization problem. `looptune` then uses `system` to optimize tunable parameters to minimize the  $H_\infty$  norm. For more information about the optimization algorithms, see [1].

`looptune` computes the  $H_\infty$  norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see <http://slicot.org>.

## References

- [1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis." *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71–86.
- [2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the  $H_\infty$ -Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

## See Also

`TuningGoal.Tracking` | `sITuner` | `looptune` (for `sITuner`) | `TuningGoal.Gain` | `TuningGoal.LoopShape` | `system` | `hinfstruct` | `looptuneOptions` | `loopview` | `loopmargin` | `genss` | `connect`

# looptuneOptions

Set options for looptune

## Syntax

```
options = looptuneOptions  
options = looptuneOptions(Name,Value)
```

## Description

`options = looptuneOptions` returns the default option set for the `looptune` command.

`options = looptuneOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`looptuneOptions` takes the following `Name` arguments:

#### 'GainMargin'

Target gain margin in decibels. `GainMargin` specifies the required gain margin for the tuned control system. For MIMO control systems, the gain margin is the multiloop disk margin. See `loopmargin` for the definition of the multiloop disk margin.

**Default:** 7.6 dB

### 'PhaseMargin'

Target phase margin in degrees. `PhaseMargin` specifies the required phase margin for the tuned control system. For MIMO control systems, the phase margin is the multiloop disk margin. See `loopmargin` for the definition of the multiloop disk margin.

**Default:** 45 degrees

### 'Display'

Amount of information to display during `looptune` runs, specified as one of the following values.

- 'off' — Run in silent mode, displaying no information during or after the run.
- 'iter' — Display optimization progress after each iteration. The display includes the value of the objective parameter `gam` after each iteration. The display also includes a `Progress` value, indicating the percent change in `gam` from the previous iteration.
- 'final' — Display a one-line summary at the end of each optimization run. The display includes the minimized value of `gam` and the number of iterations for each run.

**Default:** 'final'

### 'MaxIter'

Maximum number of iterations in each optimization run.

**Default:** 300

### 'RandomStart'

Number of additional optimizations starting from random values of the free parameters in the controller.

If `RandomStart = 0`, `looptune` performs a single optimization run starting from the initial values of the tunable parameters. Setting `RandomStart = N > 0` runs  $N$  additional optimizations starting from  $N$  randomly generated parameter values.

`looptune` tunes by finding a local minimum of a gain minimization problem. To increase the likelihood of finding parameter values that meet your design requirements, set `RandomStart > 0`. You can then use the best design that results from the multiple optimization runs.

Use with `UseParallel = true` to distribute independent optimization runs among MATLAB workers (requires Parallel Computing Toolbox™ software).

**Default:** 0

**'UseParallel'**

Parallel processing flag.

Set to `true` to enable parallel processing by distributing randomized starts among workers in a parallel pool. If there is an available parallel pool, then the software performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

- If **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, then the software starts a parallel pool using the settings in those preferences.
- If **Automatically create a parallel pool** is not selected in your preferences, then the software performs the optimization runs successively, without parallel processing.

If **Automatically create a parallel pool** is not selected in your preferences, you can manually start a parallel pool using `parpool` before running the tuning command.

Using parallel processing requires Parallel Computing Toolbox software.

**Default:** `false`

**'TargetGain'**

Target value for the objective parameter `gam`.

The `looptune` command converts your design requirements into normalized gain constraints. The command then tunes the free parameters of the control system to drive the objective parameter `gam` below 1 to enforce all requirements.

The default `TargetGain = 1` ensures that the optimization stops as soon as `gam` falls below 1. Set `TargetGain` to a smaller or larger value to continue the optimization or start sooner, respectively.

**Default:** 1

**'TolGain'**

Relative tolerance for termination.

The optimization terminates when the objective parameter `gam` decreases by less than `TolGain` over 10 consecutive iterations. Increasing `TolGain` speeds up termination, and decreasing `TolGain` yields tighter final values.

**Default:** 0.001

### 'MaxFrequency'

Maximum closed-loop natural frequency.

Setting `MaxFrequency` constrains the closed-loop poles to satisfy  $|p| < \text{MaxFrequency}$ .

To allow `looptune` to choose the closed-loop poles automatically, based upon the system's open-loop dynamics, set `MaxFrequency = Inf`. To prevent unwanted fast dynamics or high-gain control, set `MaxFrequency` to a finite value.

Specify `MaxFrequency` in units of `1/TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

**Default:** `Inf`

### 'MinDecay'

Minimum decay rate for closed-loop poles

Constrains the closed-loop poles to satisfy  $\text{Re}(p) < -\text{MinDecay}$ . Increase this value to improve the stability of closed-loop poles that do not affect the closed-loop gain due to pole/zero cancellations.

Specify `MinDecay` in units of `1/TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

**Default:** `1e-7`

## Output Arguments

### **options**

Option set containing the specified options for the `looptune` command.

## Examples

### Create Options Set for looptune

Create an options set for a `looptune` run using three random restarts. Also, set the target gain and phase margins to 6 dB and 50 degrees, respectively, and limit the closed-loop pole magnitude to 100.

```
options = looptuneOptions('RandomStart',3,'GainMargin',6,...  
                          'PhaseMargin',50,'SpecRadius',100);
```

Alternatively, use dot notation to set the values of `options`.

```
options = looptuneOptions;  
options.RandomStart = 3;  
options.GainMargin = 6;  
options.PhaseMargin = 50;  
options.SpecRadius = 100;
```

### Configure Option Set for Parallel Optimization Runs

Configure an option set for a `looptune` run using 20 random restarts. Execute these independent optimization runs concurrently on multiple workers in a parallel pool.

If you have the Parallel Computing Toolbox software installed, you can use parallel computing to speed up `looptune` tuning of fixed-structure control systems. When you run multiple randomized `looptune` optimization starts, parallel computing speeds up tuning by distributing the optimization runs among workers.

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences, manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a `looptuneOptions` set that specifies 20 random restarts to run in parallel.

```
options = looptuneOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the `looptuneOptions` set when you call `looptune`. For example, suppose you have already created a plant model `G0` and tunable controller `C0`. In this case, the following command uses parallel computing to tune the control system of `G0` and `C0` to the target `crossoverWC`.

```
[G,C,gamma] = looptune(G0,C0,wc,options);
```

### **See Also**

`looptune` (for `sITuner`) | `loopmargin` | `looptune`



# looptuneSetup

Convert tuning setup for `looptune` to tuning setup for `systeme`

## Syntax

```
[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(looptuneInputs)
```

## Description

`[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(looptuneInputs)` converts a tuning setup for `looptune` into an equivalent tuning setup for `systeme`. The argument `looptuneInputs` is a sequence of input arguments for `looptune` that specifies the tuning setup. For example,

```
[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(G0,C0,wc,Req1,Req2,loopopt)
```

generates a set of arguments such that `looptune(G0,C0,wc,Req1,Req2,loopopt)` and `systeme(T0,SoftReqs,HardReqs,sysopt)` produce the same results.

Use this command to take advantage of additional flexibility that `systeme` offers relative to `looptune`. For example, `looptune` requires that you tune all channels of a MIMO feedback loop to the same target bandwidth. Converting to `systeme` allows you to specify different crossover frequencies and loop shapes for each loop in your control system. Also, `looptune` treats all tuning requirements as soft requirements, optimizing them but not requiring that any constraint be exactly met. Converting to `systeme` allows you to enforce some tuning requirements as hard constraints, while treating others as soft requirements.

You can also use this command to probe into the tuning requirements used by `looptune`.

---

**Note:** When tuning Simulink models through an `sITuner` interface, use `looptuneSetup` for `sITuner` (requires Simulink Control Design).

---

## Examples

### Convert `looptune` Problem into `systeme` Problem

Convert a set of `looptune` inputs into an equivalent set of inputs for `systeme`.

Suppose you have a numeric plant model, `G0`, and a tunable controller model, `C0`. Suppose also that you used `looptune` to tune the feedback loop between `G0` and `C0` to within a bandwidth of `wc = [wmin, wmax]`. Convert these variables into a form that allows you to use `systeme` for further tuning.

```
[T0, SoftReqs, HardReqs, sysopt] = looptuneSetup(C0, G0, wc);
```

The command returns the closed-loop system and tuning requirements for the equivalent `systeme` command, `systeme(CLO, SoftReqs, HardReqs, sysopt)`. The arrays `SoftReqs` and `HardReqs` contain the tuning requirements implicitly imposed by `looptune`. These requirements enforce the target bandwidth and default stability margins of `looptune`.

If you used additional tuning requirements when tuning the system with `looptune`, add them to the input list of `looptuneSetup`. For example, suppose you used a `TuningGoal.Tracking` requirement, `Req1`, and a `TuningGoal.Rejection` requirement, `Req2`. Suppose also that you set algorithm options for `looptune` using `looptuneOptions`. Incorporate these requirements and options into the equivalent `systeme` command.

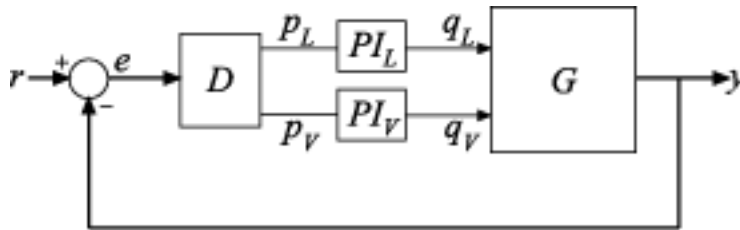
```
[T0, SoftReqs, HardReqs, sysopt] = looptuneSetup(C0, G0, wc, Req1, Req2, loopopt);
```

The resulting arguments allow you to construct an equivalent tuning problem for `systeme`. In particular, `[-, C] = looptune(C0, G0, wc, Req1, Req2, loopopt)` yields the same result as the following commands.

```
T = systeme(T0, SoftReqs, HardReqs, sysopt);  
C = setBlockValue(C0, T);
```

### Convert Distillation Column Problem for Tuning With `systeme`

Set up the following control system for tuning with `looptune`. Then convert the setup to a `systeme` problem and examine the results. These results reflect the structure of the control system model that `looptune` tunes. The results also reflect the tuning requirements implicitly enforced when tuning with `looptune`.



For this example, the 2-by-2 plant  $G$  is represented by:

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

The fixed-structure controller,  $C$ , includes three components: the 2-by-2 decoupling matrix  $D$  and two PI controllers  $PI\_L$  and  $PI\_V$ . The signals  $r$ ,  $y$ , and  $e$  are vector-valued signals of dimension 2.

Build a numeric model that represents the plant and a tunable model that represents the controller. Name all inputs and outputs as in the diagram, so that `looptune` and `looptuneSetup` know how to interconnect the plant and controller via the control and measurement signals.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL', 'qV'};
G.OutputName = {'y'};

D = tunableGain('Decoupler', eye(2));
D.InputName = 'e';
D.OutputName = {'pL', 'pV'};
PI_L = tunablePID('PI_L', 'pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';
PI_V = tunablePID('PI_V', 'pi');
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';
sum1 = sumblk('e = r - y', 2);
CO = connect(PI_L, PI_V, D, sum1, {'r', 'y'}, {'qL', 'qV'});
```

This system is now ready for tuning with `looptune`, using tuning goals that you specify. For example, specify a target bandwidth range. Create a tuning requirement that

imposes reference tracking in both channels of the system with a response time of 15 s, and a disturbance rejection requirement.

```
wc = [0.1,0.5];
TR = TuningGoal.Tracking('r','y',15,0.001,1);
DR = TuningGoal.Rejection({'qL','qV'},1/s);
DR.Focus = [0 0.1];
```

```
[G,C,gam,info] = looptune(G,CO,wc,TR,DR);
```

```
Final: Peak gain = 1, Iterations = 52
Achieved target gain value TargetGain=1.
```

`looptune` successfully tunes the system to these requirements. However, you might want to switch to `sysstune` to take advantage of additional flexibility in configuring your problem. For example, instead of tuning both channels to a loop bandwidth inside `wc`, you might want to specify different crossover frequencies for each loop. Or, you might want to enforce the tuning requirements `TR` and `DR` as hard constraints, and add other requirements as soft requirements.

Convert the `looptune` input arguments to a set of input arguments for `sysstune`.

```
[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(G,CO,wc,TR,DR);
```

This command returns a set of arguments you can provide to `sysstune` for equivalent results to tuning with `looptune`. In other words, the following command is equivalent to the previous `looptune` command.

```
[T,fsoft,ghard,info] = sysstune(T0,SoftReqs,HardReqs,sysopt);
```

```
Final: Peak gain = 1, Iterations = 52
Achieved target gain value TargetGain=1.
```

Examine the arguments returned by `looptuneSetup`.

`T0`

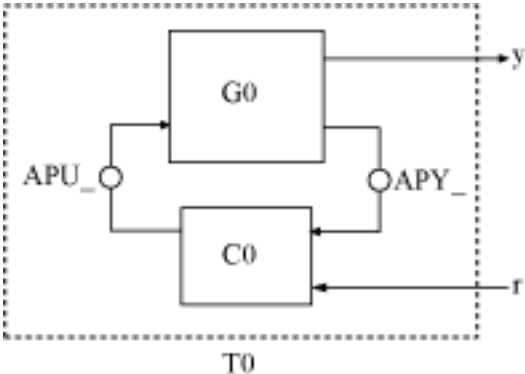
`T0 =`

```
Generalized continuous-time state-space model with 0 outputs, 2 inputs, 4 states, and
APU_: Analysis point, 2 channels, 1 occurrences.
APY_: Analysis point, 2 channels, 1 occurrences.
Decoupler: Parametric 2x2 gain, 1 occurrences.
```

PI\_L: Parametric PID controller, 1 occurrences.  
PI\_V: Parametric PID controller, 1 occurrences.

Type "ss(T0)" to see the current value, "get(T0)" to see all properties, and "T0.Blocks"

The software constructs the closed-loop control system for `system` by connecting the plant and controller at their control and measurement signals, and inserting a two-channel `AnalysisPoint` block at each of the connection locations, as illustrated in the following diagram.



When tuning the control system of this example with `looptune`, all requirements are treated as soft requirements. Therefore, `HardReqs` is empty. `SoftReqs` is an array of `TuningGoal` requirements. These requirements together enforce the bandwidth and margins of the `looptune` command, plus the additional requirements that you specified.

**SoftReqs**

SoftReqs =

5x1 heterogeneous SystemLevel (LoopShape, Tracking, Rejection, ...) array with proper

Models  
Openings  
Name

Examine the first entry in `SoftReqs`.

```
SoftReqs(1)
```

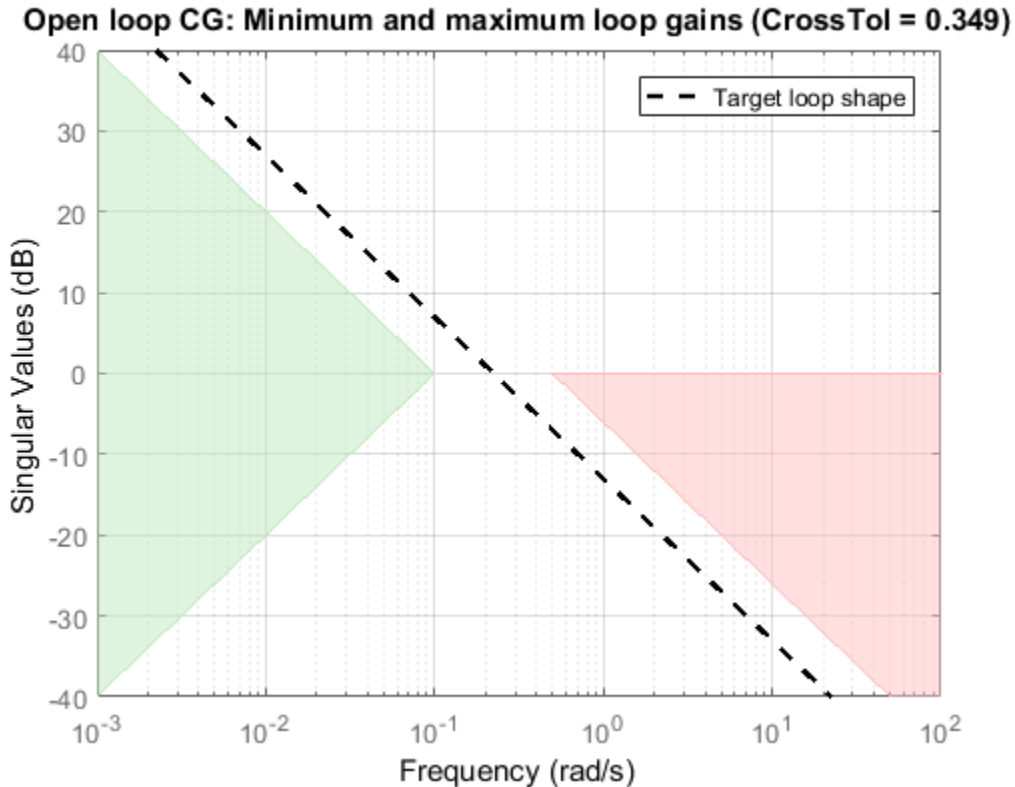
```
ans =
```

```
LoopShape with properties:
```

```
    LoopGain: [1×1 zpk]  
    CrossTol: 0.3495  
        Focus: [0 Inf]  
    Stabilize: 1  
LoopScaling: 'on'  
    Location: {2×1 cell}  
        Models: NaN  
    Openings: {0×1 cell}  
        Name: 'Open loop CG'
```

`looptuneSetup` expresses the target crossover frequency range `WC` as a `TuningGoal.LoopShape` requirement. This requirement constrains the open-loop gain profile to the loop shape stored in the `LoopGain` property, with a crossover frequency and crossover tolerance (`CrossTol`) determined by `wc`. Examine this loop shape.

```
viewSpec(SoftReqs(1))
```



The target crossover is expressed as an integrator gain profile with a crossover between 0.1 and 0.5 rad/s, as specified by `wc`. If you want to specify a different loop shape, you can alter this `TuningGoal.LoopShape` requirement before providing it to `systemtune`.

`looptune` also tunes to default stability margins that you can change using `looptuneOptions`. For `systemtune`, stability margins are specified using `TuningGoal.Margins` requirements. Here, `looptuneSetup` has expressed the default stability margins of `looptune` as soft `TuningGoal.Margins` requirements. For example, examine the fourth entry in `SoftReqs`.

```
SoftReqs(4)
```

```
ans =  
  
  Margins with properties:  
  
    GainMargin: 7.6000  
    PhaseMargin: 45  
    ScalingOrder: 0  
        Focus: [0 Inf]  
    Location: {2×1 cell}  
    Models: NaN  
    Openings: {0×1 cell}  
        Name: 'Margins at plant inputs'
```

The last entry in `SoftReqs` is a similar `TuningGoal.Margins` requirement constraining the margins at the plant outputs. `looptune` enforces these margins as soft requirements. If you want to convert them to hard constraints, pass them to `systemtune` in the input vector `HardReqs` instead of the input vector `SoftReqs`.

## Input Arguments

**looptuneInputs** — Plant, controller, and requirement inputs to `looptune`  
valid `looptune` input sequence

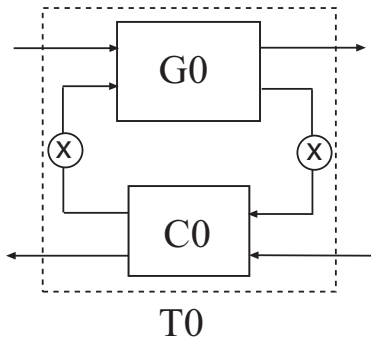
Plant, controller, and requirement inputs to `looptune`, specified as a valid `looptune` input sequence. For more information about the arguments in a valid `looptune` input sequence, see the `looptune` reference page.

## Output Arguments

**T0** — Closed-loop control system model  
generalized state-space model

Closed-loop control system model for tuning with `systemtune`, returned as a generalized state-space `genss` model. To compute `T0`, the plant, `G0`, and the controller, `C0`, are combined in the feedback configuration of the following illustration.





The connections between `C0` and `G0` are determined by matching signals using the `InputName` and `OutputName` properties of the two models. In general, the signal lines in the diagram can represent vector-valued signals. `AnalysisPoint` blocks, indicated by `X` in the diagram, are inserted between the controller and the plant. This allows definition of open-loop and closed-loop requirements on signals injected or measured at the plant inputs or outputs. For example, the bandwidth `wc` is converted into a `TuningGoal.LoopShape` requirement that imposes the desired crossover on the open-loop signal measured at the plant input.

For more information on the structure of closed-loop control system models for tuning with `systemtune`, see the `systemtune` reference page.

### **SoftReqs — Soft tuning requirements**

vector of `TuningGoal` requirement objects

Soft tuning requirements for tuning with `systemtune`, specified as a vector of `TuningGoal` requirement objects.

`looptune` expresses most of its implicit tuning requirements as soft tuning requirements. For example, a specified target loop bandwidth is expressed as a `TuningGoal.LoopShape` requirement with integral gain profile and crossover at the target frequency. Additionally, `looptune` treats all of the explicit requirements you specify (`Req1`, . . . `ReqN`) as soft requirements. `SoftReqs` contains all of these tuning requirements.

### **HardReqs — Hard tuning requirements**

vector of `TuningGoal` requirement objects

Hard tuning requirements (constraints) for tuning with `systemtune`, specified as a vector of `TuningGoal` requirement objects.

Because `looptune` treats most tuning requirements as soft requirements, `HardReqs` is usually empty. However, if you change the default `MaxFrequency` option of the `looptuneOptions` set, `loopopt`, then this requirement appears as a hard `TuningGoal.Poles` constraint.

### **sysopt — Algorithm options for systune tuning**

`systuneOptions` options set

Algorithm options for `systune` tuning, specified as a `systuneOptions` options set.

Some of the options in the `looptuneOptions` set, `loopopt`, are expressed as hard or soft requirements that are returned in `HardReqs` and `SoftReqs`. Other options correspond to options in the `systuneOptions` set.

## **Alternatives**

When tuning Simulink using an `sITuner`, interface, convert a `looptune` problem to `systune` using `looptuneSetup` for `sITuner`.

### **See Also**

`genss` | `looptune` | `looptuneOptions` | `looptuneSetup` (for `sITuner`) | `sITuner` | `systune` | `systuneOptions`

**Introduced in R2013b**

# loopview

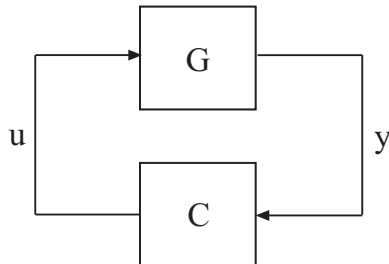
Graphically analyze MIMO feedback loops

## Syntax

```
loopview(G,C)  
loopview(G,C,info)
```

## Description

`loopview(G,C)` plots characteristics of the following positive-feedback, multi-input, multi-output (MIMO) feedback loop with plant  $G$  and controller  $C$ .



Use `loopview` to analyze the performance of a tuned control system you obtain using `looptune`.

---

**Note:** If you are tuning a Simulink model with `looptune` through an `sITuner` interface, analyze the performance of your control system using `loopview` for `sITuner` (requires Simulink Control Design).

---

`loopview` plots the singular values of:

- Open-loop frequency responses  $G \cdot C$  and  $C \cdot G$

- Sensitivity function  $S = \text{inv}(1 - G * C)$  and complementary sensitivity  $T = 1 - S$
- Maximum (target), actual (tuned), and normalized MIMO stability margins. `loopview` plots the multi-loop disk margin (see `loopmargin`). Use this plot to verify that the stability margins of the tuned system do not significantly exceed the target value.

For more information about singular values, see `sigma`.

`loopview(G,C,info)` uses the `info` structure returned by `looptune`. This syntax also plots the target and tuned values of tuning constraints imposed on the system. Additional plots include:

- Singular values of the maximum allowed  $S$  and  $T$ . The curve marked **S/T Max** shows the maximum allowed  $S$  on the low-frequency side of the plot, and the maximum allowed  $T$  on the high-frequency side. These curves are the constraints that `looptune` imposes on  $S$  and  $T$  to enforce the target crossover range `WC`.
- Target and tuned values of constraints imposed by any tuning goal requirements you used with `looptune`.

Use `loopview` with the `info` structure to assist in troubleshooting when tuning fails to meet all requirements.

## Input Arguments

### G

Numeric LTI model or tunable `genss` model representing the plant in a control system. The plant is the portion of a control system whose outputs are sensor signals (measurements), and whose inputs are actuator signals (controls).

You can obtain `G` as an output argument from `looptune` when you tune your control system.

### C

`genss` model representing the controller in a control system. The controller is the portion of your control system that receives sensor signals (measurements) as inputs and produces actuator signals (controls) as outputs.

You can obtain **C** as an output argument from `looptune` when you tune your control system.

### **info**

`info` structure returned by `looptune` during control system tuning.

## **Examples**

### **Examine Performance of Tuned Controller**

Tune a control system, and use `loopview` to examine the performance of the tuned controller.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL', 'qV'};
G.OutputName = 'y';

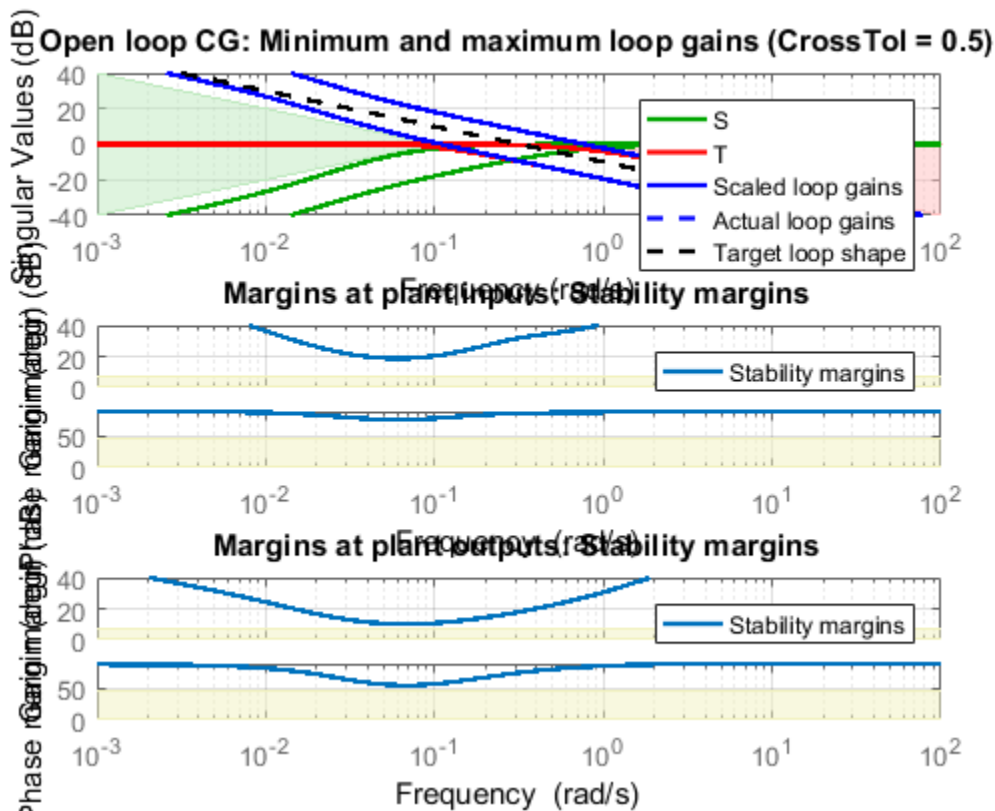
D = tunableGain('Decoupler', eye(2));
PI_L = tunablePID('PI_L', 'pi');
PI_L.OutputName = 'qL';
PI_V = tunablePID('PI_V', 'pi');
PI_V.OutputName = 'qV';

sum = sumblk('e = r - y', 2);
CO = (blkdiag(PI_L, PI_V)*D)*sum;

wc = [0.1, 1];
options = looptuneOptions('RandomStart', 5);
[G, C, gam, info] = looptune(-G, CO, wc, options);

loopview(G, C, info)

Final: Peak gain = 0.859, Iterations = 25
Achieved target gain value TargetGain=1.
```



The first plot shows that the open-loop gain crossovers fall close to the specified interval  $[0.1,1]$ . This plot also includes the tuned values of the sensitivity function  $S = \text{inv}(1 - G \cdot C)$  and complementary sensitivity  $T = 1 - S$ . These curves reflect the constraints that looptune imposes on  $S$  and  $T$  to enforce the target crossover range  $WC$ .

The second and third plots show that the MIMO stability margins of the tuned system fall well within the target range.

- “Tune MIMO Control System for Specified Bandwidth”
- “Decoupling Controller for a Distillation Column”

## Alternatives

For analyzing Simulink models tuned with `looptune` through an `sITuner` interface, use `loopview` for `sITuner` (requires Simulink Control Design).

## See Also

`looptune` (for `sITuner`) | `looptune` | `sITuner` | `loopview` (for `sITuner`)

## lyap

Continuous Lyapunov equation solution

### Syntax

```
lyap  
X = lyap(A,Q)  
X = lyap(A,B,C)  
X = lyap(A,Q,[ ],E)
```

### Description

`lyap` solves the special and general forms of the Lyapunov 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  $Q$  represent square matrices of identical sizes. If  $Q$  is a symmetric matrix, the solution  $X$  is also a symmetric matrix.

`X = lyap(A,B,C)` solves the Sylvester equation

$$AX + XB + C = 0$$

The matrices  $A$ ,  $B$ , and  $C$  must have compatible dimensions but need not be square.

`X = lyap(A,Q,[ ],E)` solves the generalized Lyapunov equation

$$AXE^T + EXA^T + Q = 0$$

where  $Q$  is a symmetric matrix. You must use empty square brackets `[ ]` for this function. If you place any values inside the brackets, the function errors out.



## Limitations

The continuous Lyapunov equation has a unique solution if the eigenvalues  $\alpha_1, \alpha_2, \dots, \alpha_n$  of  $A$  and  $\beta_1, \beta_2, \dots, \beta_n$  of  $B$  satisfy

$$\alpha_i + \beta_j \neq 0 \quad \text{for all pairs}(i, j)$$

If this condition is violated, `lyap` produces the error message:

Solution does not exist or is not unique.

## Examples

### Example 1

#### Solve Lyapunov Equation

Solve the Lyapunov equation

$$AX + XA^T + Q = 0$$

where

$$A = \begin{bmatrix} 1 & 2 \\ -3 & -4 \end{bmatrix} \quad Q = \begin{bmatrix} 3 & 1 \\ 1 & 1 \end{bmatrix}$$

The  $A$  matrix is stable, and the  $Q$  matrix is positive definite.

`A = [1 2; -3 -4];`

`Q = [3 1; 1 1];`

`X = lyap(A,Q)`

These commands return the following  $X$  matrix:

`X =`

```

6.1667    -3.8333
-3.8333     3.0000

```

You can compute the eigenvalues to see that  $X$  is positive definite.

`eig(X)`

The command returns the following result:

`ans =`

```
0.4359
8.7308
```

## Example 2

### Solve Sylvester Equation

Solve the Sylvester equation

$$AX + XB + C = 0$$

where

$$A = 5 \quad B = \begin{bmatrix} 4 & 3 \\ 4 & 3 \end{bmatrix} \quad C = [2 \quad 1]$$

```
A = 5;
B = [4 3; 4 3];
C = [2 1];
X = lyap(A,B,C)
```

These commands return the following  $X$  matrix:

```
X =
-0.2000    -0.0500
```

## More About

### Algorithms

`lyap` uses SLICOT routines SB03MD and SG03AD for Lyapunov equations and SB04MD (SLICOT) and ZTRSYL (LAPACK) for Sylvester 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] Barraud, A.Y., "A numerical algorithm to solve  $A X A - X = Q$ ," *IEEE Trans. Auto. Contr.*, AC-22, pp. 883–885, 1977.
- [3] Hammarling, S.J., "Numerical solution of the stable, non-negative definite Lyapunov equation," *IMA J. Num. Anal.*, Vol. 2, pp. 303–325, 1982.
- [4] Penzl, T., "Numerical solution of generalized Lyapunov equations," *Advances in Comp. Math.*, Vol. 8, pp. 33–48, 1998.
- [5] Golub, G.H., Nash, S. and Van Loan, C.F., "A Hessenberg-Schur method for the problem  $AX + XB = C$ ," *IEEE Trans. Auto. Contr.*, AC-24, pp. 909–913, 1979.

## See Also

covar | dlyap

**Introduced before R2006a**

## lyapchol

Square-root solver for continuous-time Lyapunov equation

### Syntax

```
R = lyapchol(A,B)
X = lyapchol(A,B,E)
```

### Description

`R = lyapchol(A,B)` computes a Cholesky factorization  $X = R' * R$  of the solution  $X$  to the Lyapunov matrix equation:

$$A * X + X * A' + B * B' = 0$$

All eigenvalues of matrix  $A$  must lie in the open left half-plane for  $R$  to exist.

`X = lyapchol(A,B,E)` computes a Cholesky factorization  $X = R' * R$  of  $X$  solving the generalized Lyapunov equation:

$$A * X * E' + E * X * A' + B * B' = 0$$

All generalized eigenvalues of  $(A,E)$  must lie in the open left half-plane for  $R$  to exist.

### More About

#### Algorithms

`lyapchol` uses SLICOT routines SB03OD and SG03BD.

### 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] Hammarling, S.J., "Numerical solution of the stable, non-negative definite Lyapunov equation," *IMA J. Num. Anal.*, Vol. 2, pp. 303-325, 1982.

[3] Penzl, T., "Numerical solution of generalized Lyapunov equations," *Advances in Comp. Math.*, Vol. 8, pp. 33-48, 1998.

## **See Also**

lyap | dlyapchol

**Introduced before R2006a**

# mag2db

Convert magnitude to decibels (dB)

## Syntax

```
ydb = mag2db(y)
```

## Description

`ydb = mag2db(y)` returns the corresponding decibel (dB) value *ydb* for a given magnitude *y*. The relationship between magnitude and decibels is  $ydb = 20 \log_{10}(y)$ .

## See Also

db2mag

Introduced in R2008a

# make1DOF

Convert 2-DOF PID controller to 1-DOF controller

## Syntax

```
C1 = make1DOF(C2)
```

## Description

`C1 = make1DOF(C2)` converts the two-degree-of-freedom PID controller `C2` to one degree of freedom by removing the terms that depend on coefficients  $b$  and  $c$ .

## Examples

### Convert 2-DOF PID controller to 1-DOF

Design a 2-DOF PID controller for a plant.

```
G = tf(1,[1 0.5 0.1]);
C2 = pidtune(G,'pidf2',1.5)
```

`C2 =`

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with  $K_p = 1.12$ ,  $K_i = 0.23$ ,  $K_d = 1.3$ ,  $T_f = 0.122$ ,  $b = 0.664$ ,  $c = 0.0136$

Continuous-time 2-DOF PIDF controller in parallel form.

Convert the controller to one degree of freedom.

```
C1 = make1DOF(C2)
```

C1 =

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f*s+1}$$

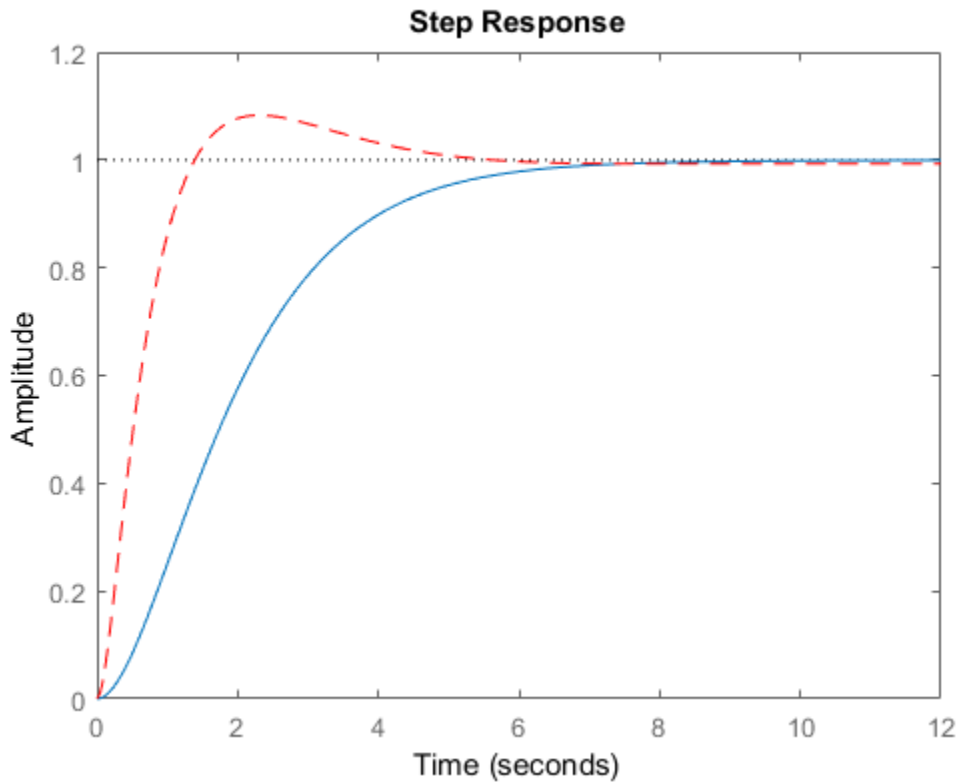
with  $K_p = 1.12$ ,  $K_i = 0.23$ ,  $K_d = 1.3$ ,  $T_f = 0.122$

Continuous-time PIDF controller in parallel form.

The new controller has the same PID gains and filter constant. However, `make1DOF` removes the terms involving the setpoint weights `b` and `c`. Therefore, in a closed loop with the plant `G`, the 2-DOF controller `C2` yields a different closed-loop response from `C1`.

```
CM = tf(C2);
T2 = CM(1)*feedback(G, -CM(2));
T1 = feedback(G*C1,1);
stepplot(T2,T1, 'r--')
```





## Input Arguments

### **C2** — 2-DOF PID controller

`pid2` object | `pidstd2` object

2-DOF PID controller, specified as a `pid2` object or a `pidstd2` object.

## Output Arguments

### **C1** — 1-DOF PID controller

`pid` object | `pidstd` object

1-DOF PID controller, returned as a `pid` or `pidstd` object. `C1` is in parallel form if `C2` is in parallel form, and standard form if `C2` is in standard form.

For example, suppose `C2` is a continuous-time, parallel-form 2-DOF `pid2` controller. The relationship between the inputs,  $r$  and  $y$ , and the output  $u$  of `C2` is given by:

$$u = K_p (br - y) + \frac{K_i}{s} (r - y) + \frac{K_d s}{T_f s + 1} (cr - y).$$

Then `C1` is a parallel-form 1-DOF `pid` controller of the form:

$$C_1 = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}.$$

The PID gains  $K_p$ ,  $K_i$ , and  $K_d$ , and the filter time constant  $T_f$  are unchanged. `make1DOF` removes the terms that depend on the setpoint weights  $b$  and  $c$ . For more information about 2-DOF PID controllers, see “Two-Degree-of-Freedom PID Controllers”.

The conversion also preserves the values of the properties `Ts`, `TimeUnit`, `SamplingGrid`, `IFormula`, and `DFormula`.

## More About

- “Two-Degree-of-Freedom PID Controllers”

## See Also

`getComponents` | `make2DOF` | `pid` | `pid2` | `pidstd` | `pidstd2`

**Introduced in R2015b**

# make2DOF

Convert 1-DOF PID controller to 2-DOF controller

## Syntax

```
C2 = make2DOF(C1)
C2 = make2DOF(C1,b)
C2 = make2DOF(C1,b,c)
```

## Description

`C2 = make2DOF(C1)` converts the one-degree-of-freedom PID controller `C1` to two degrees of freedom. The setpoint weights `b` and `c` of the 2-DOF controller are 1, and the remaining PID coefficients do not change.

`C2 = make2DOF(C1,b)` specifies the setpoint weight for the proportional term.

`C2 = make2DOF(C1,b,c)` specifies the setpoint weights for both the proportional and derivative terms.

## Examples

### Convert 1-DOF PID controller to 2-DOF

Design a 1-DOF PID controller for a plant.

```
G = tf(1,[1 0.5 0.1]);
C1 = pidtune(G,'pidf',1.5)
```

`C1 =`

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f*s+1}$$

with  $K_p = 1.12$ ,  $K_i = 0.23$ ,  $K_d = 1.3$ ,  $T_f = 0.122$

Continuous-time PIDF controller in parallel form.

Convert the controller to two degrees of freedom.

```
C2 = make2DOF(C1)
```

```
C2 =
```

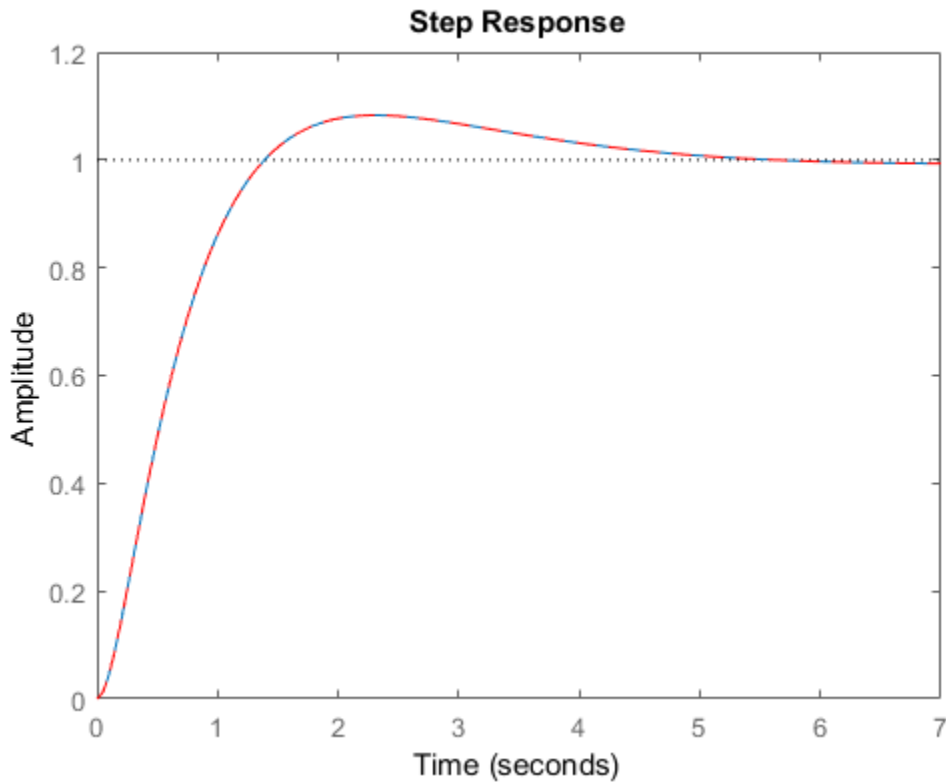
$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with  $K_p = 1.12$ ,  $K_i = 0.23$ ,  $K_d = 1.3$ ,  $T_f = 0.122$ ,  $b = 1$ ,  $c = 1$

Continuous-time 2-DOF PIDF controller in parallel form.

The new controller has the same PID gains and filter constant. It also contains new terms involving the setpoint weights  $b$  and  $c$ . By default,  $b = c = 1$ . Therefore, in a closed loop with the plant  $G$ , the 2-DOF controller  $C2$  yields the same response as  $C1$ .

```
T1 = feedback(G*C1,1);  
CM = tf(C2);  
T2 = CM(1)*feedback(G,-CM(2));  
stepplot(T1,T2,'r--')
```



Convert **C1** to a 2-DOF controller with different **b** and **c** values.

```
C2_2 = make2DOF(C1,0.5,0.75)
```

```
C2_2 =
```

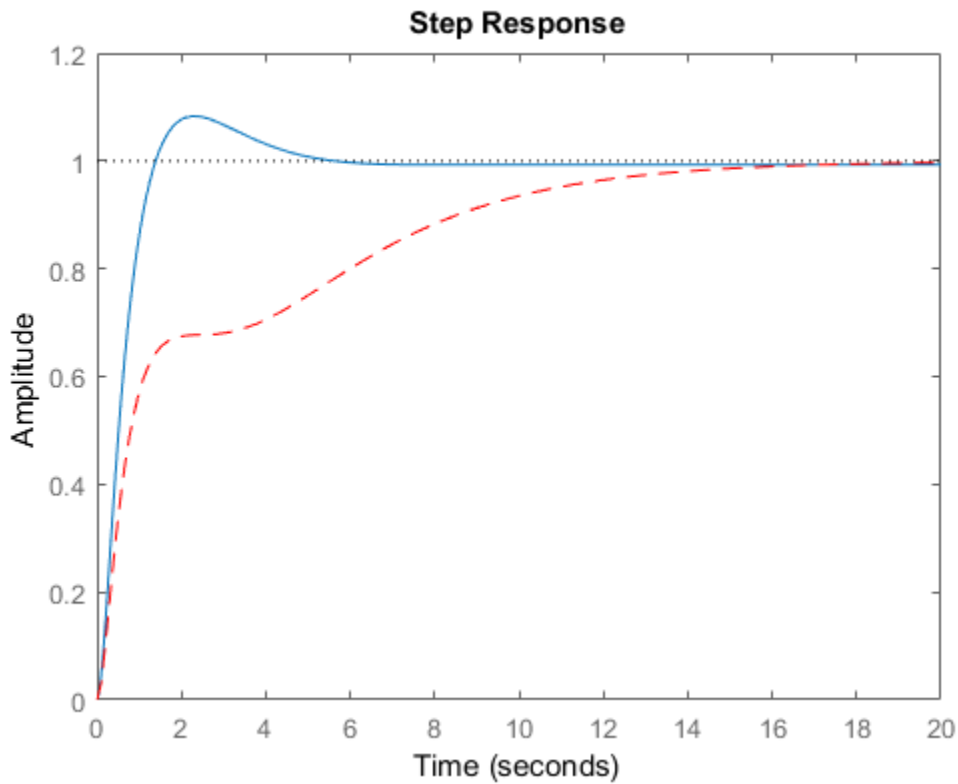
$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with  $K_p = 1.12$ ,  $K_i = 0.23$ ,  $K_d = 1.3$ ,  $T_f = 0.122$ ,  $b = 0.5$ ,  $c = 0.75$

Continuous-time 2-DOF PIDF controller in parallel form.

The PID gains and filter constant are still unchanged, but the setpoint weights now change the closed-loop response.

```
CM_2 = tf(C2_2);  
T2_2 = CM_2(1)*feedback(G, -CM_2(2));  
stepplot(T1,T2_2, 'r--')
```



## Input Arguments

**C1 — 1-DOF PID controller**  
pid object | pidstd object

1-DOF PID controller, specified as a `pid` object or a `pidstd` object.

**b — Setpoint weight on proportional term**

1 (default) | real nonnegative scalar

Setpoint weight on proportional term, specified as a real, nonnegative, finite value. If you do not specify `b`, then `C2` has `b = 1`.

**c — Setpoint weight on derivative term**

1 (default) | real nonnegative scalar

Setpoint weight on derivative term, specified as a real, nonnegative, finite value. If you do not specify `c`, then `C2` has `c = 1`.

## Output Arguments

**C2 — 2-DOF PID controller**

`pid2` object | `pidstd2` object

2-DOF PID controller, returned as a `pid2` object or `pidstd2` object. `C2` is in parallel form if `C1` is in parallel form, and standard form if `C1` is in standard form.

For example, suppose `C1` is a continuous-time, parallel-form `pid` controller of the form:

$$C_1 = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}.$$

Then `C2` is a parallel-form 2-DOF `pid2` controller, which has two inputs and one output. The relationship between the inputs,  $r$  and  $y$ , and the output  $u$  of `C2` is given by:

$$u = K_p (br - y) + \frac{K_i}{s} (r - y) + \frac{K_d s}{T_f s + 1} (cr - y).$$

The PID gains  $K_p$ ,  $K_i$ , and  $K_d$ , and the filter time constant  $T_f$  are unchanged. The setpoint weights  $b$  and  $c$  are specified by the input arguments `b` and `c`, or 1 by default. For more information about 2-DOF PID controllers, see “Two-Degree-of-Freedom PID Controllers”.

The conversion also preserves the values of the properties `Ts`, `TimeUnit`, `SamplingGrid`, `IFormula`, and `DFormula`.

## More About

- “Two-Degree-of-Freedom PID Controllers”

## See Also

`getComponents` | `make1DOF` | `pid` | `pid2` | `pidstd` | `pidstd2`

**Introduced in R2015b**



# margin

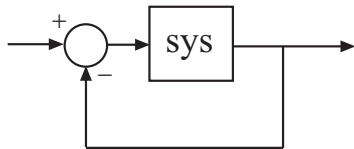
Gain margin, phase margin, and crossover frequencies

## Syntax

```
[Gm, Pm, Wgm, Wpm] = margin(sys)
[Gm, Pm, Wgm, Wpm] = margin(mag, phase, w)
margin(sys)
```

## Description

`margin` calculates the minimum gain margin,  $G_m$ , phase margin,  $P_m$ , and associated frequencies  $W_{gm}$  and  $W_{pm}$  of SISO open-loop models. The gain and phase margin of a system `sys` indicates the relative stability of the closed-loop system formed by applying unit negative feedback to `sys`, as in the following illustration.



The gain margin is the amount of gain increase or decrease required to make the loop gain unity at the frequency  $W_{gm}$  where the phase angle is  $-180^\circ$  (modulo  $360^\circ$ ). In other words, the gain margin is  $1/g$  if  $g$  is the gain at the  $-180^\circ$  phase frequency. Similarly, the phase margin is the difference between the phase of the response and  $-180^\circ$  when the loop gain is 1.0. The frequency  $W_{pm}$  at which the magnitude is 1.0 is called the *unity-gain frequency* or *gain 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, Wgm, Wpm] = margin(sys)` computes the gain margin  $G_m$ , the phase margin  $P_m$ , and the corresponding frequencies  $W_{gm}$  and  $W_{pm}$ , given the SISO open-loop dynamic system model `sys`.  $W_{gm}$  is the frequency where the gain margin is measured, which is

a  $-180$  degree phase crossing frequency. `Wpm` is the frequency where the phase margin is measured, which is a 0dB gain crossing frequency. These frequencies are expressed in radians/`TimeUnit`, where `TimeUnit` is the unit specified in the `TimeUnit` property of `sys`. When `sys` has several crossovers, `margin` returns the smallest gain and phase margins and corresponding frequencies.

The phase margin `Pm` is in degrees. The gain margin `Gm` is an absolute magnitude. You can compute the gain margin in dB by

$$Gm\_dB = 20 * \log_{10}(Gm)$$

`[Gm,Pm,Wgm,Wpm] = margin(mag,phase,w)` derives the gain and phase margins from Bode frequency response data (magnitude, phase, and frequency vector). `margin` interpolates between the frequency points to estimate the margin values. Provide the gain data `mag` in absolute units, and phase data `phase` in degrees. You can provide the frequency vector `w` in any units; `margin` returns `Wgm` and `Wpm` in the same units.

---

**Note:** When you use `margin(mag,phase,w)`, `margin` relies on interpolation to approximate the margins, which generally produces less accurate results. For example, if there is no 0 dB crossing within the `w` range, `margin` returns a phase margin of `Inf`. Therefore, if you have an analytical model `sys`, using `[Gm,Pm,Wgm,Wpm] = margin(sys)` is the most robust way to obtain the margins.

---

`margin(sys)`, without output arguments, plots the Bode response of `sys` on the screen and indicates the gain and phase margins on the plot. By default, gain margins are expressed in dB on the plot.

## Examples

### Gain and Phase Margins of Open-Loop Transfer Function

Create an open-loop discrete-time transfer function.

```
hd = tf([0.04798 0.0464],[1 -1.81 0.9048],0.1)
```

```
hd =
```

```
0.04798 z + 0.0464
```

```
-----  
z^2 - 1.81 z + 0.9048
```

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

Compute the gain and phase margins.

```
[Gm,Pm,Wgm,Wpm] = margin(hd)
```

```
Gm =
```

```
2.0517
```

```
Pm =
```

```
13.5711
```

```
Wgm =
```

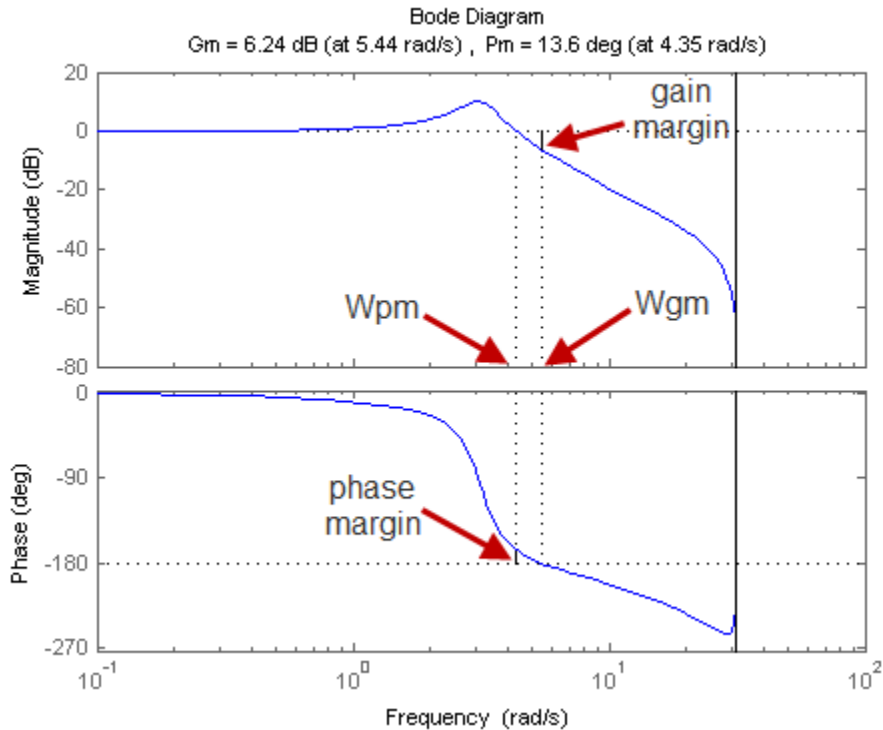
```
5.4374
```

```
Wpm =
```

```
4.3544
```

Display the gain and phase margins graphically.

```
margin(hd)
```



Solid vertical lines mark the gain margin and phase margin. The dashed vertical lines indicate the locations of  $W_{pm}$ , the frequency where the phase margin is measured, and  $W_{gm}$ , the frequency where the gain margin is measured.

## More About

### Algorithms

The phase margin is computed using  $H_\infty$  theory, and the gain margin by solving  $H(j\omega) = \overline{H(j\omega)}$  for the frequency  $\omega$ .

### See Also

bode | Linear System Analyzer

**Introduced before R2006a**

## minreal

Minimal realization or pole-zero cancelation

### Syntax

```
sysr = minreal(sys)
sysr = minreal(sys,tol)
[sysr,u] = minreal(sys,tol)
... = minreal(sys,tol,false)
... = minreal(sys,[],false)
```

### 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  $(A,B,C)$

`... = minreal(sys,tol,false)` and `... = minreal(sys,[],false)` disable the verbose output of the function. By default, `minreal` displays a message indicating the number of states removed from a state-space model `sys`.

### Examples

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 `cloop`.

```
cloop =
```

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

Continuous-time zero/pole/gain model.

To cancel the pole-zero pair at  $s = 1$ , type

```
cloopmin = minreal(cloop)
```

This command produces the following result.

```
cloopmin =
```

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

Continuous-time zero/pole/gain model.

## More About

### Algorithms

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` | Model Reducer | `modred` | `sminreal`

**Introduced before R2006a**

## modred

Eliminate states from state-space models

### Syntax

```
rsys = modred(sys,elim)
rsys = modred(sys,elim,'method')
```

### Description

`rsys = modred(sys,elim)` reduces the order of a continuous or discrete state-space model `sys` by eliminating the states found in the vector `elim`. The full state vector  $X$  is partitioned as  $X = [X1;X2]$  where  $X1$  is the reduced state vector and  $X2$  is discarded.

`elim` can be a vector of indices or a logical vector commensurate with  $X$  where true values mark states to be discarded. This function is usually used in conjunction with `balreal`. Use `balreal` to first isolate states with negligible contribution to the I/O response. If `sys` has been balanced with `balreal` and the vector `g` of Hankel singular values has  $M$  small entries, you can use `modred` to eliminate the corresponding  $M$  states. For example:

```
[sys,g] = balreal(sys) % Compute balanced realization
elim = (g<1e-8) % Small entries of g are negligible states
rsys = modred(sys,elim) % Remove negligible states
```

`rsys = modred(sys,elim,'method')` also specifies the state elimination method. Choices for 'method' include

- 'MatchDC' (default): Enforce matching DC gains. The state-space matrices are recomputed as described in “Algorithms” on page 2-594.
- 'Truncate': Simply delete  $X2$ .

The 'Truncate' option tends to produce a better approximation in the frequency domain, but the DC gains are not guaranteed to match.

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`.



## Examples

### Order Reduction by Matched-DC-Gain and Direct-Deletion Methods

Consider the following 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`.

```
h = tf([1 11 36 26],[1 14.6 74.96 153.7 99.65]);
[hb,g] = balreal(h);
```

Examine the gramians.

```
g'
```

```
ans =
```

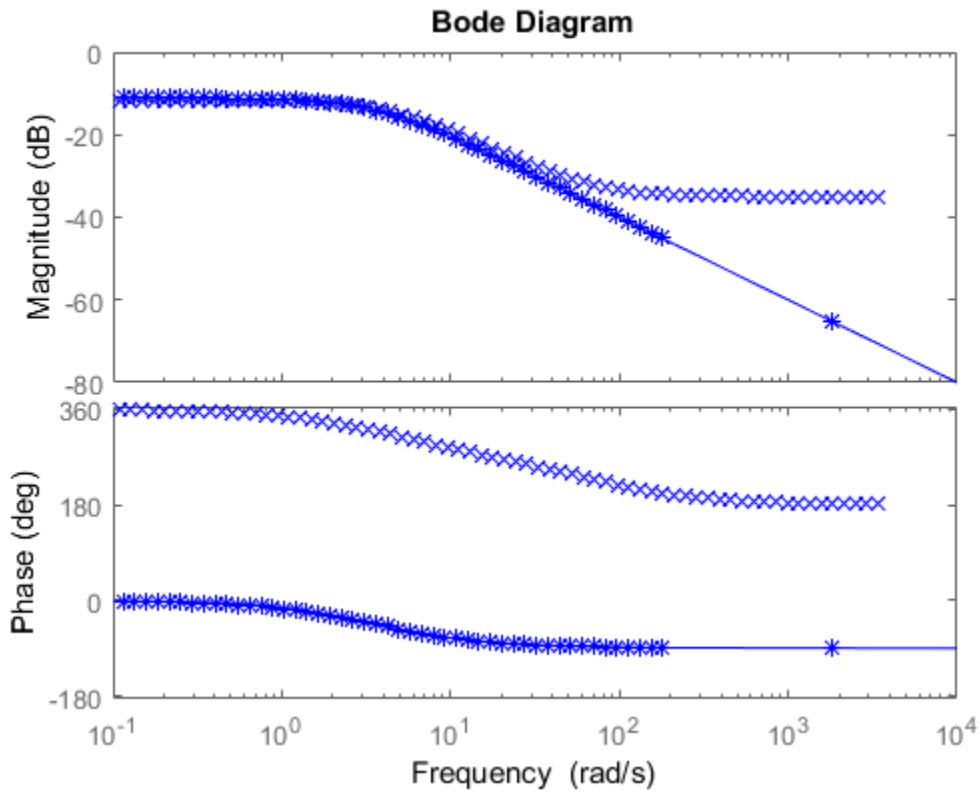
```
    0.1394    0.0095    0.0006    0.0000
```

The last three diagonal entries of the balanced gramians are relatively small. Eliminate these three least-contributing states with `modred`, using both matched-DC-gain and direct-deletion methods.

```
hmdc = modred(hb,2:4,'MatchDC');
hdel = modred(hb,2:4,'Truncate');
```

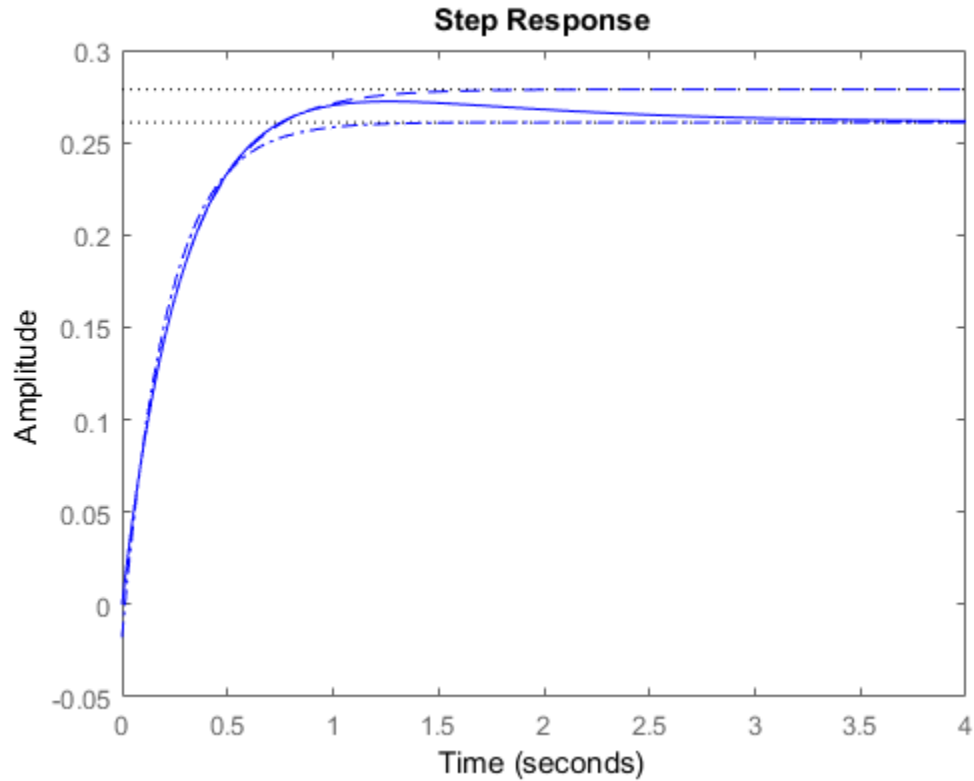
Both `hmdc` and `hdel` are first-order models. Compare their Bode responses against that of the original model.

```
bodeplot(h, '-', hmdc, 'x', hdel, '*')
```



The reduced-order model `hdel` is clearly a better frequency-domain approximation of `h`. Now compare the step responses.

```
stepplot(h, '-', hmdc, '-.-', hdel, '-.-')
```



While `hdel` accurately reflects the transient behavior, only `hmdc` gives the true steady-state response.

## 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.

## More About

### Algorithms

The algorithm for the matched DC gain method is as follows. For continuous-time models

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

the state vector is partitioned into  $x_1$ , to be kept, and  $x_2$ , to be eliminated.

$$\begin{aligned}\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} u \\ y &= [C_1 \quad C_2]x + Du\end{aligned}$$

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

$$\begin{aligned}\dot{x}_1 &= [A_{11} - A_{12}A_{22}^{-1}A_{21}]x_1 + [B_1 - A_{12}A_{22}^{-1}B_2]u \\ y &= [C_1 - C_2A_{22}^{-1}A_{21}]x + [D - C_2A_{22}^{-1}B_2]u\end{aligned}$$

The discrete-time case is treated similarly by setting

$$x_2[n+1] = x_2[n]$$

### See Also

balreal | minreal

Introduced before R2006a

# Model Reducer

Reduce complexity of linear time-invariant (LTI) models

## Description

The **Model Reducer** app lets you compute reduced-order approximations of high-order models. Working with lower-order models can simplify analysis and control design. Simpler models are also easier to understand and manipulate. You can reduce a plant model to focus on relevant dynamics before designing a controller for the plant. Or, you can use model reduction to simplify a full-order controller.

Using any of the following methods, **Model Reducer** helps you reduce model order while preserving model characteristics that are important to your application:

- **Balanced Truncation** — Remove states with relatively small energy contributions.
- **Mode Selection** — Select modes by specifying a region of interest in the complex plane.
- **Pole-Zero Simplification** — Eliminate canceling or near-canceling pole-zero pairs.

**Model Reducer** provides response plots and error plots to help ensure that the reduced-order model preserves important dynamics.

## Open the Model Reducer App

- **MATLAB Toolstrip:** On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- **MATLAB command prompt:** Enter `modelReducer`.

## Examples

- “Reduce Model Order Using the Model Reducer App”
- “Pole-Zero Simplification”
- “Balanced Truncation Model Reduction”
- “Mode-Selection Model Reduction”

## Parameters

### Balanced Truncation Tab

#### **Model** — Currently selected model for reduction

model name

Specify the model you want to reduce by selecting from the **Model** drop-down list. The list includes all models currently in the **Data Browser**. To get a model from the MATLAB workspace into the **Data Browser**, on the **Model Reducer** tab, click



**Import Model**.

#### **Reduced model orders** — Number of states in reduced model

integer | integer array

Specify the number of states in the reduced-order model. Any value is permitted that falls between the number of unstable states in the model and the number of states in a minimal realization of the system (see `minreal`). If you specify a single value, **Model Reducer** computes and displays the responses of a model of that order. If you specify multiple values, **Model Reducer** computes models of all specified orders and displays their responses on the same plot. To store reduced models in the **Data Browser**, click



For more information, see “Balanced Truncation Model Reduction”.

Example: 5

Example: 4:7

Example: [3,7,10]

#### **Preserve DC Gain** — Match DC gain of reduced model to original model

checked (default) | unchecked

When **Preserve DC Gain** is checked, the DC gain of the reduced model equals the DC gain of the original model. When the DC behavior of the model is important in your application, leave this option checked. Uncheck the option to get better matching of higher-frequency behavior.

For more information, see “Balanced Truncation Model Reduction”.

**Select frequency range — Limit analysis to specified frequencies**

unchecked (default) | checked

By default, **Model Advisor** analyzes Hankel singular values across all frequencies. Such a limit is useful when you know the model has modes outside the region of interest to your particular application. When you apply a frequency limit, **Model Reducer** determines which states are the low-energy states to truncate based on their energy contribution within the specified frequency range only.

To limit the analysis of state contributions to a particular frequency range, check **Select frequency range**. Then enter a frequency range in the text box as a vector of the form  $[f_{min}, f_{max}]$ . Units are rad/TimeUnit, where TimeUnit is the TimeUnit property of the model you are reducing.

**Mode Selection Tab****Model** — Currently selected model for reduction

model name

Specify the model you want to reduce by selecting from the **Model** drop-down list. The list includes all models currently in the **Data Browser**. To get a model from the MATLAB workspace into the **Data Browser**, on the **Model Reducer** tab, click

**Import Model**.

For more information, see “Mode-Selection Model Reduction”.

**Lower Cutoff — Lowest mode frequency**

positive scalar

Enter the frequency of the slowest dynamics to preserve in the reduced model. Poles with natural frequency below this cutoff are eliminated from the reduced model.

**Upper Cutoff — Highest mode frequency**

positive scalar

Enter the frequency of the fastest dynamics to preserve in the reduced model. Poles with natural frequency above this cutoff are eliminated from the reduced model.

**Pole/Zero Simplification Tab****Model** — Currently selected model for reduction

model name

Specify the model you want to reduce by selecting from the **Model** drop-down list. The list includes all models currently in the **Data Browser**. To get a model from the

MATLAB workspace into the **Data Browser**, on the **Model Reducer** tab, click  **Import Model**.

**Simplification of Pole-Zero Pairs — Tolerance for pole-zero cancellation**  
positive scalar

Set the tolerance for pole-zero cancellation by using the slider or entering a value in the text box. The value determines how close together a pole and zero must be for **Model Reducer** to eliminate them from the reduced model. Moving the slider to the left or entering a smaller value in the text box simplifies the model less, by cancelling fewer poles and zeros. Moving the slider to the right, or entering a larger value, simplifies the model more by cancelling poles and zeros that are further apart.

For more information, see “Pole-Zero Simplification”.

### Programmatic Use

`modelReducer` opens the **Model Reducer** app with no models in the **Data Browser**.


To import a model from the MATLAB workspace, click  **Import Model**.

`modelReducer(model)` opens app and imports the specified LTI model. `model` can be a:

- `tf`, `ss`, or `zpk` model that is proper. The model can be SISO or MIMO, and continuous or discrete. Continuous-time models must not have time delays. (See `pade` for information about approximating time delays in continuous-time models.)
- Generalized model such as a `genss` model. The **Model Reducer** app uses the current or nominal value of all control design blocks in `model` (see `getValue`).

`modelReducer(model1, ..., modelN)` opens the app and imports the specified models.

`modelReducer(sessionFile)` opens the app and loads a previously saved session. `sessionFile` is the name of a session data file in the current working directory or on the MATLAB path.

To save session data to disk, in the **Model Reducer** app, on the **Model Reducer** tab, click  **Save Session**. The saved session data includes the current plot configuration and all models in the **Data Browser**.



## See Also

### Functions

balred | freqsep | minreal

**Introduced in R2016a**

## modsep

Region-based modal decomposition

### Syntax

```
[H,H0] = modsep(G,N,REGIONFCN)
MODSEP(G,N,REGIONFCN,PARAM1,...)
```

### Description

[H,H0] = modsep(G,N,REGIONFCN) decomposes the LTI model G into a sum of n simpler models H<sub>j</sub> with their poles in disjoint regions R<sub>j</sub> of the complex plane:

$$G(s) = H0 + \sum_{j=1}^N H_j(s)$$

G can be any LTI model created with `ss`, `tf`, or `zpk`, and N is the number of regions used in the decomposition. `modsep` packs the submodels H<sub>j</sub> into an LTI array H and returns the static gain H0 separately. Use H(:, :, j) to retrieve the submodel H<sub>j</sub>(s).

To specify the regions of interest, use a function of the form

```
IR = REGIONFCN(p)
```

that assigns a region index IR between 1 and N to a given pole p. You can specify this function by its name or as a function handle, and use the syntax MODSEP(G,N,REGIONFCN,PARAM1,...) to pass extra input arguments:

```
IR = REGIONFCN(p,PARAM1,...)
```

### Examples

To decompose G into  $G(z) = H0 + H1(z) + H2(z)$  where H1 and H2 have their poles inside and outside the unit disk respectively, use

```
[H,H0] = modsep(G,2,@udsep)
```

where the function `udsep` is defined by

```
function r = udsep(p)
if abs(p)<1, r = 1; % assign r=1 to poles inside unit disk
else      r = 2; % assign r=2 to poles outside unit disk
end
```

To extract  $H_1(z)$  and  $H_2(z)$  from the LTI array  $H$ , use

```
H1 = H(:,:,1); H2 = H(:,:,2);
```

## **See Also**

`stabsep`

**Introduced before R2006a**

# nblocks

Number of blocks in Generalized matrix or Generalized LTI model

## Syntax

`N = nblocks(M)`

## Description

`N = nblocks(M)` returns the number of “Control Design Blocks” in the Generalized LTI model or Generalized matrix `M`.

## Input Arguments

**M**

A Generalized LTI model (`genss` or `genfrd` model), a Generalized matrix (`genmat`), or an array of such models.

## Output Arguments

**N**

The number of “Control Design Blocks” in `M`. If a block appears multiple times in `M`, `N` reflects the total number of occurrences.

If `M` is a model array, `N` is an array with the same dimensions as `M`. Each entry of `N` is the number of Control Design Blocks in the corresponding entry of `M`.

## Examples

### Number of Control Design Blocks in a Second-Order Filter Model

This example shows how to use `nblocks` to examine two different ways of parametrizing a model of a second-order filter.

- 1 Create a tunable (parametric) model of the second-order filter:

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

where the damping  $\zeta$  and the natural frequency  $\omega_n$  are tunable parameters.

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

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

- 2 Examine the number of tunable blocks in the model using `nblocks`.

```
nblocks(F)
```

This command returns the result:

```
ans =
```

```
6
```

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

- 3 Rewrite F for fewer occurrences of `wn`.

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

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

Use this expression to create the tunable filter:

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

- 4 Examine the number of tunable blocks in the new filter model.

```
nblocks(F)
```

This command returns the result:

```
ans =
```

```
4
```

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

### More About

- “Control Design Blocks”
- “Generalized Matrices”
- “Generalized and Uncertain LTI Models”

### See Also

`genss` | `genfrd` | `genmat` | `getValue`

**Introduced in R2011a**

## ndBasis

Basis functions for tunable gain surface

You use basis function expansions to parameterize gain surfaces for tuning gain-scheduled controllers, with the `tunableSurface` command. The complexity of such expansions grows quickly when you have multiple scheduling variables. Use `ndBasis` to build N-dimensional expansions from low-dimensional expansions. `ndBasis` is analogous to `ndgrid` in the way it spatially replicates the expansions along each dimension.

## Syntax

```
shapefcn = ndBasis(F1,F2)
shapefcn = ndBasis(F1,F2,...,FN)
```

## Description

`shapefcn = ndBasis(F1,F2)` forms the outer (tensor) product of two basis function expansions. Each basis function expansion is a function that returns a vector of expansion terms, such as returned by `polyBasis`. If

$$F_1(x_1) = [F_{1,1}(x_1), F_{1,2}(x_1), \dots, F_{1,i}(x_1)] \text{ and } F_2(x_2) = [F_{2,1}(x_2), F_{2,2}(x_2), \dots, F_{2,i}(x_2)],$$

then `shapefcn` is a vector of terms of the form:

$$F_{ij} = F_{1,i}(x_1) F_{2,j}(x_2).$$

The terms are listed in a column-oriented fashion, with  $i$  varying first, then  $j$ .

`shapefcn = ndBasis(F1,F2,...,FN)` forms the outer product of three or more basis function expansions. The terms in the vector returned by `shapefcn` are of the form:

$$F_{i_1 \dots i_N} = F_{1,i_1}(x_1) F_{2,i_2}(x_2) \dots F_{N,i_N}(x_N).$$

These terms are listed in sort order that of an N-dimensional array, with  $i_1$  varying first, then  $i_2$ , and so on. Each  $F_j$  can itself be a multi-dimensional basis function expansion.

## Examples

### Polynomial Basis Functions of Two Variables

Create a two-dimensional basis of polynomial functions to second-order in both variables.

Define a one-dimensional set of basis functions.

```
F = @(x)[x,x^2];
```

Equivalently, you can use `polyBasis` to create F.

```
F = polyBasis('canonical',2);
```

Generate a two-dimensional expansion from F.

```
F2D = ndBasis(F,F);
```

F2D is a function of two variables. The function returns a vector containing the evaluated basis functions of those two variables:

$$F2D(x,y) = [x, x^2, y, yx, yx^2, y^2, xy^2, x^2y^2].$$

To confirm this, evaluate F2D for  $x = 0.2$ ,  $y = -0.3$ .

```
F2D(0.2, -0.3)
```

```
ans =
```

```
Columns 1 through 7
```

```
0.2000    0.0400   -0.3000   -0.0600   -0.0120    0.0900    0.0180
```

```
Column 8
```

```
0.0036
```

The expansion you combine with `ndBasis` need not have the same order. For instance, combine F with first-order expansion in one variable.

```
G = @(y)[y];  
F2D2 = ndBasis(F,G);
```



The array returned by F2D2 is similar to that returned by F2D, without the terms that are quadratic in the second variable.

$$F2D2(x, y) = [x, x^2, y, yx, yx^2].$$

Evaluate F2D2 for for  $x = 0.2$ ,  $y = -0.3$  to confirm the order of terms.

```
F2D2(0.2, -0.3)
```

```
ans =
```

```
0.2000    0.0400   -0.3000   -0.0600   -0.0120
```

### Mixed Multi-Dimensional Basis Functions

Create a set of two-dimensional basis functions where the expansion is quadratic in one variable and periodic in the other variable.

First generate the one-dimensional expansions.

```
F1 = polyBasis('canonical',2);
F2 = fourierBasis(1);
```

For simplicity, this example takes only the first harmonic of the periodic variation. These expansions have basis functions given by:

$$F1(x) = [x, x^2], \quad F2(y) = [\cos(\pi y), \sin(\pi y)].$$

Create the two-dimensional basis function expansion.

```
F = ndBasis(F1,F2);
```

The array returned by F includes all multiplicative combinations of the basis functions:

$$F(x, y) = [x, x^2, \cos(\pi y), \cos(\pi y)x, \cos(\pi y)x^2, \sin(\pi y), x \sin(\pi y), x^2 \sin(\pi y)].$$

To confirm this, evaluate F for  $x = 0.2$ ,  $y = -0.3$ .

```
F(0.2, -0.3)
```

```
ans =
```

```
Columns 1 through 7
    0.2000    0.0400    0.5878    0.1176    0.0235   -0.8090   -0.1618
Column 8
   -0.0324
```

## Input Arguments

### **F** — Basis function expansion

function handle

Basis function expansion, specified as a function handle. The function must return a vector of basis functions of one or more scheduling variables. You can define these basis functions explicitly, or using `polyBasis` or `fourierBasis`.

Example: `F = @(x) [x, x^2, x^3]`

Example: `F = polyBasis(3,2)`

## Output Arguments

### **shapefcn** — Basis function expansion

function handle

Basis function expansion, specified as a function handle. `shapefcn` takes as input arguments the total number of variables in `F1`, `F2`, ..., `FN`. It returns a vector of functions of those variables, defined on the interval  $[-1,1]$  for each input variable. When you use `shapefcn` to create a gain surface, `tunableSurface` automatically generates tunable coefficients for each term in the vector.

## More About

### Tips

- The `ndBasis` operation is associative:

`ndBasis(F1,ndBasis(F2,F3)) = ndBasis(ndBasis(F1,F2),F3) = ndBasis(F1,F2,F3)`

## See Also

`fourierBasis` | `polyBasis` | `tunableSurface`

**Introduced in R2015b**

## ndims

Query number of dimensions of dynamic system model or model array

### Syntax

```
n = ndims(sys)
```

### Description

`n = ndims(sys)` is the number of dimensions of a dynamic system model or a model array `sys`. A single model has two dimensions (one for outputs, and one for inputs). A model 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))
```

### Examples

```
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`

Introduced before R2006a

# ngrid

Superimpose Nichols chart on 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.

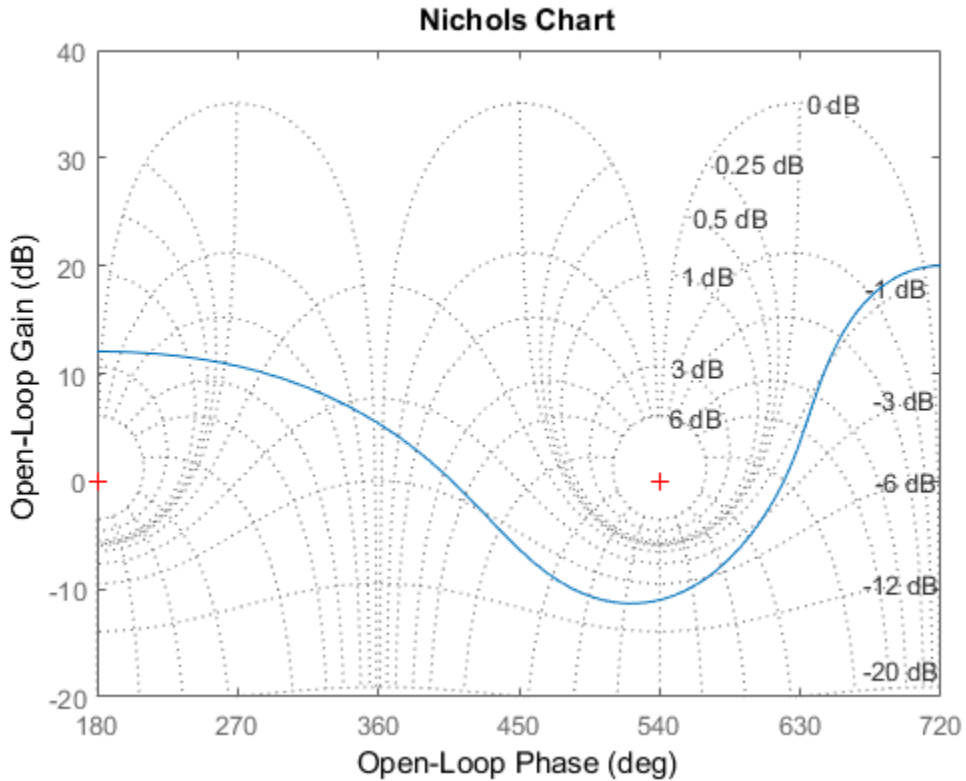
## Examples

### Nichols Response with Nichols Grid Lines

Plot the Nichols response with Nichols grid lines for the following system:

$$H(s) = \frac{-4s^4 + 48s^3 - 18s^2 + 250s + 600}{s^4 + 30s^3 + 282s^2 + 525s + 60}.$$

```
H = tf([-4 48 -18 250 600],[1 30 282 525 60]);  
nichols(H)  
ngrid
```



The right-click menu for Nichols charts includes the **Tight** option under **Zoom**. You can use this to clip unbounded branches of the Nichols chart.

### See Also

nichols

Introduced before R2006a

# nichols

Nichols chart of frequency response

## 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` creates a Nichols chart of the frequency response. A Nichols chart displays the magnitude (in dB) plotted against the phase (in degrees) of the system response. Nichols charts 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 chart.

`nichols(sys)` creates a Nichols chart of the dynamic system `sys`. This model can be continuous or discrete, SISO or MIMO. In the MIMO case, `nichols` produces an array of Nichols charts, 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)` specifies the frequency range or frequency points to be used for the chart. 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 must be in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

`nichols(sys1,sys2,...,sysN)` or `nichols(sys1,sys2,...,sysN,w)` superimposes the Nichols charts of several 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.

`[mag,phase,w] = nichols(sys)` or `[mag,phase] = nichols(sys,w)` returns the magnitude and phase (in degrees) of the frequency response at the frequencies `w` (in rad/TimeUnit). The outputs `mag` and `phase` are 3-D arrays similar to those produced by `bode` (see the `bode` reference page). They have dimensions (number of outputs) × (number of inputs) × (length of `w`)

## Examples

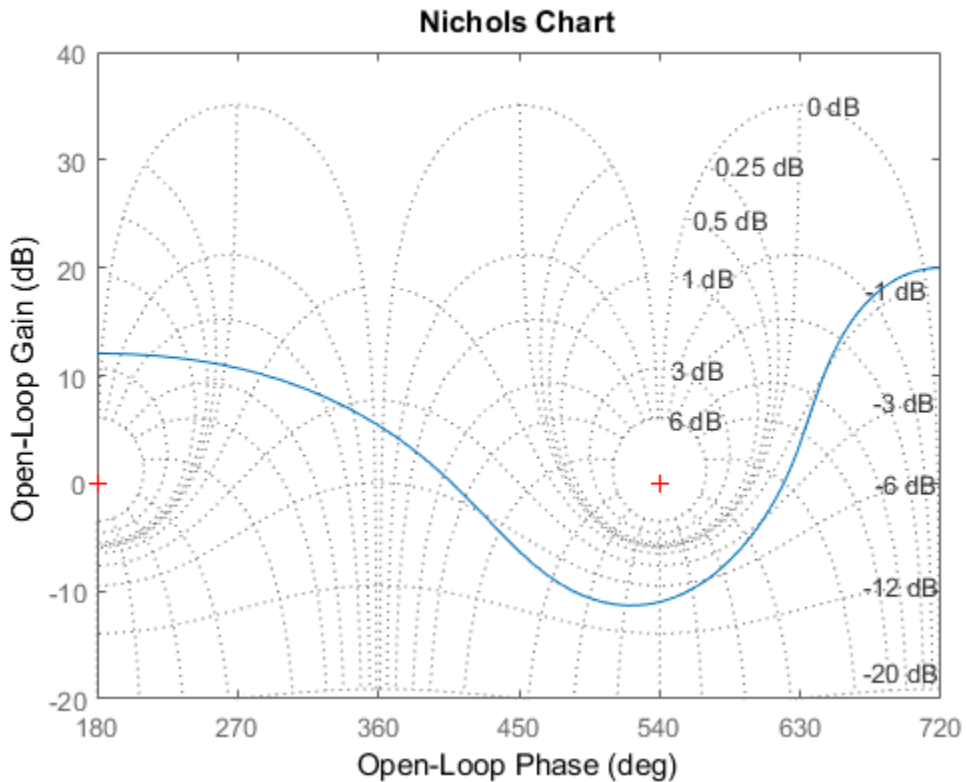
### Nichols Response with Nichols Grid Lines

Plot the Nichols response with Nichols grid lines for the following system:

$$H(s) = \frac{-4s^4 + 48s^3 - 18s^2 + 250s + 600}{s^4 + 30s^3 + 282s^2 + 525s + 60}$$

```
H = tf([-4 48 -18 250 600],[1 30 282 525 60]);  
nichols(H)  
ngrid
```





The right-click menu for Nichols charts includes the **Tight** option under **Zoom**. You can use this to clip unbounded branches of the Nichols chart.

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### Algorithms

See `bode`.

**See Also**

bode | evalfr | freqresp | Linear System Analyzer | ngrid | nyquist | sigma

**Introduced before R2006a**

# nicholsoptions

Create list of Nichols plot options

## Syntax

```
P = nicholsoptions
P = nicholsoptions('cstprefs')
```

## Description

`P = nicholsoptions` returns a list of available options for Nichols plots with default values set. You can use these options to customize the Nichols plot appearance from the command line.

`P = nicholsoptions('cstprefs')` initializes the plot options with the options you selected in the Control System Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation.

This table summarizes the Nichols plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid Specified as one of: 'off'   'on' <b>Default:</b> 'off'
GridColor	Color of the grid lines Specified as one of: Vector of RGB values in the range [0,1]   color shorthand such as 'k' or 'r'   'none'. <b>Default:</b> [0.15,0.15,0.15]
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs

Option	Description
	Specified as one of: 'none'   'inputs'   'outputs'   'all' <b>Default:</b> 'none'
InputLabels, OutputLabels	Input and output label styles.
InputVisible, OutputVisible	Visibility of input and output channels
FreqUnits	Frequency units, specified as one of: <ul style="list-style-type: none"> <li>• 'Hz'</li> <li>• 'rad/second'</li> <li>• 'rpm'</li> <li>• 'kHz'</li> <li>• 'MHz'</li> <li>• 'GHz'</li> <li>• 'rad/nanosecond'</li> <li>• 'rad/microsecond'</li> <li>• 'rad/millisecond'</li> <li>• 'rad/minute'</li> <li>• 'rad/hour'</li> <li>• 'rad/day'</li> <li>• 'rad/week'</li> <li>• 'rad/month'</li> <li>• 'rad/year'</li> <li>• 'cycles/nanosecond'</li> <li>• 'cycles/microsecond'</li> <li>• 'cycles/millisecond'</li> <li>• 'cycles/hour'</li> <li>• 'cycles/day'</li> <li>• 'cycles/week'</li> <li>• 'cycles/month'</li> <li>• 'cycles/year'</li> </ul>

Option	Description
MagLowerLimMode	Enables a lower magnitude limit Specified as one of: 'auto'   'manual' <b>Default:</b> 'auto'
MagLowerLim	Specifies the lower magnitude limit
PhaseUnits	Phase units Specified as one of: 'deg'   'rad' <b>Default:</b> 'deg'
PhaseWrapping	Enables phase wrapping Specified as one of: 'on'   'off' <b>Default:</b> 'off'
PhaseWrappingBranch	Phase value at which the plot wraps accumulated phase when PhaseWrapping is set to 'on'. <b>Default:</b> -180 (phase wraps into the interval [-180°,180°))
PhaseMatching	Enables phase matching Specified as one of: 'on'   'off' <b>Default:</b> 'off'
PhaseMatchingFreq	Frequency for matching phase
PhaseMatchingValue	The value to make the phase responses close to

## Examples

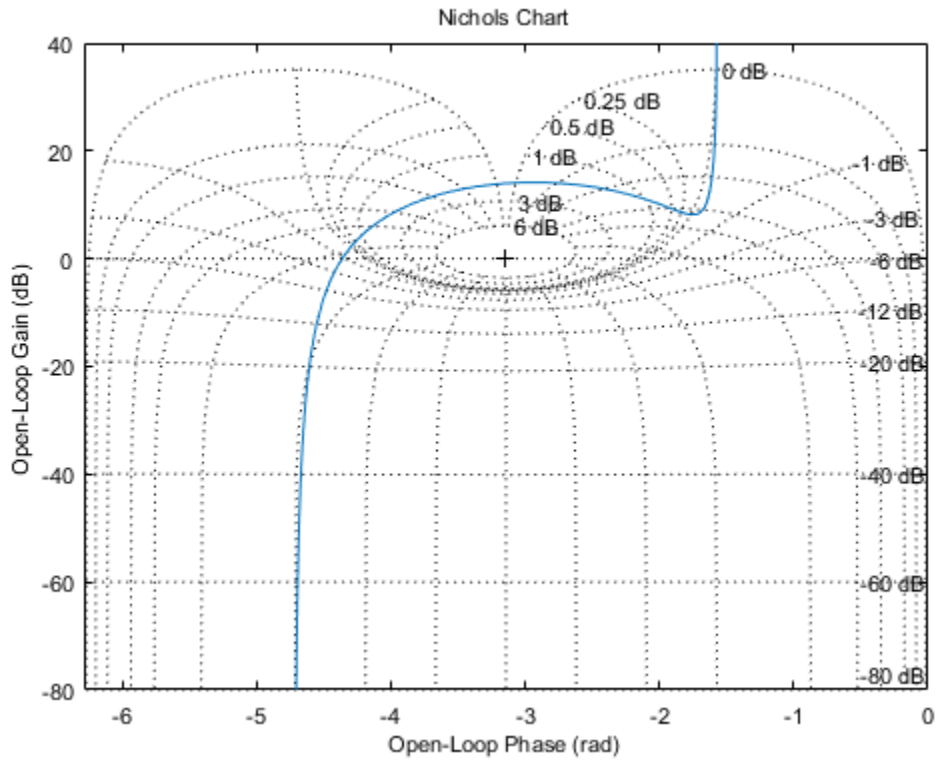
### Set Options for Nichols Plot

Create an options set, and set the phase units and grid option.

```
P = nicholsoptions;
P.PhaseUnits = 'rad';
P.Grid = 'on';
```

Use the options set to generate a Nichols plot. Note the phase units and grid in the plot.

```
h = nicholsplot(tf(1,[1,.2,1,0]),P);
```



### See Also

`getoptions` | `nicholsplot` | `setoptions`

Introduced in R2008a

# nicholsplot

Plot Nichols frequency responses and return plot handle

## Syntax

```
h = nicholsplot(sys)
nicholsplot(sys,{wmin,wmax})
nicholsplot(sys,w)
nicholsplot(sys1,sys2,...,w)
nicholsplot(AX,...)
nicholsplot(..., plotoptions)
```

## Description

`h = nicholsplot(sys)` draws the Nichols plot of the dynamic system `sys`. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help nicholsoptions
```

for a list of available plot options.

The frequency range and number of points are chosen automatically. See `bode` for details on the notion of frequency in discrete time.

`nicholsplot(sys,{wmin,wmax})` draws the Nichols plot for frequencies between `wmin` and `wmax` (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`).

`nicholsplot(sys,w)` uses the user-supplied vector `w` of frequencies, in `rad/TimeUnit`, at which the Nichols response is to be evaluated. See `logspace` to generate logarithmically spaced frequency vectors.

`nicholsplot(sys1,sys2,...,w)` draws the Nichols plots of multiple models `sys1,sys2,...` on a single plot. The frequency vector `w` is optional. You can also specify a color, line style, and marker for each system, as in

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

`nicholsplot(AX, ...)` plots into the axes with handle `AX`.

`nicholsplot(..., plotoptions)` plots the Nichols plot with the options specified in `plotoptions`. Type

`help nicholsoptions`

for more details.

## Examples

Generate Nichols plot and use plot handle to change frequency units to Hz

```
sys = rss(5);  
h = nicholsplot(sys);  
% Change units to Hz  
setoptions(h, 'FreqUnits', 'Hz');
```

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`getoptions` | `nichols` | `nicholsoptions` | `setoptions`

**Introduced before R2006a**



# nmodels

Number of models in model array

## Syntax

```
N = nmodels(sysarray)
```

## Description

`N = nmodels(sysarray)` returns the number of models in an array of dynamic system models or static models.

## Examples

### Confirm Number of Models in Array

Create a 2-by-3-by-4 array of state-space models.

```
sysarr = rss(2,2,2,2,3,4);
```

Confirm the number of models in the array.

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

```
N =
```

```
    24
```

## Input Arguments

**sysarray** — Input model array  
model array

Input model array, specified as an array of input-output models such as numeric LTI models, generalized models, or identified LTI models.

### Output Arguments

**N — Number of models in array**

positive integer

Number of models in the input model array, returned as a positive integer.

### See Also

`ndims` | `size`

**Introduced in R2013a**

## norm

Norm of linear model

### Syntax

```
n = norm(sys)
n = norm(sys,2)
n = norm(sys,inf)
[n,fpeak] = norm(sys,inf)
[...] = norm(sys,inf,tol)
```

### Description

`n = norm(sys)` or `n = norm(sys,2)` return the  $H_2$  norm of the linear dynamic system model `sys`.

`n = norm(sys,inf)` returns the  $H_\infty$  norm of `sys`.

`[n,fpeak] = norm(sys,inf)` also returns the frequency `fpeak` at which the gain reaches its peak value.

`[...] = norm(sys,inf,tol)` sets the relative accuracy of the  $H_\infty$  norm to `tol`.

### Input Arguments

#### **sys**

Continuous- or discrete-time linear dynamic system model. `sys` can also be an array of linear models.

#### **tol**

Positive real value setting the relative accuracy of the  $H_\infty$  norm.

**Default:** 0.01

## Output Arguments

**n**

$H_2$  norm or  $H_\infty$  norm of the linear model `sys`.

If `sys` is an array of linear models, `n` is an array of the same size as `sys`. In that case each entry of `n` is the norm of each entry of `sys`.

**fpeak**

Frequency at which the peak gain of `sys` occurs.

## Examples

This example uses `norm` to compute the  $H_2$  and  $H_\infty$  norms of a discrete-time linear system.

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.

To compute the  $H_2$  norm of this transfer function, enter:

```
H = tf([1 -2.841 2.875 -1.004],[1 -2.417 2.003 -0.5488],0.1)
norm(H)
```

These commands return the result:

```
ans =
    1.2438
```

To compute the  $H_\infty$  infinity norm, enter:

```
[ninf,fpeak] = norm(H,inf)
```

This command returns the result:

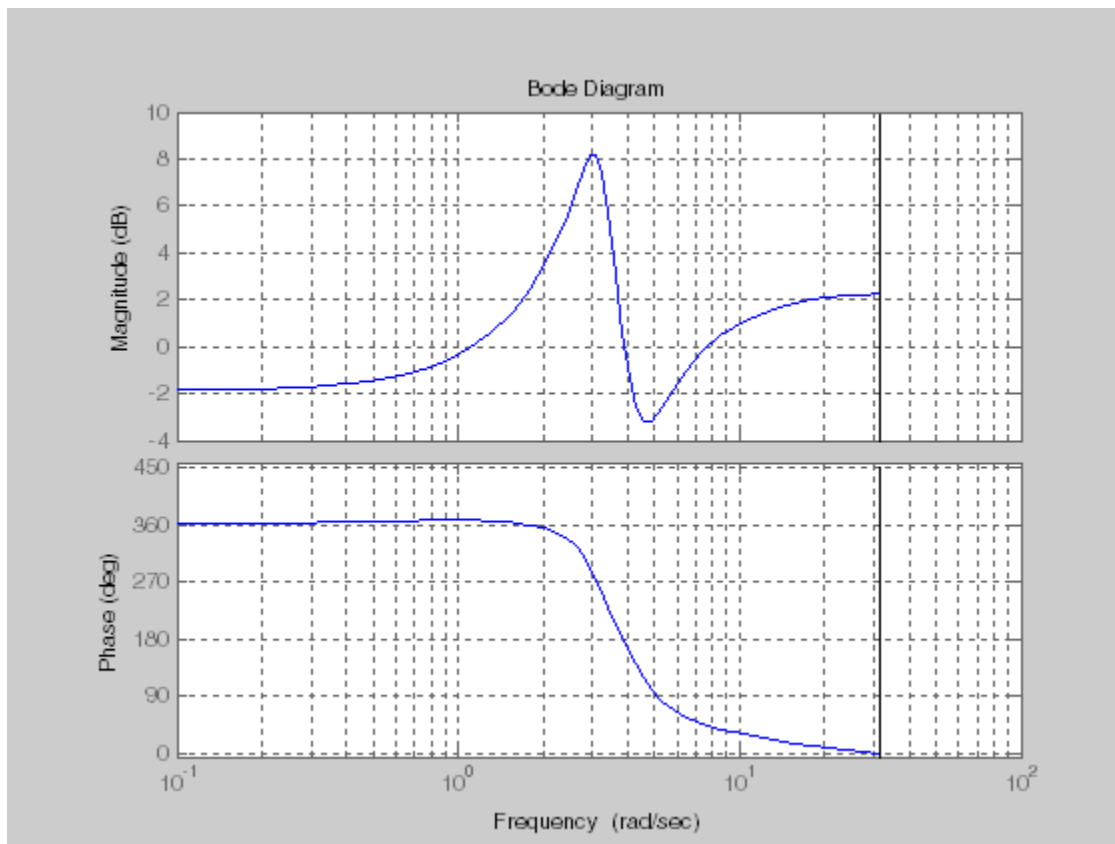
```
ninf =
```

2.5488

```
fpeak =
    3.0844
```

You can use a Bode plot of  $H(z)$  to confirm these values.

```
bode(H)
grid on;
```



The gain indeed peaks at approximately 3 rad/sec. To find the peak gain in dB, enter:

```
20*log10(ninf)
```

This command produces the following result:

ans =  
8.1268

## More About

### H2 norm

The  $H_2$  norm of a stable continuous-time system with transfer function  $H(s)$ , is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi} \int_{-\infty}^{\infty} \text{Trace} [H(j\omega)^H H(j\omega)] d\omega.}$$

For a discrete-time system with transfer function  $H(z)$ , the  $H_2$  norm is given by:

$$\|H\|_2 = \sqrt{\frac{1}{2\pi} \int_{-\pi}^{\pi} \text{Trace} [H(e^{j\omega})^H H(e^{j\omega})] d\omega.}$$

The  $H_2$  norm is equal to the root-mean-square of the impulse response of the system. The  $H_2$  norm measures the steady-state covariance (or power) of the output response  $y = Hw$  to unit white noise inputs  $w$ :

$$\|H\|_2^2 = \lim_{t \rightarrow \infty} E\{y(t)^T y(t)\}, \quad E(u(t)w(\tau)^T) = \delta(t - \tau) I.$$

The  $H_2$  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  $\omega = \infty$ ).

`norm(sys)` produces the same result as

`sqrt(trace(covar(sys,1)))`

### H-infinity norm

The  $H_\infty$  norm (also called the  $L_\infty$  norm) of a SISO linear system is the peak gain of the frequency response. For a MIMO system, the  $H_\infty$  norm is the peak gain across all input/output channels. Thus, for a continuous-time system  $H(s)$ , the  $H_\infty$  norm is given by:

$$\|H(s)\|_{\infty} = \max_{\omega} |H(j\omega)| \quad (\text{SISO})$$

$$\|H(s)\|_{\infty} = \max_{\omega} \sigma_{\max}(H(j\omega)) \quad (\text{MIMO})$$

where  $\sigma_{\max}(\cdot)$  denotes the largest singular value of a matrix.

For a discrete-time system  $H(z)$ :

$$\|H(z)\|_{\infty} = \max_{\theta \in [0, \pi]} |H(e^{j\theta})| \quad (\text{SISO})$$

$$\|H(z)\|_{\infty} = \max_{\theta \in [0, \pi]} \sigma_{\max}(H(e^{j\theta})) \quad (\text{MIMO})$$

The  $H_{\infty}$  norm is infinite if **sys** has poles on the imaginary axis (in continuous time), or on the unit circle (in discrete time).

## Algorithms

`norm` first converts **sys** to a state space model.

`norm` uses the same algorithm as `covar` for the  $H_2$  norm. For the  $H_{\infty}$  norm, `norm` uses the algorithm of [1]. `norm` computes the  $H_{\infty}$  norm (peak gain) using the SLICOT library. For more information about the SLICOT library, see <http://slicot.org>.

## References

- [1] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the  $H_{\infty}$ -Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

## See Also

`freqresp` | `sigma`

Introduced before R2006a

## nyquist

Nyquist plot of frequency response

### 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)
[re,im,w,sdre,sdim] = nyquist(sys)
```

### Description

`nyquist` creates a Nyquist plot of the frequency response of a dynamic system model. 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)` creates a Nyquist plot of a dynamic system `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 must be in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

`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')`.

`[re,im,w] = nyquist(sys)` and `[re,im] = nyquist(sys,w)` return the real and imaginary parts of the frequency response at the frequencies `w` (in `rad/TimeUnit`). `re` and `im` are 3-D arrays (see "Arguments" below for details).

`[re,im,w,sdre,sdim] = nyquist(sys)` also returns the standard deviations of `re` and `im` for the identified system `sys`.

## Arguments

The output arguments `re` and `im` are 3-D arrays with dimensions

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

For SISO systems, the scalars `re(1,1,k)` and `im(1,1,k)` are the real and imaginary parts of the response at the frequency  $\omega_k = w(k)$ .

$$\text{re}(1,1,k) = \text{Re}(h(j\omega_k))$$

$$\text{im}(1,1,k) = \text{Im}(h(j\omega_k))$$

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  $i$ .

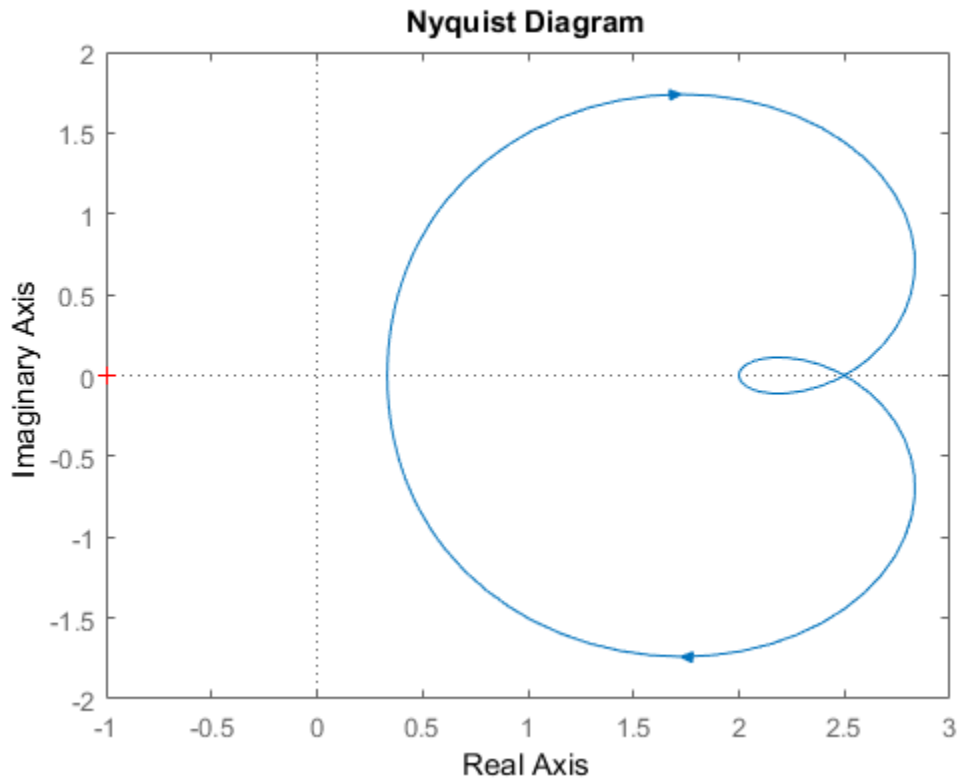
## Examples

### Nyquist Plot of Dynamic System

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)
```



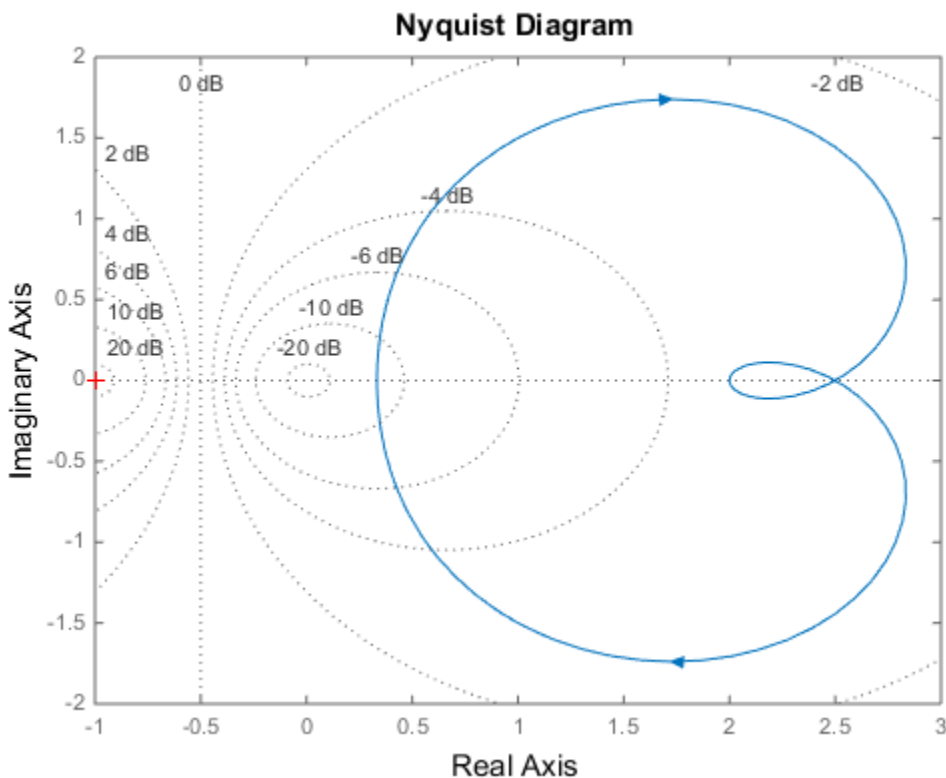
The nyquist function has support for M-circles, which are the contours of the constant closed-loop magnitude. M-circles are defined as the locus of complex numbers where

$$T(j\omega) = \left| \frac{G(j\omega)}{1 + G(j\omega)} \right|$$

is a constant value. In this equation,  $\omega$  is the frequency in radians/TimeUnit, where TimeUnit is the system time units, 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  $H$ .

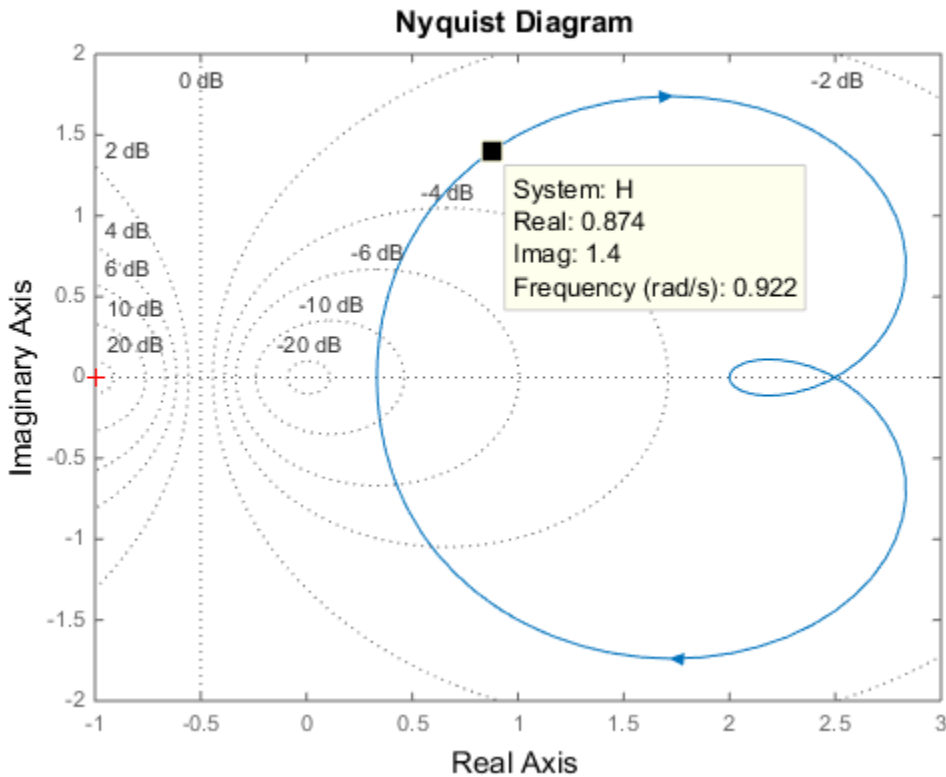


You have two zoom options available from the right-click menu that apply specifically to Nyquist plots:

- **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.



### Nyquist Plot of Identified Model with Response Uncertainty

Compute the standard deviation of the real and imaginary parts of frequency response of an identified model. Use this data to create a  $3\sigma$  plot of the response uncertainty. (Identified models require System Identification Toolbox.)

Identify a transfer function model based on data. Obtain the standard deviation data for the real and imaginary parts of the frequency response.

```
load iddata2 z2;
sys_p = tfest(z2,2);
w = linspace(-10*pi,10*pi,512);
[re, im, ~, sdre, sdim] = nyquist(sys_p,w);
```

`sys_p` is an identified transfer function model. `sdre` and `sdim` contain 1-std standard deviation uncertainty values in `re` and `im` respectively.

Create a Nyquist plot showing the response and its  $3\sigma$  uncertainty:

```
re = squeeze(re);
im = squeeze(im);
sdre = squeeze(sdre);
sdim = squeeze(sdim);
plot(re,im,'b', re+3*sdre, im+3*sdim, 'k:', re-3*sdre, im-3*sdim, 'k:')
```

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### Algorithms

See `bode`.

### See Also

`bode` | `evalfr` | `freqresp` | Linear System Analyzer | `nichols` | `sigma`

Introduced before R2006a

## nyquistoptions

List of Nyquist plot options

### Syntax

```
P = nyquistoptions
P = nyquistoptions('cstprefs')
```

### Description

`P = nyquistoptions` returns the default options for Nyquist plots. You can use these options to customize the Nyquist plot appearance using the command line.

`P = nyquistoptions('cstprefs')` initializes the plot options with the options you selected in the Control System and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation.

The following table summarizes the Nyquist plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid Specified as one of the following values: 'off'   'on' <b>Default:</b> 'off'
GridColor	Color of the grid lines Specified as one of the following: Vector of RGB values in the range [0, 1]   character vector of color name   'none'. For example, for yellow color, specify as one of the following: [1 1 0], 'yellow', or 'y'. <b>Default:</b> [0.15,0.15,0.15]
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs

---

Option	Description
	Specified as one of the following values: 'none'   'inputs'   'outputs'   'all' <b>Default:</b> 'none'
InputLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	<p>Frequency units, specified as one of the following values:</p> <ul style="list-style-type: none"> <li>• 'Hz'</li> <li>• 'rad/second'</li> <li>• 'rpm'</li> <li>• 'kHz'</li> <li>• 'MHz'</li> <li>• 'GHz'</li> <li>• 'rad/nanosecond'</li> <li>• 'rad/microsecond'</li> <li>• 'rad/millisecond'</li> <li>• 'rad/minute'</li> <li>• 'rad/hour'</li> <li>• 'rad/day'</li> <li>• 'rad/week'</li> <li>• 'rad/month'</li> <li>• 'rad/year'</li> <li>• 'cycles/nanosecond'</li> <li>• 'cycles/microsecond'</li> <li>• 'cycles/millisecond'</li> <li>• 'cycles/hour'</li> <li>• 'cycles/day'</li> <li>• 'cycles/week'</li> <li>• 'cycles/month'</li> <li>• 'cycles/year'</li> </ul> <p><b>Default:</b> 'rad/s'</p> <p>You can also specify 'auto' which uses frequency units rad/TimeUnit relative to system time units specified in the TimeUnit property. For</p>



Option	Description
	multiple systems with different time units, the units of the first system are used.
MagUnits	Magnitude units Specified as one of the following values: 'dB'   'abs' <b>Default:</b> 'dB'
PhaseUnits	Phase units Specified as one of the following values: 'deg'   'rad' <b>Default:</b> 'deg'
ShowFullContour	Show response for negative frequencies Specified as one of the following values: 'on'   'off' <b>Default:</b> 'on'
ConfidenceRegionNumber	Number of standard deviations to use to plotting the response confidence region (identified models only). <b>Default:</b> 1.
ConfidenceRegionDisplay	The frequency spacing of confidence ellipses. For identified models only. <b>Default:</b> 5, which means the confidence ellipses are shown at every 5th frequency sample.

## Examples

This example shows how to create a Nyquist plot displaying the full contour (the response for both positive and negative frequencies).

```
P = nyquistoptions;
P.ShowFullContour = 'on';
h = nyquistplot(tf(1,[1,.2,1]),P);
```

## See Also

nyquist | nyquistplot | getoptions | setoptions

**Introduced in R2011a**

## nyquistplot

Nyquist plot with additional plot customization options

### Syntax

```
h = nyquistplot(sys)
nyquistplot(sys, {wmin, wmax})
nyquistplot(sys, w)
nyquistplot(sys1, sys2, ..., w)
nyquistplot(AX, ...)
nyquistplot(..., plotoptions)
```

### Description

`h = nyquistplot(sys)` draws the Nyquist plot of the dynamic system model `sys`. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help nyquistoptions
```

for a list of available plot options.

The frequency range and number of points are chosen automatically. See `bode` for details on the notion of frequency in discrete time.

`nyquistplot(sys, {wmin, wmax})` draws the Nyquist plot for frequencies between `wmin` and `wmax` (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`).

`nyquistplot(sys, w)` uses the user-supplied vector `w` of frequencies (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`) at which the Nyquist response is to be evaluated. See `logspace` to generate logarithmically spaced frequency vectors.

`nyquistplot(sys1, sys2, ..., w)` draws the Nyquist plots of multiple models `sys1, sys2, ...` on a single plot. The frequency vector `w` is optional. You can also specify a color, line style, and marker for each system, as in

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

nyquistplot(AX,...) plots into the axes with handle AX.

nyquistplot(..., plotoptions) plots the Nyquist response with the options specified in plotoptions. Type

```
help nyquistoptions
```

for more details.

## Examples

### Example 1

#### Customize Nyquist Plot Frequency Units

Plot the Nyquist frequency response and change the units to rad/s.

```
sys = rss(5);
h = nyquistplot(sys);
% Change units to radians per second.
setoptions(h,'FreqUnits','rad/s');
```

### Example 2

Compare the frequency responses of identified state-space models of order 2 and 6 along with their 1-std confidence regions rendered at every 50th frequency sample.

```
load iddata1
sys1 = n4sid(z1, 2) % discrete-time IDSS model of order 2
sys2 = n4sid(z1, 6) % discrete-time IDSS model of order 6
```

Both models produce about 76% fit to data. However, **sys2** shows higher uncertainty in its frequency response, especially close to Nyquist frequency as shown by the plot:

```
w = linspace(10,10*pi,256);
h = nyquistplot(sys1,sys2,w);
setoptions(h,'ConfidenceRegionDisplaySpacing',50,'ShowFullContour','off');
```

Right-click to turn on the confidence region characteristic by using the **Characteristics->Confidence Region**.

### More About

#### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

#### See Also

`nyquist` | `setoptions` | `getoptions`

**Introduced before R2006a**

## obsv

Observability matrix

### Syntax

```
obsv(A,C)
Ob = obsv(sys)
```

### Description

`obsv` computes the observability matrix for state-space systems. For an  $n$ -by- $n$  matrix  $A$  and a  $p$ -by- $n$  matrix  $C$ , `obsv(A,C)` returns the observability matrix

$$Ob = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

with  $n$  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  $n$ .

### Examples

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)
```

These commands produce the following result.

```
unob =
      0
```

## More About

### Tips

`obsv` is here for educational purposes and is not recommended for serious control design. Computing the rank of the observability matrix is not recommended for observability testing. `Ob` will be numerically singular for most systems with more than a handful of states. This fact is well documented in the control literature. For example, see section III in <http://lawwww.epfl.ch/webdav/site/la/users/105941/public/NumCompCtrl.pdf>

### See Also

`obsvf`

**Introduced before R2006a**

## obsvf

Compute observability staircase form

### Syntax

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

### Description

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

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

where  $T$  is unitary and the transformed system has a *staircase* form with the unobservable modes, if any, in the upper left corner.

$$\bar{A} = \begin{bmatrix} A_{no} & A_{12} \\ 0 & A_o \end{bmatrix}, \quad \bar{B} = \begin{bmatrix} B_{no} \\ B_o \end{bmatrix}, \quad \bar{C} = [0 \ C_o]$$

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  $\bar{A}$ ,  $\bar{B}$ , and  $\bar{C}$ , as described above.  $T$  is the similarity transformation matrix and  $k$  is a vector of length  $n$ , where  $n$  is the number of states in  $A$ . Each entry of  $k$  represents the number of observable states factored out during each step of the transformation matrix calculation [1]. The number of nonzero elements in  $k$  indicates how many iterations were necessary to calculate  $T$ , and  $\text{sum}(k)$  is the number of states in  $A_o$ , the observable portion of  $\bar{A}$ .

`obsvf(A,B,C,tol)` uses the tolerance `tol` when calculating the observable/unobservable subspaces. When the tolerance is not specified, it defaults to  $10 \cdot n \cdot \text{norm}(a, 1) \cdot \text{eps}$ .

### Examples

Form the observability staircase form of

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

by typing

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

$$Abar = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$Bbar = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$Cbar = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$k = \begin{bmatrix} 2 & 0 \end{bmatrix}$$

### More About

#### Algorithms

obsvf implements the Staircase Algorithm of [1] by calling `ctrbf` and using duality.



## References

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

## See Also

ctrbf | obsv

**Introduced before R2006a**

## ord2

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` ( $\omega_n$ ) and damping factor `z` ( $\zeta$ ). 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.

### Examples

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 | ss | tf

**Introduced before R2006a**

## order

Query model order

## Syntax

```
NS = order(sys)
```

## Description

`NS = order(sys)` returns the model order `NS`. The order of a dynamic system model is the number of poles (for proper transfer functions) or the number of states (for state-space models). For improper transfer functions, the order is defined as the minimum number of states needed to build an equivalent state-space model (ignoring pole/zero cancellations).

`order(sys)` is an overloaded method that accepts SS, TF, and ZPK models. For LTI arrays, `NS` is an array of the same size listing the orders of each model in `sys`.

## Caveat

`order` does not attempt to find minimal realizations of MIMO systems. For example, consider this 2-by-2 MIMO system:

```
s=tf('s');  
h = [1, 1/(s*(s+1)); 1/(s+2), 1/(s*(s+1)*(s+2))];  
order(h)  
ans =
```

```
6
```

Although `h` has a 3rd order realization, `order` returns 6. Use

```
order(ss(h, 'min'))
```

to find the minimal realization order.

## **See Also**

pole | balred

**Introduced in R2012a**

## padé

Padé approximation of model with time delays

### Syntax

```
[num,den] = padé(T,N)
padé(T,N)
sysx = padé(sys,N)
sysx = padé(sys,NU,NY,NINT)
```

### Description

`padé` approximates time delays by rational models. Such approximations are useful to model time delay effects such as transport and computation delays within the context of continuous-time systems. The Laplace transform of a time delay of  $T$  seconds is  $\exp(-sT)$ . This exponential transfer function is approximated by a rational transfer function using Padé approximation formulas [1].

`[num,den] = padé(T,N)` returns the Padé approximation of order  $N$  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  $N$ th-order polynomials.

When invoked without output arguments, `padé(T,N)` plots the step and phase responses of the  $N$ th-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 = padé(sys,N)` produces a delay-free approximation `sysx` of the continuous delay system `sys`. All delays are replaced by their  $N$ th-order Padé approximation. See “Time Delays in Linear Systems” for more information about models with time delays.

`sysx = padé(sys,NU,NY,NINT)` specifies independent approximation orders for each input, output, and I/O or internal delay. Here `NU`, `NY`, and `NINT` are integer arrays such that

- `NU` is the vector of approximation orders for the input channel
- `NY` is the vector of approximation orders for the output channel
- `NINT` is the approximation order for I/O delays (TF or ZPK models) or internal delays (state-space models)

You can use scalar values for `NU`, `NY`, or `NINT` to specify a uniform approximation order. You can also set some entries of `NU`, `NY`, or `NINT` to `Inf` to prevent approximation of the corresponding delays.

## Examples

### Third-Order Padé Approximation

Compute a third-order Padé approximation of a 0.1-second I/O delay.

```
s = tf('s');
sys = exp(-0.1*s);
sysx = pade(sys,3)
```

```
sysx =
```

```

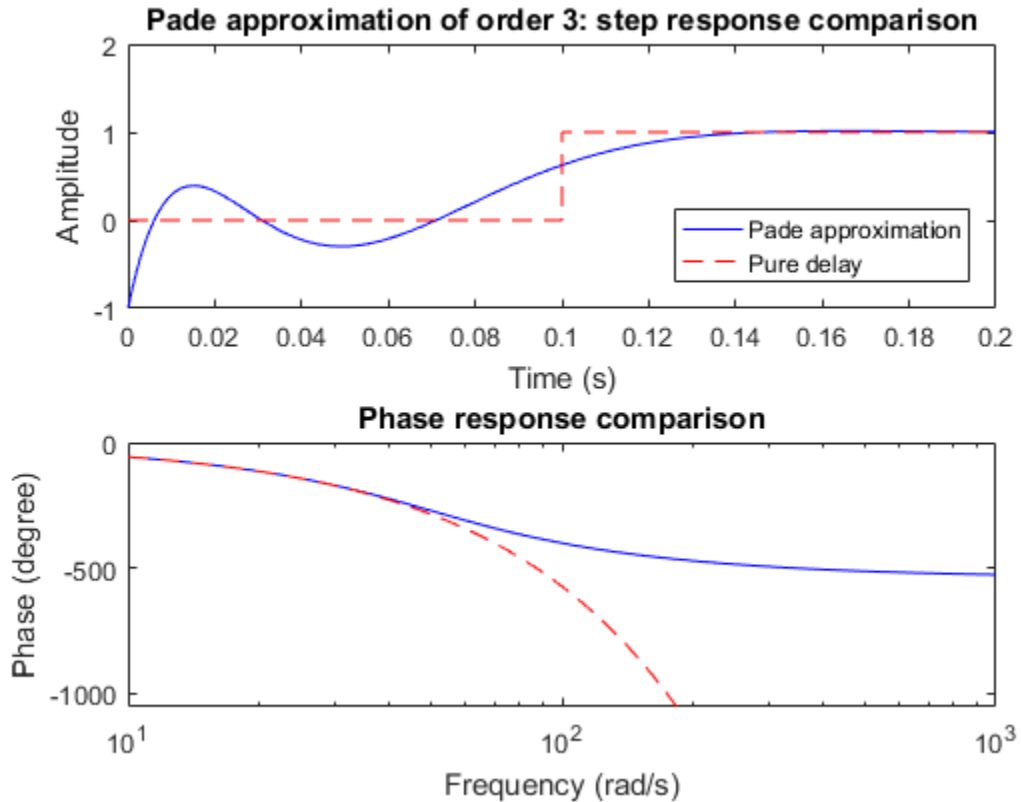
  -s^3 + 120 s^2 - 6000 s + 1.2e05
-----
  s^3 + 120 s^2 + 6000 s + 1.2e05
```

```
Continuous-time transfer function.
```

Here, `sys` is a dynamic system representation of the exact time delay of 0.1 s. `sysx` is a transfer function that approximates that delay.

Compare the time and frequency responses of the true delay and its approximation. Calling the `pade` command without output arguments generates the comparison plots. In this case the first argument to `pade` is just the magnitude of the exact time delay, rather than a dynamic system representing the time delay.

```
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.

## More About

- “Time-Delay Approximation”



## References

- [1] Golub, G. H. and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 557-558.

## See Also

c2d | absorbDelay | thiran

**Introduced before R2006a**

## parallel

Parallel connection of two models

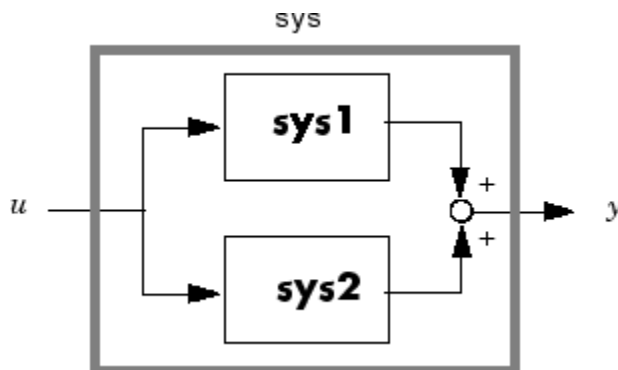
### Syntax

```
parallel
sys = parallel(sys1,sys2)
sys = parallel(sys1,sys2,inp1,inp2,out1,out2)
sys = parallel(sys1,sys2,'name')
```

### Description

`parallel` connects two model objects in parallel. This function accepts any type of 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.

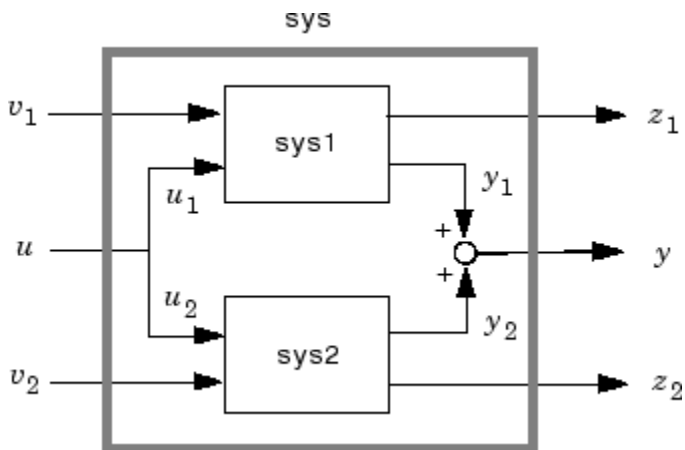
`sys = parallel(sys1,sys2)` forms the basic parallel connection shown in the following figure.



This command equals the direct addition

```
sys = sys1 + sys2
```

`sys = parallel(sys1,sys2,inp1,inp2,out1,out2)` forms the more general parallel connection shown in the following figure.



The vectors `inp1` and `inp2` contain indexes into the input channels of `sys1` and `sys2`, respectively, and define the input channels  $u_1$  and  $u_2$  in the diagram. Similarly, the vectors `out1` and `out2` contain indexes into the outputs of these two systems and define the output channels  $y_1$  and  $y_2$  in the diagram. The resulting model `sys` has  $[v_1 ; u ; v_2]$  as inputs and  $[z_1 ; y ; z_2]$  as outputs.

`sys = parallel(sys1,sys2,'name')` connects `sys1` and `sys2` by matching I/O names. You must specify all I/O names of `sys1` and `sys2`. The matching names appear in `sys` in the same order as in `sys1`. For example, the following specification:

```
sys1 = ss(eye(3),'InputName',{'C','B','A'},'OutputName',{'Z','Y','X'});
sys2 = ss(eye(3),'InputName',{'A','C','B'},'OutputName',{'X','Y','Z'});
parallel(sys1,sys2,'name')
```

returns this result:

```
d =
      C  B  A
Z  1  1  0
Y  1  1  0
X  0  0  2
```

Static gain.

---

**Note:** If `sys1` and `sys2` are model arrays, `parallel` returns model array `sys` of the same size, where `sys(:, :, k) = parallel(sys1(:, :, k), sys2(:, :, k), inp1, ...)`.

---

## Examples

See Kalman Filtering for an example.

## See Also

`append` | `feedback` | `series`

**Introduced before R2006a**

# passiveplot

Compute or plot passivity index as function of frequency

## Syntax

```
passiveplot(G)
passiveplot(G,type)
passiveplot( ____,w)
passiveplot(G1,G2,...,GN, ____)
passiveplot(G1,PlotStyle1,...,GN,PlotStyleN, ____)
```

```
[index,wout] = passiveplot(G)
[index,wout] = passiveplot(G,type)
index = passiveplot(G,w)
index = passiveplot(G,type,w)
```

## Description

`passiveplot(G)` plots the relative passivity indices of the dynamic system  $G$  as a function of frequency. When  $I + G$  is minimum phase, the relative passivity indices are the singular values of  $(I - G)(I + G)^{-1}$ . The largest singular value measures the relative excess ( $R < 1$ ) or shortage ( $R > 1$ ) at each frequency. See `getPassiveIndex` for more information about the meaning of the passivity index.

`passiveplot` automatically chooses the frequency range and number of points for the plot based on the dynamics of  $G$ .

`passiveplot(G,type)` plots the input, output, or I/O passivity index, depending on the value of `type`: 'input', 'output', or 'io', respectively.

`passiveplot( ____,w)` plots the passivity index for frequencies specified by  $w$ .

- If  $w$  is a cell array of the form  $\{w_{min}, w_{max}\}$ , then `passiveplot` plots the passivity index at frequencies ranging between  $w_{min}$  and  $w_{max}$ .
- If  $w$  is a vector of frequencies, then `passiveplot` plots the passivity index at each specified frequency.

You can use this syntax with any of the previous input-argument combinations.

`passiveplot(G1,G2,...,GN, ___)` plots the passivity index for multiple dynamic systems  $G_1, G_2, \dots, G_N$  on the same plot. You can also use this syntax with the `type` input argument, with `w` to specify frequencies to plot, or both.

`passiveplot(G1,PlotStyle1,...,GN,PlotStyleN, ___)` specifies a color, linestyle, and marker for each system in the plot.

`[index,wout] = passiveplot(G)` and `[index,wout] = passiveplot(G,type)` return the passivity index at each frequency in the vector `wout`. The output `index` is a matrix, and the value `index(:,k)` gives the passivity indices in descending order at the frequency `w(k)`. This syntax does not draw a plot.

`index = passiveplot(G,w)` and `index = passiveplot(G,type,w)` return the passivity indices at the frequencies specified by `w`.

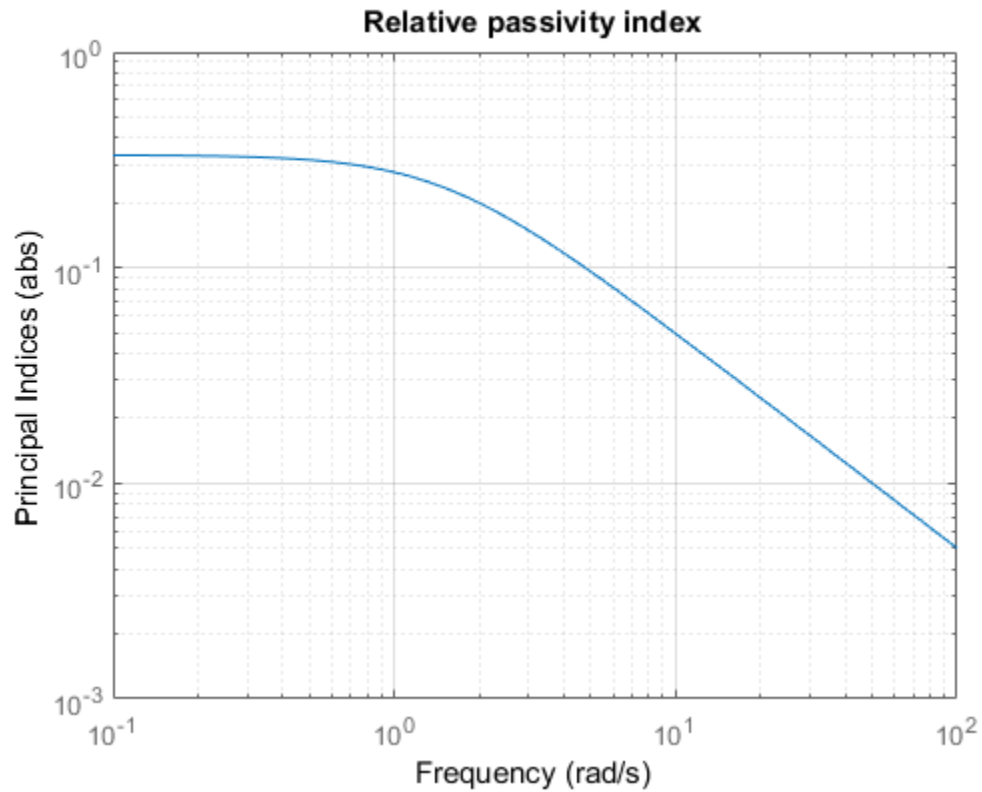
## Examples

### Plot Passivity Versus Frequency

Plot the relative passivity index as a function of frequency of the system

$$G = (s + 2)/(s + 1).$$

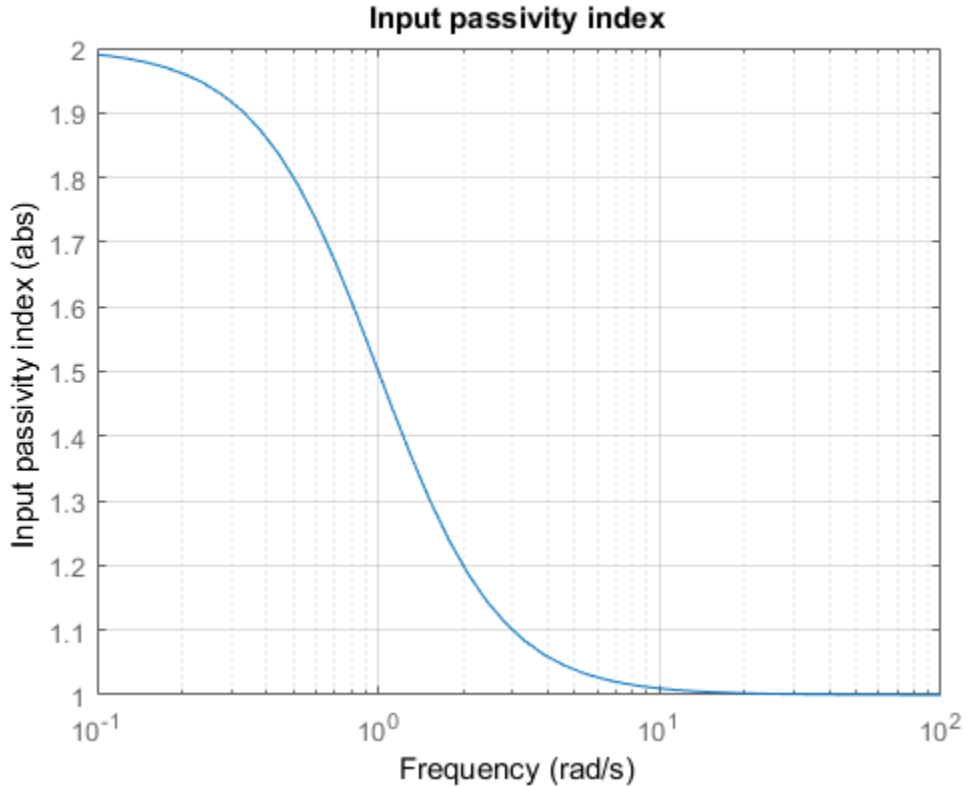
```
G = tf([1 2],[1 1]);  
passiveplot(G)
```



The plot shows that the relative passivity index is less than 1 at all frequencies. Therefore, the system **G** is passive.

Plot the input passivity index of the same system.

```
passiveplot(G, 'input')
```



The input passivity index is positive at all frequencies. Therefore, the system is input strictly passive.

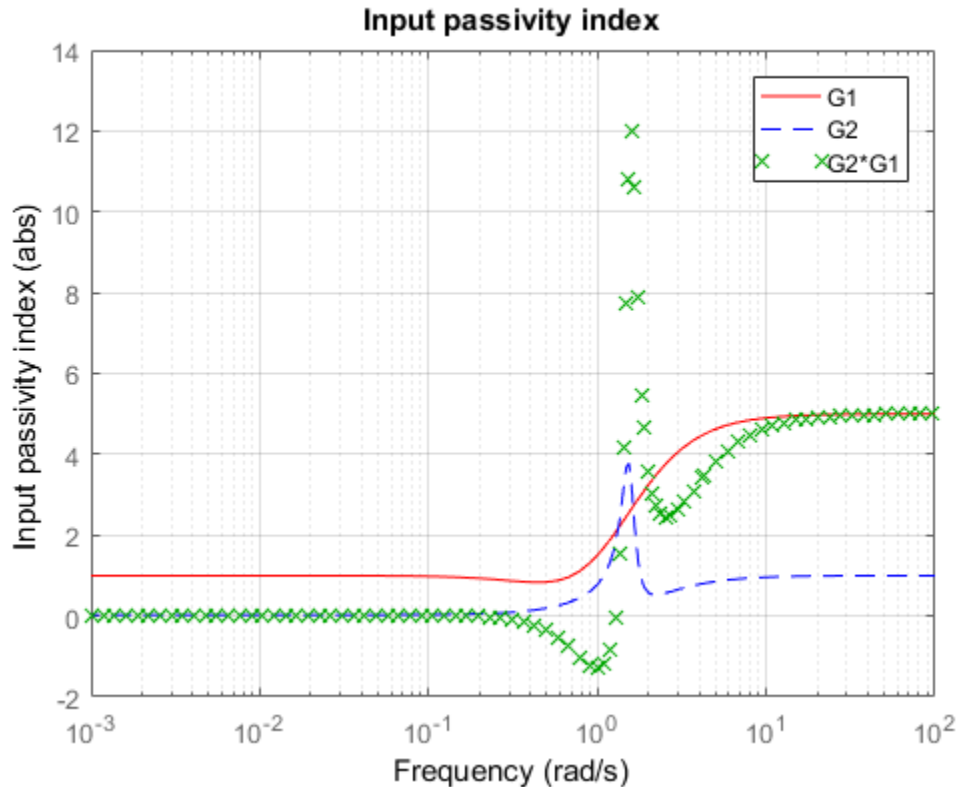
### Plot Passivity of Multiple Systems

Plot the input passivity index of two dynamic systems and their series interconnection.

```
G1 = tf([5 3 1],[1 2 1]);
G2 = tf([1 1 5 0.1],[1 2 3 4]);
H = G2*G1;

passiveplot(G1,'r',G2,'b--',H,'gx','input')
legend('G1','G2','G2*G1')
```





The input passivity index of the interconnected system dips below 0 around 1 rad/s. This plot shows that the series interconnection of two passive systems is not necessarily passive. However, passivity is preserved for parallel or feedback interconnections of passive systems.

## Input Arguments

### **G** — Model to analyze

dynamic system model | model array

Model to analyze for passivity, specified as a dynamic system model such as a `tf`, `ss`, or `genss` model. `G` can be MIMO, if the number of inputs equals the number of outputs.

**G** can be continuous or discrete. If **G** is a generalized model with tunable or uncertain blocks, `passiveplot` evaluates passivity of the current, nominal value of **G**.

If **G** is a model array, then `passiveplot` plots the passivity index of all models in the array on the same plot. When you use output arguments to get passivity data, **G** must be a single model.

### **type** — Type of passivity index

'input' | 'output' | 'io'

Type of passivity index, specified as one of the following:

- 'input' — Input passivity index (input feedforward passivity). This value is the smallest eigenvalue of  $\left(G(s) + G(s)^H\right)/2$ , for  $s = j\omega$  in continuous time, and  $s = e_{j\omega}$  in discrete time.
- 'output' — Output passivity index (output feedback passivity). When **G** is minimum phase, this value is the smallest eigenvalue of  $\left(G(s)^{-1} + G(s)^{-H}\right)/2$ , for  $s = j\omega$  in continuous time, and  $s = e_{j\omega}$  in discrete time.
- 'io' — Combined I/O passivity index. When **I + G** is minimum phase, this value is the largest  $\tau(\omega)$  such that:

$$G(s) + G(s)^H > 2\tau(\omega)\left(I + G(s)^H G(s)\right),$$

for  $s = j\omega$  in continuous time, and  $s = e_{j\omega}$  in discrete time.

See “About Passivity and Passivity Indices” for details about these indices.

### **w** — Frequencies

{wmin,wmax} | vector

Frequencies at which to compute and plot indices, specified as the cell array {wmin,wmax} or as a vector of frequency values.

- If **w** is a cell array of the form {wmin,wmax}, then the function computes the index at frequencies ranging between wmin and wmax.
- If **w** is a vector of frequencies, then the function computes the index at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically-spaced frequency values.

Specify frequencies in units of rad/TimeUnit, where TimeUnit is the TimeUnit property of the model.

### **PlotStyle — Line style, marker, and color**

character vector

Line style, marker, and color of both the line and marker, specified as a vector of one, two, or three characters. The characters can appear in any order. For more information about configuring the PlotStyle argument, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

Example: 'r--', '\*b', 'y'

## **Output Arguments**

### **index — Passivity indices**

matrix

Passivity indices as a function of frequency, returned as a matrix. `index` contains whichever type of passivity index you specify, computed at the frequencies `w` if you supplied them, or `wout` if you did not. `index` has as many columns as there are values in `w` or `wout`, and

- One row, for the input, output, or combined i/o passivity indices.
- As many rows as `G` has inputs or outputs, for the relative passivity index.

For example, suppose that `G` is a 3-input, 3-output system, and `w` is a 1-by-30 vector of frequencies. Then the following syntax returns a 3-by-30 matrix `index`.

```
index = passiveplot(G,w);
```

The entry `index(:,k)` contains the relative passivity indices of `G`, in descending order, at the frequency `w(k)`.

### **wout — Frequencies**

vector

Frequencies at which the indices are calculated, returned as a vector. The function automatically chooses the frequency range and number of points based on the dynamics of the model.

## More About

- “About Passivity and Passivity Indices”

## See Also

`getPassiveIndex` | `getSectorIndex` | `isPassive` | `sectorplot`

**Introduced in R2016a**

## permut

Rearrange array dimensions in model arrays

### Syntax

```
newarray = permut(sysarray,order)
```

### Description

`newarray = permut(sysarray,order)` rearranges the array dimensions of a model array so that the dimensions are in the specified order. The input and output dimensions of the model array are not counted as array dimensions for this operation.

### Examples

#### Permute Model Array Dimensions

Create a 1-by-2-by-3 array of state-space models.

```
sysarr = rss(2,2,2,1,2,3);
```

Rearrange the model array so that the dimensions are 3-by-2-by-1.

```
newarr = permut(sysarr,[3 2 1]);  
size(newarr)
```

```
3x2 array of state-space models.
```

```
Each model has 2 outputs, 2 inputs, and 2 states.
```

The input and output dimensions of the model array remain unchanged.

### Input Arguments

**sysarray** — Model array to rearrange  
model array

Model array to rearrange, specified as an array of input-output models such as numeric LTI models, generalized models, or identified LTI models.

**order** — Dimensions of rearranged model array

vector

Dimensions of rearranged model array, specified as a vector of positive integers. For example, to rearrange a model array into a 3-by-2 array, `order` is `[3 2]`.

Data Types: `double`

## Output Arguments

**newarray** — Rearranged model array

model array

Rearranged model array, returned as an array of input-output models with the new dimensions as specified in `order`.

## See Also

`ndims` | `reshape` | `size`

**Introduced in R2013a**

# pid

Create PID controller in parallel form, convert to parallel-form PID controller

## Syntax

```
C = pid(Kp,Ki,Kd,Tf)
C = pid(Kp,Ki,Kd,Tf,Ts)
C = pid(sys)
C = pid(Kp)
C = pid(Kp,Ki)
C = pid(Kp,Ki,Kd)
C = pid(...,Name,Value)
C = pid
```

## Description

`C = pid(Kp,Ki,Kd,Tf)` creates a continuous-time PID controller with proportional, integral, and derivative gains `Kp`, `Ki`, and `Kd` and first-order derivative filter time constant `Tf`:

$$C = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}.$$

This representation is in *parallel form*. If all of `Kp`, `Ki`, `Kd`, and `Tf` are real, then the resulting `C` is a `pid` controller object. If one or more of these coefficients is tunable (`realp` or `genmat`), then `C` is a tunable generalized state-space (`genss`) model object.

`C = pid(Kp,Ki,Kd,Tf,Ts)` creates a discrete-time PID controller with sample time `Ts`. The controller is:

$$C = K_p + K_i IF(z) + \frac{K_d}{T_f + DF(z)}.$$

$IF(z)$  and  $DF(z)$  are the *discrete integrator formulas* for the integrator and derivative filter. By default,

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

To choose different discrete integrator formulas, use the `IFormula` and `DFormula` properties. (See “Properties” on page 2-673 for more information about `IFormula` and `DFormula`). If `DFormula` = 'ForwardEuler' (the default value) and `Tf`  $\neq$  0, then `Ts` and `Tf` must satisfy `Tf` > `Ts`/2. This requirement ensures a stable derivative filter pole.

`C = pid(sys)` converts the dynamic system `sys` to a parallel form `pid` controller object.

`C = pid(Kp)` creates a continuous-time proportional (P) controller with `Ki` = 0, `Kd` = 0, and `Tf` = 0.

`C = pid(Kp,Ki)` creates a proportional and integral (PI) controller with `Kd` = 0 and `Tf` = 0.

`C = pid(Kp,Ki,Kd)` creates a proportional, integral, and derivative (PID) controller with `Tf` = 0.

`C = pid(...,Name,Value)` creates a controller or converts a dynamic system to a `pid` controller object with additional options specified by one or more `Name, Value` pair arguments.

`C = pid` creates a P controller with `Kp` = 1.

## Input Arguments

### **Kp**

Proportional gain.

`Kp` can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When `Kp` = 0, the controller has no proportional action.



**Default:** 1

### **Ki**

Integral gain.

Ki can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (**realp**) or generalized matrix (**genmat**).
- A tunable surface for gain-scheduled tuning, created using **tunableSurface**.

When  $K_i = 0$ , the controller has no integral action.

**Default:** 0

### **Kd**

Derivative gain.

Kd can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (**realp**) or generalized matrix (**genmat**).
- A tunable surface for gain-scheduled tuning, created using **tunableSurface**.

When  $K_d = 0$ , the controller has no derivative action.

**Default:** 0

### **Tf**

Time constant of the first-order derivative filter.

Tf can be:

- A real, finite, and nonnegative value.

- An array of real, finite, and nonnegative values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $T_f = 0$ , the controller has no filter on the derivative action.

**Default:** 0

### **Ts**

Sample time.

To create a discrete-time `pid` controller, provide a positive real value ( $T_s > 0$ ). `pid` does not support discrete-time controller with undetermined sample time ( $T_s = -1$ ).

$T_s$  must be a scalar value. In an array of `pid` controllers, each controller must have the same  $T_s$ .

**Default:** 0 (continuous time)

### **sys**

SISO dynamic system to convert to parallel `pid` form.

`sys` must represent a valid PID controller that can be written in parallel form with  $T_f \geq 0$ .

`sys` can also be an array of SISO dynamic systems.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Use `Name`, `Value` syntax to set the numerical integration formulas `IFormula` and `DFormula` of a discrete-time `pid` controller, or to set other object properties such as `InputName` and `OutputName`. For information about available properties of `pid` controller objects, see “Properties” on page 2-673.

## Output Arguments

### C

PID controller, represented as a `pid` controller object, an array of `pid` controller objects, a `genss` object, or a `genss` array.

- If all the gains `Kp`, `Ki`, `Kd`, and `Tf` have numeric values, then `C` is a `pid` controller object. When the gains are numeric arrays, `C` is an array of `pid` controller objects. The controller type (P, I, PI, PD, PDF, PID, PIDF) depends upon the values of the gains. For example, when `Kd = 0`, but `Kp` and `Ki` are nonzero, `C` is a PI controller.
- If one or more gains is a tunable parameter (`realp`), generalized matrix (`genmat`), or tunable gain surface (`tunableSurface`), then `C` is a generalized state-space model (`genss`).

## Properties

### `Kp`, `Ki`, `Kd`

PID controller gains.

The `Kp`, `Ki`, and `Kd` properties store the proportional, integral, and derivative gains, respectively. `Kp`, `Ki`, and `Kd` are real and finite.

### `Tf`

Derivative filter time constant.

The `Tf` property stores the derivative filter time constant of the `pid` controller object. `Tf` is real, finite, and nonnegative.

### `IFormula`

Discrete integrator formula  $IF(z)$  for the integrator of the discrete-time `pid` controller `C`:

$$C = K_p + K_i IF(z) + \frac{K_d}{T_f + DF(z)}.$$

`IFormula` can take the following values:

- 'ForwardEuler' —  $IF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the `ForwardEuler` formula can result in instability, even when discretizing a system that is stable in continuous time.

- 'BackwardEuler' —  $IF(z) = \frac{T_s z}{z-1}$ .

An advantage of the `BackwardEuler` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- 'Trapezoidal' —  $IF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the `Trapezoidal` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the `Trapezoidal` formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

When `C` is a continuous-time controller, `IFormula` is ' '.

**Default:** 'ForwardEuler'

### DFormula

Discrete integrator formula  $DF(z)$  for the derivative filter of the discrete-time pid controller `C`:

$$C = K_p + K_i IF(z) + \frac{K_d}{T_f + DF(z)}$$

`DFormula` can take the following values:

- 'ForwardEuler' —  $DF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the `ForwardEuler` formula can result in instability, even when discretizing a system that is stable in continuous time.

- `'BackwardEuler'` —  $DF(z) = \frac{T_s z}{z-1}$ .

An advantage of the `BackwardEuler` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- `'Trapezoidal'` —  $DF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the `Trapezoidal` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the `Trapezoidal` formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

The `Trapezoidal` value for `DFormula` is not available for a `pid` controller with no derivative filter (`Tf = 0`).

When `C` is a continuous-time controller, `DFormula` is `''`.

**Default:** `'ForwardEuler'`

### **InputDelay**

Time delay on the system input. `InputDelay` is always 0 for a `pid` controller object.

### **OutputDelay**

Time delay on the system Output. `OutputDelay` is always 0 for a `pid` controller object.

### **Ts**

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel name, specified as a character vector. Use this property to name the input channel of the controller model. For example, assign the name `error` to the input of a controller model `C` as follows.

```
C.InputName = 'error';
```

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `C.u` is equivalent to `C.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### **InputUnit**

Input channel units, specified as a character vector. Use this property to track input signal units. For example, assign the concentration units `mol/m^3` to the input of a controller model `C` as follows.

```
C.InputUnit = 'mol/m^3';
```

`InputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

### **InputGroup**

Input channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **OutputName**

Output channel name, specified as a character vector. Use this property to name the output channel of the controller model. For example, assign the name `control` to the output of a controller model `C` as follows.

```
C.OutputName = 'control';
```

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `C.y` is equivalent to `C.OutputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

**OutputUnit**

Output channel units, specified as a character vector. Use this property to track output signal units. For example, assign the unit `Volts` to the output of a controller model `C` as follows.

```
C.OutputUnit = 'Volts';
```

`OutputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

**OutputGroup**

Output channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

**Name**

System name, specified as a character vector. For example, 'system\_1'.

**Default:** ''

**Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, 'System is MIMO'.

**Default:** {}

**UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

**SamplingGrid**

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This



information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta,w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:,:,1,1) [zeta=0.3, w=5] =
```

```

      25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```

      25
-----
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink

Control Design commands `linearize` and `sLinearizer` populate `SamplingGrid` in this way.

**Default:** []

## Examples

### PDF Controller

Create a continuous-time controller with proportional and derivative gains and a filter on the derivative term. To do so, set the integral gain to zero. Set the other gains and the filter time constant to the desired values.

```
Kp = 1;  
Ki = 0;    % No integrator  
Kd = 3;  
Tf = 0.5;  
C = pid(Kp,Ki,Kd,Tf)
```

C =

$$K_p + K_d * \frac{s}{T_f s + 1}$$

with  $K_p = 1$ ,  $K_d = 3$ ,  $T_f = 0.5$

Continuous-time PDF controller in parallel form.

The display shows the controller type, formula, and parameter values, and verifies that the controller has no integrator term.

### Discrete-Time PI Controller

Create a discrete-time PI controller with trapezoidal discretization formula.

To create a discrete-time PI controller, set the value of `TS` and the discretization formula using `Name, Value` syntax.

```
C1 = pid(5,2.4,'Ts',0.1,'IFormula','Trapezoidal')    % Ts = 0.1s
```

C1 =

$$K_p + K_i * \frac{T_s(z+1)}{2*(z-1)}$$

with Kp = 5, Ki = 2.4, Ts = 0.1

Sample time: 0.1 seconds

Discrete-time PI controller in parallel form.

Alternatively, you can create the same discrete-time controller by supplying **Ts** as the fifth input argument after all four PID parameters, **Kp**, **Ki**, **Kd**, and **Tf**. Since you only want a PI controller, set **Kd** and **Tf** to zero.

```
C2 = pid(5,2.4,0,0,0.1, 'IFormula', 'Trapezoidal')
```

C2 =

$$K_p + K_i * \frac{T_s(z+1)}{2*(z-1)}$$

with Kp = 5, Ki = 2.4, Ts = 0.1

Sample time: 0.1 seconds

Discrete-time PI controller in parallel form.

The display shows that C1 and C2 are the same.

### PID Controller with Named Input and Output

When you create a PID controller, set the dynamic system properties **InputName** and **OutputName**. This is useful, for example, when you interconnect the PID controller with other dynamic system models using the **connect** command.

```
C = pid(1,2,3, 'InputName', 'e', 'OutputName', 'u')
```

C =

$$K_p + K_i * \frac{1}{s} + K_d * s$$

with  $K_p = 1$ ,  $K_i = 2$ ,  $K_d = 3$

Continuous-time PID controller in parallel form.

The display does not show the input and output names for the PID controller, but you can examine the property values. For instance, verify the input name of the controller.

C.InputName

```
ans =  
cell  
    'e'
```

### Array of PID Controllers

Create a 2-by-3 grid of PI controllers with proportional gain ranging from 1–2 across the array rows and integral gain ranging from 5–9 across columns.

To build the array of PID controllers, start with arrays representing the gains.

```
Kp = [1 1 1;2 2 2];  
Ki = [5:2:9;5:2:9];
```

When you pass these arrays to the `pid` command, the command returns the array.

```
pi_array = pid(Kp,Ki,'Ts',0.1,'IFormula','BackwardEuler');  
size(pi_array)
```

```
2x3 array of PID controller.  
Each PID has 1 output and 1 input.
```

Alternatively, use the `stack` command to build an array of PID controllers.

```
C = pid(1,5,0.1)           % PID controller
```

```
Cf = pid(1,5,0.1,0.5)      % PID controller with filter
pid_array = stack(2,C,Cf); % stack along 2nd array dimension
```

C =

$$K_p + K_i * \frac{1}{s} + K_d * s$$

with  $K_p = 1$ ,  $K_i = 5$ ,  $K_d = 0.1$

Continuous-time PID controller in parallel form.

Cf =

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f * s + 1}$$

with  $K_p = 1$ ,  $K_i = 5$ ,  $K_d = 0.1$ ,  $T_f = 0.5$

Continuous-time PIDF controller in parallel form.

These commands return a 1-by-2 array of controllers.

```
size(pid_array)
```

1x2 array of PID controller.

Each PID has 1 output and 1 input.

All PID controllers in an array must have the same sample time, discrete integrator formulas, and dynamic system properties such as `InputName` and `OutputName`.

### Convert PID Controller from Standard to Parallel Form

Convert a standard form `pidstd` controller to parallel form.

Standard PID form expresses the controller actions in terms of an overall proportional gain  $K_p$ , integral and derivative time constants  $T_i$  and  $T_d$ , and filter divisor  $N$ . You can convert any standard-form controller to parallel form using the `pid` command. For example, consider the following standard-form controller.

```
Kp = 2;
Ti = 3;
Td = 4;
N = 50;
C_std = pidstd(Kp,Ti,Td,N)
```

C\_std =

$$K_p * \left( 1 + \frac{1}{T_i} * \frac{1}{s} + T_d * \frac{s}{(T_d/N)*s+1} \right)$$

with Kp = 2, Ti = 3, Td = 4, N = 50

Continuous-time PIDF controller in standard form

Convert this controller to parallel form using `pid`.

```
C_par = pid(C_std)
```

C\_par =

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f*s+1}$$

with Kp = 2, Ki = 0.667, Kd = 8, Tf = 0.08

Continuous-time PIDF controller in parallel form.

### Convert Dynamic System to Parallel-Form PID Controller

Convert a continuous-time dynamic system that represents a PID controller to parallel `pid` form.

The following dynamic system, with an integrator and two zeros, is equivalent to a PID controller.

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

Create a zpk model of  $H$ . Then use the `pid` command to obtain  $H$  in terms of the PID gains  $K_p$ ,  $K_i$ , and  $K_d$ .

```
H = zpk([-1, -2], 0, 3);
C = pid(H)
```

C =

$$K_p + K_i * \frac{1}{s} + K_d * s$$

with  $K_p = 9$ ,  $K_i = 6$ ,  $K_d = 3$

Continuous-time PID controller in parallel form.

### Convert Discrete-Time Dynamic System to Parallel-Form PID Controller

Convert a discrete-time dynamic system that represents a PID controller with derivative filter to parallel `pid` form.

Create a discrete-time zpk model that represents a PIDF controller (two zeros and two poles, including the integrator pole at  $z = 1$ ).

```
sys = zpk([-0.5, -0.6], [1 -0.2], 3, 'Ts', 0.1);
```

When you convert `sys` to PID form, the result depends on which discrete integrator formulas you specify for the conversion. For instance, use the default, `ForwardEuler`, for both the integrator and the derivative.

```
Cfe = pid(sys)
```

Cfe =

$$K_p + K_i * \frac{T_s}{z-1} + K_d * \frac{1}{T_f + T_s / (z-1)}$$

with  $K_p = 2.75$ ,  $K_i = 60$ ,  $K_d = 0.0208$ ,  $T_f = 0.0833$ ,  $T_s = 0.1$

Sample time: 0.1 seconds

Discrete-time PIDF controller in parallel form.

Now convert using the Trapezoidal formula.

```
Ctrap = pid(sys, 'IFormula', 'Trapezoidal', 'DFormula', 'Trapezoidal')
```

Ctrap =

$$K_p + K_i * \frac{T_s(z+1)}{2*(z-1)} + K_d * \frac{1}{T_f + T_s/2*(z+1)/(z-1)}$$

with  $K_p = -0.25$ ,  $K_i = 60$ ,  $K_d = 0.0208$ ,  $T_f = 0.0333$ ,  $T_s = 0.1$

Sample time: 0.1 seconds

Discrete-time PIDF controller in parallel form.

The displays show the difference in resulting coefficient values and functional form.

For this particular dynamic system, you cannot write `sys` in parallel PID form using the `BackwardEuler` formula for the derivative filter. Doing so would result in  $T_f < 0$ , which is not permitted. In that case, `pid` returns an error.

### Discretize a Continuous-Time PID Controller

Discretize a continuous-time PID controller and set integral and derivative filter formulas.

Create a continuous-time controller and discretize it using the zero-order-hold method of the `c2d` command.

```
Ccon = pid(1,2,3,4); % continuous-time PIDF controller
Cdis1 = c2d(Ccon,0.1, 'zoh')
```

Cdis1 =

$$K_p + K_i * \frac{T_s}{z-1} + K_d * \frac{1}{T_f + T_s/(z-1)}$$

with  $K_p = 1$ ,  $K_i = 2$ ,  $K_d = 3.04$ ,  $T_f = 4.05$ ,  $T_s = 0.1$



```
Sample time: 0.1 seconds
Discrete-time PIDF controller in parallel form.
```

The display shows that `c2d` computes new PID gains for the discrete-time controller.

The discrete integrator formulas of the discretized controller depend on the `c2d` discretization method, as described in “Tips”. For the `zoh` method, both `IFormula` and `DFormula` are `ForwardEuler`.

```
Cdis1.IFormula
Cdis1.DFormula
```

```
ans =
```

```
ForwardEuler
```

```
ans =
```

```
ForwardEuler
```

If you want to use different formulas from the ones returned by `c2d`, then you can directly set the `Ts`, `IFormula`, and `DFormula` properties of the controller to the desired values.

```
Cdis2 = Ccon;
Cdis2.Ts = 0.1;
Cdis2.IFormula = 'BackwardEuler';
Cdis2.DFormula = 'BackwardEuler';
```

However, these commands do not compute new PID gains for the discretized controller. To see this, examine `Cdis2` and compare the coefficients to `Ccon` and `Cdis1`.

```
Cdis2
```

```
Cdis2 =
```

$$K_p + K_i * \frac{T_s * z}{z - 1} + K_d * \frac{1}{T_f + T_s * z / (z - 1)}$$

with  $K_p = 1$ ,  $K_i = 2$ ,  $K_d = 3$ ,  $T_f = 4$ ,  $T_s = 0.1$

Sample time: 0.1 seconds

Discrete-time PIDF controller in parallel form.

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

## More About

### Tips

- Use `pid` to:
  - Create a `pid` controller object from known PID gains and filter time constant.
  - Convert a `pidstd` controller object to a standard-form `pid` controller object.
  - Convert other types of dynamic system models to a `pid` controller object.
- To design a PID controller for a particular plant, use `pidtune` or `pidTuner`. To create a tunable PID controller as a control design block, use `tunablePID`.
- Create arrays of `pid` controller objects by:
  - Specifying array values for  $K_p, K_i, K_d$ , and  $T_f$
  - Specifying an array of dynamic systems `sys` to convert to `pid` controller objects
  - Using `stack` to build arrays from individual controllers or smaller arrays

In an array of `pid` controllers, each controller must have the same sample time  $T_s$  and discrete integrator formulas `IFormula` and `DFormula`.

- To create or convert to a standard-form controller, use `pidstd`. Standard form expresses the controller actions in terms of an overall proportional gain  $K_p$ , integral and derivative times  $T_i$  and  $T_d$ , and filter divisor  $N$ :

$$C = K_p \left( 1 + \frac{1}{T_i} \frac{1}{s} + \frac{T_d s}{N} \frac{1}{s+1} \right)$$

- There are two ways to discretize a continuous-time `pid` controller:
  - Use the `c2d` command. `c2d` computes new parameter values for the discretized controller. The discrete integrator formulas of the discretized controller depend upon the `c2d` discretization method you use, as shown in the following table.

<b>c2d Discretization Method</b>	<b>IFormula</b>	<b>DFormula</b>
'zoh'	ForwardEuler	ForwardEuler
'foh'	Trapezoidal	Trapezoidal
'tustin'	Trapezoidal	Trapezoidal
'impulse'	ForwardEuler	ForwardEuler
'matched'	ForwardEuler	ForwardEuler

For more information about `c2d` discretization methods, See the `c2d` reference page. For more information about `IFormula` and `DFormula`, see “Properties” on page 2-673 .

- If you require different discrete integrator formulas, you can discretize the controller by directly setting `Ts`, `IFormula`, and `DFormula` to the desired values. (See “Discretize a Continuous-Time PID Controller” on page 2-686.) However, this method does not compute new gain and filter-constant values for the discretized controller. Therefore, this method might yield a poorer match between the continuous- and discrete-time `pid` controllers than using `c2d`.
- “What Are Model Objects?”

## See Also

`pidstd` | `pid2` | `piddata` | `make2DOF` | `pidtune` | `pidTuner` | `tunablePID` | `genss` | `realp`

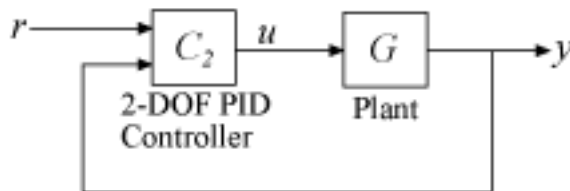
**Introduced in R2010b**

## pid2

Create 2-DOF PID controller in parallel form, convert to parallel-form 2-DOF PID controller

`pid2` controller objects represent two-degree-of-freedom (2-DOF) PID controllers in parallel form. Use `pid2` either to create a `pid2` controller object from known coefficients or to convert a dynamic system model to a `pid2` object.

Two-degree-of-freedom (2-DOF) PID controllers include setpoint weighting on the proportional and derivative terms. A 2-DOF PID controller can achieve fast disturbance rejection without significant increase of overshoot in setpoint tracking. 2-DOF PID controllers are also useful to mitigate the influence of changes in the reference signal on the control signal. The following illustration shows a typical control architecture using a 2-DOF PID controller.



## Syntax

```

C2 = pid2(Kp,Ki,Kd,Tf,b,c)
C2 = pid2(Kp,Ki,Kd,Tf,b,c,Ts)
C2 = pid2(sys)
C2 = pid2( ____,Name,Value)

```

## Description

`C2 = pid2(Kp,Ki,Kd,Tf,b,c)` creates a continuous-time 2-DOF PID controller with proportional, integral, and derivative gains `Kp`, `Ki`, and `Kd` and first-order derivative filter time constant `Tf`. The controller also has setpoint weighting `b` on the proportional

term, and setpoint weighting  $c$  on the derivative term. The relationship between the 2-DOF controller output ( $u$ ) and its two inputs ( $r$  and  $y$ ) is given by:

$$u = K_p (br - y) + \frac{K_i}{s} (r - y) + \frac{K_d s}{T_f s + 1} (cr - y).$$

This representation is in *parallel form*. If all coefficients are real-valued, then the resulting `C2` is a `pid2` controller object. If one or more of these coefficients is tunable (`realp` or `genmat`), then `C2` is a tunable generalized state-space (`genss`) model object.

`C2 = pid2(Kp, Ki, Kd, Tf, b, c, Ts)` creates a discrete-time 2-DOF PID controller with sample time  $T_s$ . The relationship between the controller output and inputs is given by:

$$u = K_p (br - y) + K_i IF(z)(r - y) + \frac{K_d}{T_f + DF(z)} (cr - y).$$

$IF(z)$  and  $DF(z)$  are the *discrete integrator formulas* for the integrator and derivative filter. By default,

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

To choose different discrete integrator formulas, use the `IFormula` and `DFormula` properties. (See “Properties” on page 2-695 for more information). If `DFormula` = 'ForwardEuler' (the default value) and  $T_f \neq 0$ , then  $T_s$  and  $T_f$  must satisfy  $T_f > T_s/2$ . This requirement ensures a stable derivative filter pole.

`C2 = pid2(sys)` converts the dynamic system `sys` to a parallel form `pid2` controller object.

`C2 = pid2( ____, Name, Value)` specifies additional properties as comma-separated pairs of `Name, Value` arguments.

## Input Arguments

### **Kp**

Proportional gain.

$K_p$  can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $K_p = 0$ , the controller has no proportional action.

**Default:** 1

### **Ki**

Integral gain.

$K_i$  can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $K_i = 0$ , the controller has no integral action.

**Default:** 0

### **Kd**

Derivative gain.

$K_d$  can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $K_d = 0$ , the controller has no derivative action.

**Default:** 0

**Tf**

Time constant of the first-order derivative filter.

Tf can be:

- A real, finite, and nonnegative value.
- An array of real, finite, and nonnegative values.
- A tunable parameter (**realp**) or generalized matrix (**genmat**).
- A tunable surface for gain-scheduled tuning, created using **tunableSurface**.

When Tf = 0, the controller has no filter on the derivative action.

**Default:** 0

**b**

Setpoint weighting on proportional term.

b can be:

- A real, nonnegative, and finite value.
- An array of real, nonnegative, finite values.
- A tunable parameter (**realp**) or generalized matrix (**genmat**).
- A tunable surface for gain-scheduled tuning, created using **tunableSurface**.

When b = 0, changes in setpoint do not feed directly into the proportional term.

**Default:** 1

**c**

Setpoint weighting on derivative term.

c can be:

- A real, nonnegative, and finite value.

- An array of real, nonnegative, finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $c = 0$ , changes in setpoint do not feed directly into the proportional term.

**Default:** 1

### **Ts**

Sample time.

To create a discrete-time `pid2` controller, provide a positive real value ( $T_s > 0$ ). `pid2` does not support discrete-time controller with undetermined sample time ( $T_s = -1$ ).

$T_s$  must be a scalar value. In an array of `pid2` controllers, each controller must have the same  $T_s$ .

**Default:** 0 (continuous time)

### **sys**

SISO dynamic system to convert to parallel `pid2` form.

`sys` must be a two-input, one-output system. `sys` must represent a valid 2-DOF PID controller that can be written in parallel form with  $T_f \geq 0$ .

`sys` can also be an array of SISO dynamic systems.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1`, `Value1`, ..., `NameN`, `ValueN`.

Use `Name`, `Value` syntax to set the numerical integration formulas `IFormula` and `DFormula` of a discrete-time `pid2` controller, or to set other object properties such as `InputName` and `OutputName`. For information about available properties of `pid2` controller objects, see “Properties” on page 2-695.



## Output Arguments

### **C2**

2-DOF PID controller, returned as a `pid2` controller object, an array of `pid2` controller objects, a `genss` object, or a `genss` array.

- If all the coefficients have scalar numeric values, then **C2** is a `pid2` controller object.
- If one or more coefficients is a numeric array, **C2** is an array of `pid2` controller objects. The controller type (such as PI, PID, or PDF) depends upon the values of the gains. For example, when  $K_d = 0$ , but  $K_p$  and  $K_i$  are nonzero, **C2** is a PI controller.
- If one or more coefficients is a tunable parameter (`realp`), generalized matrix (`genmat`), or tunable gain surface (`tunableSurface`), then **C2** is a generalized state-space model (`genss`).

## Properties

### **b, c**

Setpoint weights on the proportional and derivative terms, respectively. **b** and **c** values are real, finite, and positive. When you use the `pid2` command to create a 2-DOF PID controller, the **b**, and **c** input arguments, respectively, set the initial values of these properties.

### **Kp, Ki, Kd**

PID controller gains.

Proportional, integral, and derivative gains, respectively. **Kp**, **Ki**, and **Kd** values are real and finite. When you use the `pid2` command to create a 2-DOF PID controller, the **Kp**, **Ki**, and **Kd** input arguments, respectively, set the initial values of these properties.

### **Tf**

Derivative filter time constant.

The **Tf** property stores the derivative filter time constant of the `pid2` controller object. **Tf** is real, finite, and greater than or equal to zero. When you create a 2-DOF PID controller using the `pid2` command, the **Tf** input argument sets the initial value of this property.

**IFormula**

Discrete integrator formula  $IF(z)$  for the integrator of the discrete-time `pid2` controller `C2`. The relationship between the inputs and output of `C2` is given by:

$$u = K_p (br - y) + K_i IF(z)(r - y) + \frac{K_d}{T_f + DF(z)} (cr - y).$$

`IFormula` can take the following values:

- 'ForwardEuler' —  $IF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the `ForwardEuler` formula can result in instability, even when discretizing a system that is stable in continuous time.

- 'BackwardEuler' —  $IF(z) = \frac{T_s z}{z-1}$ .

An advantage of the `BackwardEuler` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- 'Trapezoidal' —  $IF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the `Trapezoidal` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the `Trapezoidal` formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

When `C2` is a continuous-time controller, `IFormula` is ''.

**Default:** 'ForwardEuler'

**DFormula**

Discrete integrator formula  $DF(z)$  for the derivative filter of the discrete-time `pid2` controller `C2`. The relationship between the inputs and output of `C2` is given by:

$$u = K_p (br - y) + K_i IF(z)(r - y) + \frac{K_d}{T_f + DF(z)} (cr - y).$$

DFormula can take the following values:

- 'ForwardEuler' —  $DF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the `ForwardEuler` formula can result in instability, even when discretizing a system that is stable in continuous time.

- 'BackwardEuler' —  $DF(z) = \frac{T_s z}{z-1}$ .

An advantage of the `BackwardEuler` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- 'Trapezoidal' —  $DF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the `Trapezoidal` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the `Trapezoidal` formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

The `Trapezoidal` value for `DFormula` is not available for a `pid2` controller with no derivative filter (`Tf = 0`).

When `C2` is a continuous-time controller, `DFormula` is `' '`.

**Default:** `'ForwardEuler'`

### **InputDelay**

Time delay on the system input. `InputDelay` is always 0 for a `pid2` controller object.

### **OutputDelay**

Time delay on the system Output. `OutputDelay` is always 0 for a `pid2` controller object.

### **Ts**

Sample time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time  $T_s$ , and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel name, specified as a character vector or a 2-by-1 cell array of character vectors. Use this property to name the input channels of the controller model. For

example, assign the names `setpoint` and `measurement` to the inputs of a 2-DOF PID controller model `C` as follows.

```
C.InputName = {'setpoint'; 'measurement'};
```

Alternatively, use automatic vector expansion to assign both input names. For example:

```
C.InputName = 'C-input';
```

The input names automatically expand to `'C-input(1)'; 'C-input(2)'`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `C.u` is equivalent to `C.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** `{ '' ; '' }`

### **InputUnit**

Input channel units, specified as a 2-by-1 cell array of character vectors. Use this property to track input signal units. For example, assign the units `Volts` to the reference input and the concentration units `mol/m^3` to the measurement input of a 2-DOF PID controller model `C` as follows.

```
C.InputUnit = {'Volts'; 'mol/m^3'};
```

`InputUnit` has no effect on system behavior.

**Default:** `{ '' ; '' }`

### **InputGroup**

Input channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **OutputName**

Output channel name, specified as a character vector. Use this property to name the output channel of the controller model. For example, assign the name `control` to the output of a controller model `C` as follows.

```
C.OutputName = 'control';
```

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `C.y` is equivalent to `C.OutputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### **OutputUnit**

Output channel units, specified as a character vector. Use this property to track output signal units. For example, assign the unit `Volts` to the output of a controller model `C` as follows.

```
C.OutputUnit = 'Volts';
```

`OutputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

### **OutputGroup**

Output channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **Name**

System name, specified as a character vector. For example, `'system_1'`.

**Default:** ''

### **Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** {}

**UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

**SamplingGrid**

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta,w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:, :, 1, 1) [zeta=0.3, w=5] =
```

```
      25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

$$\frac{25}{s^2 + 3.5 s + 25}$$

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sLinearizer` populate `SamplingGrid` in this way.

**Default:** []

## Examples

### 2-DOF PDF Controller

Create a continuous-time 2-DOF controller with proportional and derivative gains and a filter on the derivative term. To do so, set the integral gain to zero. Set the other gains and the filter time constant to the desired values.

```
Kp = 1;
Ki = 0; % No integrator
Kd = 3;
Tf = 0.1;
b = 0.5; % setpoint weight on proportional term
c = 0.5; % setpoint weight on derivative term
C2 = pid2(Kp,Ki,Kd,Tf,b,c)
```

C2 =

$$u = K_p (b*r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with  $K_p = 1$ ,  $K_d = 3$ ,  $T_f = 0.1$ ,  $b = 0.5$ ,  $c = 0.5$

Continuous-time 2-DOF PDF controller in parallel form.



The display shows the controller type, formula, and parameter values, and verifies that the controller has no integrator term.

### Discrete-Time 2-DOF PI Controller

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

```
Kp = 5;
Ki = 2.4;
Kd = 0;
Tf = 0;
b = 0.5;
c = 0;
Ts = 0.1;
C2 = pid2(Kp,Ki,Kd,Tf,b,c,Ts, 'IFormula', 'Trapezoidal')
```

C2 =

$$u = K_p (b*r - y) + K_i \frac{T_s*(z+1)}{2*(z-1)} (r - y)$$

with  $K_p = 5$ ,  $K_i = 2.4$ ,  $b = 0.5$ ,  $T_s = 0.1$

Sample time: 0.1 seconds

Discrete-time 2-DOF PI controller in parallel form.

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

### 2-DOF PID Controller with Named Inputs and Output

Create a 2-DOF PID controller, and set the dynamic system properties `InputName` and `OutputName`. Naming inputs and outputs is useful, for example, when you interconnect the PID controller with other dynamic system models using the `connect` command.

```
C2 = pid2(1,2,3,0,1,1, 'InputName', {'r', 'y'}, 'OutputName', 'u')
```

C2 =

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d*s (c*r-y)$$

with  $K_p = 1$ ,  $K_i = 2$ ,  $K_d = 3$ ,  $b = 1$ ,  $c = 1$

Continuous-time 2-DOF PID controller in parallel form.

A 2-DOF PID controller has two inputs and one output. Therefore, the 'InputName' property is an array containing two names, one for each input. The model display does not show the input and output names for the PID controller, but you can examine the property values to see them. For instance, verify the input name of the controller.

C2.InputName

ans =

2×1 cell array

'r'  
'y'

### Array of 2-DOF PID Controllers

Create a 2-by-3 grid of 2-DOF PI controllers with proportional gain ranging from 1–2 across the array rows and integral gain ranging from 5–9 across columns.

To build the array of PID controllers, start with arrays representing the gains.

```
Kp = [1 1 1;2 2 2];  
Ki = [5:2:9;5:2:9];
```

When you pass these arrays to the `pid2` command, the command returns the array of controllers.

```
pi_array = pid2(Kp,Ki,0,0,0.5,0,'Ts',0.1,'IFormula','BackwardEuler');  
size(pi_array)
```

2x3 array of 2-DOF PID controller.  
Each PID has 1 output and 2 inputs.

If you provide scalar values for some coefficients, `pid2` automatically expands them and assigns the same value to all entries in the array. For instance, in this example, `Kd = Tf = 0`, so that all entries in the array are PI controllers. Also, all entries in the array have `b = 0.5`.

Access entries in the array using array indexing. For dynamic system arrays, the first two dimensions are the I/O dimensions of the model, and the remaining dimensions are the array dimensions. Therefore, the following command extracts the (2,3) entry in the array.

```
pi23 = pi_array(:,:,2,3)
```

```
pi23 =
```

$$u = K_p (b*r-y) + K_i \frac{T_s*z}{z-1} (r-y)$$

```
with Kp = 2, Ki = 9, b = 0.5, Ts = 0.1
```

```
Sample time: 0.1 seconds
```

```
Discrete-time 2-DOF PI controller in parallel form.
```

You can also build an array of PID controllers using the `stack` command.

```
C2 = pid2(1,5,0.1,0,0.5,0.5);           % PID controller
C2f = pid2(1,5,0.1,0.5,0.5,0.5);       % PID controller with filter
pid_array = stack(2,C2,C2f);           % stack along 2nd array dimension
```

These commands return a 1-by-2 array of controllers.

```
size(pid_array)
```

```
1x2 array of 2-DOF PID controller.
```

```
Each PID has 1 output and 2 inputs.
```

All PID controllers in an array must have the same sample time, discrete integrator formulas, and dynamic system properties such as `InputName` and `OutputName`.

### Convert 2-DOF PID Controller from Standard to Parallel Form

Convert a standard-form `pidstd2` controller to parallel form.

Standard PID form expresses the controller actions in terms of an overall proportional gain  $K_p$ , integrator and derivative time constants  $T_i$  and  $T_d$ , and filter divisor  $N$ . You can convert any 2-DOF standard-form controller to parallel form using the `pid2` command. For example, consider the following standard-form controller.

```
Kp = 2;
Ti = 3;
Td = 4;
N = 50;
b = 0.1;
c = 0.5;
C2_std = pidstd2(Kp,Ti,Td,N,b,c)
```

C2\_std =

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{1}{s} * (r-y) + T_d * \frac{s}{(T_d/N)*s+1} * (c*r-y)]$$

with  $K_p = 2$ ,  $T_i = 3$ ,  $T_d = 4$ ,  $N = 50$ ,  $b = 0.1$ ,  $c = 0.5$

Continuous-time 2-DOF PIDF controller in standard form

Convert this controller to parallel form using `pid2`.

```
C2_par = pid2(C2_std)
```

C2\_par =

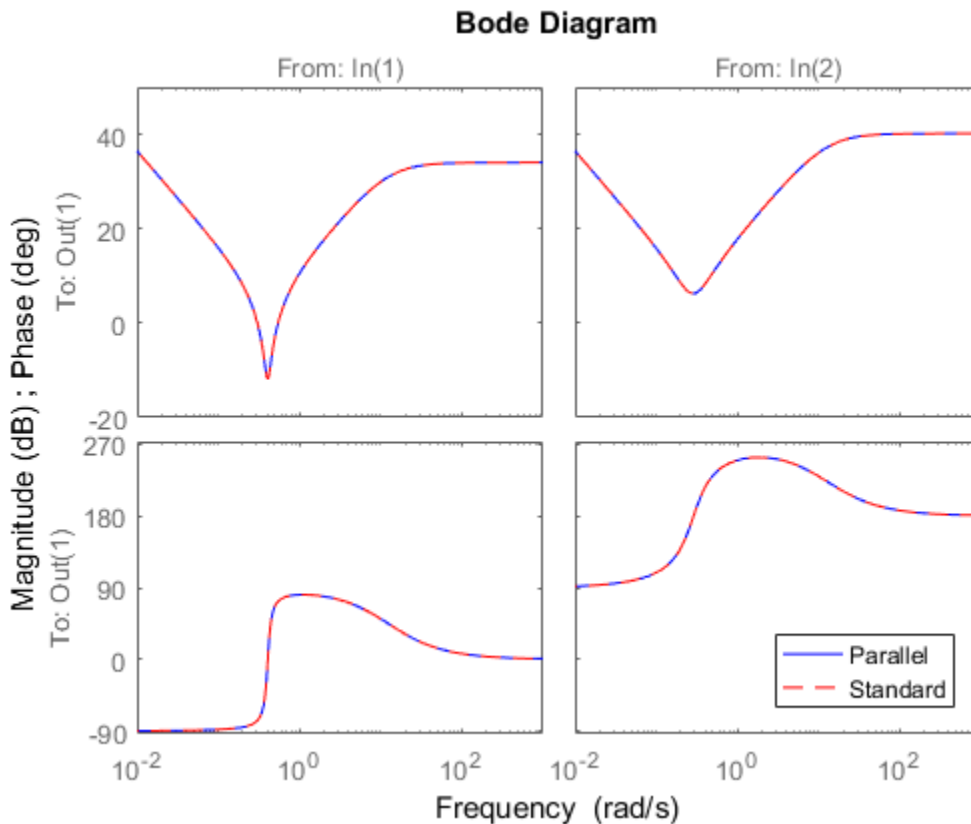
$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with  $K_p = 2$ ,  $K_i = 0.667$ ,  $K_d = 8$ ,  $T_f = 0.08$ ,  $b = 0.1$ ,  $c = 0.5$

Continuous-time 2-DOF PIDF controller in parallel form.

A response plot confirms that the two forms are equivalent.

```
bodeplot(C2_par, 'b-', C2_std, 'r--')
legend('Parallel', 'Standard', 'Location', 'Southeast')
```



### Convert Dynamic System to Parallel-Form PID Controller

Convert a continuous-time dynamic system that represents a PID controller to parallel pid form.

The following dynamic system, with an integrator and two zeros, is equivalent to a PID controller.

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

Create a zpk model of  $H$ . Then use the pid command to obtain  $H$  in terms of the PID gains  $K_p$ ,  $K_i$ , and  $K_d$ .

```
H = zpk([-1, -2], 0, 3);
C = pid(H)
```

C =

$$K_p + K_i * \frac{1}{s} + K_d * s$$

with Kp = 9, Ki = 6, Kd = 3

Continuous-time PID controller in parallel form.

### Convert Dynamic System to 2-DOF Parallel-Form PID Controller

Convert a discrete-time dynamic system that represents a 2-DOF PID controller with derivative filter to parallel `pid2` form.

The following state-space matrices represent a discrete-time 2-DOF PID controller with a sample time of 0.1 s.

```
A = [1,0;0,0.99];
B = [0.1, -0.1; -0.005, 0.01];
C = [3, 0.2];
D = [2.6, -5.2];
Ts = 0.1;
sys = ss(A,B,C,D,Ts);
```

When you convert `sys` to 2-DOF PID form, the result depends on which discrete integrator formulas you specify for the conversion. For instance, use the default, `ForwardEuler`, for both the integrator and the derivative.

```
C2fe = pid2(sys)
```

C2fe =

$$u = K_p (b*r-y) + K_i \frac{T_s}{z-1} (r-y) + K_d \frac{1}{T_f+T_s/(z-1)} (c*r-y)$$

with Kp = 5, Ki = 3, Kd = 2, Tf = 10, b = 0.5, c = 0.5, Ts = 0.1

Sample time: 0.1 seconds  
Discrete-time 2-DOF PIDF controller in parallel form.

Now convert using the Trapezoidal formula.

```
C2trap = pid2(sys, 'IFormula', 'Trapezoidal', 'DFormula', 'Trapezoidal')
```

C2trap =

$$u = K_p (b*r-y) + K_i \frac{T_s(z+1)}{2*(z-1)} (r-y) + K_d \frac{1}{T_f+T_s/2*(z+1)/(z-1)} (c*r-y)$$

with  $K_p = 4.85$ ,  $K_i = 3$ ,  $K_d = 2$ ,  $T_f = 9.95$ ,  $b = 0.485$ ,  $c = 0.5$ ,  $T_s = 0.1$

Sample time: 0.1 seconds  
Discrete-time 2-DOF PIDF controller in parallel form.

The displays show the difference in resulting coefficient values and functional form.

### Discretize a Continuous-Time 2-DOF PID Controller

Discretize a continuous-time 2-DOF PID controller and specify the integral and derivative filter formulas.

Create a continuous-time controller and discretize it using the zero-order-hold method of the `c2d` command.

```
C2con = pid2(10,5,3,0.5,1,1); % continuous-time 2-DOF PIDF controller
C2dis1 = c2d(C2con,0.1,'zoh')
```

C2dis1 =

$$u = K_p (b*r-y) + K_i \frac{T_s}{z-1} (r-y) + K_d \frac{1}{T_f+T_s/(z-1)} (c*r-y)$$

with  $K_p = 10$ ,  $K_i = 5$ ,  $K_d = 3.31$ ,  $T_f = 0.552$ ,  $b = 1$ ,  $c = 1$ ,  $T_s = 0.1$

Sample time: 0.1 seconds  
Discrete-time 2-DOF PIDF controller in parallel form.

The display shows that `c2d` computes new PID coefficients for the discrete-time controller.

The discrete integrator formulas of the discretized controller depend on the `c2d` discretization method, as described in “Tips”. For the `zoh` method, both `IFormula` and `DFormula` are `ForwardEuler`.

```
C2dis1.IFormula
C2dis1.DFormula
```

```
ans =
```

```
ForwardEuler
```

```
ans =
```

```
ForwardEuler
```

If you want to use different formulas from the ones returned by `c2d`, then you can directly set the `Ts`, `IFormula`, and `DFormula` properties of the controller to the desired values.

```
C2dis2 = C2con;
C2dis2.Ts = 0.1;
C2dis2.IFormula = 'BackwardEuler';
C2dis2.DFormula = 'BackwardEuler';
```

However, these commands do not compute new PID gains for the discretized controller. To see this, examine `C2dis2` and compare the coefficients to `C2con` and `C2dis1`.

```
C2dis2
```

```
C2dis2 =
```

$$u = K_p (b*r-y) + K_i \frac{T_s*z}{z-1} (r-y) + K_d \frac{1}{T_f+T_s*z/(z-1)} (c*r-y)$$

```
with Kp = 10, Ki = 5, Kd = 3, Tf = 0.5, b = 1, c = 1, Ts = 0.1
```



Sample time: 0.1 seconds  
 Discrete-time 2-DOF PIDF controller in parallel form.

- “Two-Degree-of-Freedom PID Controllers”
- “Discrete-Time Proportional-Integral-Derivative (PID) Controllers”

## More About

### Tips

- To design a PID controller for a particular plant, use `pidtune` or `pidTuner`. To create a tunable 2-DOF PID controller as a control design block, use `tunablePID2`.
- To break a 2-DOF controller into two SISO control components, such as a feedback controller and a feedforward controller, use `getComponents`.
- Create arrays of `pid2` controller objects by:
  - Specifying array values for one or more of the coefficients `Kp`, `Ki`, `Kd`, `Tf`, `b`, and `c`.
  - Specifying an array of dynamic systems `sys` to convert to `pid2` controller objects.
  - Using `stack` to build arrays from individual controllers or smaller arrays.
  - Passing an array of plant models to `pidtune`.

In an array of `pid2` controllers, each controller must have the same sample time `Ts` and discrete integrator formulas `IFormula` and `DFormula`.

- To create or convert to a standard-form controller, use `pidstd2`. Standard form expresses the controller actions in terms of an overall proportional gain  $K_p$ , integral and derivative times  $T_i$  and  $T_d$ , and filter divisor  $N$ . For example, the relationship between the inputs and output of a continuous-time standard-form 2-DOF PID controller is given by:

$$u = K_p \left[ (br - y) + \frac{1}{T_i s} (r - y) + \frac{T_d s}{N s + 1} (cr - y) \right].$$

- There are two ways to discretize a continuous-time `pid2` controller:

- Use the `c2d` command. `c2d` computes new parameter values for the discretized controller. The discrete integrator formulas of the discretized controller depend upon the `c2d` discretization method you use, as shown in the following table.

<b>c2d Discretization Method</b>	<b>IFormula</b>	<b>DFormula</b>
'zoh'	ForwardEuler	ForwardEuler
'foh'	Trapezoidal	Trapezoidal
'tustin'	Trapezoidal	Trapezoidal
'impulse'	ForwardEuler	ForwardEuler
'matched'	ForwardEuler	ForwardEuler

For more information about `c2d` discretization methods, See the `c2d` reference page. For more information about `IFormula` and `DFormula`, see “Properties” on page 2-695 .

- If you require different discrete integrator formulas, you can discretize the controller by directly setting `Ts`, `IFormula`, and `DFormula` to the desired values. (See “Discretize a Continuous-Time 2-DOF PID Controller” on page 2-709.) However, this method does not compute new gain and filter-constant values for the discretized controller. Therefore, this method might yield a poorer match between the continuous- and discrete-time `pid2` controllers than using `c2d`.
- “What Are Model Objects?”

### See Also

`pidstd2` | `pid` | `piddata2` | `getComponents` | `make1DOF` | `pidtune` | `pidTuner` | `tunablePID2` | `genss` | `realp`

**Introduced in R2015b**

# piddata

Access coefficients of parallel-form PID controller

## Syntax

```
[Kp,Ki,Kd,Tf] = piddata(sys)
[Kp,Ki,Kd,Tf,Ts] = piddata(sys)
[Kp,Ki,Kd,Tf,Ts] = piddata(sys,J1,...,JN)
```

## Description

`[Kp,Ki,Kd,Tf] = piddata(sys)` returns the PID gains `Kp`, `Ki`, `Kd` and the filter time constant `Tf` of the parallel-form controller represented by the dynamic system `sys`.

`[Kp,Ki,Kd,Tf,Ts] = piddata(sys)` also returns the sample time `Ts`.

`[Kp,Ki,Kd,Tf,Ts] = piddata(sys,J1,...,JN)` extracts the data for a subset of entries in `sys`, where `sys` is an  $N$ -dimensional array of dynamic systems. The indices `J` specify the array entry to extract.

## Input Arguments

### **sys**

SISO dynamic system or array of SISO dynamic systems. If `sys` is not a `pid` object, it must represent a valid PID controller that can be written in parallel PID form.

### **J**

Integer indices of  $N$  entries in the array `sys` of dynamic systems. For example, suppose `sys` is a 4-by-5 (two-dimensional) array of `pid` controllers or dynamic system models that represent PID controllers. The following command extracts the data for entry (2,3) in the array.

```
[Kp,Ki,Kd,Tf,Ts] = piddata(sys,2,3);
```

## Output Arguments

### **Kp**

Proportional gain of the parallel-form PID controller represented by dynamic system `sys`.

If `sys` is a `pid` controller object, the output `Kp` is equal to the `Kp` value of `sys`.

If `sys` is not a `pid` object, `Kp` is the proportional gain of a parallel PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, `Kp` is an array of the same dimensions as `sys`.

### **Ki**

Integral gain of the parallel-form PID controller represented by dynamic system `sys`.

If `sys` is a `pid` controller object, then the output `Ki` is equal to the `Ki` value of `sys`.

If `sys` is not a `pid` object, then `Ki` is the integral gain of a parallel PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, then `Ki` is an array of the same dimensions as `sys`.

### **Kd**

Derivative gain of the parallel-form PID controller represented by dynamic system `sys`.

If `sys` is a `pid` controller object, then the output `Kd` is equal to the `Kd` value of `sys`.

If `sys` is not a `pid` object, then `Kd` is the derivative gain of a parallel PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, then `Kd` is an array of the same dimensions as `sys`.

### **Tf**

Filter time constant of the parallel-form PID controller represented by dynamic system `sys`.

If `sys` is a `pid` controller object, the output `Tf` is equal to the `Tf` value of `sys`.

If `sys` is not a `pid` object, `Tf` is the filter time constant of a parallel PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, `Tf` is an array of the same dimensions as `sys`.

### **Ts**

Sample time of the dynamic system `sys`. `Ts` is always a scalar value.

## **Examples**

Extract the proportional, integral, and derivative gains and the filter time constant from a parallel-form `pid` controller.

For the following `pid` object:

```
sys = pid(1,4,0.3,10);
```

you can extract the parameter values from `sys` by entering:

```
[Kp Ki Kd Tf] = piddata(sys);
```

Extract the parallel form proportional and integral gains from an equivalent standard-form PI controller.

For a standard-form PI controller, such as:

```
sys = pidstd(2,3);
```

you can extract the gains of an equivalent parallel-form PI controller by entering:

```
[Kp Ki] = piddata(sys)
```

These commands return the result:

```
Kp =
```

```
2
```

```
Ki =
```

0.6667

Extract parameters from a dynamic system that represents a PID controller.

The dynamic system

$$H(z) = \frac{(z-0.5)(z-0.6)}{(z-1)(z+0.8)}$$

represents a discrete-time PID controller with a derivative filter. Use `piddata` to extract the parallel-form PID parameters.

```
H = zpk([0.5 0.6],[1,-0.8],1,0.1); % sample time Ts = 0.1s
[Kp Ki Kd Tf Ts] = piddata(H);
```

the `piddata` function uses the default `ForwardEuler` discrete integrator formula for `IFormula` and `DFormula` to compute the parameter values.

Extract the gains from an array of PI controllers.

```
sys = pid(rand(2,3),rand(2,3)); % 2-by-3 array of PI controllers
[Kp Ki Kd Tf] = piddata(sys);
```

The parameters `Kp`, `Ki`, `Kd`, and `Tf` are also 2-by-3 arrays.

Use the index input `J` to extract the parameters of a subset of `sys`.

```
[Kp Ki Kd Tf] = piddata(sys,5);
```

## More About

### Tips

If `sys` is not a `pid` controller object, `piddata` returns the PID gains `Kp`, `Ki`, `Kd` and the filter time constant `Tf` of a parallel-form controller equivalent to `sys`.

For discrete-time `sys`, `piddata` returns the parameters of an equivalent parallel-form controller. This controller has discrete integrator formulas `IFormula` and `DFormula` set to `ForwardEuler`. See the `pid` reference page for more information about discrete integrator formulas.

## **See Also**

pid | pidstd | get

**Introduced in R2010b**

## piddata2

Access coefficients of parallel-form 2-DOF PID controller

### Syntax

```
[Kp,Ki,Kd,Tf,b,c] = piddata2(sys)
[Kp,Ki,Kd,Tf,b,c,Ts] = piddata2(sys)
[Kp,Ki,Kd,Tf,b,c,Ts] = piddata2(sys,J1,...,JN)
```

### Description

`[Kp,Ki,Kd,Tf,b,c] = piddata2(sys)` returns the PID gains `Kp`, `Ki`, `Kd`, the filter time constant `Tf`, and the setpoint weights `b` and `c` of the parallel-form 2-DOF PID controller represented by the dynamic system `sys`.

If `sys` is a `pid2` controller object, then each output argument is the corresponding coefficient in `sys`.

If `sys` is not a `pid2` object, then each output argument is the corresponding coefficient of the parallel-form 2-DOF PID controller that is equivalent to `sys`.

If `sys` is an array of dynamic systems, then each output argument is an array of the same dimensions as `sys`.

`[Kp,Ki,Kd,Tf,b,c,Ts] = piddata2(sys)` also returns the sample time `Ts`. For discrete-time `sys` that is not a `pid2` object, `piddata2` calculates the coefficient values using the default `ForwardEuler` discrete integrator formula for both `IFormula` and `DFormula`. See the `pid2` reference page for more information about discrete integrator formulas.

`[Kp,Ki,Kd,Tf,b,c,Ts] = piddata2(sys,J1,...,JN)` extracts the data for a subset of entries in `sys`, where `sys` is an N-dimensional array of dynamic systems. The indices `J` specify the array entry to extract.



## Examples

### Extract Coefficients from Parallel-Form 2-DOF PID Controller

Typically, you extract coefficients from a controller obtained from another function, such as `pidtune` or `getBlockValue`. For this example, create a 2-DOF PID controller that has random coefficients.

```
rng('default'); % for reproducibility
C2 = pid2(rand,rand,rand,rand,rand,rand);
```

Extract the PID coefficients, filter time constant, and setpoint weights.

```
[Kp,Ki,Kd,Tf,b,c] = piddata2(C2);
```

### Extract Parallel-Form Gains from Standard-Form Controller

Create a 2-DOF PID controller in standard form.

```
C2 = pidstd2(2,3,4,10,0.5,0.5)
```

C2 =

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{1}{s} * (r-y) + T_d * \frac{s}{(T_d/N)*s+1} * (c*r-y)]$$

with  $K_p = 2$ ,  $T_i = 3$ ,  $T_d = 4$ ,  $N = 10$ ,  $b = 0.5$ ,  $c = 0.5$

Continuous-time 2-DOF PIDF controller in standard form

Compute the coefficients of an equivalent parallel-form PID controller.

```
[Kp,Ki,Kd,Tf,b,c] = piddata2(C2);
```

Check some of the coefficients to confirm that they are different from the standard-form coefficients.

Ki

Ki =

0.6667

Kd

Kd =

8

### Extract 2-DOF PID Coefficients from Equivalent System

Extract coefficients from a two-input, one-output dynamic system that represents a valid 2-DOF parallel-form PID controller.

The following A, B, C, and D matrices form a discrete-time state-space model that represents a 2-DOF PID controller.

```
A = [1,0;0.09975,0.995];
B = [0.00625, -0.00625;0.1245, -0.1241];
C = [0,4];
D = [2.875, -5.75];
sys = ss(A,B,C,D,0.1)
```

sys =

A =

	x1	x2
x1	1	0
x2	0.09975	0.995

B =

	u1	u2
x1	0.00625	-0.00625
x2	0.1245	-0.1241

C =

	x1	x2
y1	0	4

D =

	u1	u2
y1	2.875	-5.75

Sample time: 0.1 seconds  
Discrete-time state-space model.

Extract the PID gains, filter time constant, and setpoint weights of the model.

```
[Kp,Ki,Kd,Tf,b,c,Ts] = piddata2(sys);
```

For a discrete-time system, `piddata2` calculates the coefficient values using the default ForwardEuler discrete integrator formula for both `IFormula` and `DFormula`.

### Extract Coefficients from 2-DOF PI Controller Array

Typically, you obtain an array of controllers by using `pidtune` on an array of plant models. For this example, create a 2-by-3 array of 2-DOF PI controllers with random values of `Kp`, `Ki`, and `b`.

```
rng('default');  
C2 = pid2(rand(2,3),rand(2,3),0,0,rand(2,3),0);
```

Extract all the coefficients from the array.

```
[Kp,Ki,Kd,Tf,b,c] = piddata2(C2);
```

Each of the outputs is itself a 2-by-3 array. For example, examine `Ki`.

`Ki`

`Ki =`

```
    0.2785    0.9575    0.1576  
    0.5469    0.9649    0.9706
```

Extract only the coefficients of entry (2,1) in the array.

```
[Kp21,Ki21,Kd21,Tf21,b21,c21] = piddata2(C2,2,1);
```

Each of these outputs is a scalar.

`Ki21`

`Ki21 =`

0.5469

### Input Arguments

#### **sys** — 2-DOF PID controller

pid2 controller object | dynamic system model | dynamic system array

2-DOF PID controller in parallel form, specified as a `pid2` controller object, a dynamic system model, or a dynamic system array. If `sys` is not a `pid2` controller object, it must be a two-input, one-output model that represents a valid 2-DOF PID controller that can be written in parallel form.

#### **J** — Indices

positive integers

Indices of entry to extract from a model array `sys`, specified as positive integers. Provide as many indices as there are array dimensions in `sys`. For example, suppose `sys` is a 4-by-5 (two-dimensional) array of `pid2` controllers or dynamic system models that represent 2-DOF PID controllers. The following command extracts the data for entry (2,3) in the array.

```
[Kp,Ki,Kd,Tf,b,c,Ts] = piddata2(sys,2,3);
```

### Output Arguments

#### **Kp** — Proportional gain

scalar | array

Proportional gain of the parallel-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

If `sys` is a `pid2` controller object, then `Kp` is the `Kp` value of `sys`.

If `sys` is not a `pid2` object, then `Kp` is the proportional gain of the parallel-form 2-DOF PID controller that is equivalent to `sys`.

If `sys` is an array of dynamic systems, then `Kp` is an array of the same dimensions as `sys`.

**Ki — Integral gain**

scalar | array

Integral gain of the parallel-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**Kd — Derivative gain**

scalar | array

Derivative gain of the parallel-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**Tf — Filter time constant**

scalar | array

Filter time constant of the parallel-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**b — Setpoint weight on proportional term**

scalar | array

Setpoint weight on the proportional term of the parallel-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**c — Setpoint weight on derivative term**

scalar | array

Setpoint weight on the derivative term of the parallel-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**Ts — Sample time**

scalar

Sample time of the `pid2` controller, dynamic system `sys`, or dynamic system array, returned as a scalar.

**See Also**`pid2` | `piddata` | `pidstdata2`**Introduced in R2015b**

## pidstd

Create a PID controller in standard form, convert to standard-form PID controller

### Syntax

```
C = pidstd(Kp,Ti,Td,N)
C = pidstd(Kp,Ti,Td,N,Ts)
C = pidstd(sys)
C = pidstd(Kp)
C = pidstd(Kp,Ti)
C = pidstd(Kp,Ti,Td)
C = pidstd(...,Name,Value)
C = pidstd
```

### Description

`C = pidstd(Kp,Ti,Td,N)` creates a continuous-time PIDF (PID with first-order derivative filter) controller object in standard form. The controller has proportional gain  $K_p$ , integral and derivative times  $T_i$  and  $T_d$ , and first-order derivative filter divisor  $N$ :

$$C = K_p \left( 1 + \frac{1}{T_i} \frac{1}{s} + \frac{T_d s}{N s + 1} \right).$$

`C = pidstd(Kp,Ti,Td,N,Ts)` creates a discrete-time controller with sample time  $T_s$ . The discrete-time controller is:

$$C = K_p \left( 1 + \frac{1}{T_i} IF(z) + \frac{T_d}{\frac{T_d}{N} + DF(z)} \right).$$

$IF(z)$  and  $DF(z)$  are the *discrete integrator formulas* for the integrator and derivative filter. By default,

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

To choose different discrete integrator formulas, use the `IFormula` and `DFormula` inputs. (See “Properties” on page 2-728 for more information about `IFormula` and `DFormula`). If `DFormula` = 'ForwardEuler' (the default value) and  $N \neq \text{Inf}$ , then  $T_s$ ,  $T_d$ , and  $N$  must satisfy  $T_d/N > T_s/2$ . This requirement ensures a stable derivative filter pole.

`C = pidstd(sys)` converts the dynamic system `sys` to a standard form `pidstd` controller object.

`C = pidstd(Kp)` creates a continuous-time proportional (P) controller with  $T_i = \text{Inf}$ ,  $T_d = 0$ , and  $N = \text{Inf}$ .

`C = pidstd(Kp, Ti)` creates a proportional and integral (PI) controller with  $T_d = 0$  and  $N = \text{Inf}$ .

`C = pidstd(Kp, Ti, Td)` creates a proportional, integral, and derivative (PID) controller with  $N = \text{Inf}$ .

`C = pidstd(..., Name, Value)` creates a controller or converts a dynamic system to a `pidstd` controller object with additional options specified by one or more `Name, Value` pair arguments.

`C = pidstd` creates a P controller with  $K_p = 1$ .

## Input Arguments

### **Kp**

Proportional gain.

`Kp` can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).

- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

**Default:** 1

### **Ti**

Integrator time.

Ti can be:

- A real and positive value.
- An array of real and positive values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

**Default:** Inf

### **Td**

Derivative time.

Td can be:

- A real, finite, and nonnegative value.
- An array of real, finite, and nonnegative values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $T_d = 0$ , the controller has no derivative action.

**Default:** 0

### **N**

Derivative filter divisor.

N can be:

- A real and positive value.
- An array of real and positive values.



- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $N = \text{Inf}$ , the controller has no filter on the derivative action.

**Default:** `Inf`

### **Ts**

Sample time.

To create a discrete-time `pidstd` controller, provide a positive real value ( $T_s > 0$ ). `pidstd` does not support discrete-time controller with undetermined sample time ( $T_s = -1$ ).

$T_s$  must be a scalar value. In an array of `pidstd` controllers, each controller must have the same  $T_s$ .

**Default:** `0` (continuous time)

### **sys**

SISO dynamic system to convert to standard `pidstd` form.

`sys` must represent a valid controller that can be written in standard form with  $T_i > 0$ ,  $T_d \geq 0$ , and  $N > 0$ .

`sys` can also be an array of SISO dynamic systems.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Use `Name, Value` syntax to set the numerical integration formulas `IFormula` and `DFormula` of a discrete-time `pidstd` controller, or to set other object properties such as `InputName` and `OutputName`. For information about available properties of `pidstd` controller objects, see “Properties” on page 2-728.

## Output Arguments

### **C**

`pidstd` object representing a single-input, single-output PID controller in standard form.

The controller type (P, PI, PD, PDF, PID, PIDF) depends upon the values of `Kp`, `Ti`, `Td`, and `N`. For example, when `Td = Inf` and `Kp` and `Ti` are finite and nonzero, `C` is a PI controller. Enter `getType(C)` to obtain the controller type.

When the inputs `Kp`, `Ti`, `Td`, and `N` or the input `sys` are arrays, `C` is an array of `pidstd` objects.

## Properties

### **Kp**

Proportional gain. `Kp` must be real and finite.

### **Ti**

Integral time. `Ti` must be real, finite, and greater than or equal to zero.

### **Td**

Derivative time. `Td` must be real, finite, and greater than or equal to zero.

### **N**

Derivative filter divisor. `N` must be real, and greater than or equal to zero.

### **IFormula**

Discrete integrator formula  $IF(z)$  for the integrator of the discrete-time `pidstd` controller `C`:

$$C = K_p \left( 1 + \frac{1}{T_i} IF(z) + \frac{T_d}{\frac{T_d}{N} + DF(z)} \right).$$

`IFormula` can take the following values:

- 'ForwardEuler' —  $IF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the **ForwardEuler** formula can result in instability, even when discretizing a system that is stable in continuous time.

- 'BackwardEuler' —  $IF(z) = \frac{T_s z}{z-1}$ .

An advantage of the **BackwardEuler** formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- 'Trapezoidal' —  $IF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the **Trapezoidal** formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the **Trapezoidal** formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

When **C** is a continuous-time controller, **IFormula** is ' '.

**Default:** 'ForwardEuler'

### **DFormula**

Discrete integrator formula  $DF(z)$  for the derivative filter of the discrete-time **pidstd** controller **C**:

$$C = K_p \left( 1 + \frac{1}{T_i} IF(z) + \frac{T_d}{\frac{T_d}{N} + DF(z)} \right)$$

**DFormula** can take the following values:

- 'ForwardEuler' —  $DF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the `ForwardEuler` formula can result in instability, even when discretizing a system that is stable in continuous time.

- 'BackwardEuler' —  $DF(z) = \frac{T_s z}{z-1}$ .

An advantage of the `BackwardEuler` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- 'Trapezoidal' —  $DF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the `Trapezoidal` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the `Trapezoidal` formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

The `Trapezoidal` value for `DFormula` is not available for a `pidstd` controller with no derivative filter (`N = Inf`).

When `C` is a continuous-time controller, `DFormula` is ''.

**Default:** 'ForwardEuler'

### **InputDelay**

Time delay on the system input. `InputDelay` is always 0 for a `pidstd` controller object.

### **OutputDelay**

Time delay on the system Output. `OutputDelay` is always 0 for a `pidstd` controller object.

### **Ts**

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel name, specified as a character vector. Use this property to name the input channel of the controller model. For example, assign the name `error` to the input of a controller model `C` as follows.

```
C.InputName = 'error';
```

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `C.u` is equivalent to `C.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### **InputUnit**

Input channel units, specified as a character vector. Use this property to track input signal units. For example, assign the concentration units `mol/m^3` to the input of a controller model `C` as follows.

```
C.InputUnit = 'mol/m^3';
```

`InputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

### **InputGroup**

Input channel groups. This property is not needed for PID controller models.

**Default:** `struct` with no fields

### **OutputName**

Output channel name, specified as a character vector. Use this property to name the output channel of the controller model. For example, assign the name `control` to the output of a controller model `C` as follows.

```
C.OutputName = 'control';
```

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `C.y` is equivalent to `C.OutputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### OutputUnit

Output channel units, specified as a character vector. Use this property to track output signal units. For example, assign the unit `Volts` to the output of a controller model `C` as follows.

```
C.OutputUnit = 'Volts';
```

`OutputUnit` has no effect on system behavior.

**Default:** Empty character vector, `''`

### OutputGroup

Output channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### Name

System name, specified as a character vector. For example, `'system_1'`.

**Default:** `''`

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** `{}`

### UserData

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** `[]`

### SamplingGrid

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta,w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:,:,1,1) [zeta=0.3, w=5] =
```

```

      25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```

      25
-----
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sLinearizer` populate `SamplingGrid` in this way.

**Default:** []



## Examples

Create a continuous-time standard-form PDF controller with proportional gain 1, derivative time 3, and a filter divisor of 6.

```
C = pidstd(1,Inf,3,6);
```

```
C =
```

$$K_p * (1 + T_d * \frac{s}{(T_d/N)*s+1})$$

```
with Kp = 1, Td = 3, N = 6
```

Continuous-time PDF controller in standard form

The display shows the controller type, formula, and coefficient values.

Create a discrete-time PI controller with trapezoidal discretization formula.

To create a discrete-time controller, set the value of `Ts` using `Name, Value` syntax.

```
C = pidstd(1,0.5,'Ts',0.1,'IFormula','Trapezoidal') % Ts = 0.1s
```

This command produces the result:

Discrete-time PI controller in standard form:

$$K_p * (1 + \frac{1}{T_i} * \frac{T_s*(z+1)}{2*(z-1)})$$

```
with Kp = 1, Ti = 0.5, Ts = 0.1
```

Alternatively, you can create the same discrete-time controller by supplying `Ts` as the fifth argument after all four PID parameters `Kp`, `Ti`, `Td`, and `N`.

```
C = pidstd(5,2.4,0,Inf,0.1,'IFormula','Trapezoidal');
```

Create a PID controller and set dynamic system properties `InputName` and `OutputName`.

```
C = pidstd(1,0.5,3,'InputName','e','OutputName','u')
```

Create a 2-by-3 grid of PI controllers with proportional gain ranging from 1–2 and integral time ranging from 5–9.

Create a grid of PI controllers with proportional gain varying row to row and integral time varying column to column. To do so, start with arrays representing the gains.

```
Kp = [1 1 1;2 2 2];
Ti = [5:2:9;5:2:9];
pi_array = pidstd(Kp,Ti,'Ts',0.1,'IFormula','BackwardEuler');
```

These commands produce a 2-by-3 array of discrete-time `pidstd` objects. All `pidstd` objects in an array must have the same sample time, discrete integrator formulas, and dynamic system properties (such as `InputName` and `OutputName`).

Alternatively, you can use the `stack` command to build arrays of `pidstd` objects.

```
C = pidstd(1,5,0.1) % PID controller
Cf = pidstd(1,5,0.1,0.5) % PID controller with filter
pid_array = stack(2,C,Cf); % stack along 2nd array dimension
```

These commands produce a 1-by-2 array of controllers. Enter the command:

```
size(pid_array)
```

to see the result

```
1x2 array of PID controller.
Each PID has 1 output and 1 input.
```

Convert a parallel-form `pid` controller to standard form.

Parallel PID form expresses the controller actions in terms of an proportional, integral, and derivative gains  $K_p$ ,  $K_i$ , and  $K_d$ , and a filter time constant  $T_f$ . You can convert a parallel form controller `parsys` to standard form using `pidstd`, provided that:

- `parsys` is not a pure integrator (I) controller.
- The gains  $K_p$ ,  $K_i$ , and  $K_d$  of `parsys` all have the same sign.

```
parsys = pid(2,3,4,5); % Standard-form controller
stdsys = pidstd(parsys)
```

These commands produce a parallel-form controller:

Continuous-time PIDF controller in standard form:

$$K_p * \left( 1 + \frac{1}{T_i} * \frac{1}{s} + T_d * \frac{s}{(T_d/N)*s+1} \right)$$

with  $K_p = 2$ ,  $T_i = 0.66667$ ,  $T_d = 2$ ,  $N = 0.4$

Convert a continuous-time dynamic system that represents a PID controller to standard `pidstd` form.

The dynamic system

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

represents a PID controller. Use `pidstd` to obtain  $H(s)$  in terms of the standard-form PID parameters  $K_p$ ,  $T_i$ , and  $T_d$ .

```
H = zpk([-1, -2],0,3);
C = pidstd(H)
```

These commands produce the result:

Continuous-time PID controller in standard form:

$$K_p * \left( 1 + \frac{1}{T_i} * \frac{1}{s} + T_d * s \right)$$

with  $K_p = 9$ ,  $T_i = 1.5$ ,  $T_d = 0.33333$

Convert a discrete-time dynamic system that represents a PID controller with derivative filter to standard `pidstd` form.

```
% PIDF controller expressed in zpk form
sys = zpk([-0.5, -0.6], [1 -0.2], 3, 'Ts', 0.1)
```

The resulting `pidstd` object depends upon the discrete integrator formula you specify for `IFormula` and `DFormula`.

For example, if you use the default `ForwardEuler` for both formulas:

```
C = pidstd(sys)
```

you obtain the result:

Discrete-time PIDF controller in standard form:

$$K_p * \left( 1 + \frac{1}{T_i} * \frac{T_s}{z-1} + T_d * \frac{1}{(T_d/N)+T_s/(z-1)} \right)$$

with  $K_p = 2.75$ ,  $T_i = 0.045833$ ,  $T_d = 0.0075758$ ,  $N = 0.090909$ ,  $T_s = 0.1$

For this particular `sys`, you cannot write `sys` in standard PID form using the `BackwardEuler` formula for the `DFormula`. Doing so would result in  $N < 0$ , which is not permitted. In that case, `pidstd` returns an error.

Similarly, you cannot write `sys` in standard form using the `Trapezoidal` formula for both integrators. Doing so would result in negative  $T_i$  and  $T_d$ , which also returns an error.

Discretize a continuous-time `pidstd` controller.

First, discretize the controller using the 'zoh' method of `c2d`.

```
Cc = pidstd(1,2,3,4) % continuous-time pidf controller
Cd1 = c2d(Cc,0.1,'zoh')
```

`c2d` computes new parameters for the discrete-time controller:

Discrete-time PIDF controller in standard form:

$$K_p * \left( 1 + \frac{1}{T_i} * \frac{T_s}{z-1} + T_d * \frac{1}{(T_d/N)+T_s/(z-1)} \right)$$

with  $K_p = 1$ ,  $T_i = 2$ ,  $T_d = 3.2044$ ,  $N = 4$ ,  $T_s = 0.1$

The resulting discrete-time controller uses `ForwardEuler` ( $T_s/(z-1)$ ) for both `IFormula` and `DFormula`.

The discrete integrator formulas of the discretized controller depend upon the `c2d` discretization method, as described in “Tips” on page 2-739. To use a different `IFormula` and `DFormula`, directly set `Ts`, `IFormula`, and `DFormula` to the desired values:

```
Cd2 = Cc;
Cd2.Ts = 0.1;
Cd2.IFormula = 'BackwardEuler';
```

```
Cd2.DFormula = 'BackwardEuler';
```

These commands do not compute new parameter values for the discretized controller. To see this, enter:

```
Cd2
```

to obtain the result:

Discrete-time PIDF controller in standard form:

$$K_p * \left( 1 + \frac{1}{T_i} * \frac{T_s * z}{z-1} + T_d * \frac{1}{(T_d/N) + T_s * z / (z-1)} \right)$$

with  $K_p = 1$ ,  $T_i = 2$ ,  $T_d = 3$ ,  $N = 4$ ,  $T_s = 0.1$

## More About

### Tips

- Use `pidstd` either to create a `pidstd` controller object from known PID gain, integral and derivative times, and filter divisor, or to convert a dynamic system model to a `pidstd` object.
- To tune a PID controller for a particular plant, use `pidtune` or `pidTuner`.
- Create arrays of `pidstd` controllers by:
  - Specifying array values for  $K_p, T_i, T_d$ , and  $N$
  - Specifying an array of dynamic systems `sys` to convert to standard PID form
  - Using `stack` to build arrays from individual controllers or smaller arrays

In an array of `pidstd` controllers, each controller must have the same sample time  $T_s$  and discrete integrator formulas `IFormula` and `DFormula`.

- To create or convert to a parallel-form controller, use `pid`. Parallel form expresses the controller actions in terms of proportional, integral, and derivative gains  $K_p$ ,  $K_i$  and  $K_d$ , and a filter time constant  $T_f$ :

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

- There are two ways to discretize a continuous-time `pidstd` controller:
  - Use the `c2d` command. `c2d` computes new parameter values for the discretized controller. The discrete integrator formulas of the discretized controller depend upon the `c2d` discretization method you use, as shown in the following table.

<b>c2d Discretization Method</b>	<b>IFormula</b>	<b>DFormula</b>
'zoh'	ForwardEuler	ForwardEuler
'foh'	Trapezoidal	Trapezoidal
'tustin'	Trapezoidal	Trapezoidal
'impulse'	ForwardEuler	ForwardEuler
'matched'	ForwardEuler	ForwardEuler

For more information about `c2d` discretization methods, See the `c2d` reference page. For more information about `IFormula` and `DFormula`, see “Properties” on page 2-728 .

- If you require different discrete integrator formulas, you can discretize the controller by directly setting `Ts`, `IFormula`, and `DFormula` to the desired values. (See this example.) However, this method does not compute new gain and filter-constant values for the discretized controller. Therefore, this method might yield a poorer match between the continuous- and discrete-time `pidstd` controllers than using `c2d`.
- “What Are Model Objects?”

## See Also

`pidstd2` | `pidstddata` | `pidtune` | `pidTuner`

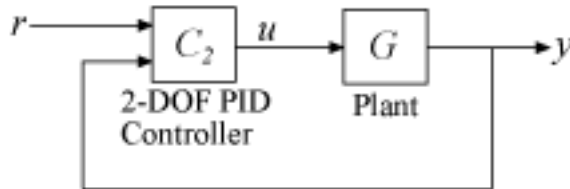
**Introduced in R2010b**

## pidstd2

Create 2-DOF PID controller in standard form, convert to standard-form 2-DOF PID controller

`pid2` controller objects represent two-degree-of-freedom (2-DOF) PID controllers in parallel form. Use `pid2` either to create a `pid2` controller object from known coefficients or to convert a dynamic system model to a `pid2` object.

Two-degree-of-freedom (2-DOF) PID controllers include setpoint weighting on the proportional and derivative terms. A 2-DOF PID controller is capable of fast disturbance rejection without significant increase of overshoot in setpoint tracking. 2-DOF PID controllers are also useful to mitigate the influence of changes in the reference signal on the control signal. The following illustration shows a typical control architecture using a 2-DOF PID controller.



## Syntax

```

C2 = pidstd2(Kp,Ti,Td,N,b,c)
C2 = pidstd2(Kp,Ti,Td,N,b,c,Ts)
C2 = pidstd2(sys)
C2 = pid2(__,Name,Value)

```

## Description

`C2 = pidstd2(Kp,Ti,Td,N,b,c)` creates a continuous-time 2-DOF PID controller with proportional gain  $K_p$ , integrator and derivative time constants  $T_i$ , and  $T_d$ ,

and derivative filter divisor  $N$ . The controller also has setpoint weighting  $b$  on the proportional term, and setpoint weighting  $c$  on the derivative term. The relationship between the 2-DOF controller's output ( $u$ ) and its two inputs ( $r$  and  $y$ ) is given by:

$$u = K_p \left[ (br - y) + \frac{1}{T_i s} (r - y) + \frac{T_d s}{N s + 1} (cr - y) \right].$$

This representation is in *standard form*. If all of the coefficients are real-valued, then the resulting `C2` is a `pidstd2` controller object. If one or more of these coefficients is tunable (`realp` or `genmat`), then `C2` is a tunable generalized state-space (`genss`) model object.

`C2 = pidstd2(Kp, Ti, Td, N, b, c, Ts)` creates a discrete-time 2-DOF PID controller with sample time  $T_s$ . The relationship between the controller's output and inputs is given by:

$$u = K_p \left[ (br - y) + \frac{1}{T_i} IF(z)(r - y) + \frac{T_d}{N + DF(z)} (cr - y) \right].$$

$IF(z)$  and  $DF(z)$  are the *discrete integrator formulas* for the integrator and derivative filter. By default,

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

To choose different discrete integrator formulas, use the `IFormula` and `DFormula` properties. (See “Properties” on page 2-746 for more information). If `DFormula = 'ForwardEuler'` (the default value) and  $N \neq \text{Inf}$ , then  $T_s$ ,  $T_d$ , and  $N$  must satisfy  $T_d/N > T_s/2$ . This requirement ensures a stable derivative filter pole.

`C2 = pidstd2(sys)` converts the dynamic system `sys` to a standard form `pidstd2` controller object.

`C2 = pid2( ___, Name, Value)` specifies additional properties as comma-separated pairs of `Name, Value` arguments.



## Input Arguments

### **Kp**

Proportional gain.

Kp can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (**realp**) or generalized matrix (**genmat**).
- A tunable surface for gain-scheduled tuning, created using **tunableSurface**.

**Default:** 1

### **Ti**

Integrator time.

Ti can be:

- A real and positive value.
- An array of real and positive values.
- A tunable parameter (**realp**) or generalized matrix (**genmat**).
- A tunable surface for gain-scheduled tuning, created using **tunableSurface**.

When  $T_i = \text{Inf}$ , the controller has no integral action.

**Default:** Inf

### **Td**

Derivative time.

Td can be:

- A real, finite, and nonnegative value.
- An array of real, finite, and nonnegative values.
- A tunable parameter (**realp**) or generalized matrix (**genmat**).

- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $Td = 0$ , the controller has no derivative action.

**Default:** 0

### **N**

Derivative filter divisor.

N can be:

- A real and positive value.
- An array of real and positive values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $N = \text{Inf}$ , the controller has no filter on the derivative action.

**Default:** Inf

### **b**

Setpoint weighting on proportional term.

b can be:

- A real, nonnegative, and finite value.
- An array of real, nonnegative, finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $b = 0$ , changes in setpoint do not feed directly into the proportional term.

**Default:** 1

### **c**

Setpoint weighting on derivative term.

$c$  can be:

- A real, nonnegative, and finite value.
- An array of real, nonnegative, finite values.
- A tunable parameter (`realp`) or generalized matrix (`genmat`).
- A tunable surface for gain-scheduled tuning, created using `tunableSurface`.

When  $c = 0$ , changes in setpoint do not feed directly into the proportional term.

**Default:** 1

### **Ts**

Sample time.

To create a discrete-time `pidstd2` controller, provide a positive real value ( $T_s > 0$ ). `pidstd2` does not support discrete-time controller with undetermined sample time ( $T_s = -1$ ).

$T_s$  must be a scalar value. In an array of `pidstd2` controllers, each controller must have the same  $T_s$ .

**Default:** 0 (continuous time)

### **sys**

SISO dynamic system to convert to standard `pidstd2` form.

`sys` be a two-input, one-output system. `sys` must represent a valid 2-DOF controller that can be written in standard form with  $T_i > 0$ ,  $T_d \geq 0$ , and  $N > 0$ .

`sys` can also be an array of SISO dynamic systems.

## **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Use `Name, Value` syntax to set the numerical integration formulas `IFormula` and `DFormula` of a discrete-time `pidstd2` controller, or to set other object properties such as `InputName` and `OutputName`. For information about available properties of `pidstd2` controller objects, see “Properties” on page 2-746.

## Output Arguments

### **C2**

2-DOF PID controller, returned as a `pidstd2` controller object, an array of `pidstd2` controller objects, a `genss` object, or a `genss` array.

- If all the coefficients have scalar numeric values, then **C2** is a `pidstd2` controller object.
- If one or more coefficients is a numeric array, **C2** is an array of `pidstd2` controller objects. The controller type (such as PI, PID, or PDF) depends upon the values of the gains. For example, when `Td = 0`, but `Kp` and `Ti` are nonzero and finite, **C2** is a PI controller.
- If one or more coefficients is a tunable parameter (`realp`), generalized matrix (`genmat`), or tunable gain surface (`tunableSurface`), then **C2** is a generalized state-space model (`genss`).

## Properties

### **b, c**

Setpoint weights on the proportional and derivative terms, respectively. **b** and **c** values are real, finite, and positive. When you create a 2-DOF PID controller using the `pidstd2` command, the initial values of these properties are set by the **b**, and **c** input arguments, respectively.

### **Kp**

Proportional gain.

The value of **Kp** is real and finite. When you create a 2-DOF PID controller using the `pidstd2` command, the initial value of this property is set by the **Kp** input argument.

**Ti**

Integrator time. **Ti** is real and positive. When you create a 2-DOF PID controller using the `pidstd2` command, the initial value of this property is set by the **Ti** input argument. When **Ti** = `Inf`, the controller has no integral action.

**Td**

Derivative time. **Td** is real, finite, and nonnegative. When you create a 2-DOF PID controller using the `pidstd2` command, the initial value of this property is set by the **Td** input argument. When **Td** = 0, the controller has no derivative action.

**N**

Derivative filter divisor. **N** must be real and positive. When you create a 2-DOF PID controller using the `pidstd2` command, the initial value of this property is set by the **N** input argument.

**IFormula**

Discrete integrator formula  $IF(z)$  for the integrator of the discrete-time `pidstd2` controller **C2**. The relationship between the inputs and output of **C2** is given by:

$$u = K_p \left[ (br - y) + \frac{1}{T_i} IF(z)(r - y) + \frac{T_d}{\frac{T_d}{N} + DF(z)} (cr - y) \right].$$

**IFormula** can take the following values:

- 'ForwardEuler' —  $IF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the **ForwardEuler** formula can result in instability, even when discretizing a system that is stable in continuous time.

- 'BackwardEuler' —  $IF(z) = \frac{T_s z}{z-1}$ .

An advantage of the `BackwardEuler` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- 'Trapezoidal' —  $IF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the `Trapezoidal` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the `Trapezoidal` formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

When `C2` is a continuous-time controller, `IFormula` is ' '.

**Default:** 'ForwardEuler'

### DFormula

Discrete integrator formula  $DF(z)$  for the derivative filter of the discrete-time `pidstd2` controller `C2`. The relationship between the inputs and output of `C2` is given by:

$$u = K_p \left[ (br - y) + \frac{1}{T_i} IF(z)(r - y) + \frac{T_d}{\frac{T_d}{N} + DF(z)} (cr - y) \right].$$

`DFormula` can take the following values:

- 'ForwardEuler' —  $DF(z) = \frac{T_s}{z-1}$ .

This formula is best for small sample time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sample time, the `ForwardEuler` formula can result in instability, even when discretizing a system that is stable in continuous time.

- 'BackwardEuler' —  $DF(z) = \frac{T_s z}{z-1}$ .

An advantage of the `BackwardEuler` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result.

- 'Trapezoidal' —  $DF(z) = \frac{T_s}{2} \frac{z+1}{z-1}$ .

An advantage of the `Trapezoidal` formula is that discretizing a stable continuous-time system using this formula always yields a stable discrete-time result. Of all available integration formulas, the `Trapezoidal` formula yields the closest match between frequency-domain properties of the discretized system and the corresponding continuous-time system.

The `Trapezoidal` value for `DFormula` is not available for a `pidstd2` controller with no derivative filter (`N = Inf`).

When `C2` is a continuous-time controller, `DFormula` is `''`.

**Default:** `'ForwardEuler'`

### **InputDelay**

Time delay on the system input. `InputDelay` is always 0 for a `pidstd2` controller object.

### **OutputDelay**

Time delay on the system Output. `OutputDelay` is always 0 for a `pidstd2` controller object.

### **Ts**

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

**TimeUnit**

Units for the time variable, the sample time  $T_s$ , and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

**InputName**

Input channel name, specified as a character vector or a 2-by-1 cell array of character vectors. Use this property to name the input channels of the controller model. For example, assign the names `setpoint` and `measurement` to the inputs of a 2-DOF PID controller model `C` as follows.

```
C.InputName = {'setpoint'; 'measurement'};
```

Alternatively, use automatic vector expansion to assign both input names. For example:

```
C.InputName = 'C-input';
```

The input names automatically expand to `{'C-input(1)'; 'C-input(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `C.u` is equivalent to `C.InputName`.



Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** { '' ; '' }

### **InputUnit**

Input channel units, specified as a 2-by-1 cell array of character vectors. Use this property to track input signal units. For example, assign the units `Volts` to the reference input and the concentration units `mol/m^3` to the measurement input of a 2-DOF PID controller model `C` as follows.

```
C.InputUnit = {'Volts'; 'mol/m^3'};
```

`InputUnit` has no effect on system behavior.

**Default:** { '' ; '' }

### **InputGroup**

Input channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **OutputName**

Output channel name, specified as a character vector. Use this property to name the output channel of the controller model. For example, assign the name `control` to the output of a controller model `C` as follows.

```
C.OutputName = 'control';
```

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `C.y` is equivalent to `C.OutputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### **OutputUnit**

Output channel units, specified as a character vector. Use this property to track output signal units. For example, assign the unit `Volts` to the output of a controller model `C` as follows.

```
C.OutputUnit = 'Volts';
```

`OutputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

### **OutputGroup**

Output channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **Name**

System name, specified as a character vector. For example, 'system\_1'.

**Default:** ''

### **Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, 'System is MIMO'.

**Default:** {}

### **UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

### **SamplingGrid**

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta,w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:,:,1,1) [zeta=0.3, w=5] =
```

```
      25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```
      25
-----
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the

variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sLinearizer` populate `SamplingGrid` in this way.

**Default:** []

## Examples

### 2-DOF PDF Controller

Create a continuous-time 2-DOF PDF controller in standard form. To do so, set the integral time constant to `Inf`. Set the other gains and the filter divisor to the desired values.

```
Kp = 1;
Ti = Inf;    % No integrator
Td = 3;
N = 6;
b = 0.5;    % setpoint weight on proportional term
c = 0.5;    % setpoint weight on derivative term
C2 = pidstd2(Kp,Ti,Td,N,b,c)
```

C2 =

$$u = K_p * [(b*r-y) + T_d * \frac{s}{(T_d/N)*s+1} * (c*r-y)]$$

with  $K_p = 1$ ,  $T_d = 3$ ,  $N = 6$ ,  $b = 0.5$ ,  $c = 0.5$

Continuous-time 2-DOF PDF controller in standard form

The display shows the controller type, formula, and parameter values, and verifies that the controller has no integrator term.

### Discrete-Time 2-DOF PI Controller in Standard Form

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

```

Kp = 1;
Ti = 2.4;
Td = 0;
N = Inf;
b = 0.5;
c = 0;
Ts = 0.1;
C2 = pidstd2(Kp,Ti,Td,N,b,c,Ts, 'IFormula', 'Trapezoidal')

```

C2 =

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{T_s*(z+1)}{2*(z-1)} * (r-y)]$$

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

Sample time: 0.1 seconds

Discrete-time 2-DOF PI controller in standard form

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

## 2-DOF PID Controller with Named Inputs and Output

Create a 2-DOF PID controller in standard form, and set the dynamic system properties InputName and OutputName. Naming inputs and outputs is useful, for example, when you interconnect the PID controller with other dynamic system models using the connect command.

```

C2 = pidstd2(1,2,3,10,1,1, 'InputName', {'r', 'y'}, 'OutputName', 'u')

```

C2 =

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{1}{s} * (r-y) + T_d * \frac{s}{(T_d/N)*s+1} * (c*r-y)]$$

with Kp = 1, Ti = 2, Td = 3, N = 10, b = 1, c = 1

Continuous-time 2-DOF PIDF controller in standard form

A 2-DOF PID controller has two inputs and one output. Therefore, the 'InputName' property is an array containing two names, one for each input. The model display does not show the input and output names for the PID controller, but you can examine the property values to see them. For instance, verify the input name of the controller.

C2.InputName

```
ans =  
  
2x1 cell array  
  
    'r'  
    'y'
```

### Array of 2-DOF PID Controllers

Create a 2-by-3 grid of 2-DOF PI controllers in standard form. The proportional gain ranges from 1–2 across the array rows, and the integrator time constant ranges from 5–9 across columns.

To build the array of PID controllers, start with arrays representing the gains.

```
Kp = [1 1 1;2 2 2];  
Ti = [5:2:9;5:2:9];
```

When you pass these arrays to the `pidstd2` command, the command returns the array of controllers.

```
pi_array = pidstd2(Kp,Ti,0,Inf,0.5,0,'Ts',0.1,'IFormula','BackwardEuler');  
size(pi_array)
```

```
2x3 array of 2-DOF PID controller.  
Each PID has 1 output and 2 inputs.
```

If you provide scalar values for some coefficients, `pidstd2` automatically expands them and assigns the same value to all entries in the array. For instance, in this example, `Td = 0`, so that all entries in the array are PI controllers. Also, all entries in the array have `b = 0.5`.

Access entries in the array using array indexing. For dynamic system arrays, the first two dimensions are the I/O dimensions of the model, and the remaining dimensions are the array dimensions. Therefore, the following command extracts the (2,3) entry in the array.

```
pi23 = pi_array(:,:,2,3)
```

```
pi23 =
```

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{T_s*z}{z-1} * (r-y)]$$

```
with Kp = 2, Ti = 9, b = 0.5, Ts = 0.1
```

```
Sample time: 0.1 seconds
```

```
Discrete-time 2-DOF PI controller in standard form
```

You can also build an array of PID controllers using the `stack` command.

```
C2 = pidstd2(1,5,0.1,Inf,0.5,0.5);           % PID controller
C2f = pidstd2(1,5,0.1,0.5,0.5,0.5);        % PID controller with filter
pid_array = stack(2,C2,C2f);               % stack along 2nd array dimension
```

These commands return a 1-by-2 array of controllers.

```
size(pid_array)
```

```
1x2 array of 2-DOF PID controller.
Each PID has 1 output and 2 inputs.
```

All PID controllers in an array must have the same sample time, discrete integrator formulas, and dynamic system properties such as `InputName` and `OutputName`.

### Convert 2-DOF PID Controller from Parallel to Standard Form

Convert a parallel-form `pid2` controller to standard form.

Parallel PID form expresses the controller actions in terms of proportional, integral, and derivative gains  $K_p$ ,  $K_i$ , and  $K_d$ , and filter time constant  $T_f$ . You can convert a parallel-form `pid2` controller to standard form using the `pidstd2` command, provided that both of the following are true:

- The `pid2` controller can be expressed in valid standard form.
- The gains `Kp`, `Ki`, and `Kd` of the `pid2` controller all have the same sign.

For example, consider the following parallel-form controller.

```
Kp = 2;
Ki = 3;
Kd = 4;
Tf = 2;
b = 0.1;
c = 0.5;
C2_par = pid2(Kp,Ki,Kd,Tf,b,c)
```

`C2_par =`

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with `Kp = 2`, `Ki = 3`, `Kd = 4`, `Tf = 2`, `b = 0.1`, `c = 0.5`

Continuous-time 2-DOF PIDF controller in parallel form.

Convert this controller to parallel form using `pidstd2`.

```
C2_std = pidstd2(C2_par)
```

`C2_std =`

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{1}{s} * (r-y) + T_d * \frac{s}{(T_d/N)*s+1} * (c*r-y)]$$

with `Kp = 2`, `Ti = 0.667`, `Td = 2`, `N = 1`, `b = 0.1`, `c = 0.5`

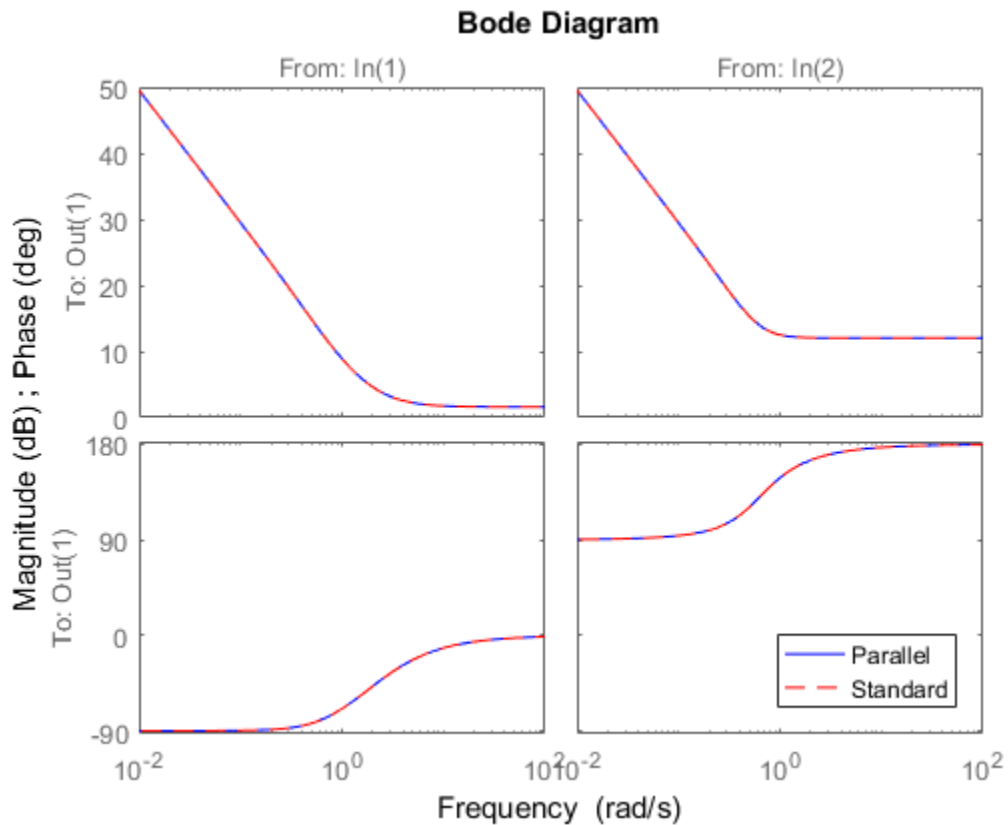
Continuous-time 2-DOF PIDF controller in standard form

The display confirms the new standard form. A response plot confirms that the two forms are equivalent.

```
bodeplot(C2_par, 'b-', C2_std, 'r--')
```



```
legend('Parallel', 'Standard', 'Location', 'Southeast')
```



### Convert Dynamic System to Standard-Form 2-DOF PID Controller

Convert a two-input, one-output continuous-time dynamic system that represents a 2-DOF PID controller to a standard-form `pidstd2` controller.

The following state-space matrices represent a 2-DOF PID controller.

```
A = [0,0;0,-8.181];
B = [1,-1;-0.1109,8.181];
C = [0.2301,10.66];
D = [0.8905,-11.79];
sys = ss(A,B,C,D);
```

Rewrite `sys` in terms of the standard-form PID parameters `Kp`, `Ti`, `Td`, and `N`, and the setpoint weights `b` and `c`.

```
C2 = pidstd2(sys)
```

```
C2 =
```

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{1}{s} * (r-y) + T_d * \frac{s}{(T_d/N)*s+1} * (c*r-y)]$$

```
with Kp = 1.13, Ti = 4.91, Td = 1.15, N = 9.43, b = 0.66, c = 0.0136
```

Continuous-time 2-DOF PIDF controller in standard form

### Convert Discrete-Time Dynamic System to 2-DOF Standard-Form PID Controller

Convert a discrete-time dynamic system that represents a 2-DOF PID controller with derivative filter to standard `pidstd2` form.

The following state-space matrices represent a discrete-time 2-DOF PID controller with a sample time of 0.05 s.

```
A = [1,0;0,0.6643];
B = [0.05,-0.05; -0.004553,0.3357];
C = [0.2301,10.66];
D = [0.8905,-11.79];
Ts = 0.05;
sys = ss(A,B,C,D,Ts);
```

When you convert `sys` to 2-DOF PID form, the result depends on which discrete integrator formulas you specify for the conversion. For instance, use the default, `ForwardEuler`, for both the integrator and the derivative.

```
C2fe = pidstd2(sys)
```

```
C2fe =
```

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{T_s}{z-1} * (r-y) + T_d * \frac{1}{(T_d/N)+T_s/(z-1)} * (c*r-y)]$$

with  $K_p = 1.13$ ,  $T_i = 4.91$ ,  $T_d = 1.41$ ,  $N = 9.43$ ,  $b = 0.66$ ,  $c = 0.0136$ ,  $T_s = 0.05$

Sample time: 0.05 seconds  
Discrete-time 2-DOF PIDF controller in standard form

Now convert using the Trapezoidal formula.

```
C2trap = pidstd2(sys, 'IFormula', 'Trapezoidal', 'DFormula', 'Trapezoidal')
```

C2trap =

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{T_s*(z+1)}{2*(z-1)} * (r-y) + T_d * \frac{1}{(T_d/N)+T_s/2*(z+1)/(z-1)} * (c*r-y)]$$

with  $K_p = 1.12$ ,  $T_i = 4.89$ ,  $T_d = 1.41$ ,  $N = 11.4$ ,  $b = 0.658$ ,  $c = 0.0136$ ,  $T_s = 0.05$

Sample time: 0.05 seconds  
Discrete-time 2-DOF PIDF controller in standard form

The displays show the difference in resulting coefficient values and functional form.

For some dynamic systems, attempting to use the Trapezoidal or BackwardEuler integrator formulas yields invalid results, such as negative  $T_i$ ,  $T_d$ , or  $N$  values. In such cases, `pidstd2` returns an error.

### Discretize a Standard-Form 2-DOF PID Controller

Discretize a continuous-time standard-form 2-DOF PID controller and specify the integral and derivative filter formulas.

Create a continuous-time `pidstd2` controller and discretize it using the zero-order-hold method of the `c2d` command.

```
C2con = pidstd2(10,5,3,0.5,1,1); % continuous-time 2-DOF PIDF controller
C2dis1 = c2d(C2con,0.1,'zoh')
```

C2dis1 =

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{T_s}{z-1} * (r-y) + T_d * \frac{1}{(T_d/N)+T_s/(z-1)} * (c*r-y)]$$

with  $K_p = 10$ ,  $T_i = 5$ ,  $T_d = 3.03$ ,  $N = 0.5$ ,  $b = 1$ ,  $c = 1$ ,  $T_s = 0.1$

Sample time: 0.1 seconds

Discrete-time 2-DOF PIDF controller in standard form

The display shows that `c2d` computes new PID coefficients for the discrete-time controller.

The discrete integrator formulas of the discretized controller depend on the `c2d` discretization method, as described in “Tips”. For the `zoh` method, both `IFormula` and `DFormula` are `ForwardEuler`.

```
C2dis1.IFormula
C2dis1.DFormula
```

```
ans =
```

```
ForwardEuler
```

```
ans =
```

```
ForwardEuler
```

If you want to use different formulas from the ones returned by `c2d`, then you can directly set the `Ts`, `IFormula`, and `DFormula` properties of the controller to the desired values.

```
C2dis2 = C2con;
C2dis2.Ts = 0.1;
C2dis2.IFormula = 'BackwardEuler';
C2dis2.DFormula = 'BackwardEuler';
```

However, these commands do not compute new coefficients for the discretized controller. To see this, examine `C2dis2` and compare the coefficients to `C2con` and `C2dis1`.

```
C2dis2
```

C2dis2 =

$$u = K_p * [(b*r-y) + \frac{1}{T_i} * \frac{T_s*z}{z-1} * (r-y) + T_d * \frac{1}{(T_d/N)+T_s*z/(z-1)} * (c*r-y)]$$

with  $K_p = 10$ ,  $T_i = 5$ ,  $T_d = 3$ ,  $N = 0.5$ ,  $b = 1$ ,  $c = 1$ ,  $T_s = 0.1$

Sample time: 0.1 seconds

Discrete-time 2-DOF PIDF controller in standard form

- “Two-Degree-of-Freedom PID Controllers”
- “Discrete-Time Proportional-Integral-Derivative (PID) Controllers”

## More About

### Tips

- To design a PID controller for a particular plant, use `pidtune` or `pidTuner`. To create a tunable 2-DOF PID controller as a control design block, use `tunablePID2`.
- To break a 2-DOF controller into two SISO control components, such as a feedback controller and a feedforward controller, use `getComponents`.
- Create arrays of `pidstd2` controllers by:
  - Specifying array values for one or more of the coefficients  $K_p$ ,  $T_i$ ,  $T_d$ ,  $N$ ,  $b$ , and  $c$ .
  - Specifying an array of dynamic systems `sys` to convert to `pid2` controller objects.
  - Using `stack` to build arrays from individual controllers or smaller arrays.
  - Passing an array of plant models to `pidtune`.

In an array of `pidstd2` controllers, each controller must have the same sample time  $T_s$  and discrete integrator formulas `IFormula` and `DFormula`.

- To create or convert to a parallel-form controller, use `pid2`. Parallel form expresses the controller actions in terms of proportional, integral, and derivative gains  $K_p$ ,  $K_i$  and  $K_d$ , and a filter time constant  $T_f$ . For example, the relationship between the inputs and output of a continuous-time parallel-form 2-DOF PID controller is given by:

$$u = K_p (br - y) + \frac{K_i}{s} (r - y) + \frac{K_d s}{T_f s + 1} (cr - y).$$

- There are two ways to discretize a continuous-time `pidstd2` controller:
  - Use the `c2d` command. `c2d` computes new parameter values for the discretized controller. The discrete integrator formulas of the discretized controller depend upon the `c2d` discretization method you use, as shown in the following table.

<b>c2d Discretization Method</b>	<b>IFormula</b>	<b>DFormula</b>
'zoh'	ForwardEuler	ForwardEuler
'foh'	Trapezoidal	Trapezoidal
'tustin'	Trapezoidal	Trapezoidal
'impulse'	ForwardEuler	ForwardEuler
'matched'	ForwardEuler	ForwardEuler

For more information about `c2d` discretization methods, See the `c2d` reference page. For more information about `IFormula` and `DFormula`, see “Properties” on page 2-746 .

- If you require different discrete integrator formulas, you can discretize the controller by directly setting `Ts`, `IFormula`, and `DFormula` to the desired values. (See “Discretize a Standard-Form 2-DOF PID Controller” on page 2-761.) However, this method does not compute new gain and filter-constant values for the discretized controller. Therefore, this method might yield a poorer match between the continuous- and discrete-time `pidstd2` controllers than using `c2d`.
- “What Are Model Objects?”

## See Also

`pid2` | `pidstddata2` | `pidtune` | `pidTuner` | `getComponents`

**Introduced in R2015b**

# pidstdata

Access coefficients of standard-form PID controller

## Syntax

```
[Kp,Ti,Td,N] = pidstdata(sys)
[Kp,Ti,Td,N,Ts] = pidstdata(sys)
[Kp,Ti,Td,N,Ts] = pidstdata(sys, J1,...,JN)
```

## Description

`[Kp,Ti,Td,N] = pidstdata(sys)` returns the proportional gain `Kp`, integral time `Ti`, derivative time `Td`, and filter divisor `N` of the standard-form controller represented by the dynamic system `sys`.

`[Kp,Ti,Td,N,Ts] = pidstdata(sys)` also returns the sample time `Ts`.

`[Kp,Ti,Td,N,Ts] = pidstdata(sys, J1,...,JN)` extracts the data for a subset of entries in the array of `sys` dynamic systems. The indices `J` specify the array entries to extract.

## Input Arguments

### **sys**

SISO dynamic system or array of SISO dynamic systems. If `sys` is not a `pidstd` object, it must represent a valid PID controller that can be written in standard PID form.

### **J**

Integer indices of `N` entries in the array `sys` of dynamic systems.

## Output Arguments

### **Kp**

Proportional gain of the standard-form PID controller represented by dynamic system `sys`.

If `sys` is a `pidstd` controller object, the output `Kp` is equal to the `Kp` value of `sys`.

If `sys` is not a `pidstd` object, `Kp` is the proportional gain of a standard-form PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, `Kp` is an array of the same dimensions as `sys`.

### **Ti**

Integral time constant of the standard-form PID controller represented by dynamic system `sys`.

If `sys` is a `pidstd` controller object, the output `Ti` is equal to the `Ti` value of `sys`.

If `sys` is not a `pidstd` object, `Ti` is the integral time constant of a standard-form PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, `Ti` is an array of the same dimensions as `sys`.

### **Td**

Derivative time constant of the standard-form PID controller represented by dynamic system `sys`.

If `sys` is a `pidstd` controller object, the output `Td` is equal to the `Td` value of `sys`.

If `sys` is not a `pidstd` object, `Td` is the derivative time constant of a standard-form PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, `Td` is an array of the same dimensions as `sys`.

### **N**

Filter divisor of the standard-form PID controller represented by dynamic system `sys`.

If `sys` is a `pidstd` controller object, the output `N` is equal to the `N` value of `sys`.



If `sys` is not a `pidstd` object, `N` is the filter time constant of a standard-form PID controller equivalent to `sys`.

If `sys` is an array of dynamic systems, `N` is an array of the same dimensions as `sys`.

### **Ts**

Sample time of the dynamic system `sys`. `Ts` is always a scalar value.

## **Examples**

Extract the proportional, integral, and derivative gains and the filter time constant from a standard-form `pidstd` controller.

For the following `pidstd` object:

```
sys = pidstd(1,4,0.3,10);
```

you can extract the parameter values from `sys` by entering:

```
[Kp Ti Td N] = pidstddata(sys);
```

Extract the standard-form proportional and integral gains from an equivalent parallel-form PI controller.

For a standard-form PI controller, such as:

```
sys = pid(2,3);
```

you can extract the gains of an equivalent parallel-form PI controller by entering:

```
[Kp Ti] = pidstddata(sys)
```

These commands return the result:

```
Kp =
```

```
    2
```

```
Ti =
```

0.6667

Extract parameters from a dynamic system that represents a PID controller.

The dynamic system

$$H(z) = \frac{(z-0.5)(z-0.6)}{(z-1)(z+0.8)}$$

represents a discrete-time PID controller with a derivative filter. Use `pidstddata` to extract the standard-form PID parameters.

```
H = zpk([0.5 0.6],[1,-0.8],1,0.1); % sample time Ts = 0.1s
[Kp Ti Td N Ts] = pidstddata(H);
```

the `pidstddata` function uses the default `ForwardEuler` discrete integrator formula for `Iformula` and `Dformula` to compute the parameter values.

Extract the gains from an array of PI controllers.

```
sys = pidstd(rand(2,3),rand(2,3)); % 2-by-3 array of PI controllers
[Kp Ti Td N] = pidstddata(sys);
```

The parameters `Kp`, `Ti`, `Td`, and `N` are also 2-by-3 arrays.

Use the index input `J` to extract the parameters of a subset of `sys`.

```
[Kp Ti Td N] = pidstddata(sys,5);
```

## More About

### Tips

If `sys` is not a `pidstd` controller object, `pidstddata` returns `Kp`, `Ti`, `Td` and `N` values of a standard-form controller equivalent to `sys`.

For discrete-time `sys`, `piddata` returns parameters of an equivalent `pidstd` controller. This controller has discrete integrator formulas `Iformula` and `Dformula` set to `ForwardEuler`. See the `pidstd` reference page for more information about discrete integrator formulas.

## **See Also**

pidstd | pid | get

**Introduced in R2010b**

## pidstdata2

Access coefficients of standard-form 2-DOF PID controller

### Syntax

```
[Kp,Ti,Td,N,b,c] = pidstdata2(sys)
[Kp,Ti,Td,N,b,c,Ts] = pidstdata2(sys)
[Kp,Ti,Td,N,b,c,Ts] = pidstdata2(sys,J1,...,JN)
```

### Description

`[Kp,Ti,Td,N,b,c] = pidstdata2(sys)` returns the proportional gain `Kp`, integral time `Ti`, derivative time `Td`, the filter divisor `N`, and the setpoint weights `b` and `c` of the standard-form 2-DOF PID controller represented by the dynamic system `sys`.

If `sys` is a `pidstd2` controller object, then each output argument is the corresponding coefficient in `sys`.

If `sys` is not a `pidstd2` object, then each output argument is the corresponding coefficient of the standard-form 2-DOF PID controller that is equivalent to `sys`.

If `sys` is an array of dynamic systems, then each output argument is an array of the same dimensions as `sys`.

`[Kp,Ti,Td,N,b,c,Ts] = pidstdata2(sys)` also returns the sample time `Ts`. For discrete-time `sys` that is not a `pidstd2` object, `pidstdata2` calculates the coefficient values using the default `ForwardEuler` discrete integrator formula for both `IFormula` and `DFormula`. See the `pidstd2` reference page for more information about discrete integrator formulas.

`[Kp,Ti,Td,N,b,c,Ts] = pidstdata2(sys,J1,...,JN)` extracts the data for a subset of entries in `sys`, where `sys` is an `N`-dimensional array of dynamic systems. The indices `J` specify the array entry to extract.

## Examples

### Extract Coefficients from Standard-Form 2-DOF PID Controller

Typically, you extract coefficients from a controller obtained from another function, such as `pidtune` or `getBlockValue`. For this example, create a standard-form 2-DOF PID controller that has random coefficients.

```
rng('default'); % for reproducibility
C2 = pidstd2(rand,rand,rand,rand,rand,rand);
```

Extract the PID coefficients, filter divisor, and setpoint weights.

```
[Kp,Ti,Td,N,b,c] = pidstddata2(C2);
```

### Extract Standard-Form Coefficients from Parallel-Form Controller

Create a 2-DOF PID controller in parallel form.

```
C2 = pid2(2,3,4,10,0.5,0.5)
```

```
C2 =
```

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

with  $K_p = 2$ ,  $K_i = 3$ ,  $K_d = 4$ ,  $T_f = 10$ ,  $b = 0.5$ ,  $c = 0.5$

Continuous-time 2-DOF PIDF controller in parallel form.

Compute the coefficients of an equivalent parallel-form PID controller.

```
[Kp,Ti,Td,N,b,c] = pidstddata2(C2);
```

Check some of the coefficients to confirm that they are different from the parallel-form coefficients.

```
Ti
```

```
Ti =
```

0.6667

Td

Td =

2

### Extract Standard-Form 2-DOF PID Coefficients from Equivalent System

Extract coefficients from a two-input, one-output dynamic system that represents a valid 2-DOF standard-form PID controller.

The following A, B, C, and D matrices form a discrete-time state-space model that represents a 2-DOF PID controller in standard form.

```
A = [1,0;0,0.5];
B = [0.1, -0.1;-0.25,0.5];
C = [4,400];
D = [220, -440];
sys = ss(A,B,C,D,0.1)
```

sys =

A =

	x1	x2
x1	1	0
x2	0	0.5

B =

	u1	u2
x1	0.1	-0.1
x2	-0.25	0.5

C =

	x1	x2
y1	4	400

D =

	u1	u2
y1	220	-440

Sample time: 0.1 seconds  
Discrete-time state-space model.

Extract the PID coefficients, filter divisor, and setpoint weights of the model.

```
[Kp,Ti,Td,N,b,c,Ts] = pidstddata2(sys);
```

For a discrete-time system, `pidstddata2` calculates the coefficient values using the default `ForwardEuler` discrete integrator formula for both `IFormula` and `DFormula`.

### Extract Standard-Form Coefficients from 2-DOF PI Controller Array

Typically, you obtain an array of controllers by using `pidtune` on an array of plant models. For this example, create a 2-by-3 array of standard-form 2-DOF PI controllers with random values of `Kp`, `Ti`, and `b`.

```
rng('default');  
C2 = pidstd2(rand(2,3),rand(2,3),0,10,rand(2,3),0);
```

Extract all the coefficients from the array.

```
[Kp,Ti,Td,N,b,c] = pidstddata2(C2);
```

Each of the outputs is itself a 2-by-3 array. For example, examine `Ki`.

`Ti`

```
Ti =
```

```
    0.2785    0.9575    0.1576  
    0.5469    0.9649    0.9706
```

Extract only the coefficients of entry (2,1) in the array.

```
[Kp21,Ti21,Td21,N21,b21,c21] = pidstddata2(C2,2,1);
```

Each of these outputs is a scalar.

`Ti21`

```
Ti21 =
```

0.5469

## Input Arguments

### **sys** — 2-DOF PID controller

pidstd2 controller object | dynamic system model | dynamic system array

2-DOF PID controller in standard form, specified as a `pidstd2` controller object, a dynamic system model, or a dynamic system array. If `sys` is not a `pidstd2` controller object, it must be a two-input, one-output model that represents a valid 2-DOF PID controller that can be written in standard form.

### **J** — Indices

positive integers

Indices of entry to extract from a model array `sys`, specified as positive integers. Provide as many indices as there are array dimensions in `sys`. For example, suppose `sys` is a 4-by-5 (two-dimensional) array of `pidstd2` controllers or dynamic system models that represent 2-DOF PID controllers. The following command extracts the data for entry (2,3) in the array.

```
[Kp,Ti,Td,N,b,c,Ts] = piddstdata2(sys,2,3);
```

## Output Arguments

### **Kp** — Proportional gain

scalar | array

Proportional gain of the standard-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

If `sys` is a `pidstd2` controller object, then `Kp` is the `Kp` value of `sys`.

If `sys` is not a `pidstd2` object, then `Kp` is the proportional gain of the standard-form 2-DOF PID controller that is equivalent to `sys`.

If `sys` is an array of dynamic systems, then `Kp` is an array of the same dimensions as `sys`.



**Ti — Integral time constant**

scalar | array

Integral time constant of the standard-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**Td — Derivative time constant**

scalar | array

Derivative time constant of the standard-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**N — Filter divisor**

scalar | array

Filter divisor of the parallel-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**b — Setpoint weight on proportional term**

scalar | array

Setpoint weight on the proportional term of the standard-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**c — Setpoint weight on derivative term**

scalar | array

Setpoint weight on the derivative term of the standard-form 2-DOF PID controller represented by `sys`, returned as a scalar or array.

**Ts — Sample time**

scalar

Sample time of the `pidstd2` controller, dynamic system `sys`, or dynamic system array, returned as a scalar.

**See Also**`piddata2` | `pidstd2` | `pidstddata`**Introduced in R2015b**

## pidtool

Open PID Tuner for PID tuning

---

**Note:** pidtool has been removed. Use pidTuner instead.

---

**Introduced in R2010b**

# pidtune

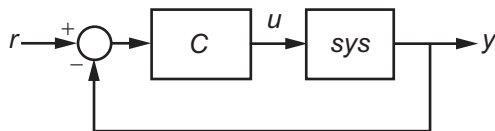
PID tuning algorithm for linear plant model

## Syntax

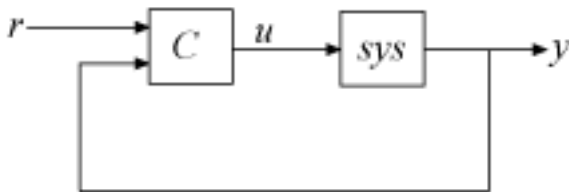
```
C = pidtune(sys,type)
C = pidtune(sys,C0)
C = pidtune(sys,type,wc)
C = pidtune(sys,C0,wc)
C = pidtune(sys,...,opts)
[C,info] = pidtune(...)
```

## Description

`C = pidtune(sys,type)` designs a PID controller of type `type` for the plant `sys`. If `type` specifies a one-degree-of-freedom (1-DOF) PID controller, then the controller is designed for the unit feedback loop as illustrated:



If `type` specifies a two-degree-of-freedom (2-DOF) PID controller, then `pidtune` designs a 2-DOF controller as in the feedback loop of this illustration:



`pidtune` tunes the parameters of the PID controller `C` to balance performance (response time) and robustness (stability margins).

`C = pidtune(sys,C0)` designs a controller of the same type and form as the controller `C0`. If `sys` and `C0` are discrete-time models, `C` has the same discrete integrator formulas as `C0`.

`C = pidtune(sys,type,wc)` and `C = pidtune(sys,C0,wc)` specify a target value `wc` for the first 0 dB gain crossover frequency of the open-loop response.

`C = pidtune(sys,...,opts)` uses additional tuning options, such as the target phase margin. Use `pidtuneOptions` to specify the option set `opts`.

`[C,info] = pidtune(...)` returns the data structure `info`, which contains information about closed-loop stability, the selected open-loop gain crossover frequency, and the actual phase margin.

## Input Arguments

### `sys`

Single-input, single-output dynamic system model of the plant for controller design. `sys` can be:

- Any type of SISO dynamic system model, including Numeric LTI models and identified models. If `sys` is a tunable or uncertain model, `pidtune` designs a controller for the current or nominal value of `sys`.
- A continuous- or discrete-time model.
- Stable, unstable, or integrating. A plant with unstable poles, however, might not be stabilizable under PID control.
- A model that includes any type of time delay. A plant with long time delays, however, might not achieve adequate performance under PID control.
- An array of plant models. If `sys` is an array, `pidtune` designs a separate controller for each plant in the array.

If the plant has unstable poles, and `sys` is one of the following:

- A `frd` model

- A `ss` model with internal time delays that cannot be converted to I/O delays

you must use `pidtuneOptions` to specify the number of unstable poles in the plant, if any.

### **type**

Controller type of the controller to design, specified as a character vector. The term *controller type* refers to which terms are present in the controller action. For example, a PI controller has only a proportional and an integral term, while a PIDF controller contains proportional, integrator, and filtered derivative terms. `type` can take the values summarized below. For more detailed information about these controller types, see “PID Controller Types for Tuning”

## **1-DOF Controllers**

- 'P' — Proportional only
- 'I' — Integral only
- 'PI' — Proportional and integral
- 'PD' — Proportional and derivative
- 'PDF' — Proportional and derivative with first-order filter on derivative term
- 'PID' — Proportional, integral, and derivative
- 'PIDF' — Proportional, integral, and derivative with first-order filter on derivative term

## **2-DOF Controllers**

- 'PI2' — 2-DOF proportional and integral
- 'PD2' — 2-DOF proportional and derivative
- 'PDF2' — 2-DOF proportional and derivative with first-order filter on derivative term
- 'PID2' — 2-DOF proportional, integral, and derivative
- 'PIDF2' — 2-DOF proportional, integral, and derivative with first-order filter on derivative term

For more information about 2-DOF PID controllers generally, see “Two-Degree-of-Freedom PID Controllers”.

### 2-DOF Controllers with Fixed Setpoint Weights

- 'I-PD' — 2-DOF PID with  $b = 0, c = 0$
- 'I-PDF' — 2-DOF PIDF with  $b = 0, c = 0$
- 'ID-P' — 2-DOF PID with  $b = 0, c = 1$
- 'IDF-P' — 2-DOF PIDF with  $b = 0, c = 1$
- 'PI-D' — 2-DOF PID with  $b = 1, c = 0$
- 'PI-DF' — 2-DOF PIDF with  $b = 1, c = 0$

For more detailed information about fixed-setpoint-weight 2-DOF PID controllers, see “PID Controller Types for Tuning”.

### Controller Form

When you use the `type` input, `pidtune` designs a controller in parallel (`pid` or `pid2`) form. Use the input `C0` instead of `type` if you want to design a controller in standard (`pidstd` or `pidstd2`) form.

If `sys` is a discrete-time model with sample time `Ts`, `pidtune` designs a discrete-time controller with the same `Ts`. The controller has the `ForwardEuler` discrete integrator formula for both integral and derivative actions. Use the input `C0` instead of `type` if you want to design a controller having a different discrete integrator formula.

For more information about PID controller forms and formulas, see:

- “Proportional-Integral-Derivative (PID) Controllers”
- “Two-Degree-of-Freedom PID Controllers”
- “Discrete-Time Proportional-Integral-Derivative (PID) Controllers”

#### **C0**

PID controller setting properties of the designed controller, specified as a `pid`, `pidstd`, `pid2`, or `pidstd2` object. If you provide `C0`, `pidtune`:

- Designs a controller of the type represented by `C0`.
- Returns a `pid` controller, if `C0` is a `pid` controller.
- Returns a `pidstd` controller, if `C0` is a `pidstd` controller.
- Returns a 2-DOF `pid2` controller, if `C0` is a `pid2` controller.
- Returns a 2-DOF `pidstd2` controller, if `C0` is a `pidstd2` controller.
- Returns a controller with the same `Iformula` and `Dformula` values as `C0`, if `sys` is a discrete-time system. See the `pid`, `pid2`, `pidstd`, and `pidstd2` reference pages for more information about `Iformula` and `Dformula`.

### **wc**

Target value for the 0 dB gain crossover frequency of the tuned open-loop response. Specify `wc` in units of radians/`TimeUnit`, where `TimeUnit` is the time unit of `sys`. The crossover frequency `wc` roughly sets the control bandwidth. The closed-loop response time is approximately  $1/wc$ .

Increase `wc` to speed up the response. Decrease `wc` to improve stability. When you omit `wc`, `pidtune` automatically chooses a value, based on the plant dynamics, that achieves a balance between response and stability.

### **opts**

Option set specifying additional tuning options for the `pidtune` design algorithm, such as target phase margin or design focus. Use `pidtuneOptions` to create `opts`.

## **Output Arguments**

### **c**

Controller designed for `sys`. If `sys` is an array of linear models, `pidtune` designs a controller for each linear model and returns an array of PID controllers.

#### **Controller form:**

- If the second argument to `pidtune` is `type`, `C` is a `pid` or `pid2` controller.
- If the second argument to `pidtune` is `C0`:
  - `C` is a `pid` controller, if `C0` is a `pid` object.

- **C** is a `pidstd` controller, if **CO** is a `pidstd` object.
- **C** is a `pid2` controller, if **CO** is a `pid2` object.
- **C** is a `pidstd2` controller, if **CO** is a `pidstd2` object.

### Controller type:

- If the second argument to `pidtune` is `type`, **C** generally has the specified type.
- If the second argument to `pidtune` is `CO`, **C** generally has the same type as `CO`.

In either case, however, where the algorithm can achieve adequate performance and robustness using a lower-order controller than specified with `type` or `CO`, `pidtune` returns a **C** having fewer actions than specified. For example, **C** can be a PI controller even though `type` is `'PIDF'`.

### Time domain:

- **C** has the same time domain as `sys`.
- If `sys` is a discrete-time model, **C** has the same sample time as `sys`.
- If you specify `CO`, **C** has the same `Iformula` and `Dformula` as `CO`. If no `CO` is specified, both `Iformula` and `Dformula` are `Forward Euler`. See the `pid`, `pid2`, `pidstd`, and `pidstd2` reference pages for more information about `Iformula` and `Dformula`.

If you specify `CO`, **C** also obtains model properties such as `InputName` and `OutputName` from `CO`. For more information about model properties, see the reference pages for each type of dynamic system model.

### info

Data structure containing information about performance and robustness of the tuned PID loop. The fields of `info` are:

- `Stable` — Boolean value indicating closed-loop stability. `Stable` is 1 if the closed loop is stable, and 0 otherwise.
- `CrossoverFrequency` — First 0 dB crossover frequency of the open-loop system `C*sys`, in `rad/TimeUnit`, where `TimeUnit` is the time units specified in the `TimeUnit` property of `sys`.
- `PhaseMargin` — Phase margin of the tuned PID loop, in degrees.

If `sys` is an array of plant models, `info` is an array of data structures containing information about each tuned PID loop.



## Examples

### PID Controller Design at the Command Line

This example shows how to design a PID controller for the plant given by:

$$sys = \frac{1}{(s + 1)^3}.$$

As a first pass, create a model of the plant and design a simple PI controller for it.

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

```
C_pi =
```

$$K_p + K_i * \frac{1}{s}$$

```
with Kp = 1.14, Ki = 0.454
```

Continuous-time PI controller in parallel form.

```
info =
```

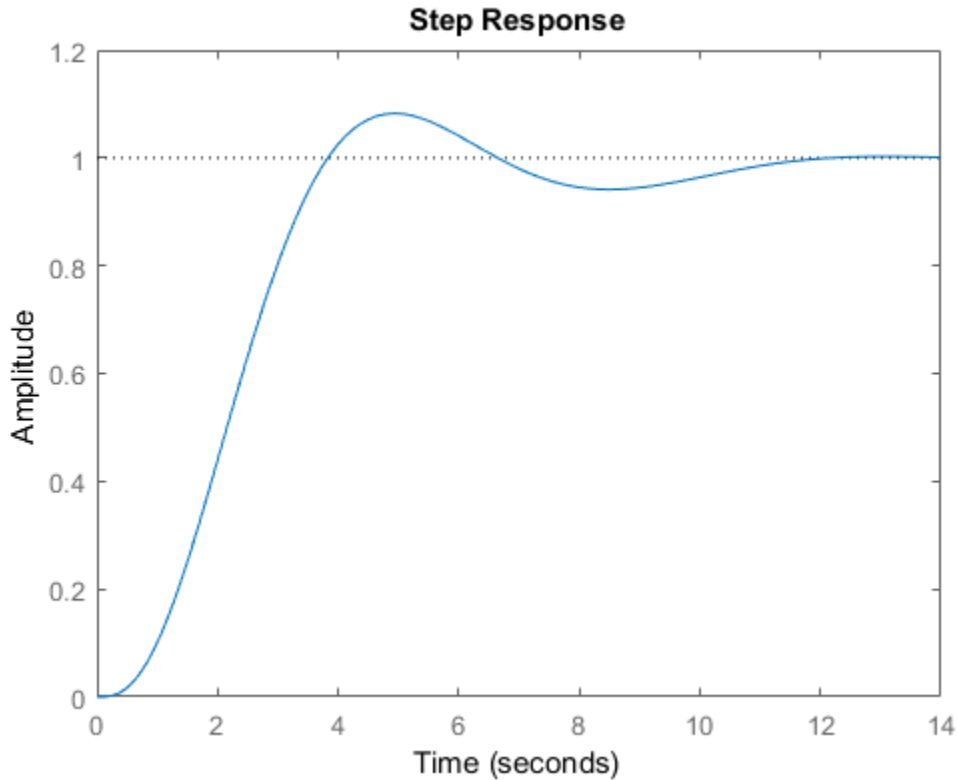
```
struct with fields:
```

```
Stable: 1
CrossoverFrequency: 0.5205
PhaseMargin: 60.0000
```

`C_pi` is a `pid` controller object that represents a PI controller. The fields of `info` show that the tuning algorithm chooses an open-loop crossover frequency of about 0.52 rad/s.

Examine the closed-loop step response (reference tracking) of the controlled system.

```
T_pi = feedback(C_pi*sys, 1);
step(T_pi)
```



To improve the response time, you can set a higher target crossover frequency than the result that `pidtune` automatically selects, 0.52. Increase the crossover frequency to 1.0.

```
[C_pi_fast,info] = pidtune(sys,'PI',1.0)
```

```
C_pi_fast =
```

$$K_p + K_i * \frac{1}{s}$$

with  $K_p = 2.83$ ,  $K_i = 0.0495$

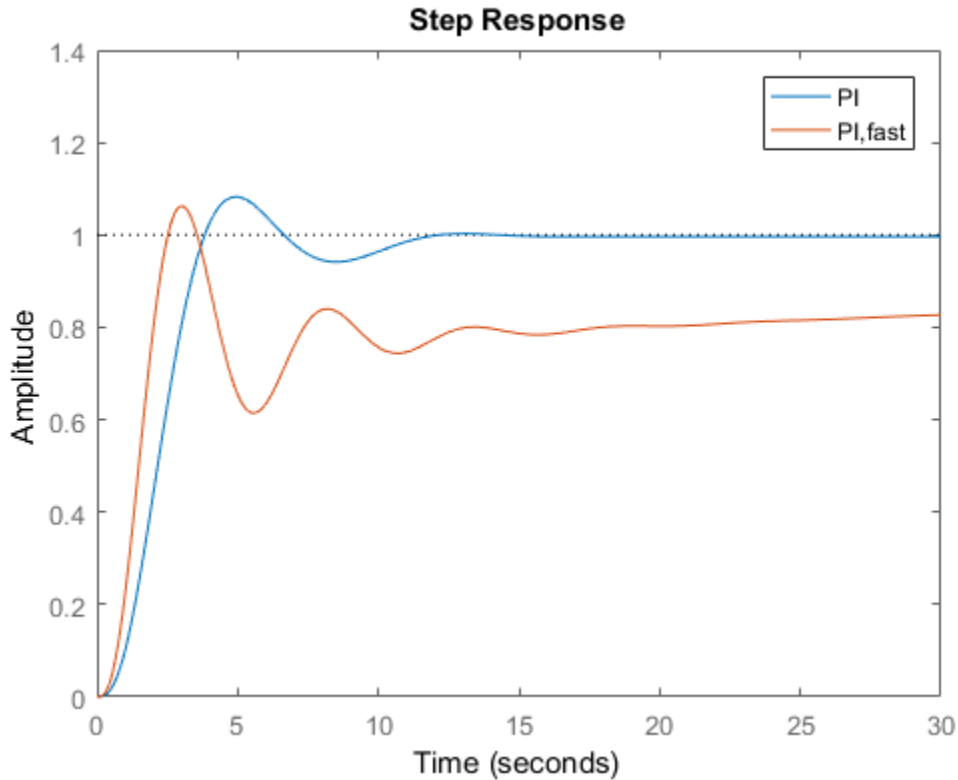
Continuous-time PI controller in parallel form.

```
info =  
  struct with fields:  
    Stable: 1  
    CrossoverFrequency: 1  
    PhaseMargin: 43.9973
```

The new controller achieves the higher crossover frequency, but at the cost of a reduced phase margin.

Compare the closed-loop step response with the two controllers.

```
T_pi_fast = feedback(C_pi_fast*sys,1);  
step(T_pi,T_pi_fast)  
axis([0 30 0 1.4])  
legend('PI', 'PI,fast')
```



This reduction in performance results because the PI controller does not have enough degrees of freedom to achieve a good phase margin at a crossover frequency of 1.0 rad/s. Adding a derivative action improves the response.

Design a PIDF controller for GC with the target crossover frequency of 1.0 rad/s.

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

```
C_pidf_fast =
```

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f*s+1}$$

with  $K_p = 2.72$ ,  $K_i = 0.985$ ,  $K_d = 1.72$ ,  $T_f = 0.00875$

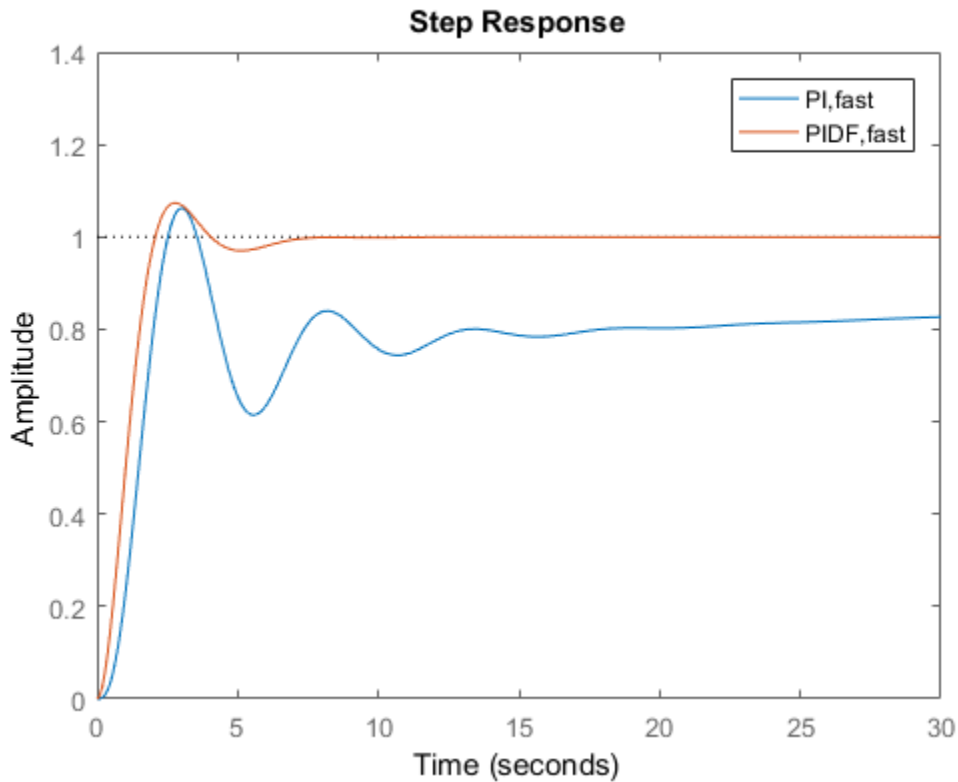
Continuous-time PIDF controller in parallel form.

```
info =  
  
    struct with fields:  
  
                Stable: 1  
    CrossoverFrequency: 1  
                PhaseMargin: 60.0000
```

The fields of `info` show that the derivative action in the controller allows the tuning algorithm to design a more aggressive controller that achieves the target crossover frequency with a good phase margin.

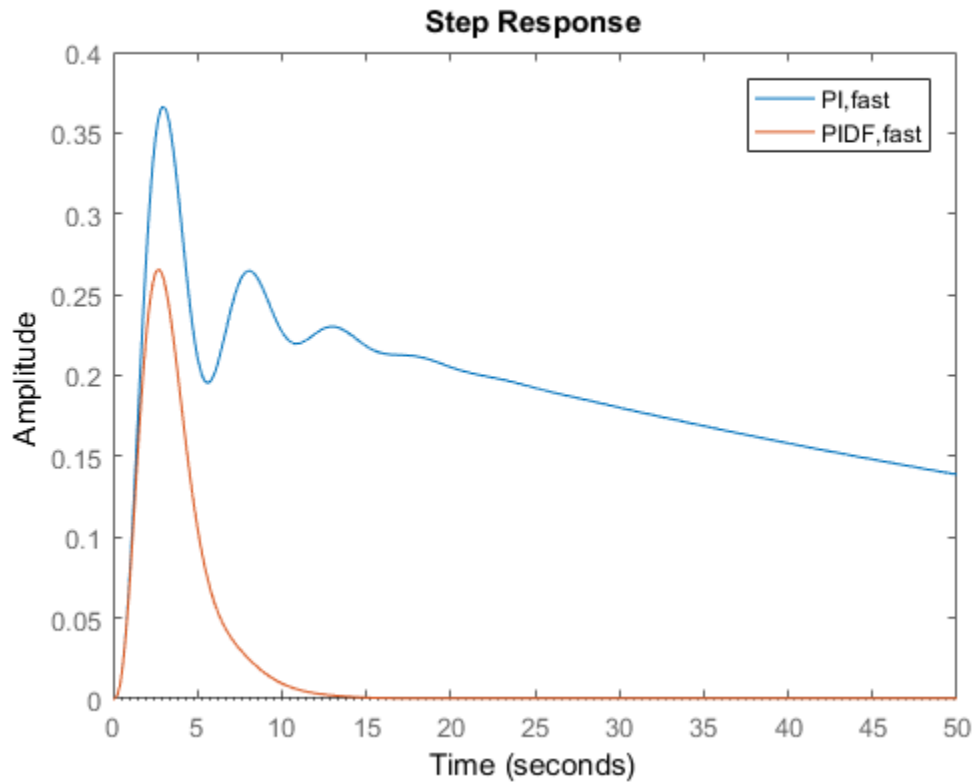
Compare the closed-loop step response and disturbance rejection for the fast PI and PIDF controllers.

```
T_pidf_fast = feedback(C_pidf_fast*sys,1);  
step(T_pi_fast, T_pidf_fast);  
axis([0 30 0 1.4]);  
legend('PI,fast', 'PIDF,fast');
```



You can compare the input (load) disturbance rejection of the controlled system with the fast PI and PIDF controllers. To do so, plot the response of the closed-loop transfer function from the plant input to the plant output.

```
S_pi_fast = feedback(sys,C_pi_fast);  
S_pidf_fast = feedback(sys,C_pidf_fast);  
step(S_pi_fast,S_pidf_fast);  
axis([0 50 0 0.4]);  
legend('PI,fast', 'PIDF,fast');
```



This plot shows that the PIDF controller also provides faster disturbance rejection.

## Design Standard-Form PID Controller

Design a PID controller in standard form for the plant defined by

$$sys = \frac{1}{(s+1)^3}.$$

To design a controller in standard form, use a standard-form controller as the `C0` argument to `pidtune`.

```
sys = zpk([],[-1 -1 -1],1);
C0 = pidstd(1,1,1);
C = pidtune(sys,C0)
```

C =

$$K_p * \left( 1 + \frac{1}{T_i} * \frac{1}{s} + T_d * s \right)$$

with Kp = 2.18, Ti = 2.36, Td = 0.591

Continuous-time PID controller in standard form

## Specify Integrator Discretization Method

Design a discrete-time PI controller using a specified method to discretize the integrator.

If your plant is in discrete time, `pidtune` automatically returns a discrete-time controller using the default Forward Euler integration method. To specify a different integration method, use `pid` or `pidstd` to create a discrete-time controller having the desired integration method.

```
sys = c2d(tf([1 1],[1 5 6]),0.1);
C0 = pid(1,1,'Ts',0.1,'IFormula','BackwardEuler');
C = pidtune(sys,C0)
```

C =

$$K_p + K_i * \frac{T_s * z}{z-1}$$

with Kp = -0.518, Ki = 10.4, Ts = 0.1

Sample time: 0.1 seconds

Discrete-time PI controller in parallel form.

Using `C0` as an input causes `pidtune` to design a controller `C` of the same form, type, and discretization method as `C0`. The display shows that the integral term of `C` uses the Backward Euler integration method.

Specify a Trapezoidal integrator and compare the resulting controller.



```
C0_tr = pid(1,1,'Ts',0.1,'IFormula','Trapezoidal');
Ctr = pidtune(sys,C_tr)
```

```
Ctr =
```

$$K_i * \frac{T_s(z+1)}{2*(z-1)}$$

```
with Ki = 10.4, Ts = 0.1
```

```
Sample time: 0.1 seconds
Discrete-time I-only controller.
```

## Design 2-DOF PID Controller

Design a 2-DOF PID Controller for the plant given by the transfer function:

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

Use a target bandwidth of 1.5 rad/s.

```
wc = 1.5;
G = tf(1,[1 0.5 0.1]);
C2 = pidtune(G,'PID2',wc)
```

```
C2 =
```

$$u = K_p (b*r-y) + K_i \frac{1}{s} (r-y) + K_d*s (c*r-y)$$

```
with Kp = 1.26, Ki = 0.255, Kd = 1.38, b = 0.665, c = 0
```

```
Continuous-time 2-DOF PID controller in parallel form.
```

Using the type 'PID2' causes `pidtune` to generate a 2-DOF controller, represented as a `pid2` object. The display confirms this result. The display also shows that `pidtune` tunes

all controller coefficients, including the setpoint weights **b** and **c**, to balance performance and robustness.

## Alternatives

For interactive PID tuning, use PID Tuner. See “PID Controller Design for Fast Reference Tracking” for an example of designing a controller using PID Tuner.

PID Tuner cannot design controllers for multiple plants at once.

## More About

### Tips

By default, `pidtune` with the `type` input returns a `pid` controller in parallel form. To design a controller in standard form, use a `pidstd` controller as input argument `C0`. For more information about parallel and standard controller forms, see the `pid` and `pidstd` reference pages.

### Algorithms

For information about the MathWorks® PID tuning algorithm, see “PID Tuning Algorithm”.

- “PID Controller Types for Tuning”
- “PID Tuning Algorithm”

## References

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

### See Also

`pid` | `pid2` | `pidstd` | `pidstd2` | `pidtuneOptions` | `pidTuner`

Introduced in R2010b

# pidtuneOptions

Define options for pidtune command

## Syntax

```
opt = pidtuneOptions  
opt = pidtuneOptions(Name,Value)
```

## Description

`opt = pidtuneOptions` returns the default option set for the `pidtune` command.

`opt = pidtuneOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

#### 'PhaseMargin'

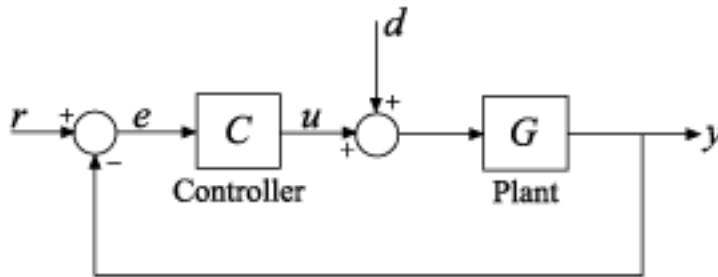
Target phase margin in degrees. `pidtune` attempts to design a controller such that the phase margin is at least the value specified for `PhaseMargin`. The selected crossover frequency could restrict the achievable phase margin. Typically, higher phase margin improves stability and overshoot, but limits bandwidth and response speed.

**Default:** 60

#### 'DesignFocus'

Closed-loop performance objective to favor in the design. For a given target phase margin, `pidtune` chooses a controller design that balances the two measures of

performance, reference tracking and disturbance rejection. When you change the `DesignFocus` option, the tuning algorithm attempts to adjust the PID gains to favor either reference tracking or disturbance rejection while achieving the same target phase margin. In the control architecture assumed by `pidtune`, shown in the following diagram, reference tracking is the response at  $y$  to signals at  $r$ , and disturbance rejection is the suppression at  $y$  of signals at  $d$ .



The `DesignFocus` option can take the following values:

- 'balanced' (default) — For a given robustness, tune the controller to balance reference tracking and disturbance rejection.
- 'reference-tracking' — Tune the controller to favor reference tracking, if possible.
- 'disturbance-rejection' — Tune the controller to favor disturbance rejection, if possible.

The more tunable parameters there are in the system, the more likely it is that the PID algorithm can achieve the desired design focus without sacrificing robustness. For example, setting the design focus is more likely to be effective for PID controllers than for P or PI controllers. In all cases, how much you can fine-tune the performance of the system depends strongly on the properties of your plant.

For an example illustrating the effect of this option, see “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (Command Line)”.

**Default:** 'balanced'

### 'NumUnstablePoles'

Number of unstable poles in the plant. When your plant is a `frd` model or a state-space model with internal delays, you must specify the number of open-loop unstable poles (if any). Incorrect values might result in PID controllers that fail to stabilize the real plant. (`pidtune` ignores this option for other model types.)

Unstable poles are poles located at:

- $\text{Re}(s) > 0$ , for continuous-time plants
- $|z| > 1$ , for discrete-time plants

A pure integrator in the plant ( $s = 0$ ) or ( $|z| > 1$ ) does not count as an unstable pole for `NumUnstablePoles`. If your plant is a `frd` model of a plant with a pure integrator, for best results, ensure that your frequency response data covers a low enough frequency to capture the integrator slope.

**Default:** 0

## Output Arguments

### `opt`

Object containing the specified options for `pidtune`.

## Examples

Tune a PIDF controller with a target phase margin of 45 degrees, favoring the disturbance-rejection measure of performance.

```
sys = tf(1,[1 3 3 1]);  
opts = pidtuneOptions('PhaseMargin',45,'DesignFocus','disturbance-rejection');  
[C,info] = pidtune(sys,'pid',opts);
```

## More About

### Tips

- When using the `pidtune` command to design a PID controller for a plant with unstable poles, if your plant model is one of the following:

- A `frd` model
- A `ss` model with internal delays that cannot be converted to I/O delays

then use `pidtuneOptions` to specify the number of unstable poles in the plant.

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

### **See Also**

`pidtune`

**Introduced in R2010b**

# PID Tuner

Tune PID controllers

## Description

The **PID Tuner** app automatically tunes the gains of a PID controller for a SISO plant to achieve a balance between performance and robustness. You can specify the controller type, such as PI, PID with derivative filter, or two-degree-of-freedom (2-DOF) PID controllers. Analysis plots let you examine controller performance in time and frequency domains. You can interactively refine the performance of the controller to adjust loop bandwidth and phase margin, or to favor setpoint tracking or disturbance rejection.

You can use **PID Tuner** with a plant represented by a numeric LTI model such as a transfer function (**tf**) or state-space (**ss**) model. If you have Simulink Control Design software, you can use **PID Tuner** to tune a PID Controller or PID Controller (2DOF) block in a Simulink model. If you have System Identification Toolbox software, you can use the app to estimate a plant from measured or simulated data and design a controller for the estimated plant.

## Open the PID Tuner App

- MATLAB Toolstrip: On the **Apps** tab, under **Control System Design and Analysis**, click the app icon.
- MATLAB command prompt: Enter `pidTuner`.
- Simulink model: In the PID Controller or PID Controller (2DOF) block dialog box, click **Tune**.

## Examples

- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)”
- “PID Controller Tuning in Simulink”

# Parameters

## Plant — Current plant

LTI model in Data Browser | Import | ...

The **Plant** menu displays the name of the current plant that **PID Tuner** is using for controller design.

Change the current plant using the following menu options:

- A list of the LTI models present in the **PID Tuner** Data Browser.
- **Import** — Import a new LTI model from the MATLAB workspace.
- **Re-Linearize Closed Loop** — Linearize the plant at a different snapshot time. See “Tune at a Different Operating Point”. This option is available only when tuning a PID Controller or PID Controller (2DOF) block in a Simulink model.
- **Identify New Plant** — Use system identification to obtain a plant from measured or simulated system response data (requires System Identification Toolbox software). See:
  - “Interactively Estimate Plant Parameters from Response Data”, when tuning a PID controller for an LTI model.
  - “Interactively Estimate Plant from Measured or Simulated Response Data”, when tuning a PID Controller block in a Simulink model.

If you are tuning a PID controller for a plant represented by an LTI model, the default plant is:

- **Plant** = 1, if you opened **PID Tuner** from the **Apps** tab in the MATLAB Toolstrip, or if you used the `pidTuner` command without an input argument.
- The plant you specified as an input argument to `pidTuner`.

If you are tuning a PID Controller or PID Controller (2DOF) block in a Simulink model, then the default plant is linearized at the operating point specified by the model initial conditions. See “What Plant Does PID Tuner See?”

## Type — Controller type

'PI' | 'PIDF' | 'PID2' | ...

The controller type specifies which terms are present in the PID controller. For instance, a PI controller has a proportional and an integral term. A PDF controller has a proportional term and a filtered derivative term.



- If you are tuning a controller for a plant represented by an LTI model, use the **Type** menu to specify controller type. When you change controller type, **PID Tuner** automatically designs a new controller. Available controller types include 2-DOF PID controllers for more flexibility in the trade-off between disturbance rejection and reference tracking. For details on available controller types, see “PID Controller Types for Tuning”.
- If you are tuning a PID Controller or PID Controller (2DOF) block in a Simulink model, the **Type** field displays the controller type specified in the block dialog box.

### Form — Controller form

'Parallel' | 'Standard'

This field displays the controller form.

- If you are tuning a controller for a plant represented by an LTI model, use the **Form** menu to specify controller form. For information about parallel and standard forms, see the `pid` and `pidstd` reference pages.
- If you are tuning a PID Controller or PID Controller (2DOF) block in a Simulink model, the **Form** field displays the controller form specified in the block dialog box.

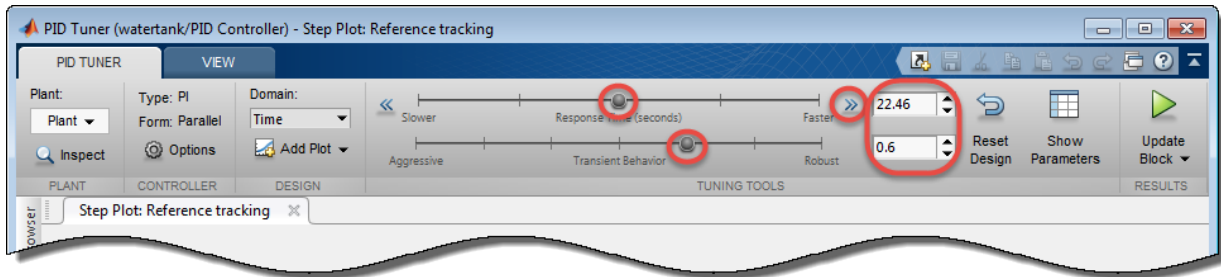
### Domain — Domain for specifying performance targets

'Time' | 'Frequency'

The **Domain** menu enables you to switch the domain in which PID Tuner displays the target performance parameters.

- **Time** — Sliders set the **Response Time** and **Transient Behavior**.
- **Frequency** — Sliders set the **Bandwidth** and **Phase Margin**.

To refine the controller design, you adjust the target performance parameters using the sliders or the corresponding numeric values.



For more information, see:

- “Refine the Design” (tuning a controller for an LTI model)
- “Refine the Design” (tuning PID Controller or PID Controller (2DOF) block in Simulink model)

### **Add Plot — Create analysis plots**

Reference Tracking | Input Disturbance Rejection | Controller Effort  
| ...

Create time-domain and frequency-domain analysis plots to help analyze the performance of the PID controller. For detailed information about the available response plots, see:

- “Analyze Design in PID Tuner” (tuning a controller for an LTI model)
- “Analyze Design in PID Tuner” (tuning PID Controller or PID Controller (2DOF) block in Simulink model)

## **Programmatic Use**

pidTuner

## **More About**

- “Designing PID Controllers with PID Tuner”
- “Introduction to Automatic PID Tuning in Simulink”

## **See Also**

### **Functions**

pidtune

**Introduced in R2010b**

# pidTuner

Open PID Tuner for PID tuning

## Syntax

```
pidTuner(sys,type)
pidTuner(sys,Cbase)
pidTuner(sys)
pidTuner
```

## Description

`pidTuner(sys,type)` launches the PID Tuner app and designs a controller of type `type` for plant `sys`.

`pidTuner(sys,Cbase)` launches PID Tuner with a baseline controller `Cbase` so that you can compare performance between the designed controller and the baseline controller. If `Cbase` is a `pid`, `pidstd`, `pid2` or `pidstd2` controller object, PID Tuner designs a controller of the same form, type, and discrete integrator formulas as `Cbase`.

`pidTuner(sys)` designs a parallel-form PI controller.

`pidTuner` launches PID Tuner with default plant of 1 and proportional (P) controller of 1.

## Input Arguments

### **sys**

Plant model for controller design. `sys` can be:


- Any SISO LTI system (such as `ss`, `tf`, `zpk`, or `frd`).
- Any System Identification Toolbox SISO linear model (`idtf`, `idfrd`, `idgrey`, `idpoly`, `idproc`, or `idss`).
- A continuous- or discrete-time model.

- Stable, unstable, or integrating. However, you might not be able to stabilize a plant with unstable poles under PID control.
- A model that includes any type of time delay. A plant with long time delays, however, might not achieve adequate performance under PID control.

If the plant has unstable poles, and `sys` is either:

- A `frd` model
- A `ss` model with internal time delays that cannot be converted to I/O delays

then you must specify the number of unstable poles in the plant. To do this, after opening

PID Tuner, in the **Plant** menu, select  **Import**. In the Import Linear System dialog box, reimport `sys`, specifying the number of unstable poles where prompted.

### **type**

Controller type of the controller to design, specified as a character vector. The term *controller type* refers to which terms are present in the controller action. For example, a PI controller has only a proportional and an integral term, while a PIDF controller contains proportional, integrator, and filtered derivative terms. `type` can take the values summarized below. For more detailed information about these controller types, see “PID Controller Types for Tuning”

## **1-DOF Controllers**

- 'P' — Proportional only
- 'I' — Integral only
- 'PI' — Proportional and integral
- 'PD' — Proportional and derivative
- 'PDF' — Proportional and derivative with first-order filter on derivative term
- 'PID' — Proportional, integral, and derivative
- 'PIDF' — Proportional, integral, and derivative with first-order filter on derivative term

## **2-DOF Controllers**

- 'PI2' — 2-DOF proportional and integral

- 'PD2' — 2-DOF proportional and derivative
- 'PDF2' — 2-DOF proportional and derivative with first-order filter on derivative term
- 'PID2' — 2-DOF proportional, integral, and derivative
- 'PIDF2' — 2-DOF proportional, integral, and derivative with first-order filter on derivative term

For more information about 2-DOF PID controllers generally, see “Two-Degree-of-Freedom PID Controllers”.

## 2-DOF Controllers with Fixed Setpoint Weights

- 'I-PD' — 2-DOF PID with  $b = 0$ ,  $c = 0$
- 'I-PDF' — 2-DOF PIDF with  $b = 0$ ,  $c = 0$
- 'ID-P' — 2-DOF PID with  $b = 0$ ,  $c = 1$
- 'IDF-P' — 2-DOF PIDF with  $b = 0$ ,  $c = 1$
- 'PI-D' — 2-DOF PID with  $b = 1$ ,  $c = 0$
- 'PI-DF' — 2-DOF PIDF with  $b = 1$ ,  $c = 0$

For more detailed information about fixed-setpoint-weight 2-DOF PID controllers, see “PID Controller Types for Tuning”.

## Controller Form

When you use the `type` input, PID Tuner designs a controller in parallel form. If you want to design a controller in standard form, Use the input `Cbase` instead of `type`, or select `Standard` from the **Form** menu. For more information about parallel and standard forms, see the `pid` and `pidstd` reference pages.

If `sys` is a discrete-time model with sample time `Ts`, PID Tuner designs a discrete-time `pid` controller using the `ForwardEuler` discrete integrator formula. To design a controller having a different discrete integrator formula:

- Use the input argument `Cbase` instead of `type`. PID Tuner reads controller type, form, and discrete integrator formulas from the baseline controller `Cbase`.

- In PID Tuner, click **Options** to open the Controller Options dialog box. Select discrete integrator formulas from the **Integral Formula** and **Derivative Formula** menus.

For more information about discrete integrator formulas, see the `pid` and `pidstd` reference pages.

### **Cbase**

A dynamic system representing a baseline controller, permitting comparison of the performance of the designed controller to the performance of **Cbase**.

If **Cbase** is a `pid` or `pidstd` object, PID Tuner also uses it to configure the type, form, and discrete integrator formulas of the designed controller. The designed controller:

- Is the type represented by **Cbase**.
- Is a parallel-form controller, if **Cbase** is a `pid` controller object.
- Is a standard-form controller, if **Cbase** is a `pidstd` controller object.
- Is a parallel-form 2-DOF controller, if **Cbase** is a `pid2` controller object.
- Is a standard-form 2-DOF controller, if **Cbase** is a `pidstd2` controller object.
- Has the same **Iformula** and **Dformula** values as **Cbase**. For more information about **Iformula** and **Dformula**, see the `pid` and `pidstd` reference pages .

If **Cbase** is any other dynamic system, PID Tuner designs a parallel-form PI controller. You can change the controller form and type using the **Form** and **Type** menus after launching PID Tuner.

## **Examples**

### **Interactive PID Tuning of Parallel-Form Controller**

Launch PID Tuner to design a parallel-form PIDF controller for a discrete-time plant:

```
Gc = zpk([], [-1 -1 -1], 1);  
Gd = c2d(Gc, 0.1);           % Create discrete-time plant  
  
pidTuner(Gd, 'pidf')        % Launch PID Tuner
```

### **Interactive PID Tuning of Standard-Form Controller Using Integrator Discretization Method**

Design a standard-form PIDF controller using `BackwardEuler` discrete integrator formula:

```
Gc = zpk([],[-1 -1 -1],1);
Gd = c2d(Gc,0.1);           % Create discrete-time plant

% Create baseline controller.
Cbase = pidstd(1,2,3,4,'Ts',0.1,...
    'IFormula','BackwardEuler','DFormula','BackwardEuler')

pidTuner(Gd,Cbase)         % Launch PID Tuner
```

PID Tuner designs a controller for `Gd` having the same form, type, and discrete integrator formulas as `Cbase`. For comparison, you can display the response plots of `Cbase` with the response plots of the designed controller by clicking the **Show baseline** checkbox in PID Tuner.

## Alternatives

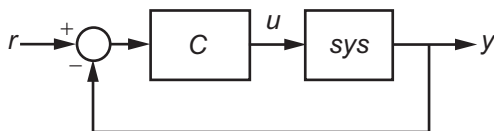
You can open PID Tuner from the MATLAB desktop, in the **Apps** tab. When you do so, use the **Plant** menu in PID Tuner to specify your plant model.

For PID tuning at the command line, use `pidtune`. The `pidtune` command can design a controller for multiple plants at once.

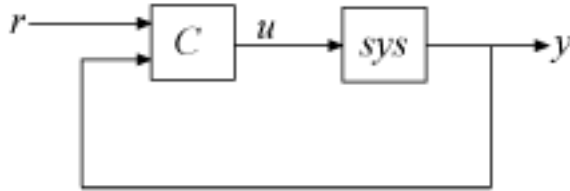
## More About

### Tips

- If `type` or `Cbase` specifies a one-degree-of-freedom (1-DOF) PID controller, then `pidTuner` designs a controller for the unit feedback loop as illustrated:



- If `type` or `Cbase` specifies a two-degree-of-freedom (2-DOF) PID controller, then `pidTuner` designs a 2-DOF controller as in the feedback loop of this illustration:



- PID Tuner has a default target phase margin of 60 degrees and automatically tunes the PID gains to balance performance (response time) and robustness (stability margins). Use the **Response time** or **Bandwidth** and **Phase Margin** sliders to tune the controller's performance to your requirements. Increasing performance typically decreases robustness, and vice versa.
- Select response plots from the **Response** menu to analyze the controller's performance.
- If you provide **Cbase**, check **Show baseline** to display the response of the baseline controller.
- For more detailed information about using PID Tuner, see “Designing PID Controllers with PID Tuner”.

### Algorithms

For information about the MathWorks PID tuning algorithm, see “PID Tuning Algorithm”.

- “Designing PID Controllers with PID Tuner”
- “PID Controller Types for Tuning”

### References

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

### See Also

pid | pidstd | pid2 | pidstd2 | pidtune



**Introduced in R2014b**

## place

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)** places the desired closed-loop poles **p** by computing a state-feedback gain matrix **K**. All the inputs of the plant are assumed to be 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)** 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).'
```

## Examples

### Pole Placement Design

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)
```

## More About

### Algorithms

`place` uses the algorithm of [1] which, for multi-input systems, optimizes the choice of eigenvectors for a robust solution.

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.

## References

- [1] Kautsky, J., N.K. Nichols, and P. Van Dooren, "Robust Pole Assignment in Linear State Feedback," *International Journal of 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.

### See Also

`lqr` | `rlocus`

Introduced before R2006a

## pole

Compute poles of dynamic system

### Syntax

```
pole(sys)
```

### Description

`pole(sys)` computes the poles  $p$  of the SISO or MIMO dynamic system model `sys`.

If `sys` has internal delays, poles are obtained by first setting all internal delays to zero (creating a zero-order Padé approximation) so that the system has a finite number of zeros. For some systems, setting delays to 0 creates singular algebraic loops, which result in either improper or ill-defined, zero-delay approximations. For these systems, `pole` returns an error. This error does not imply a problem with the model `sys` itself.

### Limitations

Multiple poles are numerically sensitive and cannot be computed to high accuracy. A pole  $\lambda$  with multiplicity  $m$  typically gives rise to a cluster of computed poles distributed on a circle with center  $\lambda$  and radius of order

$$\rho \approx \varepsilon^{1/m}$$

where  $\varepsilon$  is the relative machine precision (`eps`).

### More About

#### Algorithms

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.

### **See Also**

`pzmap` | `zero` | `damp` | `esort` | `dsort`

**Introduced before R2006a**

## polyBasis

Polynomial basis functions for tunable gain surface

You use basis function expansions to parameterize gain surfaces for tuning gain-scheduled controllers. `polyBasis` generates standard polynomial expansions in any number of scheduling variables. Use the resulting functions to create tunable gain surfaces with `tunableSurface`.

### Syntax

```
shapefcn = polyBasis('canonical',degree)
shapefcn = polyBasis('chebyshev',degree)
shapefcn = polyBasis( ____,nvars)
```

### Description

`shapefcn = polyBasis('canonical',degree)` generates a function that evaluates the powers of an input variable,  $x$ , up to `degree`:

$$\text{shapefcn}(x) = [x, x^2, \dots, x^{\text{order}}].$$

`shapefcn = polyBasis('chebyshev',degree)` generates a function that evaluates Chebyshev polynomials up to `degree`:

$$\text{shapefcn}(x) = [T_1(x), \dots, T_{\text{order}}(x)].$$

The Chebyshev polynomials are defined recursively by:

$$T_0(x) = 1; \quad T_1(x) = x; \quad T_{i+1}(x) = 2xT_i(x) - T_{i-1}(x).$$

`shapefcn = polyBasis( ____,nvars)` generates an `nvars`-dimensional polynomial expansion by taking the outer product of `nvars` 1-D polynomial expansions. The

resulting function `shapefcn` takes `nvars` input arguments and returns a vector with  $(\text{degree}+1)^{(\text{nvars}-1)}$  entries. For example, for `nvars = 3` and canonical polynomials,

$$\text{shapefcn}(x,y,z) = [x^i y^j z^k : 0 \leq i,j,k \leq \text{order}, i+j+k > 0].$$

Thus, to specify a bilinear function in two scheduling variables, use:

```
shapefcn = polyBasis('canonical',1,2);
```

Using the resulting function with `tunableSurface` defines a variable gain of the form:

$$K(x,y) = K_0 + K_1 x + K_2 y + K_3 xy.$$

Here,  $x$  and  $y$  are the normalized scheduling variables, whose values lie in the range  $[-1,1]$ . (See `tunableSurface` for more information.)

To specify basis functions in multiple scheduling variables where the expansions are different for each variable, use `ndBasis`.

## Examples

### Polynomial Basis Functions of One Scheduling Variable

Create basis functions for a gain that varies as a cubic function of one scheduling variable.

```
shapefcn = polyBasis('canonical',3);
```

`shapefcn` is a handle to a function of one variable that returns an array of values corresponding to the first three powers of its input. In other words,  $\text{shapefcn}(x) = [x \ x^2 \ x^3]$ . For example, examine `shapefcn(-0.2)`.

```
x = -0.2;
shapefcn(x)
```

```
ans =
```

```
-0.2000    0.0400   -0.0080
```

Evaluating `[x x^2 x^3]` for `x = -0.2` returns the same result.

```
[x x^2 x^3]
```

```
ans =
```

```
-0.2000    0.0400   -0.0080
```

Use `shapefcn` as an input argument to `tunableSurface` to define a polynomial gain surface. This `shapefcn` is equivalent to using:

```
shapefcn = @(x) [x x^2 x^3];
```

### Chebyshev Basis Functions

Create a set of basis functions that are Chebyshev polynomials of a single variable, up to third degree.

```
shapefcn = polyBasis('chebyshev',3);
```

### Bilinear Function of Two Variables

Create basis functions for a bilinear gain surface,  $[x, y, xy]$ .

```
shapefcn = polyBasis('canonical',1,2);
```

Confirm the values returned by `shapefcn` for a particular  $(x, y)$  pair.

```
x = 0.2;  
y = -0.5;  
shapefcn(x,y)
```

```
ans =
```

```
0.2000   -0.5000   -0.1000
```

This `shapefcn` is equivalent to:

```
shapefcn = @(x,y)[x,y,x*y];
```



The basis functions of `shapefcn` are first-order in each of the two variables. To create a set of basis functions in different degrees for each variable, use `ndBasis`.

## Input Arguments

### **degree** — Degree of expansion

positive integer

Degree of the polynomial expansion, specified as a positive integer.

Example:

### **nvars** — Number of variables

1 (default) | positive integer

Number of scheduling variables, specified as a positive integer.

Example:

## Output Arguments

### **shapefcn** — Polynomial expansion

function handle

Polynomial expansion, specified as a function handle. `shapefcn` takes as input arguments the number of variables specified by `nvars`. The function evaluates polynomials in those variables up to the specified degree, and returns the resulting values in a vector. When you use `shapefcn` to create a gain surface, `tunableSurface` automatically generates tunable coefficients for each polynomial term in the vector.

## See Also

`fourierBasis` | `ndBasis` | `tunableSurface`

**Introduced in R2015b**

## predict

Predict state and state estimation error covariance at next time step using extended or unscented Kalman filter

The `predict` command predicts the state and state estimation error covariance of an `extendedKalmanFilter` or `unscentedKalmanFilter` object at the next time step. To implement the extended or unscented Kalman filter algorithms, use the `predict` and `correct` commands together. If the current output measurement exists, you can use `predict` and `correct`. If the measurement is missing, you can only use `predict`. For information about the order in which to use the commands, see “Using predict and correct Commands” on page 2-821.

## Syntax

```
[PredictedState,PredictedStateCovariance] = predict(obj)
[PredictedState,PredictedStateCovariance] = predict(obj,Us1,...,Usn)
```

## Description

`[PredictedState,PredictedStateCovariance] = predict(obj)` predicts state estimate and state estimation error covariance of an extended or unscented Kalman filter object `obj` at the next time step.

You create `obj` using the `extendedKalmanFilter` or `unscentedKalmanFilter` commands. You specify the state transition function and measurement function of your nonlinear system in `obj`. You also specify whether the process and measurement noise terms are additive or nonadditive in these functions. The `State` property of the object stores the latest estimated state value. Assume that at time step  $k$ , `obj.State` is  $\hat{x}[k|k]$ . This value is the state estimate for time  $k$ , estimated using measured outputs until time  $k$ . When you use the `predict` command, the software returns  $\hat{x}[k+1|k]$  in the `PredictedState` output. Where  $\hat{x}[k+1|k]$  is the state estimate for time  $k+1$ , estimated using measured output until time  $k$ . The command returns the state estimation error covariance of  $\hat{x}[k+1|k]$  in the `PredictedStateCovariance` output. The software also updates the `State` and `StateCovariance` properties of `obj` with these corrected values.

Use this syntax if the state transition function  $f$  that you specified in `obj.StateTransitionFcn` has one of the following forms:

- $x(k) = f(x(k-1))$  — for additive process noise.
- $x(k) = f(x(k-1), w(k-1))$  — for nonadditive process noise.

Where  $x$  and  $w$  are the state and process noise of the system. The only inputs to  $f$  are the states and process noise.

`[PredictedState, PredictedStateCovariance] = predict(obj, Us1, ... Usn)` specifies additional input arguments, if the state transition function of the system requires these inputs. You can specify multiple arguments.

Use this syntax if your state transition function  $f$  has one of the following forms:

- $x(k) = f(x(k-1), Us1, \dots Usn)$  — for additive process noise.
- $x(k) = f(x(k-1), w(k-1), Us1, \dots Usn)$  — for nonadditive process noise.

## Examples

### Estimate States Online Using Unscented Kalman Filter

Estimate the states of a van der Pol oscillator using an unscented Kalman filter algorithm and measured output data. The oscillator has two states and one output.

Create an unscented Kalman filter object for the oscillator. Use previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to a van der Pol oscillator with nonlinearity parameter,  $\mu$ , equal to 1. The functions assume additive process and measurement noise in the system. Specify the initial state values for the two states as `[1;0]`. This is the guess for the state value at initial time  $k$ , using knowledge of system outputs until time  $k-1$ ,  $\hat{x}[k|k-1]$ .

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[1;0]);
```

Load the measured output data,  $y$ , from the oscillator. In this example, use simulated static data for illustration. The data is stored in the `vdp_data.mat` file.

```
load vdp_data.mat y
```

Specify the process noise and measurement noise covariances of the oscillator.

```
obj.ProcessNoise = 0.01;  
obj.MeasurementNoise = 0.16;
```

Implement the unscented Kalman filter algorithm to estimate the states of the oscillator by using the `correct` and `predict` commands. You first correct  $\hat{x}[k|k-1]$  using measurements at time  $k$  to get  $\hat{x}[k|k]$ . Then, you predict the state value at next time step,  $\hat{x}[k+1|k]$ , using  $\hat{x}[k|k]$ , the state estimate at time step  $k$  that is estimated using measurements until time  $k$ .

To simulate real-time data measurements, use the measured data one time step at a time.

```
for k = 1:size(y)  
    [CorrectedState,CorrectedStateCovariance] = correct(obj,y(k));  
    [PredictedState,PredictedStateCovariance] = predict(obj);  
end
```

When you use the `correct` command, `obj.State` and `obj.StateCovariance` are updated with the corrected state and state estimation error covariance values for time step  $k$ , `CorrectedState` and `CorrectedStateCovariance`. When you use the `predict` command, `obj.State` and `obj.StateCovariance` are updated with the predicted values for time step  $k+1$ , `PredictedState` and `PredictedStateCovariance`.

In this example, you used `correct` before `predict` because the initial state value was  $\hat{x}[k|k-1]$ , a guess for the state value at initial time  $k$  using system outputs until time  $k-1$ . If your initial state value is  $\hat{x}[k-1|k-1]$ , the value at previous time  $k-1$  using measurement until  $k-1$ , then use the `predict` command first. For more information about the order of using `predict` and `correct`, see “Using `predict` and `correct` Commands”.

### Specify State Transition and Measurement Functions with Additional Inputs

Consider a nonlinear system with input  $u$  whose state  $x$  and measurement  $y$  evolve according to the following state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise  $w$  of the system is additive while the measurement noise  $v$  is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input  $u$ .

```
f = @(x,u) (sqrt(x+u));
h = @(x,v,u) (x+2*u+v^2);
```

$f$  and  $h$  are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive,  $v$  is also specified as an input. Note that  $v$  is specified as an input before the additional input  $u$ .

Create an extended Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = extendedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of  $u$  to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement  $y[k]=0.8$  and input  $u[k]=0.2$  at time step  $k$ .

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given  $u[k]=0.2$ .

```
predict(obj,0.2)
```

- “Nonlinear State Estimation Using Unscented Kalman Filter”
- “Generate Code for Online State Estimation in MATLAB”

## Input Arguments

**obj** — Extended or unscented Kalman filter object

extendedKalmanFilter object | unscentedKalmanFilter object

Extended or unscented Kalman filter object for online state estimation, created using one of the following commands:

- `extendedKalmanFilter` — Uses the extended Kalman filter algorithm.
- `unscentedKalmanFilter` — Uses the unscented Kalman filter algorithm.

### **Us1, . . . Usn — Additional input arguments to state transition function**

input arguments of any type

Additional input arguments to state transition function, specified as input arguments of any type. The state transition function,  $f$ , is specified in the `StateTransitionFcn` property of the object. If the function requires input arguments in addition to the state and process noise values, you specify these inputs in the `predict` command syntax.

For example, suppose that your state transition function calculates the predicted state  $x$  at time step  $k$  using system inputs  $u(k-1)$  and time  $k-1$ , in addition to the state  $x(k-1)$ :

$$x(k) = f(x(k-1), u(k-1), k-1)$$

Then when you perform online state estimation at time step  $k$ , specify these additional inputs in the `predict` command syntax:

```
[PredictedState, PredictedStateCovariance] = predict(obj, u(k-1), k-1);
```

## Output Arguments

### **PredictedState — Predicted state estimate**

vector

Predicted state estimate, returned as a vector of size  $M$ , where  $M$  is the number of states of the system. If you specify the initial states of `obj` as a column vector then  $M$  is returned as a column vector, otherwise  $M$  is returned as a row vector.

For information about how to specify the initial states of the object, see the `extendedKalmanFilter` and `unscentedKalmanFilter` reference pages.

### **PredictedStateCovariance — Predicted state estimation error covariance**

matrix

Predicted state estimation error covariance, returned as an  $M$ -by- $M$  matrix, where  $M$  is the number of states of the system.

## More About

### Using `predict` and `correct` Commands

After you have created an extended or unscented Kalman filter object, `obj`, to implement the extended or unscented Kalman filter algorithms, use the `correct` and `predict` commands together.

At time step  $k$ , `correct` command returns the corrected value of states and state estimation error covariance using measured system outputs  $y[k]$  at the same time step. If your measurement function has additional input arguments  $U_m$ , you specify these as inputs to the `correct` command. The command passes these values to the measurement function.

```
[CorrectedState,CorrectedCovariance] = correct(obj,y,Um)
```

The `correct` command updates the `State` and `StateCovariance` properties of the object with the estimated values, `CorrectedState` and `CorrectedCovariance`.

The `predict` command returns the prediction of state and state estimation error covariance at the next time step. If your state transition function has additional input arguments  $U_s$ , you specify these as inputs to the `predict` command. The command passes these values to the state transition function.

```
[PredictedState,PredictedCovariance] = predict(obj,U_s)
```

The `predict` command updates the `State` and `StateCovariance` properties of the object with the predicted values, `PredictedState` and `PredictedCovariance`.

If the current output measurement exists at a given time step, you can use `correct` and `predict`. If the measurement is missing, you can only use `predict`. For details about how these commands implement the algorithms, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

The order in which you implement the commands depends on the availability of measured data  $y$ ,  $U_s$ , and  $U_m$  for your system:

- `correct` then `predict` — Assume that at time step  $k$ , the value of `obj.State` is  $\hat{x}[k|k-1]$ . This value is the state of the system at time  $k$ , estimated using measured outputs until time  $k-1$ . You also have the measured output  $y[k]$  and inputs  $U_s[k]$  and  $U_m[k]$  at the same time step.

Then you first execute the `correct` command with measured system data  $y[k]$  and additional inputs  $U_m[k]$ . The command updates the value of `obj.State` to be  $\hat{x}[k|k]$ , the state estimate for time  $k$ , estimated using measured outputs up to time  $k$ . When you then execute the `predict` command with input  $U_s[k]$ , `obj.State` now stores  $\hat{x}[k+1|k]$ . The algorithm uses this state value as an input to the `correct` command in the next time step.

- `predict` then `correct` — Assume that at time step  $k$ , the value of `obj.State` is  $\hat{x}[k-1|k-1]$ . You also have the measured output  $y[k]$  and input  $U_m[k]$  at the same time step but you have  $U_s[k-1]$  from the previous time step.

Then you first execute the `predict` command with input  $U_s[k-1]$ . The command updates the value of `obj.State` to  $\hat{x}[k|k-1]$ . When you then execute the `correct` command with input arguments  $y[k]$  and  $U_m[k]$ , `obj.State` is updated with  $\hat{x}[k|k]$ . The algorithm uses this state value as an input to the `predict` command in the next time step.

Thus, while in both cases the state estimate for time  $k$ ,  $\hat{x}[k|k]$  is the same, if at time  $k$  you do not have access to the current state transition function inputs  $U_s[k]$ , and instead have  $U_s[k-1]$ , then use `predict` first and then `correct`.

For an example of estimating states using the `predict` and `correct` commands, see “Estimate States Online Using Unscented Kalman Filter” on page 2-817.

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”

### See Also

`clone` | `correct` | `extendedKalmanFilter` | `unscentedKalmanFilter`

**Introduced in R2016b**



# prescale

Optimal scaling of state-space models

## Syntax

```
scaledsys = prescale(sys)
scaledsys = prescale(sys,focus)
[scaledsys,info] = prescale(...)
prescale(sys)
```

## Description

`scaledsys = prescale(sys)` scales the entries of the state vector of a state-space model `sys` to maximize the accuracy of subsequent frequency-domain analysis. The scaled model `scaledsys` is equivalent to `sys`.

`scaledsys = prescale(sys,focus)` specifies a frequency interval `focus = {fmin,fmax}` (in rad/TimeUnit, where `TimeUnit` is the system's time units specified in the `TimeUnit` property of `sys`) over which to maximize accuracy. This is useful when `sys` has a combination of slow and fast dynamics and scaling cannot achieve high accuracy over the entire dynamic range. By default, `prescale` attempts to maximize accuracy in the frequency band with dominant dynamics.

`[scaledsys,info] = prescale(...)` also returns a structure `info` with the fields shown in the following table.

SL	Left scaling factors
SR	Right scaling factors
Freqs	Frequencies used to test accuracy
RelAcc	Guaranteed relative accuracy at these frequencies

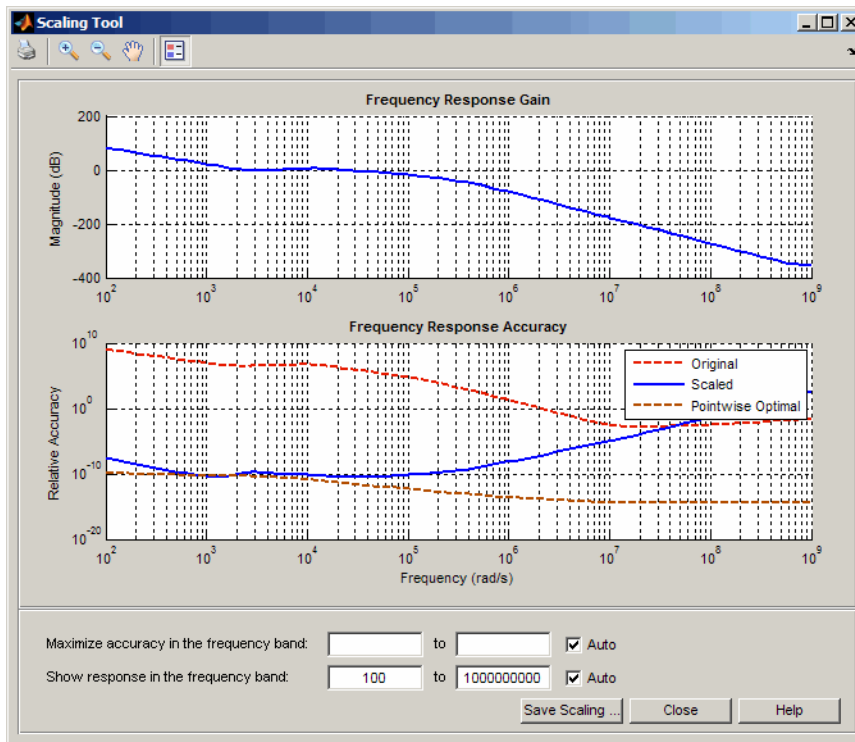
The test frequencies lie in the frequency interval `focus` when specified. The scaled state-space matrices are

$$\begin{aligned}
 A_s &= T_L A T_R \\
 B_s &= T_L B \\
 C_s &= C T_R \\
 E_s &= T_L E T_R
 \end{aligned}$$

where  $T_L = \text{diag}(SL)$  and  $T_R = \text{diag}(SR)$ .  $T_L$  and  $T_R$  are inverse of each other for explicit models ( $E = []$ ).

`prescale(sys)` opens an interactive GUI for:

- Visualizing accuracy trade-offs for `sys`.
- Adjusting the frequency interval where the accuracy of `sys` is maximized.



For more information on scaling and using the Scaling Tool GUI, see “Scaling State-Space Models”.

## More About

### Tips

Most frequency-domain analysis commands perform automatic scaling equivalent to `scaledsys = prescale(sys)`.

You do not need to scale for time-domain simulations and doing so may invalidate the initial condition `x0` used in `initial` and `lsim` simulations.

### See Also

`ss`

**Introduced in R2008b**

## pzmap

Pole-zero plot of dynamic system

### Syntax

```
pzmap(sys)  
pzmap(sys1,sys2,...,sysN)  
[p,z] = pzmap(sys)
```

### Description

`pzmap(sys)` creates a pole-zero plot of the continuous- or discrete-time dynamic system 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)` creates the pole-zero plot of multiple models on a single figure. The models can have different numbers of inputs and outputs and can be a mix of continuous and discrete systems.

`[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.

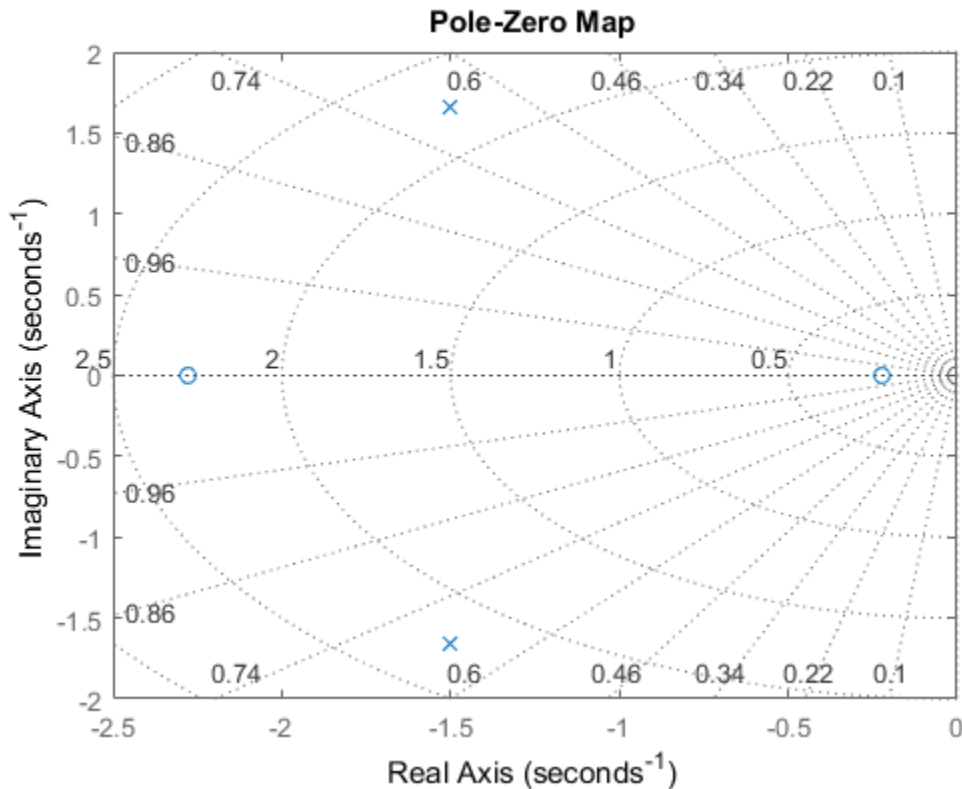
### Examples

#### Pole-Zero Plot of Dynamic System

Plot the poles and zeros of the continuous-time system represented by the following transfer function:

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

```
H = tf([2 5 1],[1 3 5]);
pzmap(H)
grid on
```



Turning on the grid displays lines of constant damping ratio ( $\zeta$ ) and lines of constant natural frequency ( $\omega_n$ ). This system has two real zeros, marked by  $o$  on the plot. The system also has a pair of complex poles, marked by  $x$ .

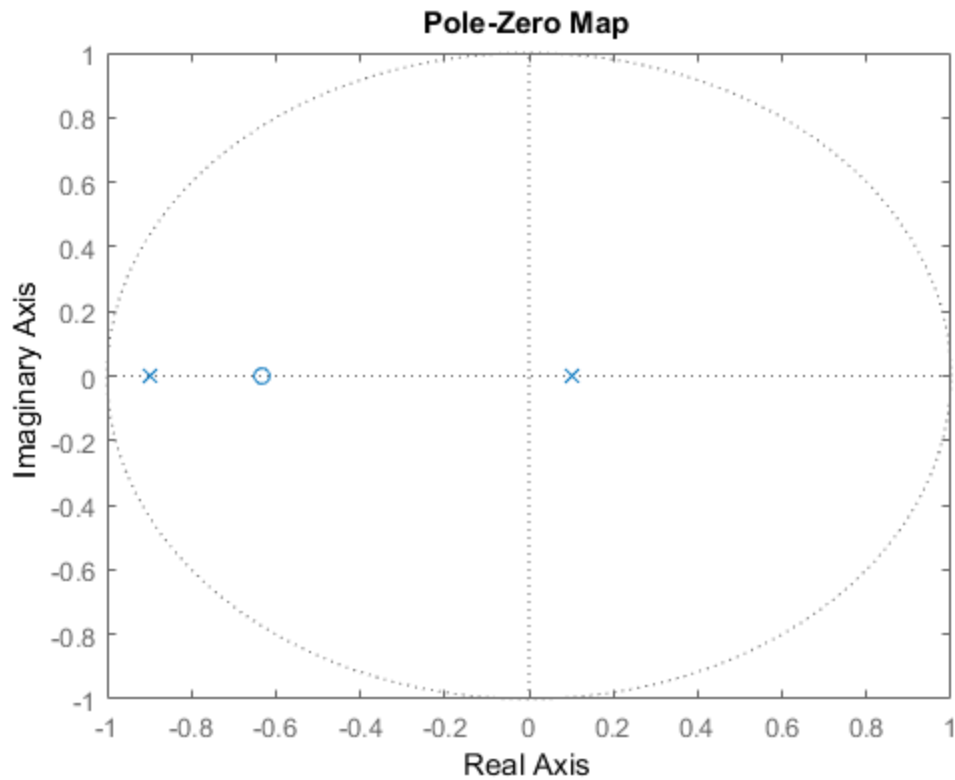
### Pole-Zero Plot of Identified System

Plot the pole-zero map of a discrete time identified state-space (`idss`) model. In practice you can obtain an `idss` model by estimation based on input-output measurements of a system. For this example, create one from state-space data.

```
A = [0.1 0; 0.2 -0.9];  
B = [.1 ; 0.1];  
C = [10 5];  
D = [0];  
sys = idss(A,B,C,D, 'Ts',0.1);
```

Examine the pole-zero map.

```
pzmap(sys)
```



System poles are marked by x, and zeros are marked by o.

## More About

### Tips

- For MIMO models, `pzmap` shows all system poles and transmission zeros on a single plot. To map poles and zeros for individual I/O pairs, use `iopzmap`.
- For additional options for customizing the appearance of the pole-zero plot, use `pzplot`.

### Algorithms

`pzmap` uses a combination of `pole` and `zero`.

### See Also

`pole` | `sgrid` | `zgrid` | `zero` | `iopzmap` | `pzplot` | `damp` | `esort` | `dsort` | `rlocus`

**Introduced before R2006a**

## pzplot

Pole-zero map of dynamic system model with plot customization options

### Syntax

```
h = pzplot(sys)
pzplot(sys1,sys2,...)
pzplot(AX,...)
pzplot(..., plotoptions)
```

### Description

`h = pzplot(sys)` computes the poles and (transmission) zeros of the dynamic system model `sys` and plots them in the complex plane. The poles are plotted as x's and the zeros are plotted as o's. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help pzoptions
```

for a list of available plot options. For more information on the ways to change properties of your plots, see “Ways to Customize Plots”.

`pzplot(sys1,sys2,...)` shows the poles and zeros of multiple models `sys1,sys2,...` on a single plot. You can specify distinctive colors for each model, as in

```
pzplot(sys1, 'r',sys2, 'y',sys3, 'g')
```

`pzplot(AX,...)` plots into the axes with handle `AX`.

`pzplot(..., plotoptions)` plots the poles and zeros with the options specified in `plotoptions`. Type

```
help pzoptions
```

for more detail.

The function `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 dynamic system models, `pzmap` plots the poles and zeros of each model in the array on the same diagram.

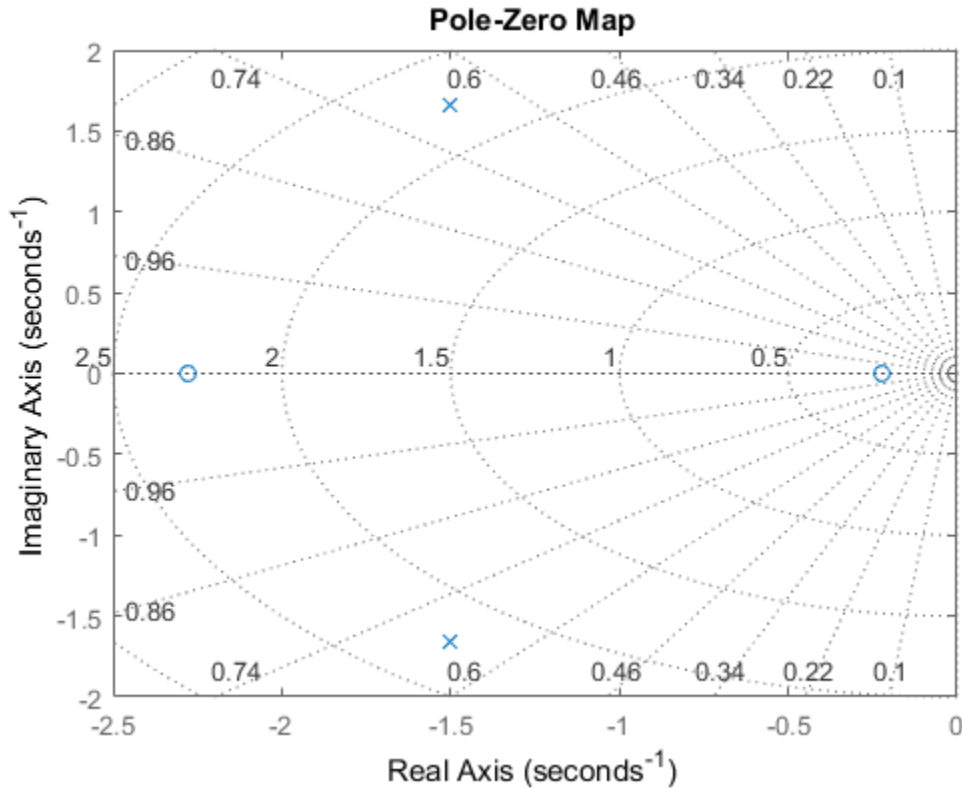
## Examples

### Pole-Zero Plot with Custom Plot Title

Plot the poles and zeros of the continuous-time system represented by the following transfer function:

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

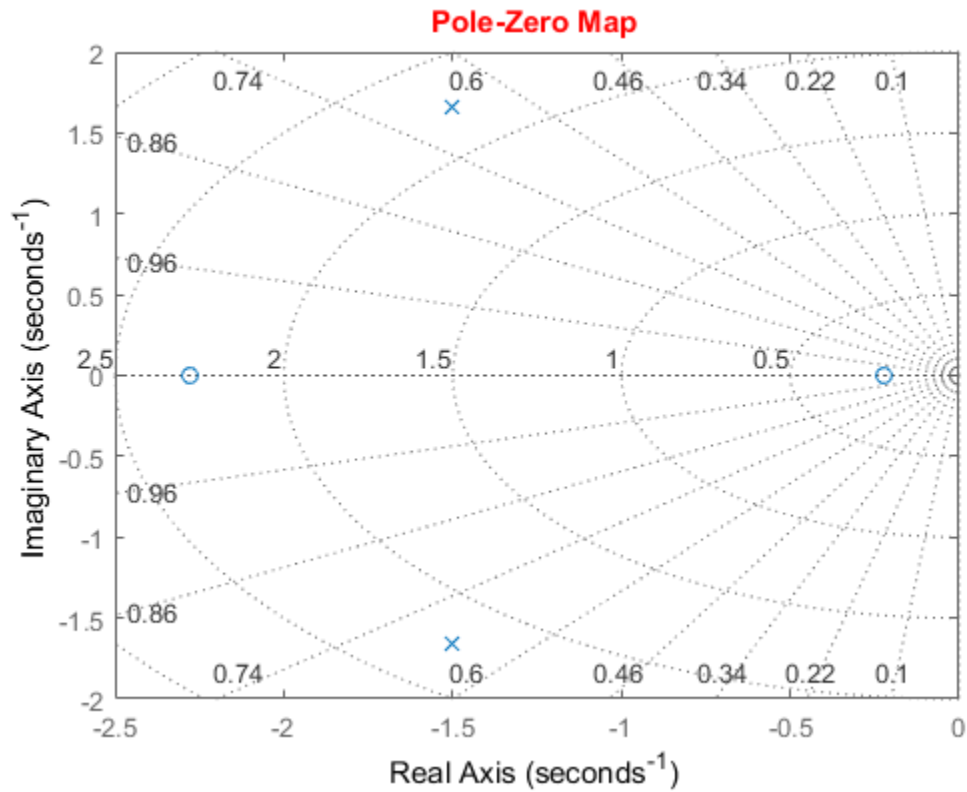
```
sys = tf([2 5 1],[1 3 5]);  
h = pzplot(sys);  
grid on
```



Turning on the grid displays lines of constant damping ratio ( $\zeta$ ) and lines of constant natural frequency ( $\omega_n$ ). This system has two real zeros, marked by o on the plot. The system also has a pair of complex poles, marked by x.

Change the color of the plot title. To do so, use the plot handle, h.

```
p = getoptions(h);
p.Title.Color = [1,0,0];
setoptions(h,p);
```



### See Also

`pzmap` | `setoptions` | `iopzplot` | `getoptions`

Introduced before R2006a

## pzoptions

Create list of pole/zero plot options

### Syntax

```
P = pzoptions
P = pzoption('cstprefs')
```

### Description

`P = pzoptions` returns a list of available options for pole/zero plots (pole/zero, input-output pole/zero and root locus) with default values set.. You can use these options to customize the pole/zero plot appearance from the command line.

`P = pzoption('cstprefs')` initializes the plot options with the options you selected in the Control System and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation.

This table summarizes the available pole/zero plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid, specified as one of the following values: 'off'   'on' <b>Default:</b> 'off'
GridColor	Color of the grid lines, specified as one of the following: Vector of RGB values in the range [0,1]   character vector of color name   'none'. For example, for yellow color, specify as one of the following: [1 1 0], 'yellow', or 'y'. <b>Default:</b> [0.15,0.15,0.15]
XlimMode, YlimMode	Limit modes

Option	Description
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs, specified as one of the following values: 'none'   'inputs'   'outputs'   'all' <b>Default:</b> 'none'
InputLabelLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	<p>Frequency units, specified as one of the following values:</p> <ul style="list-style-type: none"> <li>• 'Hz'</li> <li>• 'rad/second'</li> <li>• 'rpm'</li> <li>• 'kHz'</li> <li>• 'MHz'</li> <li>• 'GHz'</li> <li>• 'rad/nanosecond'</li> <li>• 'rad/microsecond'</li> <li>• 'rad/millisecond'</li> <li>• 'rad/minute'</li> <li>• 'rad/hour'</li> <li>• 'rad/day'</li> <li>• 'rad/week'</li> <li>• 'rad/month'</li> <li>• 'rad/year'</li> <li>• 'cycles/nanosecond'</li> <li>• 'cycles/microsecond'</li> <li>• 'cycles/millisecond'</li> <li>• 'cycles/hour'</li> <li>• 'cycles/day'</li> <li>• 'cycles/week'</li> <li>• 'cycles/month'</li> <li>• 'cycles/year'</li> </ul> <p><b>Default:</b> 'rad/s'</p> <p>You can also specify 'auto' which uses frequency units rad/TimeUnit relative</p>

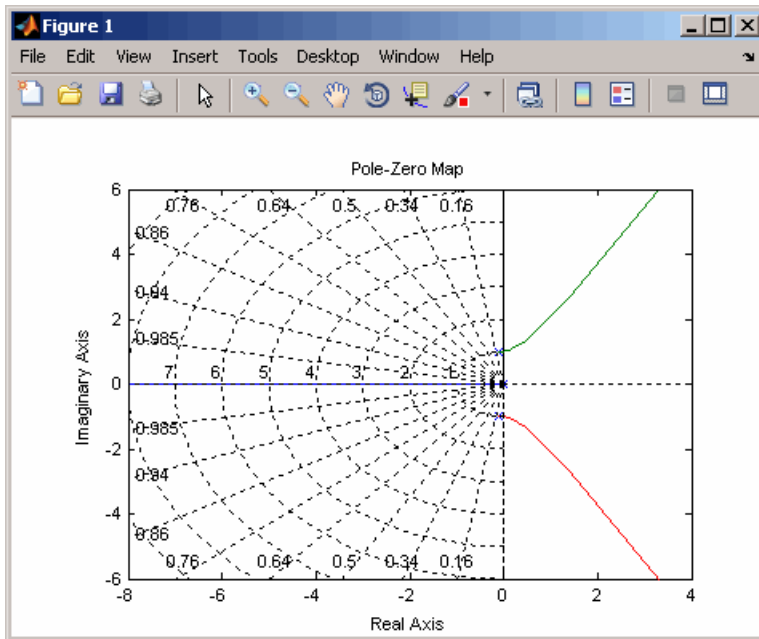
Option	Description
	to system time units specified in the <code>TimeUnit</code> property. For multiple systems with different time units, the units of the first system are used.
TimeUnits	<p>Time units, specified as one of the following values:</p> <ul style="list-style-type: none"> <li>• 'nanoseconds'</li> <li>• 'microseconds'</li> <li>• 'milliseconds'</li> <li>• 'seconds'</li> <li>• 'minutes'</li> <li>• 'hours'</li> <li>• 'days'</li> <li>• 'weeks'</li> <li>• 'months'</li> <li>• 'years'</li> </ul> <p><b>Default:</b> 'seconds'</p> <p>You can also specify 'auto' which uses time units specified in the <code>TimeUnit</code> property of the input system. For multiple systems with different time units, the units of the first system is used.</p>
ConfidenceRegionNumberSD	Number of standard deviations to use when displaying the confidence region characteristic for identified models (valid only <code>iopzplot</code> ).

## Examples

In this example, you enable the grid option before creating a plot.

```
P = pzoptions; % Create set of plot options P
P.Grid = 'on'; % Set the grid to on in options
h = rlocusplot(tf(1,[1,.2,1,0]),P);
```

The following root locus plot is created with the grid enabled.



### See Also

[iopzplot](#) | [pzplot](#) | [setoptions](#) | [getoptions](#)

**Introduced in R2008a**



# realp

Real tunable parameter

## Syntax

```
p = realp(paramname,initvalue)
```

## Description

`p = realp(paramname,initvalue)` creates a tunable real-valued parameter with name specified by `paramname` and initial value `initvalue`. Tunable real parameters can be scalar- or matrix- valued.

## Input Arguments

### **paramname**

Name of the `realp` parameter `p`, specified as a character vector such as `'a'` or `'zeta'`. This input argument sets the value of the `Name` property of `p`.

### **initvalue**

Initial numeric value of the parameter `p`. `initvalue` can be a real scalar value or a 2-dimensional matrix.

## Output Arguments

### **p**

`realp` parameter object.

## Properties

### Name

Name of the `realp` parameter object, stored as a character vector. The value of `Name` is set by the `paramname` input argument to `realp` and cannot be changed.

### Value

Value of the tunable parameter.

`Value` can be a real scalar value or a 2-dimensional matrix. The initial value is set by the `initvalue` input argument. The dimensions of `Value` are fixed on creation of the `realp` object.

### Minimum

Lower bound for the parameter value. The dimension of the `Minimum` property matches the dimension of the `Value` property.

For matrix-valued parameters, use indexing to specify lower bounds on individual elements:

```
p = realp('K',eye(2));  
p.Minimum([1 4]) = -5;
```

Use scalar expansion to set the same lower bound for all matrix elements:

```
p.Minimum = -5;
```

**Default:** -Inf for all entries

### Maximum

Upper bound for the parameter value. The dimension of the `Maximum` property matches the dimension of the `Value` property.

For matrix-valued parameters, use indexing to specify upper bounds on individual elements:

```
p = realp('K',eye(2));  
p.Maximum([1 4]) = 5;
```

Use scalar expansion to set the same upper bound for all matrix elements:

```
p.Maximum = 5;
```

**Default:** Inf for all entries

### Free

Boolean value specifying whether the parameter is free to be tuned. Set the **Free** property to 1 (**true**) for tunable parameters, and 0 (**false**) for fixed parameters.

The dimension of the **Free** property matches the dimension of the **Value** property.

**Default:** 1 (**true**) for all entries

## Examples

### Create Tunable Low-Pass Filter

This example shows how to create a low-pass filter with one tunable parameter  $a$ :

$$F = \frac{a}{s + a}$$

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

Create a tunable real parameter with an initial value of 10.

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

Use `tf` to create the tunable filter  $F$ .

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

$F$  is a `genss` object which has the tunable parameter  $a$  in its **Blocks** property. You can connect  $F$  with other tunable or numeric models to create more complex control system models. For example, see “Control System with Tunable Components”.

### Create Parametric Diagonal Matrix

Create a matrix with tunable diagonal elements and with off-diagonal elements fixed to zero.

Create a parametric matrix whose initial value is the identity matrix.

```
p = realp('P',eye(2));
```

`p` is a 2-by-2 parametric matrix. Since the initial value is the identity matrix, the off-diagonal initial values are zero.

Fix the values of the off-diagonal elements by setting the `Free` property to `false`.

```
p.Free(1,2) = false;  
p.Free(2,1) = false;
```

## More About

### Tips

- Use arithmetic operators (+, -, \*, /, \, and ^) to combine `realp` objects into rational expressions or matrix expressions. You can use the resulting expressions in model-creation functions such as `tf`, `zpk`, and `ss` to create tunable models. For more information about tunable models, see “Models with Tunable Coefficients” in the *Control System Toolbox User's Guide*.
- “Models with Tunable Coefficients”

### See Also

`genmat` | `genss` | `ss` | `tf`

**Introduced in R2011a**

## reg

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.

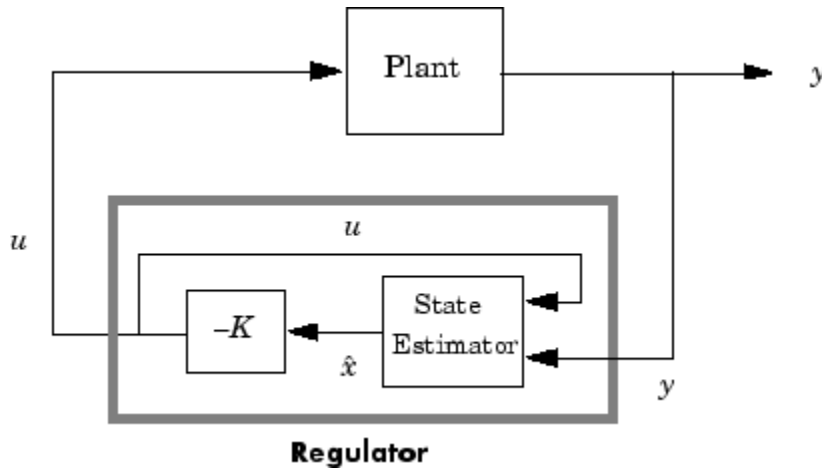
This syntax assumes that all inputs of `sys` are controls, and all outputs are measured. The regulator `rsys` is obtained by connecting the state-feedback law  $u = -Kx$  and the state estimator with gain matrix `L` (see `estim`). For a plant with equations

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

this yields the regulator

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

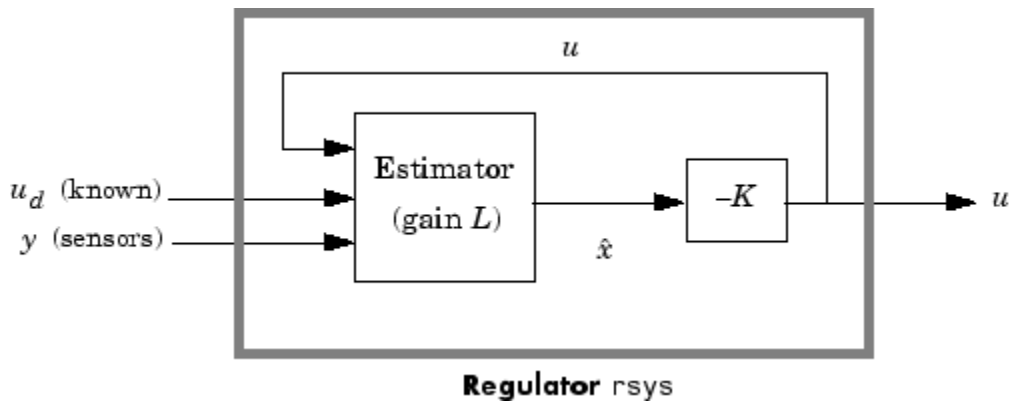
This regulator should be connected to the plant using *positive* feedback.



`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 next figure).



## Examples

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` | `kalman` | `lqr` | `dlqr` | `place` | `lqgreg`

**Introduced before R2006a**

## replaceBlock

Replace or update Control Design Blocks in Generalized LTI model

### Syntax

```
Mnew = replaceBlock(M,Block1,Value1,...,BlockN,ValueN)
Mnew = replaceBlock(M,blockvalues)
Mnew = replaceBlock(...,mode)
```

### Description

`Mnew = replaceBlock(M,Block1,Value1,...,BlockN,ValueN)` replaces the Control Design Blocks `Block1, ..., BlockN` of `M` with the specified values `Value1, ..., ValueN`. `M` is a Generalized LTI model or a Generalized matrix.

`Mnew = replaceBlock(M,blockvalues)` specifies the block names and replacement values as field names and values of the structure `blockvalues`.

`Mnew = replaceBlock(...,mode)` performs block replacement on an array of models `M` using the substitution mode specified by `mode`.

### Input Arguments

**M**

Generalized LTI model, Generalized matrix, or array of such models.

**Block1, ..., BlockN**

Names of Control Design Blocks in `M`. The `replaceBlock` command replaces each listed block of `M` with the corresponding values `Value1, ..., ValueN` that you supply.

If a specified `Block` is not a block of `M`, `replaceBlock` that block and the corresponding value.



**Value1, ..., ValueN**

Replacement values for the corresponding blocks **Block1, ..., BlockN**.

The replacement value for a block can be any value compatible with the size of the block, including a different Control Design Block, a numeric matrix, or an LTI model. If any value is `[]`, the corresponding block is replaced by its nominal (current) value.

**blockvalues**

Structure specifying blocks of **M** to replace and the values with which to replace those blocks.

The field names of **blockvalues** match names of Control Design Blocks of **M**. Use the field values to specify the replacement values for the corresponding blocks of **M**. The replacement values may be numeric values, Numeric LTI models, Control Design Blocks, or Generalized LTI models.

**mode**

Block replacement mode for an input array **M** of Generalized matrices or LTI models, specified as one of the following values:

- `'-once'` (default) — Vectorized block replacement across the model array **M**. Each block is replaced by a single value, but the value may change from model to model across the array.

For vectorized block replacement, use a structure array for the input **blockvalues**, or cell arrays for the **Value1, ..., ValueN** inputs. For example, if **M** is a 2-by-3 array of models:

- `Mnew = replaceBlock(M,blockvalues,'-once')`, where **blockvalues** is a 2-by-3 structure array, specifies one set of block values **blockvalues(k)** for each model **M(:, :, k)** in the array.
- `Mnew = replaceBlock(M,Block,Value,'-once')`, where **Value** is a 2-by-3 cell array, replaces **Block** by **Value{k}** in the model **M(:, :, k)** in the array.
- `'-batch'` — Batch block replacement. Each block is replaced by an array of values, and the same array of values is used for each model in **M**. The resulting array of model **Mnew** is of size `[size(M) Asize]`, where **Asize** is the size of the replacement value.

When the input **M** is a single model, `'-once'` and `'-batch'` return identical results.

Default: ' -once '

## Output Arguments

### Mnew

Matrix or linear model or matrix where the specified blocks are replaced by the specified replacement values.

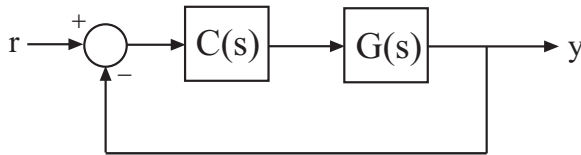
Mnew is a numeric array or numeric LTI model when all the specified replacement values are numeric values or numeric LTI models.

## Examples

### Replace Control Design Block with Numeric Values

This example shows how to replace a tunable PID controller (`tunablePID`) in a Generalized LTI model by a pure gain, a numeric PI controller, or the current value of the tunable controller.

- 1 Create a Generalized LTI model of the following system:



where the plant  $G(s) = \frac{(s-1)}{(s+1)^3}$ , and  $C$  is a tunable PID controller.

```
G = zpk(1,[-1,-1,-1],1);
C = tunablePID('C','pid');
Try = feedback(G*C,1)
```

- 2 Replace  $C$  by a pure gain of 5.

```
T1 = replaceBlock(Try,'C',5);
```

T1 is a ss model that equals `feedback(G*5,1)`.

- 3 Replace C by a PI controller with proportional gain of 5 and integral gain of 0.1.

```
C2 = pid(5,0.1);
T2 = replaceBlock(Try, 'C', C2);
```

T2 is a ss model that equals `feedback(G*C2,1)`.

- 4 Replace C by its current (nominal) value.

```
T3 = replaceBlock(Try, 'C', []);
```

T3 is a ss model where C has been replaced by `getValue(C)`.

## Sample Tunable Model Over Grid of Values

Consider the second-order filter represented by:

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

Sample this filter at varying values of the damping constant  $\zeta$  and the natural frequency  $\omega_n$ . Create a tunable model of the filter by using tunable elements for  $\zeta$  and  $\omega_n$ .

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

F =

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

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

Create a grid of sample values.

```
wvals = [3;5];
```

```
zetavals = [0.6 0.8 1.0];  
[wngrid,zetagrid] = ndgrid(wnvals,zetavals);  
Fsample = replaceBlock(F,'wn',wngrid,'zeta',zetagrid);  
size(Fsample)
```

2x3 array of state-space models.  
Each model has 1 outputs, 1 inputs, and 2 states.

The `ndgrid` command produces a full 2-by-3 grid of parameter combinations. Thus, `Fsample` is a 2-by-3 array of state-space models. Each entry in the array is a state-space model that represents `F` evaluated at the corresponding (`wn`, `zeta`) pair. For example, `Fsample(:, :, 2, 3)` has `wn = 5` and `zeta = 1.0`.

```
damp(Fsample(:, :, 2, 3))
```

Pole	Damping	Frequency (rad/seconds)	Time Constant (seconds)
-5.00e+00	1.00e+00	5.00e+00	2.00e-01
-5.00e+00	1.00e+00	5.00e+00	2.00e-01

## More About

### Tips

- Use `replaceBlock` to perform parameter studies by sampling Generalized LTI models across a grid of parameters, or to evaluate tunable models for specific values of the tunable blocks. See “Examples” on page 2-848.
- For additional options for sampling control design blocks, including concurrent sampling, use `sampleBlock`.
- To take random samples of control design blocks, see `rsampleBlock`
- “Generalized Matrices”
- “Generalized and Uncertain LTI Models”
- “Models with Tunable Coefficients”

### See Also

`getValue` | `genss` | `genmat` | `nblocks` | `sampleBlock` | `rsampleBlock`

**Introduced in R2011a**

## repsys

Replicate and tile models

### Syntax

```
rsys = repsys(sys,[M N])  
rsys = repsys(sys,N)  
rsys = repsys(sys,[M N S1,...,Sk])
```

### Description

`rsys = repsys(sys,[M N])` replicates the model `sys` into an M-by-N tiling pattern. The resulting model `rsys` has `size(sys,1)*M` outputs and `size(sys,2)*N` inputs.

`rsys = repsys(sys,N)` creates an N-by-N tiling.

`rsys = repsys(sys,[M N S1,...,Sk])` replicates and tiles `sys` along both I/O and array dimensions to produce a model array. The indices `S` specify the array dimensions. The size of the array is [`size(sys,1)*M`, `size(sys,2)*N`, `size(sys,3)*S1`, ...].

### Input Arguments

#### **sys**

Model to replicate.

#### **M**

Number of replications of `sys` along the output dimension.

#### **N**

Number of replications of `sys` along the input dimension.

**S**

Numbers of replications of `sys` along array dimensions.

**Output Arguments****rsys**

Model having `size(sys,1)*M` outputs and `size(sys,2)*N` inputs.

If you provide array dimensions `S1, ..., Sk`, `rsys` is an array of dynamic systems which each have `size(sys,1)*M` outputs and `size(sys,2)*N` inputs. The size of `rsys` is `[size(sys,1)*M, size(sys,2)*N, size(sys,3)*S1, ...]`.

**Examples**

Replicate a SISO transfer function to create a MIMO transfer function that has three inputs and two outputs.

```
sys = tf(2,[1 3]);
rsys = repsys(sys,[2 3]);
```

The preceding commands produce the same result as:

```
sys = tf(2,[1 3]);
rsys = [sys sys sys; sys sys sys];
```

Replicate a SISO transfer function into a 3-by-4 array of two-input, one-output transfer functions.

```
sys = tf(2,[1 3]);
rsys = repsys(sys, [1 2 3 4]);
```

To check the size of `rsys`, enter:

```
size(rsys)
```

This command produces the result:

```
3x4 array of transfer functions.
Each model has 1 outputs and 2 inputs.
```

## More About

### Tips

`rsys = repsys(sys,N)` produces the same result as `rsys = repsys(sys,[N N])`.  
To produce a diagonal tiling, use `rsys = sys*eye(N)`.

### See Also

`append`

**Introduced in R2010b**



# reshape

Change shape of model 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-...-by-`sk` model array. With either syntax, there must be `s1*s2*...*sk` models in `sys` to begin with.

## Examples

Change the shape of a model array from 2x3 to 6x1.

```
% Create a 2x3 model array.
sys = rss(4,1,1,2,3);
% Confirm the size of the array.
size(sys)
```

This input produces the following output:

```
2x3 array of state-space models
Each model has 1 output, 1 input, and 4 states.
```

Change the shape of the array.

```
sys1 = reshape(sys,6,1);
size(sys1)
```

This input produces the following output:

```
6x1 array of state-space models
Each model has 1 output, 1 input, and 4 states.
```

**See Also**

size | ndims

**Introduced before R2006a**

# rlocus

Root locus plot of dynamic system

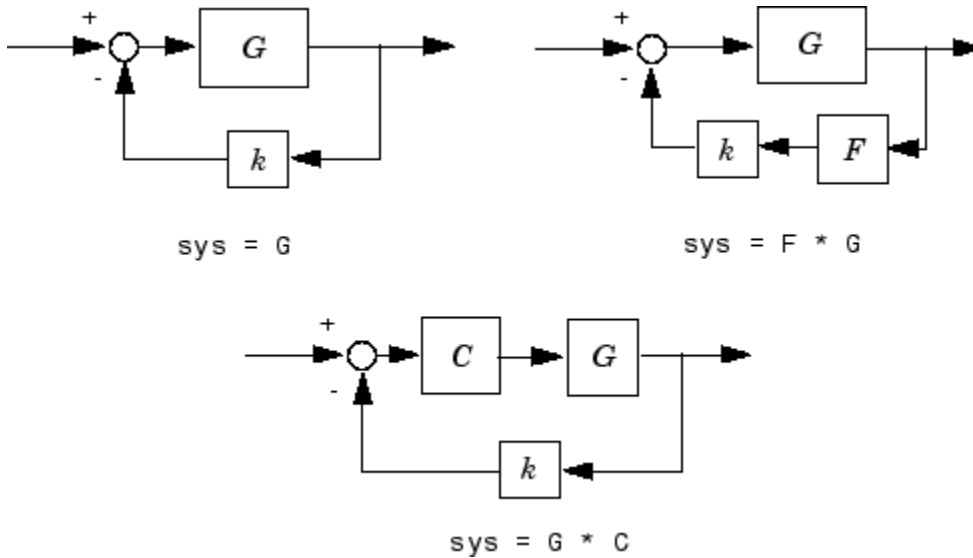
## Syntax

```
rlocus(sys)
rlocus(sys1,sys2,...)
[r,k] = rlocus(sys)
r = rlocus(sys,k)
```

## Description

`rlocus` computes the root locus of a SISO open-loop model. The root locus gives the closed-loop pole trajectories as a function of the feedback gain  $k$  (assuming negative feedback). Root loci are used to study the effects of varying feedback gains on closed-loop pole locations. In turn, these locations provide indirect information on the time and frequency responses.

`rlocus(sys)` calculates and plots the root locus of the open-loop SISO model `sys`. This function can be applied to any of the following *negative* feedback loops by setting `sys` appropriately.



If `sys` has transfer function

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

the closed-loop poles are the roots of

$$d(s) + kn(s) = 0$$

`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')`.

`[r,k] = rlocus(sys)` and `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  $k(j)$ .

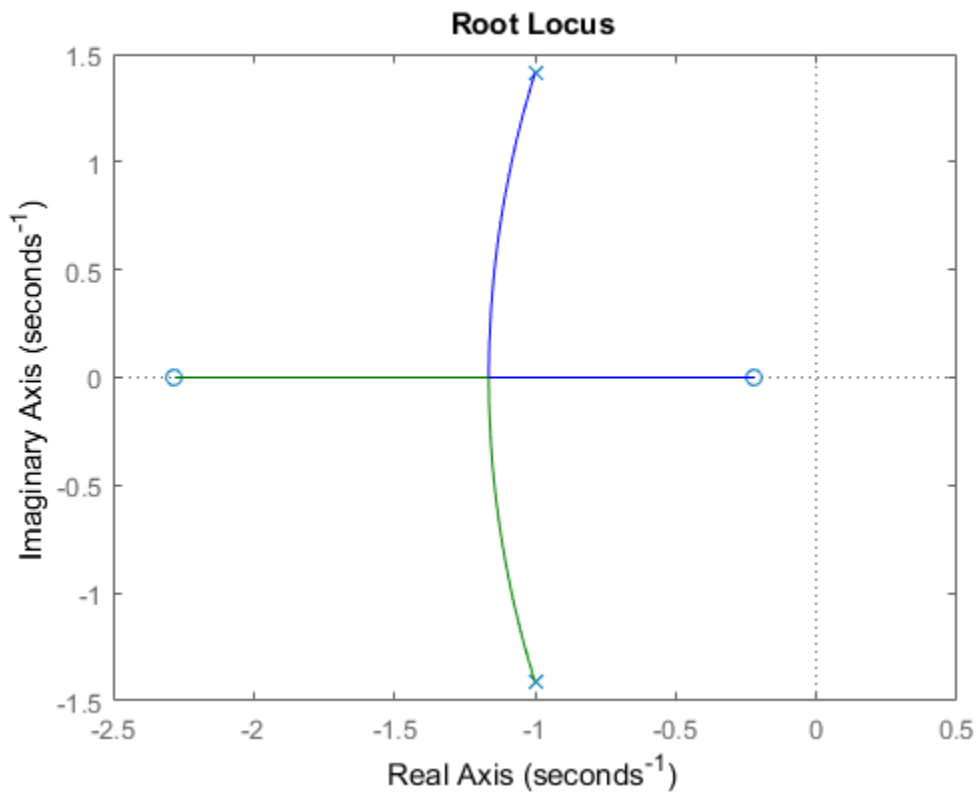
## Examples

### Root Locus Plot of Dynamic System

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.

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`pole` | `pzmap`

**Introduced before R2006a**

# rlocusplot

Plot root locus and return plot handle

## Syntax

```
h = rlocusplot(sys)
rlocusplot(sys,k)
rlocusplot(sys1,sys2,...)
rlocusplot(AX,...)
rlocusplot(..., plotoptions)
```

## Description

`h = rlocusplot(sys)` computes and plots the root locus of the single-input, single-output LTI model `sys`. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help pzoptions
```

for a list of available plot options.

See `rlocus` for a discussion of the feedback structure and algorithms used to calculate the root locus.

`rlocusplot(sys,k)` uses a user-specified vector `k` of gain values.

`rlocusplot(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

```
rlocusplot(sys1,'r',sys2,'y:',sys3,'gx')
```

`rlocusplot(AX,...)` plots into the axes with handle `AX`.

`rlocusplot(..., plotoptions)` plots the root locus with the options specified in `plotoptions`. Type

```
help pzoptions
```

for more details.

## Examples

Use the plot handle to change the title of the plot.

```
sys = rss(3);  
h = rlocusplot(sys);  
p = getoptions(h); % Get options for plot.  
p.Title.String = 'My Title'; % Change title in options.  
setoptions(h,p); % Apply options to plot.
```

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`getoptions` | `rlocus` | `pzoptions` | `setoptions`

**Introduced before R2006a**



# rsampleBlock

Randomly sample Control Design blocks in generalized model

## Syntax

```
Msamp = rsampleBlock(M, names, N)
Msamp = rsampleBlock(M, names1, N1, names2, N2, ..., namesM, NM)
[Msamp, samples] = rsampleBlock( ____ )
```

## Description

`Msamp = rsampleBlock(M, names, N)` randomly samples a subset of the Control Design blocks in the generalized model `M`. The `names` argument specifies which blocks to sample, and `N` specifies how many samples to take. The result `Msamp` is a model array of size `[size(M) N]` obtained by replacing the sampled blocks with their randomized values.

`Msamp = rsampleBlock(M, names1, N1, names2, N2, ..., namesM, NM)` takes `N1` samples of the blocks listed in `names1`, `N2` samples of the blocks listed in `names2`, and so on. The result `Msamp` is a model array of size `[size(M) N1 N2 ... NM]`.

`[Msamp, samples] = rsampleBlock( ____ )` also returns a data structure containing the block replacement values for each sampling point. You can use this syntax with any of the preceding input argument combinations.

## Examples

### Randomly Sample Parameter in Tunable Model

Create the first-order model  $G(s) = 1/(\tau s + 1)$ , where  $\tau$  is a tunable real parameter.

```
tau = realp('tau',5);
G = tf(1,[tau 1]);
```

Restrain `tau` to nonnegative values only.

```
G.Blocks.tau.Minimum = 0;
```

Generate 20 random samples of **G**. The result is a 20-by-1 array of first-order models with random values of **tau** taken from the range of **tau**.

```
Gs = rsampleBlock(G, 'tau', 20);  
size(Gs)
```

20x1 array of state-space models.  
Each model has 1 outputs, 1 inputs, and 1 states.

### Randomly Sample Multiple Parameters

Take random samples of a model with both tunable and uncertain blocks. Using uncertain blocks requires Robust Control Toolbox™. Random sampling of tunable blocks works the same way as shown in this example.

Create an uncertain model of  $G(s) = a/(\tau s + 1)$ , where  $a$  is an uncertain parameter that varies in the interval [3,5], and  $\tau = 0.5 \pm 30\%$ . Also, create a tunable PI controller, and form a closed-loop system from the tunable controller and uncertain system.

```
a = ureal('a', 4);  
tau = ureal('tau', .5, 'Percentage', 30);  
G = tf(a, [tau 1]);  
C = tunablePID('C', 'pi');  
T = feedback(G*C, 1);
```

**T** is a generalized state-space model with two uncertain blocks, **a** and **tau**, and one tunable block, **C**. Sample **T** at 20 random (**a**, **tau**) pairs.

```
[Ts, samples] = rsampleBlock(T, {'a', 'tau'}, 20);
```

**Ts** is a 20-by-1 array of **genss** models. The tunable block **C**, which is not sampled, is preserved in **Ts**. The structure **samples** has fields **samples.a** and **samples.tau** that contain the values at which those blocks are sampled.

Grouping **a** and **tau** into a cell array causes **rsampleBlock** to sample them together, as (**a**, **tau**) pairs. Sampling the blocks independently generates a higher-dimensionality arrays. For example, independently taking 10 random samples of **a** and 5 samples of **tau** generates a 10-by-5 model array.

```
[TsInd, samples] = rsampleBlock(T, 'a', 10, 'tau', 5);  
TsInd
```

TsInd =

10x5 array of generalized continuous-time state-space models.  
 Each model has 1 outputs, 1 inputs, 2 states, and the following blocks:  
 C: Parametric PID controller, 1 occurrences.

Type "ss(TsInd)" to see the current value, "get(TsInd)" to see all properties, and "TsInd"

In this array, **a** varies along one dimension and **tau** varies along the other.

## Input Arguments

### M — Model to sample

generalized model | uncertain model | generalized matrix | uncertain matrix

Model to sample, specified as a:

- Generalized model (**genss** or **genfrd**)
- Generalized matrix (**genmat**)
- Uncertain model (**uss** or **ufrd**)
- Uncertain matrix (**umat**)

### names — Control Design blocks

character vector | cell array of character vectors

Control Design blocks to sample, specified as a character vector or cell array of character vectors. The entries in **names** correspond to the names of at least a subset of the Control Design blocks in **M**. For example, suppose that **M** is a **genss** model with tunable blocks **t1** and **t2**, and uncertain blocks **u1** and **u2**. Then, {'t1', 'u2'} is one possible value for **names**.

Grouping block names together in a cell array generates samples of the group rather than independent samples of each block. For example, the following code generates a 10-by-1 array of models, where each entry in the array has a random value for the pair (**t1**, **u2**).

```
Msamp = rsampleBlock(M,{'t1', 'u2'},10);
```

To sample parameters independently, do not group them. For example, the following code generates a 10-by-20 array of models, where `t1` varies along the first dimension and `u2` varies along the second dimension.

```
Msamp = rsampleBlock(M, 't1', 10, 'u2', 20);
```

`rsampleBlock` ignores any entry in `names` that does not appear in `M`.

### **N — Number of samples**

positive integer

Number of samples to take of the preceding block or blocks, specified as a positive integer.

## Output Arguments

### **Msamp — Array of model samples**

generalized model array | `ss` array | `frd` array | numeric array

Array of model samples, returned as a generalized model array, `ss` array, `frd` array, or numeric array. `Msamp` is of the same type as `M`, unless all blocks are sampled. In that case, `Msamp` is a numeric array, `ss` array, or `frd` array. For example, suppose that `M` is a `uss` model with uncertain blocks `u1` and `u2`. The following command returns an array of `uss` models, with uncertain block `u2`.

```
Msamp1 = rsampleBlock(M, 'u1', 10);
```

The following command samples both blocks and returns an array of `ss` models.

```
Msamp2 = rsampleBlock(M, {'u1', 'u2'}, 10);
```

`rsampleBlock` uses values that fall within the uncertainty range when sampling uncertain blocks, and within the maximum and minimum parameter values when sampling tunable blocks.

### **samples — Block sample values**

structure

Block sample values, returned as a structure. The fields of `samples` are the names of the sampled blocks. The values are arrays containing the corresponding random values used to generate the entries in `Msamp`. For instance, suppose that you run the following command, where `M` is a `genss` model with tunable blocks `t1` and `t2`.

```
[Msamp,samples] = rsampleBlock(M,{'t1','t2'},10);
```

Then, `samples.t1` contains the 10 values of `t1` and `samples.t2` contains the 10 values of `t2`. If you sample a block that is not scalar valued, the corresponding field of `samples` contains values compatible with the block. For instance, if you sample a `tunablePID` block, `samples` contains an array of state-space models that represent PID controllers.

## More About

- “Generalized Models”

## See Also

`genmat` | `genss` | `getValue` | `replaceBlock` | `sampleBlock` | `uss`

**Introduced in R2016a**

### **rss**

Generate random continuous test model

### **Syntax**

```
rss(n)
rss(n,p)
rss(n,p,m,s1,...,sn)
```

### **Description**

`rss(n)` generates an  $n$ -th order model with one input and one output and returns the model in the state-space object `sys`. The poles of `sys` are random and stable with the possible exception of poles at  $s = 0$  (integrators).

`rss(n,p)` generates an  $n$ th order model with one input and  $p$  outputs, and `rss(n,p,m)` generates an  $n$ -th order model with  $m$  inputs and  $p$  outputs. The output `sys` is always a state-space model.

`rss(n,p,m,s1,...,sn)` generates an  $s_1$ -by-...-by- $s_n$  array of  $n$ -th order 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.

### **Examples**

#### **Generate State-Space Models**

Generate a random SISO state-space model with two states.

```
sys2 = rss(2)
```

```
sys2 =
```

```
A =
      x1      x2
x1  -1.101  0.3733
x2   0.3733 -0.9561
```

```
B =
      u1
x1   0.7254
x2  -0.06305
```

```
C =
      x1      x2
y1     0  -0.205
```

```
D =
      u1
y1  -0.1241
```

Continuous-time state-space model.

Generate a model with four states, three outputs, and two inputs. The input arguments to `rss` are arranged in the order states, outputs, inputs.

```
sys4 = rss(4,3,2)
```

```
sys4 =
```

```
A =
      x1      x2      x3      x4
x1  -0.6722  -3.145  -4.692  -4.391
x2   2.312  -0.3352   8.041   6.791
x3   5.398   -7.51  -0.5229   1.114
x4   4.087  -7.059  -0.3362  -0.4294
```

```
B =
      u1      u2
x1     0  -0.2256
x2   1.533     0
x3  -0.7697     0
x4     0  0.03256
```

```
C =
      x1      x2      x3      x4
```

```
y1  0.5525  0.08593  -1.062  0.7481
y2  1.101   0        2.35  -0.1924
y3  1.544   0       -0.6156  0.8886
```

```
D =
      u1      u2
y1      0  0.4882
y2  -1.402  0
y3      0 -0.1961
```

Continuous-time state-space model.

### Generate Array of Random Models

Generate a 4-by-5 array of SISO models with three states each.

```
sysarray = rss(3,1,1,4,5);
size(sysarray)
```

4x5 array of state-space models.  
Each model has 1 outputs, 1 inputs, and 3 states.

### See Also

[drss](#) | [frd](#) | [tf](#) | [zpk](#)

**Introduced before R2006a**



# sampleBlock

Sample Control Design blocks in generalized model

## Syntax

```
Msamp = sampleBlock(M,name,vals)
Msamp = sampleBlock(M,nameset,valset)
Msamp= sampleBlock(
M, nameset1, valset1, nameset2, valset2, ..., namesetM, valsetM)
[Msamp, samples] = sampleBlock( ___ )
```

## Description

`Msamp = sampleBlock(M,name,vals)` samples one Control Design block in the generalized model `M`. The result `Msamp` is a model array of size `[size(M) N]` obtained by replacing the block with the specified values, where `N` is the number of values in `vals`.

`Msamp = sampleBlock(M, nameset, valset)` concurrently samples multiple blocks specified as a cell array of block names. `valset` is a cell array of `N` sample values for each block. The result `Msamp` is a model array of size `[size(M) N]`.

`Msamp= sampleBlock(M, nameset1, valset1, nameset2, valset2, ..., namesetM, valsetM)` independently samples multiple blocks. `nameset1, nameset2, ..., namesetM` can each be a single block name (see `name`) or a cell array of names (see `nameset`). The model `M` is sampled over a grid of size `[N1 N2 ... NM]`, where `N1` is the number of values in `valset1`, `N2` is the number of values in `valset2`, and so on. The resulting `Msamp` is an array of size `[size(M) N1 N2 ... NM]`.

`[Msamp, samples] = sampleBlock( ___ )` also returns a data structure containing the block replacement values for each sampling point. You can use this syntax with any of the preceding input argument combinations.

## Examples

### Sample Real Parameter in Tunable Model

Create the first-order model  $G(s) = 1/(\tau s + 1)$ , where  $\tau$  is a tunable real parameter.

```
tau = realp('tau',5);  
G = tf(1,[tau 1]);
```

Evaluate this transfer function for  $\tau = 3,4,\dots,7$ . The result is a 5-by-1 array of first-order models.

```
Gs = sampleBlock(G,'tau',3:7);  
size(Gs)
```

```
5x1 array of state-space models.  
Each model has 1 outputs, 1 inputs, and 1 states.
```

### Sample Multiple Parameters in Tunable Model

Create a model with a pole at  $s = a$  and a gain of  $b*c$ , where  $a$ ,  $b$ , and  $c$  are tunable scalars.

```
a = realp('a',1);  
b = realp('b',3);  
c = realp('c',1);  
G = tf(b*c,[1 a]);
```

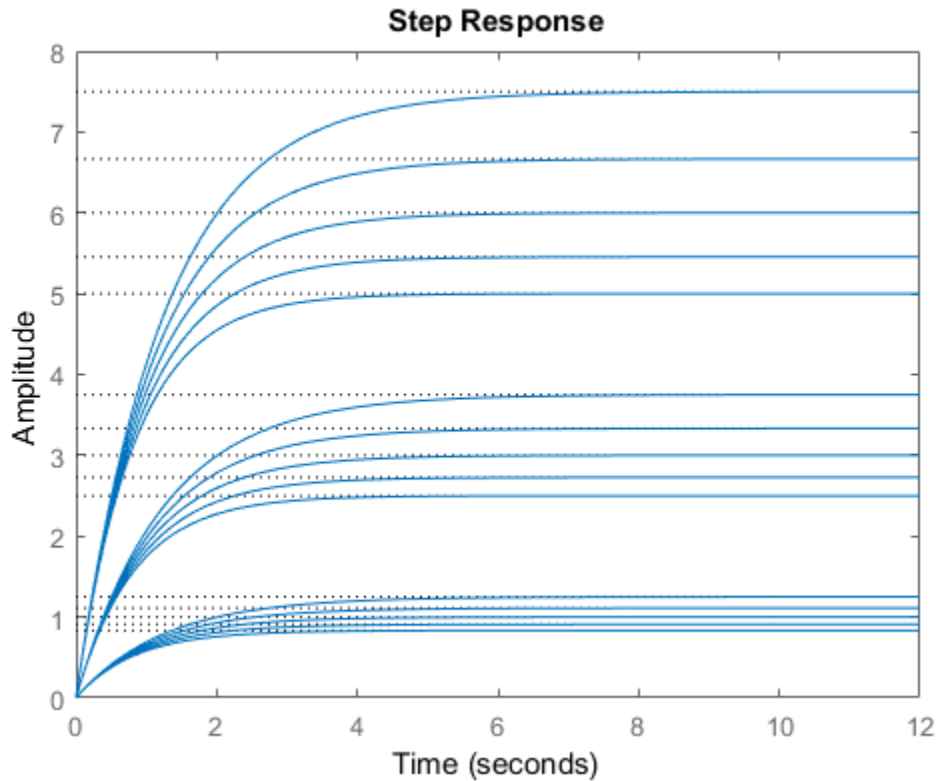
Pick 5 samples for  $a$  and 3 samples for  $(b, c)$  pairs. Evaluate  $G$  over the corresponding 5-by-3 grid of  $(a, b, c)$  combinations.

```
as = 0.8:0.1:1.2;  
bs = 2:4;  
cs = [0.5 1 1.5];  
Gs = sampleBlock(G,'a',as,{'b','c'},{bs,cs});
```

Grouping the values for  $b$  and  $c$  in cell arrays causes `sampleBlock` to treat them as the  $(b, c)$  pairs, (2,0.5), (3,1), and (1,5).  $Gs$  is a 5-by-3 array of state-space models, in which  $a$  varies along the first dimension and  $(b, c)$  varies along the second dimension. Thus, for example,  $Gs(:, :, 3, 2)$  corresponds to  $a = 1$ ,  $(b, c) = (3, 1)$ .

A step plot shows a set of responses for each of the three  $(b, c)$  pairs. Each set contains a response for each of the five  $a$  values.

```
stepplot(Gs)
```



If you do not group the values, `sampleBlock` replaces all values independently, resulting in a 5-by-3-by-3 model array.

```
GsInd = sampleBlock(G, 'a', as, 'b', bs, 'c', cs);
size(GsInd)
```

```
5x3x3 array of state-space models.
Each model has 1 outputs, 1 inputs, and 1 states.
```

For example, in `GsInd`, `Gs(:, :, 3, 2, 1)` is a model with  $a = 1$ ,  $b = 3$ , and  $c = 0.5$ .

- “Study Parameter Variation by Sampling Tunable Model”

## Input Arguments

### **M** — Model to sample

generalized model | uncertain model | generalized matrix | uncertain matrix

Model to sample, specified as a:

- Generalized model (`genss` or `genfrd`)
- Generalized matrix (`genmat`)
- Uncertain model (`uss` or `ufrd`)
- Uncertain matrix (`umat`)

### **name** — Control Design block

character vector

Control Design block to sample, specified as a character vector. For example, suppose that **M** is a `genss` model with tunable blocks `t1` and `t2`. Then, either `'t1'` or `'t2'` is a possible value for **name**.

### **vals** — Sample block values

numeric array | model array

Sample block values, specified as a numeric array or a model array. Values must be compatible with the block type. For example, if **name** is a tunable real parameter (`realp`), then **vals** is a numerical array of length *N*, the number of samples. If **name** is a tunable PID controller (`tunablePID`), then **vals** is an array of LTI models compatible with PID structure.

### **nameset** — Control Design blocks

cell array of character vectors

Control Design blocks to sample concurrently, specified as cell array of character vectors. The entries in **nameset** correspond to the names of at least a subset of the Control Design blocks in **M**. For example, suppose that **M** is a `genss` model with tunable blocks `t1` and `t2`, and uncertain blocks `u1` and `u2`. Then, `{'t1', 'u2'}` is one possible value for **nameset**.

Grouping block names together in a cell array generates samples of the group rather than independent samples. For example, the following code generates a 10-by-1 array of models, where each entry in the array has the corresponding value for the pair (`t1`, `u2`).

```
t1s = 1:10;
```

```
u2s = 2:2:20;
valset = {t1s,t2s};
Msamp = sampleBlock(M,{'t1','u2'},valset);
```

sampleBlock ignores any entry in nameset that does not appear in M.

### valset — Sample block values

cell array

Sample block values, specified as a cell array. Each entry in the cell array is itself an array of N sample values for each block in nameset. For example, the following code samples a model M at the (t1,u2) pairs (1,2), (2,4), ... (10,20).

```
t1s = 1:10;
u2s = 2:2:20;
valset = {t1s,t2s};
Msamp = sampleBlock(M,{'t1','u2'},valset);
```

Values in valset must be compatible with the corresponding block type.

## Output Arguments

### Msamp — Array of model samples

generalized model array | ss array | frd array | numeric array

Array of model samples, returned as a generalized model array, ss array, frd array, or numeric array. Msamp is of the same type as M, unless all blocks are sampled. In that case, Msamp is a numeric array, ss array, or frd array. For example, suppose that M is a uss model with uncertain blocks u1 and u2. The following command returns an array of uss models, with uncertain block u2.

```
Msamp1 = sampleBlock(M,'u1',1:10);
```

The following command samples both blocks and returns an array of ss models.

```
Msamp2 = sampleBlock(M,{'u1','u2'},{1:10,2:20});
```

### samples — Block sample values

structure

Block sample values, returned as a structure. The fields of samples are the names of the sampled blocks. The values are arrays containing the corresponding values used to generate the entries in Msamp.

## **More About**

- “Generalized Models”

## **See Also**

`genmat` | `genss` | `getValue` | `replaceBlock` | `rsampleBlock` | `uss`

**Introduced in R2016a**

# sectorplot

Compute or plot sector index as function of frequency

## Syntax

```
sectorplot(H,Q)
sectorplot(H,Q,w)
sectorplot(H1,H2,...,HN,Q)
sectorplot(H1,H2,...,HN,Q,w)
sectorplot(H1,PlotStyle1,...,HN,PlotStyleN,Q)
sectorplot(H1,PlotStyle1,...,HN,PlotStyleN,Q,w)
```

```
[index,wout] = sectorplot(H,Q)
index = sectorplot(H,Q,w)
```

## Description

`sectorplot(H,Q)` plots the relative sector indices for the dynamic system  $H$  and a given sector matrix  $Q$ . These indices measure by how much the sector bound is satisfied (index less than 1) or violated (index greater than 1) at a given frequency. (See “About Sector Bounds and Sector Indices” for more information about the meaning of the sector index.) `sectorplot` automatically chooses the frequency range and number of points based on the dynamics of  $H$ .

Let the following be an orthogonal decomposition of the symmetric matrix  $Q$  into its positive and negative parts.

$$Q = W_1 W_1^T - W_2 W_2^T, \quad W_1^T W_2 = 0.$$

The sector index plot is only meaningful if  $W_2^T H$  has a proper stable inverse. In that case, the sector indices are the singular values of:

$$\left( W_1^T H(j\omega) \right) \left( W_2^T H(j\omega) \right)^{-1}.$$

`sectorplot(H,Q,w)` plots the sector index for frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `sectorplot` plots the sector index at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `sectorplot` plots the sector index at each specified frequency.

`sectorplot(H1,H2,...,HN,Q)` and `sectorplot(H1,H2,...,HN,Q,w)` plot the sector index for multiple dynamic systems `H1,H2,...,HN` on the same plot.

`sectorplot(H1,PlotStyle1,...,HN,PlotStyleN,Q)` and `sectorplot(H1,PlotStyle1,...,HN,PlotStyleN,Q,w)` specify a color, linestyle, and marker for each system in the plot.

`[index,wout] = sectorplot(H,Q)` returns the sector index at each frequency in the vector `wout`. The output `index` is a matrix, and the value `index(:,k)` gives the sector indices in descending order at the frequency `w(k)`. This syntax does not draw a plot.

`index = sectorplot(H,Q,w)` returns the sector indices at the frequencies specified by `w`.

## Examples

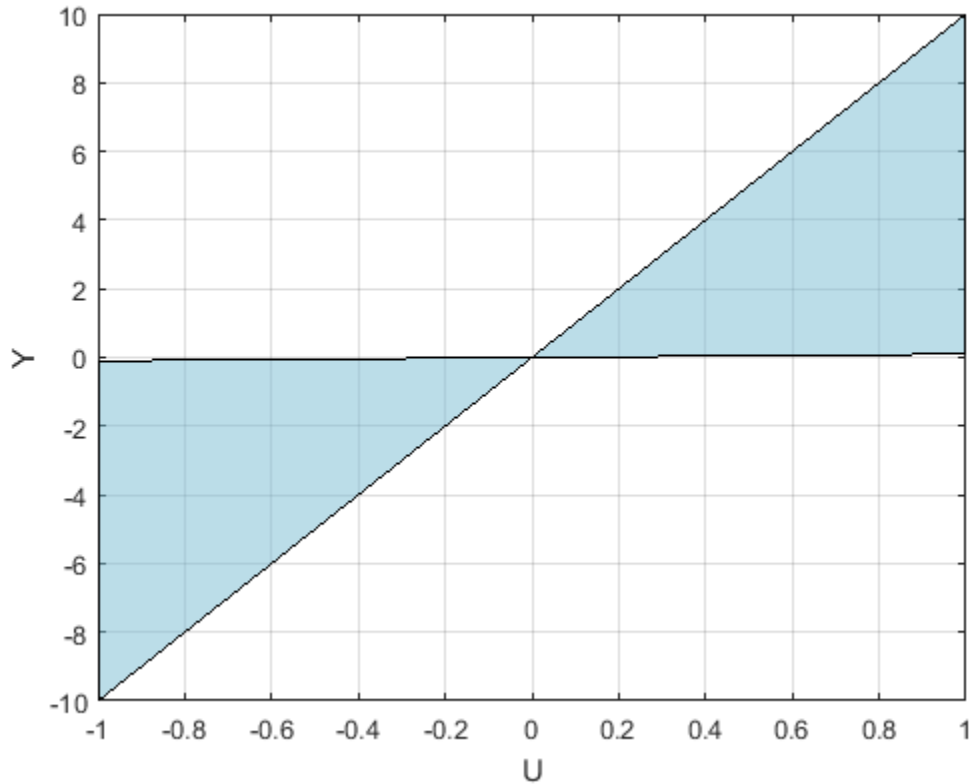
### Plot Sector Index Versus Frequency

Plot the sector index to visualize the frequencies at which the I/O trajectories of  $G(s) = (s + 2) / (s + 1)$  lie within the sector defined by:

$$S = \{(y, u) : 0.1u^2 < uy < 10u^2\}.$$

In  $U/Y$  space, this sector is the shaded region of the following diagram.





The Q matrix for this sector is given by:

$$\begin{aligned} a &= 0.1; \\ b &= 10; \\ Q &= [1 \quad -(a+b)/2 \quad ; \quad -(a+b)/2 \quad a*b]; \end{aligned}$$

A trajectory  $y(t) = Gu(t)$  lies within the sector  $S$  when for all  $T > 0$ ,

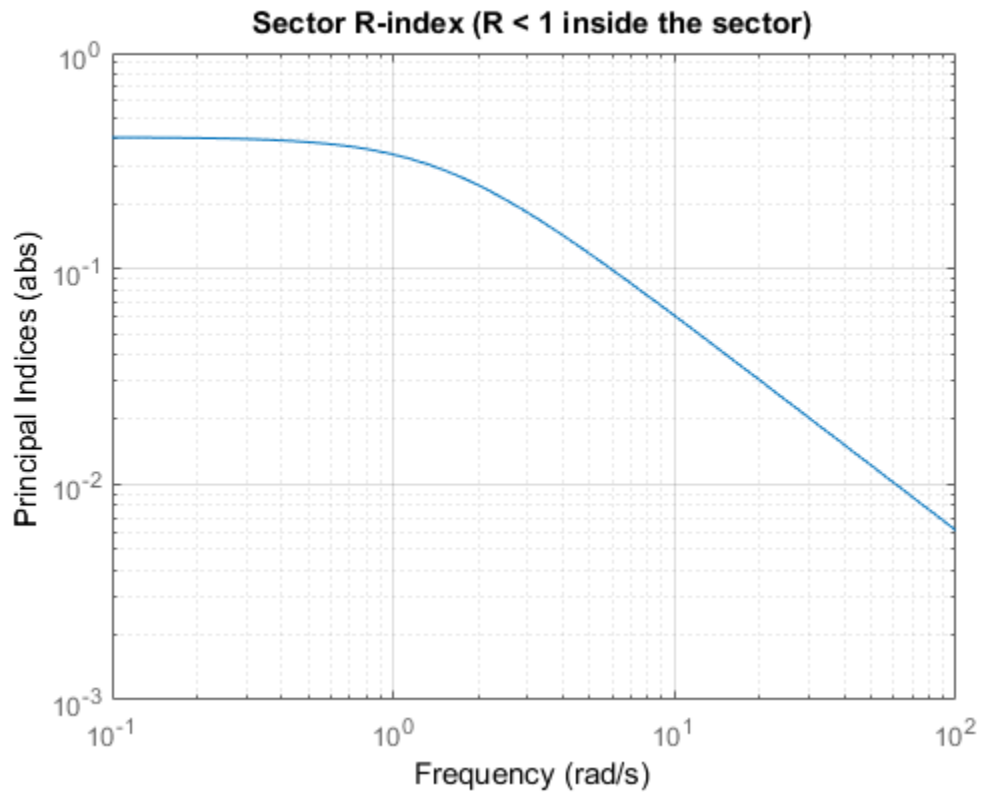
$$0.1 \int_0^T u(t)^2 dt < \int_0^T u(t)y(t) dt < 10 \int_0^T u(t)^2 dt.$$

In the frequency domain, this same condition can be expressed as:

$$\begin{pmatrix} G(j\omega) \\ 1 \end{pmatrix}^H Q \begin{pmatrix} G(j\omega) \\ 1 \end{pmatrix} < 0.$$

To check whether G satisfies or violates this condition at any frequency, plot the sector index for H = [G; 1].

```
G = tf([1 2],[1 1]);
sectorplot([G;1],Q)
```

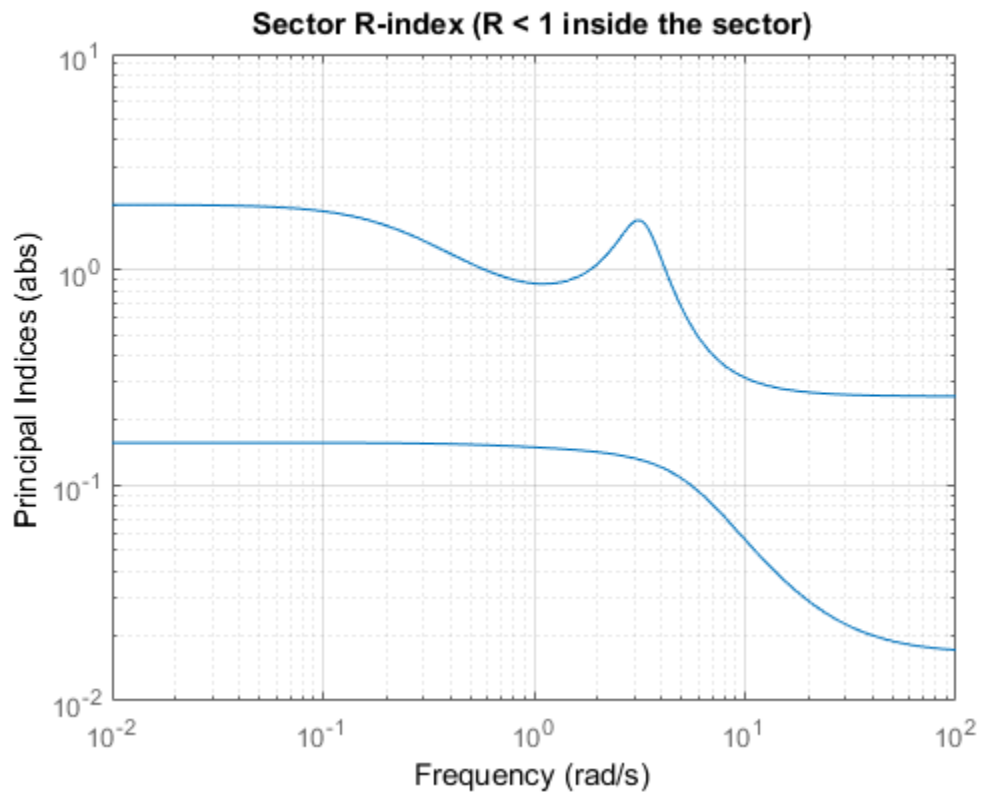


The plot shows that the sector index is less than 1 at all frequencies. Therefore, the trajectories of  $G(s)$  fit within in the specified sector  $Q$  at all frequencies.

### Sector Plot with MIMO System

Examine the sector plot of a 2-output, 2-input system for a particular sector.

```
rng(4);
H = rss(3,4,2);
Q = [-5.12  2.16  -2.04  2.17
      2.16  -1.22  -0.28  -1.11
      -2.04  -0.28  -3.35  0.00
      2.17  -1.11   0.00  0.18];
sectorplot(H,Q)
```



Because  $H$  is 2-by-2, there are two lines on the sector plot. The largest value of the sector index exceeds 1 below about 0.5 rad/s and in a narrow band around 3 rad/s. Therefore,  $H$  does not satisfy the sector bound represented by  $Q$ .

## Input Arguments

### **H** — Model to analyze

dynamic system model | model array

Model to analyze against sector bounds, specified as a dynamic system model such as a `tf`, `ss`, or `genss` model.  $H$  can be continuous or discrete. If  $H$  is a generalized model with tunable or uncertain blocks, `sectorplot` analyzes the current, nominal value of  $H$ .

To analyze whether all I/O trajectories  $(u(t), y(t))$  of a linear system  $G$  lie in a particular sector, use  $H = [G; I]$ , where  $I = \text{eyes}(nu)$ , and  $nu$  is the number of inputs of  $G$ .

If  $H$  is a model array, then `sectorplot` plots the sector index of all models in the array on the same plot. When you use output arguments to get sector-index data,  $H$  must be a single model.

### **Q** — Sector geometry

matrix | LTI model

Sector geometry, specified as:

- A matrix, for constant sector geometry.  $Q$  is a symmetric square matrix that is  $ny$  on a side, where  $ny$  is the number of outputs of  $H$ .
- An LTI model, for frequency-dependent sector geometry.  $Q$  satisfies  $Q(s)' = Q(-s)$ . In other words,  $Q(s)$  evaluates to a Hermitian matrix at each frequency.

The matrix  $Q$  must be indefinite to describe a well-defined conic sector. An indefinite matrix has both positive and negative eigenvalues.

For more information, see “About Sector Bounds and Sector Indices”.

### **w** — Frequencies

{`wmin`, `wmax`} | vector

Frequencies at which to compute and plot indices, specified as the cell array `{wmin, wmax}` or as a vector of frequency values.

- If  $w$  is a cell array of the form  $\{w_{min}, w_{max}\}$ , then the function computes the index at frequencies ranging between  $w_{min}$  and  $w_{max}$ .
- If  $w$  is a vector of frequencies, then the function computes the index at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically-spaced frequency values.

Specify frequencies in units of  $\text{rad}/\text{TimeUnit}$ , where `TimeUnit` is the `TimeUnit` property of the model.

### **PlotStyle** — Line style, marker, and color

character vector

Line style, marker, and color of both the line and marker, specified as a vector of one, two, or three characters. The characters can appear in any order. For more information about configuring the `PlotStyle` argument, see “Specify Line Style, Color, and Markers” in the MATLAB documentation.

Example: `'r--', '*b', 'y'`

## Output Arguments

### **index** — Sector indices

matrix

Sector indices as a function of frequency, returned as a matrix. `index` contains the sector indices computed at the frequencies  $w$  if you supplied them, or `wout` if you did not. `index` has as many columns as there are values in  $w$  or `wout`, and as many rows as  $H$  has inputs. Thus the value `index(:,k)` gives the sector indices in descending order at the frequency  $w(k)$ .

For example, suppose that  $G$  is a 3-input, 3-output system,  $Q$  is a suitable sector matrix, and  $w$  is a 1-by-30 vector of frequencies, then the following syntax returns a 3-by-30 matrix `index`.

```
H = [G;eyes(3)]
index = sectorplot(H,Q,w);
```

The entry `index(:,k)` contains the three sector indices for  $H$ , in descending order, at the frequency  $w(k)$ .

For more information, see “About Sector Bounds and Sector Indices”.

### **wout — Frequencies**

vector

Frequencies at which the indices are calculated, returned as a vector. The function automatically chooses the frequency range and number of points based on the dynamics of the model.

### **More About**

- “About Sector Bounds and Sector Indices”

**Introduced in R2016a**

## series

Series connection of two models

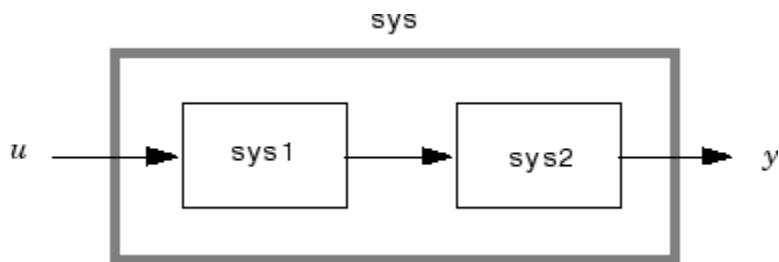
### Syntax

```
series  
sys = series(sys1,sys2)  
sys = series(sys1,sys2,outputs1,inputs2)
```

### Description

`series` connects two model objects in series. This function accepts any type of 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.

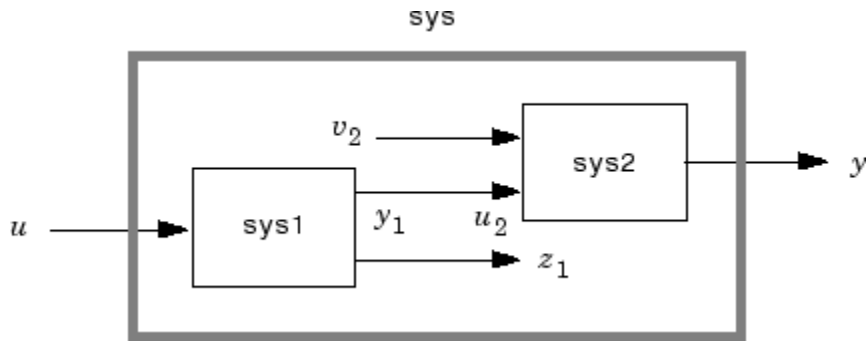
`sys = series(sys1,sys2)` forms the basic series connection shown below.



This command is equivalent to the direct multiplication

```
sys = sys2 * sys1
```

`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.

## Examples

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` | `feedback` | `parallel`

Introduced before R2006a



## set

Set or modify model properties

### Syntax

```
set(sys, 'Property', Value)
set(sys, 'Property1', Value1, 'Property2', Value2, ...)
sysnew = set( ___ )
set(sys, 'Property')
```

### Description

`set` is used to set or modify the properties of a dynamic system model using property name/property value pairs.

`set(sys, 'Property', Value)` assigns the value `Value` to the property of the model `sys`. `'Property'` 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. For a complete list of available system properties for any linear model type, see the reference page for that model type. This syntax is equivalent to `sys.Property = Value`.

`set(sys, 'Property1', Value1, 'Property2', Value2, ...)` sets multiple property values with a single statement. Each property name/property value pair updates one particular property.

`sysnew = set( ___ )` returns the modified dynamic system model, and can be used with any of the previous syntaxes.

`set(sys, 'Property')` displays help for the property specified by `'Property'`.

### Examples

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: []
  StateName: {' '}
InternalDelay: [0x1 double]
      Ts: 0
  InputDelay: 0.1
OutputDelay: 0
  InputName: {'torque'}
OutputName: {' '}
  InputGroup: [1x1 struct]
OutputGroup: [1x1 struct]
      Name: ''
      Notes: {}
  UserData: -2
```

## More About

### Tips

For discrete-time transfer functions, the convention used to represent the numerator and denominator depends on the choice of variable (see `tf` 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' 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.

---

- “What Are Model Objects?”

## See Also

get | frd | ss | tf | zpk

**Introduced before R2006a**

## setDelayModel

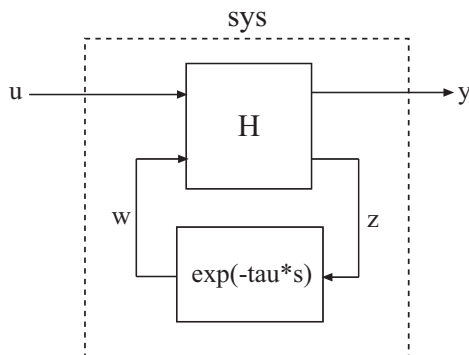
Construct state-space model with internal delays

### Syntax

```
sys = setDelayModel(H,tau)
sys = setDelayModel(A,B1,B2,C1,C2,D11,D12,D21,D22,tau)
```

### Description

`sys = setDelayModel(H,tau)` constructs the state-space model `sys` obtained by LFT interconnection of the state-space model `H` with the vector of internal delays `tau`, as shown:



`sys = setDelayModel(A,B1,B2,C1,C2,D11,D12,D21,D22,tau)` constructs the state-space model `sys` described by the following equations:

$$\begin{aligned}\frac{dx(t)}{dt} &= Ax(t) + B_1u(t) + B_2w(t) \\ y(t) &= C_1x(t) + D_{11}u(t) + D_{12}w(t) \\ z(t) &= C_2x(t) + D_{21}u(t) + D_{22}w(t) \\ w(t) &= z(t - \tau).\end{aligned}$$

$\tau$  ( $\tau$ ) is the vector of internal delays in `sys`.

## Input Arguments

### **H**

State-space (ss) model to interconnect with internal delays `tau`.

### **tau**

Vector of internal delays of `sys`.

For continuous-time models, express `tau` in seconds.

For discrete-time models, express `tau` as integer values that represent multiples of the sample time.

### **A, B1, B2, C1, C2, D11, D12, D21, D22**

Set of state-space matrices that, with the internal delay vector `tau`, explicitly describe the state-space model `sys`.

## Output Arguments

### **sys**

State-space (ss) model with internal delays `tau`.

## More About

### Tips

- `setDelayModel` is an advanced operation and is not the natural way to construct models with internal delays. See “Time Delays in Linear Systems” for recommended ways of creating internal delays.
- The syntax `sys = setDelayModel(A, B1, B2, C1, C2, D11, D12, D21, D22, tau)` constructs a continuous-time model. You can construct the discrete-time model described by the state-space equations

$$\begin{aligned}x[k+1] &= Ax[k] + B_1u[k] + B_2w[k] \\y[k] &= C_1x[k] + D_{11}u[k] + D_{12}w[k] \\z[k] &= C_2x[k] + D_{21}u[k] + D_{22}w[k] \\w[k] &= z[k - \tau].\end{aligned}$$

To do so, first construct `sys` using `sys = setDelayModel(A,B1,B2,C1,C2,D11,D12,D21,D22,tau)`. Then, use `sys.Ts` to set the sample time.

- “Internal Delays”
- “Time Delays in Linear Systems”

### See Also

`getDelayModel` | `lft` | `ss`

**Introduced in R2007a**

## setoptions

Set plot options for response plot

### Syntax

```
setoptions(h, PlotOpts)
setoptions(h, 'Property1', 'value1', ...)
setoptions(h, PlotOpts, 'Property1', 'value1', ...)
```

### Description

`setoptions(h, PlotOpts)` sets preferences for response plot using the plot handle. `h` is the plot handle, `PlotOpts` is a plot options handle containing information about plot options.

There are two ways to create a plot options handle:

- Use `getoptions`, which accepts a plot handle and returns a plot options handle.

```
p = getoptions(h)
```

- Create a default plot options handle using one of the following commands:

- `bodeoptions` — Bode plots
- `hsvoptions` — Hankel singular values plots
- `nicholsoptions` — Nichols plots
- `nyquistoptions` — Nyquist plots
- `pzoptions` — Pole/zero plots
- `sigmaoptions` — Sigma plots
- `timeoptions` — Time plots (step, initial, impulse, etc.)

For example,

```
p = bodeoptions
```

returns a plot options handle for Bode plots.

`setoptions(h, 'Property1', 'value1', ...)` assigns values to property pairs instead of using `PlotOpts`. To find out what properties and values are available for a particular plot, type `help <function>options`. For example, for Bode plots type

```
help bodeoptions
```

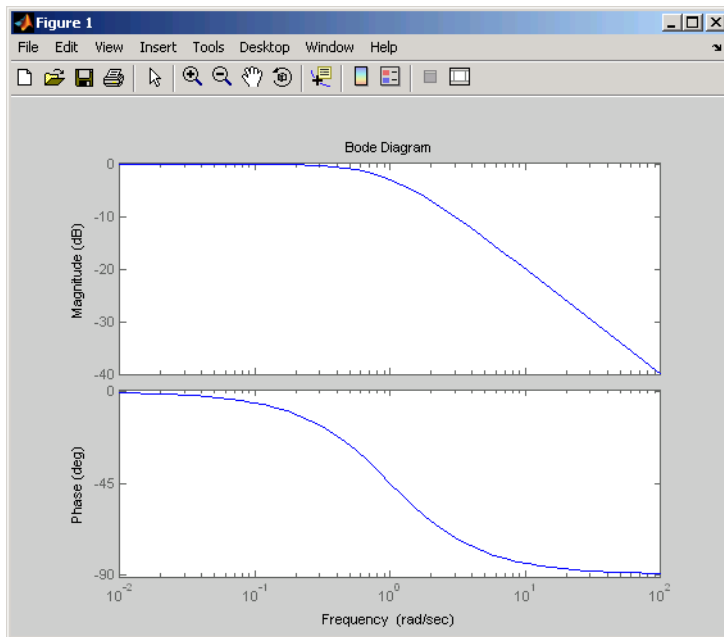
For a list of the properties and values available for each plot type, see “Properties and Values Reference”.

`setoptions(h, PlotOpts, 'Property1', 'value1', ...)` first assigns plot properties as defined in `@PlotOptions`, and then overrides any properties governed by the specified property/value pairs.

## Examples

To change frequency units, first create a Bode plot.

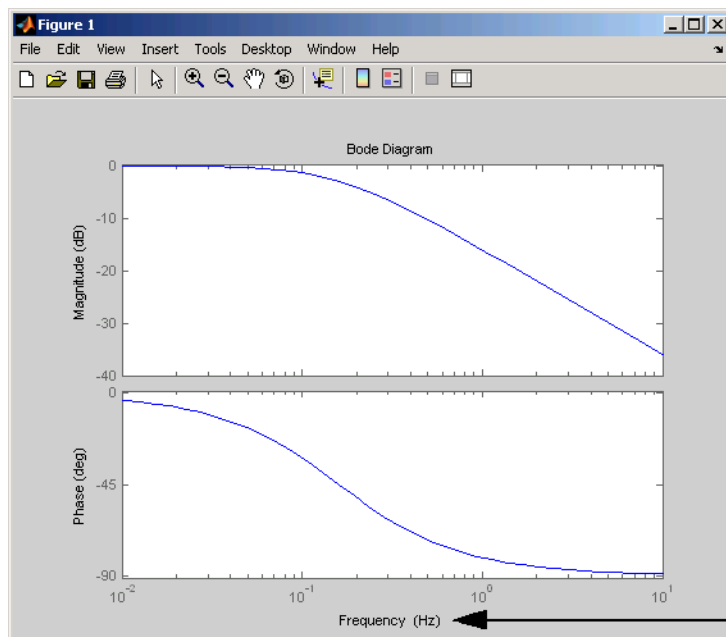
```
sys=tf(1,[1 1]);  
h=bodeplot(sys) % Create a Bode plot with plot handle h.
```





Now, change the frequency units from rad/s to Hz.

```
p=getoptions(h); % Create a plot options handle p.
p.FreqUnits = 'Hz'; % Modify frequency units.
setoptions(h,p); % Apply plot options to the Bode plot and
                % render.
```



The frequency units  
are now Hz.

To change the frequency units using property/value pairs, use this code.

```
sys=tf(1,[1 1]);
h=bodeplot(sys);
setoptions(h,'FreqUnits','Hz');
```

The result is the same as the first example.

## See Also

getoptions

Introduced before R2006a

# setBlockValue

Modify value of Control Design Block in Generalized Model

## Syntax

```
M = setBlockValue(M0,blockname,val)
M = setBlockValue(M0,blockvalues)
M = setBlockValue(M0,Mref)
```

## Description

`M = setBlockValue(M0,blockname,val)` modifies the current or nominal value of the Control Design Block `blockname` in the Generalized Model `M0` to the value specified by `val`.

`M = setBlockValue(M0,blockvalues)` modifies the value of several Control Design Blocks at once. The structure `blockvalues` specifies the blocks and replacement values. Blocks of `M0` not listed in `blockvalues` are unchanged.

`M = setBlockValue(M0,Mref)` takes replacement values from Control Design blocks in the Generalized Model `Mref`. This syntax modifies the Control Design Blocks in `M0` to match the current values of all corresponding blocks in `Mref`.

Use this syntax to propagate block values, such as tuned parameter values, from one parametric model to other models that depend on the same parameters.

## Input Arguments

### **M0**

Generalized Model containing the blocks whose current or nominal value is modified to `val`. For the syntax `M = setBlockValue(M0,Mref)` `M0` can be a single Control Design Block whose value is modified to match the value of the corresponding block in `Mref`.

**blockname**

Name of the Control Design Block in the model `M0` whose current or nominal value is modified.

To get a list of the Control Design Blocks in `M0`, enter `M0.Blocks`.

**val**

Replacement value for the current or nominal value of the Control Design Block, `blockname`. The value `val` can be any value that is compatible with `blockname` without changing the size, type, or sample time of `blockname`.

For example, you can set the value of a tunable PID block (`tunablePID`) to a `pid` controller model, or to a transfer function (`tf`) model that represents a PID controller.

**blockvalues**

Structure specifying Control Design Blocks of `M0` to modify, and the corresponding replacement values. The fields of the structure are the names of the blocks to modify. The value of each field specifies the replacement current or nominal value for the corresponding block.

**Mref**

Generalized Model that shares some Control Design Blocks with `M0`. The values of these blocks in `Mref` are used to update their counterparts in `M0`.

## Output Arguments

**M**

Generalized Model obtained from `M0` by updating the values of the specified blocks.

## Examples

### Update Controller Model with Tuned Values

Propagate the values of tuned parameters to other Control Design Blocks.

You can use tuning commands such as `systemtune`, `looptune`, or the Robust Control Toolbox™ command `hinfstruct` to tune blocks in a closed-loop model of a control system. If you do so, the tuned controller parameters are embedded in a generalized model. You can use `setBlockValue` to propagate those parameters to a controller model.

Create a tunable model of the closed-loop response of a control system, and tune the parameters using `systemtune`.

```
s = tf('s');
num = 33000*(s^2 - 200*s + 90000);
den = (s + 12.5)*(s^2 + 25*s + 63000);
G = num/den;

C0 = tunablePID('C0','pi');
a = realp('a',1);
F0 = tf(a,[1 a]);
T0 = feedback(G*C0,F0);
T0.InputName = 'r';
T0.OutputName = 'y';
```

T0 is a generalized model of the closed-loop control system and contains two tunable blocks:

- C0 - Tunable PID controller
- a - Real tunable parameter

Create a tuning requirement for the output `y` to track the input `r`, and tune the system to meet that requirement.

```
Req = TuningGoal.Tracking('r','y',0.05);
[T,fSoft,~] = systemtune(T0,Req);
```

```
Final: Soft = 1.43, Hard = -Inf, Iterations = 58
```

The generalized model T contains the tuned values of C0 and a.

Propagate the tuned values of the controller in T to the controller model C0.

```
C = setBlockValue(C0,T)
```

```
C =
```

Parametric continuous-time PID controller "C0" with formula:

$$K_p + K_i * \frac{1}{s}$$

and tunable parameters  $K_p$ ,  $K_i$ .

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

**C** is still a **tunablePID** controller. The current PID gains in **C** are set to the values of the controller in **T**.

Obtain a numeric LTI model of the tuned controller using **getValue**.

```
CVal = getValue(C,T);
```

This command returns a numerical state-space model of the tuned controller.

## See Also

genss | getBlockValue | getValue | hinfstruct | looptune | showBlockValue  
| systune

**Introduced in R2011b**

## setData

Set values of tunable-surface coefficients

### Syntax

```
Knew = setData(K,Kco)
Knew = getData(K,J,KcoJ)
```

### Description

`Knew = setData(K,Kco)` sets the current values of the tunable surface `K`. `K` is a `tunableSurface` object that represents the parametric gain surface:

$$K(n(\sigma)) = K_0 + K_1 F_1(n(\sigma)) + \dots + K_M F_M(n(\sigma)).$$

$F_1, \dots, F_M$  are basis functions, and  $n(\sigma)$  is a normalization function that maps the range of each scheduling-variable  $\sigma$  onto  $[-1,1]$ . `Kco` is an array of new values for  $[K_0, \dots, K_M]$ .

`Knew = getData(K,J,KcoJ)` sets the current value of the coefficient of the  $J$ th basis function  $F_J$  to `KcoJ`. Use `J = 0` to set the constant coefficient  $K_0$ .

### Input Arguments

#### **K** — Gain surface

`tunableSurface` object

Gain surface, specified as a `tunableSurface` object,

#### **Kco** — New coefficient values

array

New coefficient values of the tunable surface, specified as an array.

If the tunable surface `K` is a scalar-valued gain, then the length of `K` is  $(M+1)$ , where  $M$  is the number of basis functions in the parameterization. For example, if `K` represents the tunable gain surface:

$$K(\alpha, V) = K_0 + K_1\alpha + K_2V + K_3\alpha V,$$

then `KCO` is the 1-by-4 vector  $[K_0, K_1, K_2, K_3]$ .

For array-valued gains, each coefficient expands to the I/O dimensions of the gain. These expanded coefficients are concatenated horizontally in `KCO`. (See `tunableSurface`.) For example, for a two-input, two-output gain surface, `KCO` has dimensions  $[2, 2(M+1)]$ . See `evalSurf` for an example that uses `setData` on an array-valued gain.

### **J** — Index of basis function

nonnegative integer

Index of basis function, specified as a nonnegative integer. To set the constant coefficient  $K_0$ , use `J = 0`.

### **KCOJ** — Coefficient of *J*th basis function

scalar | array

Coefficient of the *J*th basis function in the tunable surface parameterization, specified as a scalar or an array.

If the tunable surface `K` is a scalar-valued gain, then `KCOJ` is a scalar. If `K` is an array-valued gain, then `KCOJ` is an array that matches the I/O dimensions of the gain.

## Output Arguments

### **Knew** — Gain surface

`tunableSurface` object

Gain surface with new coefficient values, returned as a `tunableSurface` object.

## See Also

`evalSurf` | `getData` | `tunableSurface` | `viewSurf`

Introduced in R2015b

# setValue

Modify current value of Control Design Block

## Syntax

```
blk = setValue(blk0, val)
```

## Description

`blk = setValue(blk0, val)` modifies the parameter values in the tunable Control Design Block, `blk0`, to best match the values specified by `val`. An exact match can only occur when `val` is compatible with the structure of `blk0`.

## Input Arguments

### **blk0**

Control Design Block whose value is modified.

### **val**

Specifies the replacement parameters values for `blk0`. The value `val` can be any value that is compatible with `blk0` without changing the size, type, or sample time of `blk0`. For example, if `blk0` is a `tunablePID` block, valid types for `val` include `tunablePID`, a numeric `pid` controller model, or a numeric `tf` model that represents a PID controller. `setValue` uses the parameter values of `val` to set the current value of `blockname`.

## Output Arguments

### **blk**

Control Design Block of the same type as `blk0`, whose parameters are updated to best match the parameters of `val`.



## **See Also**

getValue | setBlockValue | getBlockValue

**Introduced in R2011b**

## sgrid

Generate 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.

### Examples

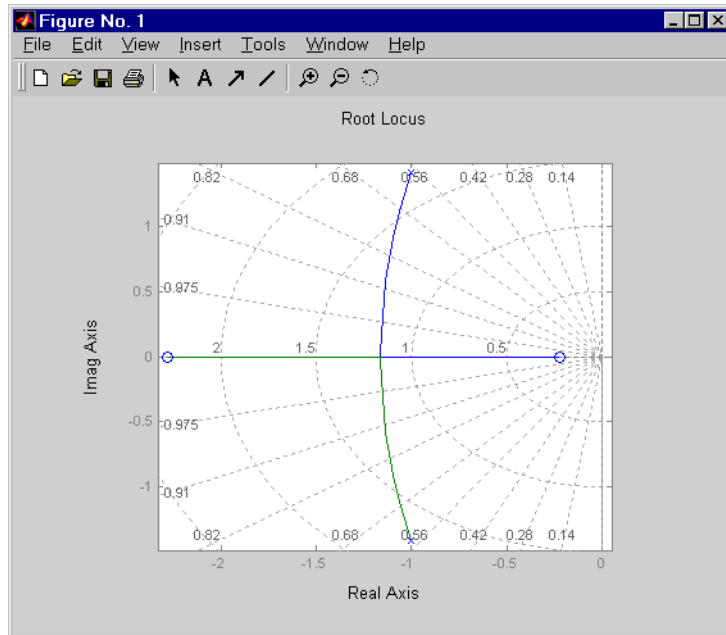
Plot s-plane grid lines on the root locus for the following system.

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

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

[zgrid](#) | [pzmap](#) | [rlocus](#)

Introduced before R2006a

## showBlockValue

Display current value of Control Design Blocks in Generalized Model

### Syntax

```
showBlockValue(M)
```

### Description

`showBlockValue(M)` displays the current values of all Control Design Blocks in the Generalized Model, `M`. (For uncertain blocks, the “current value” is the nominal value of the block.)

### Input Arguments

**M**

Generalized Model.

### Examples

Create a tunable `genss` model, and display the current value of its tunable elements.

```
G = zpk([], [-1, -1], 1);  
C = tunablePID('C', 'PID');  
a = realp('a', 10);  
F = tf(a, [1 a]);  
T = feedback(G*C, 1)*F;
```

```
showBlockValue(T)
```

```
C =  
Continuous-time I-only controller:
```

1

$$K_i * \frac{---}{s}$$

With  $K_i = 0.001$

-----  
 $a = 10$

## More About

### Tips

- Displaying the current values of a model is useful, for example, after you have tuned the free parameters of the model using a tuning command such as `systemtune`.
- `showBlockValue` displays the current values of all Control Design Blocks in a model, including tunable, uncertain, and switch blocks. To display the current values of only the tunable blocks, use `showTunable`.

### See Also

`genss` | `getBlockValue` | `setBlockValue` | `showTunable`

**Introduced in R2011b**

## showTunable

Display current value of tunable Control Design Blocks in Generalized Model

### Syntax

`showTunable(M)`

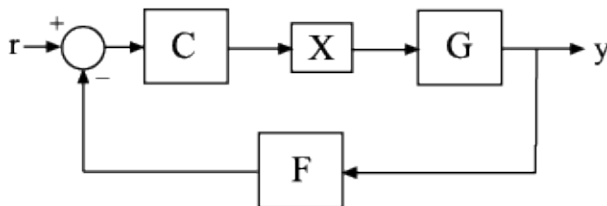
### Description

`showTunable(M)` displays the current values of all tunable Control Design Blocks in a generalized LTI model. Tunable control design blocks are parametric blocks such as `realp`, `tunableTF`, and `tunablePID`.

### Examples

#### Display Block Values of Tuned Control System Model

Tune the following control system using `systune`, and display the values of the tunable blocks.



The control structure includes a PI controller **C** and a tunable low-pass filter in the feedback path. The plant **G** is a third-order system.

Create models of the system components and connect them together to create a tunable closed-loop model of the control system.

```

s = tf('s');
num = 33000*(s^2 - 200*s + 90000);
den = (s + 12.5)*(s^2 + 25*s + 63000);
G = num/den;

C0 = tunablePID('C','pi');
a = realp('a',1);
F0 = tf(a,[1 a]);
X = AnalysisPoint('X');

T0 = feedback(G*X*C0,F0);
T0.InputName = 'r';
T0.OutputName = 'y';

```

T0 is a **genss** model that has two tunable blocks, the PI controller, C, and the parameter, a. T0 also contains the switch block X.

Create a tuning requirement that forces the output y to track the input r, and tune the system to meet that requirement.

```

Req = TuningGoal.Tracking('r','y',0.05);
[T,fSoft,~] = systune(T0,Req);

```

```

Final: Soft = 1.43, Hard = -Inf, Iterations = 58

```

**systune** finds values for the tunable parameters that optimally meet the tracking requirement. The output T is a **genss** model with the same Control Design Blocks as T0. The current values of those blocks are the tuned values.

Examine the tuned values of the tunable blocks of the control system.

```

showTunable(T)

```

```

C =

```

$$K_p + K_i * \frac{1}{s}$$

```

with Kp = 0.000433, Ki = 0.00525

```

```

Name: C

```

```

Continuous-time PI controller in parallel form.
-----

```

```

a = 67.9

```

`showTunable` displays the values of the tunable blocks only. If you use `showBlockValue` instead, the display also includes the switch block X.

## Input Arguments

### M — Input model

generalized LTI model

Input model of which to display tunable block values, specified as a generalized LTI model such as a `genss` model.

## More About

### Tips

- Displaying the current values of tunable blocks is useful, for example, after you have tuned the free parameters of the model using a tuning command such as `systemtune`.
- `showTunable` displays the current values of the tunable blocks only. To display the current values of all Control Design Blocks in a model, including tunable, uncertain, and switch blocks, use `showBlockValue`.
- “Generalized Models”
- “Control Design Blocks”

### See Also

`genss` | `getBlockValue` | `setBlockValue` | `showBlockValue` | `systemtune`

Introduced in R2012b



## sigma

Singular values plot of dynamic system

### Syntax

```
sigma(sys)
sigma(sys,w)
sigma(sys,[],type)
sigma(sys,w,type)
sigma(sys1,sys2,...,sysN,w,type)
sigma(sys1,'PlotStyle1',...,sysN,'PlotStyleN',w,type)
sv = sigma(sys,w)
[sv,w] = sigma(sys)
```

### Description

`sigma` calculates the singular values of the frequency response of a dynamic system `sys`. For an FRD model, `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  $\omega$ . 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  $\omega$  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 a 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  $[w_{\min}, w_{\max}]$ , set  $w = \{w_{\min}, w_{\max}\}$ . To use particular frequency points, set  $w$  to the corresponding vector of frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. Frequencies must be in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`.

`sigma(sys,[],type)` or `sigma(sys,w,type)` plots the following modified singular value responses:

<code>type = 1</code>	Singular values of the frequency response $H^{-1}$ , where $H$ is the frequency response of <code>sys</code> .
<code>type = 2</code>	Singular values of the frequency response $I + H$ .
<code>type = 3</code>	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.

`sigma(sys1,sys2,...,sysN,w,type)` plots the singular value plots of several LTI models on a single figure. The arguments  $w$  and `type` are optional. The models `sys1,sys2,...,sysN` need not have the same number of inputs and outputs. Each model can be either continuous- or discrete-time.

`sigma(sys1,'PlotStyle1',...,sysN,'PlotStyleN',w,type)` specifies a distinctive color, linestyle, and/or marker for each system plot. See `bode` for an example.

`sv = sigma(sys,w)` and `[sv,w] = sigma(sys)` return the singular values `sv` of the frequency response at the frequencies  $w$ . For a system with  $N_u$  input and  $N_y$  outputs, the array `sv` has  $\min(N_u, N_y)$  rows and as many columns as frequency points (length of  $w$ ). The singular values at the frequency  $w(k)$  are given by `sv(:,k)`.

## Examples

### Compute and Plot Singular Values

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

$$H(s) = \begin{bmatrix} 0 & \frac{3s}{s^2 + s + 10} \\ \frac{s+1}{s+5} & \frac{2}{s+6} \end{bmatrix}.$$

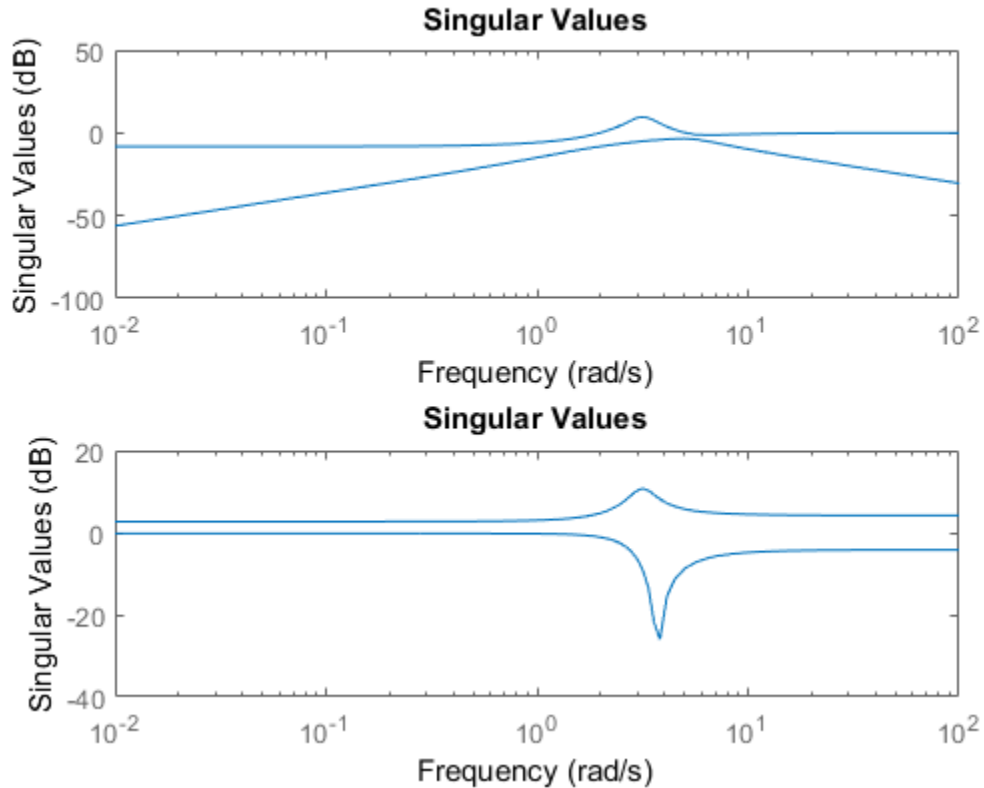
Compute the singular value responses of  $H(s)$  and  $I + H(s)$ .

```
H = [0, tf([3 0],[1 1 10]) ; tf([1 1],[1 5]), tf(2,[1 6])];  
[svH,wH] = sigma(H);  
[svIH,wIH] = sigma(H,[],2);
```

In the last command, the input 2 selects the second response type,  $I + H(s)$ . The vectors svH and svIH contain the singular value response data, at the frequencies in wH and wIH.

Plot the singular value responses of both systems.

```
subplot(211)  
sigma(H)  
subplot(212)  
sigma(H,[],2)
```



## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### Algorithms

`sigma` uses the MATLAB function `svd` to compute the singular values of a complex matrix.

For TF, ZPK, and SS models, `sigma` computes the frequency response using the `freqresp` algorithms. As a result, small discrepancies may exist between the `sigma` responses for equivalent TF, ZPK, and SS representations of a given model.

**See Also**

`bode` | `evalfr` | `freqresp` | Linear System Analyzer | `nichols` | `nyquist`

## sigmaoptions

Create list of singular-value plot options

### Syntax

```
P = sigmaoptions
P = sigmaoptions('cstprefs')
```

### Description

`P = sigmaoptions` returns a list of available options for singular value plots with default values set. You can use these options to customize the singular value plot appearance from the command line.

`P = sigmaoptions('cstprefs')` initializes the plot options with the options you selected in the Control System Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation.

This table summarizes the sigma plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid Specified as one of the following: 'off'   'on' <b>Default:</b> 'off'
GridColor	Color of the grid lines Specified as one of the following: Vector of RGB values in the range [0,1]   color   'none'. <b>Default:</b> [0.15,0.15,0.15]
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits

---

<b>Option</b>	<b>Description</b>
IOGrouping	Grouping of input-output pairs Specified as one of the following: 'none'   'inputs'   'outputs'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	<p>Frequency units, specified as one of the following:</p> <ul style="list-style-type: none"> <li>• 'Hz'</li> <li>• 'rad/second'</li> <li>• 'rpm'</li> <li>• 'kHz'</li> <li>• 'MHz'</li> <li>• 'GHz'</li> <li>• 'rad/nanosecond'</li> <li>• 'rad/microsecond'</li> <li>• 'rad/millisecond'</li> <li>• 'rad/minute'</li> <li>• 'rad/hour'</li> <li>• 'rad/day'</li> <li>• 'rad/week'</li> <li>• 'rad/month'</li> <li>• 'rad/year'</li> <li>• 'cycles/nanosecond'</li> <li>• 'cycles/microsecond'</li> <li>• 'cycles/millisecond'</li> <li>• 'cycles/hour'</li> <li>• 'cycles/day'</li> <li>• 'cycles/week'</li> <li>• 'cycles/month'</li> <li>• 'cycles/year'</li> </ul> <p><b>Default:</b> 'rad/s'</p> <p>You can also specify 'auto' which uses frequency units rad/TimeUnit relative</p>



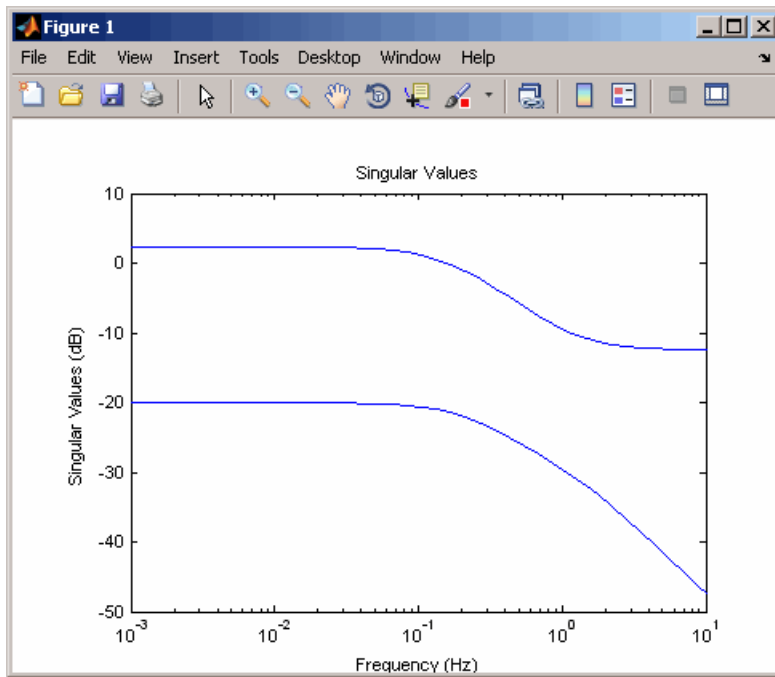
Option	Description
	to system time units specified in the <code>TimeUnit</code> property. For multiple systems with different time units, the units of the first system are used.
<code>FreqScale</code>	Frequency scale Specified as one of the following: 'linear'   'log' <b>Default:</b> 'log'
<code>MagUnits</code>	Magnitude units Specified as one of the following: 'dB'   'abs' <b>Default:</b> 'dB'
<code>MagScale</code>	Magnitude scale Specified as one of the following: 'linear'   'log' <b>Default:</b> 'linear'

## Examples

In this example, set the frequency units to Hz before creating a plot.

```
P = sigmaoptions; % Set the frequency units to Hz in options
P.FreqUnits = 'Hz'; % Create plot with the options specified by P
h = sigmaplot(rss(2,2,3),P);
```

The following singular value plot is created with the frequency units in Hz.



### See Also

`getoptions` | `setoptions` | `sigmaplot`

Introduced in R2008a

# sigmaplot

Plot singular values of frequency response and return plot handle

## Syntax

```
h = sigmaplot(sys)
sigmaplot(sys,{wmin,wmax})
sigmaplot(sys,w)
sigmaplot(sys,w,TYPE)
sigmaplot(AX,...)
sigmaplot(..., plotoptions)
```

## Description

`h = sigmaplot(sys)` produces a singular value (SV) plot of the frequency response of the dynamic system `sys`. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help sigmaoptions
```

for a list of available plot options.

The frequency range and number of points are chosen automatically. See `bode` for details on the notion of frequency in discrete time.

`sigmaplot(sys,{wmin,wmax})` draws the SV plot for frequencies ranging between `wmin` and `wmax` (in `rad/TimeUnit`, where `TimeUnit` is the time units of the input dynamic system, specified in the `TimeUnit` property of `sys`).

`sigmaplot(sys,w)` uses the user-supplied vector `w` of frequencies, in `rad/TimeUnit`, at which the frequency response is to be evaluated. See `logspace` to generate logarithmically spaced frequency vectors.

`sigmaplot(sys,w,TYPE)` or `sigmaplot(sys,[],TYPE)` draws the following modified SV plots depending on the value of `TYPE`:

TYPE = 1	-->	SV of inv(SYS)
TYPE = 2	-->	SV of I + SYS
TYPE = 3	-->	SV of I + inv(SYS)

sys should be a square system when using this syntax.

sigmaplot(Ax, ...) plots into the axes with handle AX.

sigmaplot(..., plotoptions) plots the singular values with the options specified in plotoptions. Type

help sigmaoptions

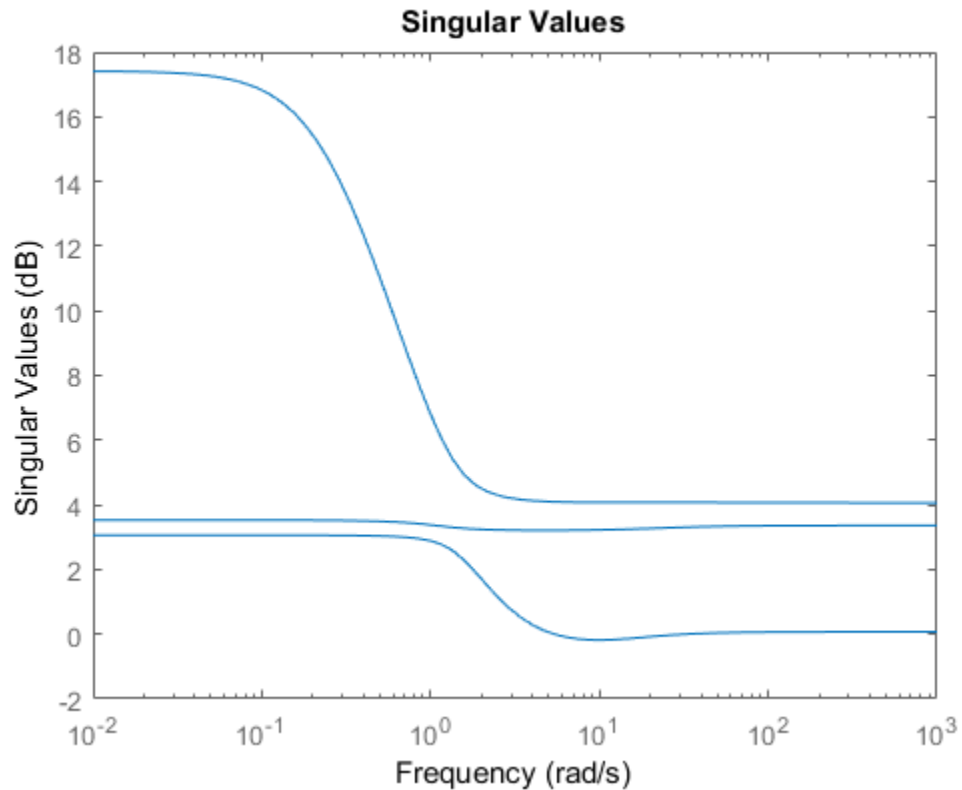
for more details.

## Examples

### Singular Value Response Plot with Custom Plot Options

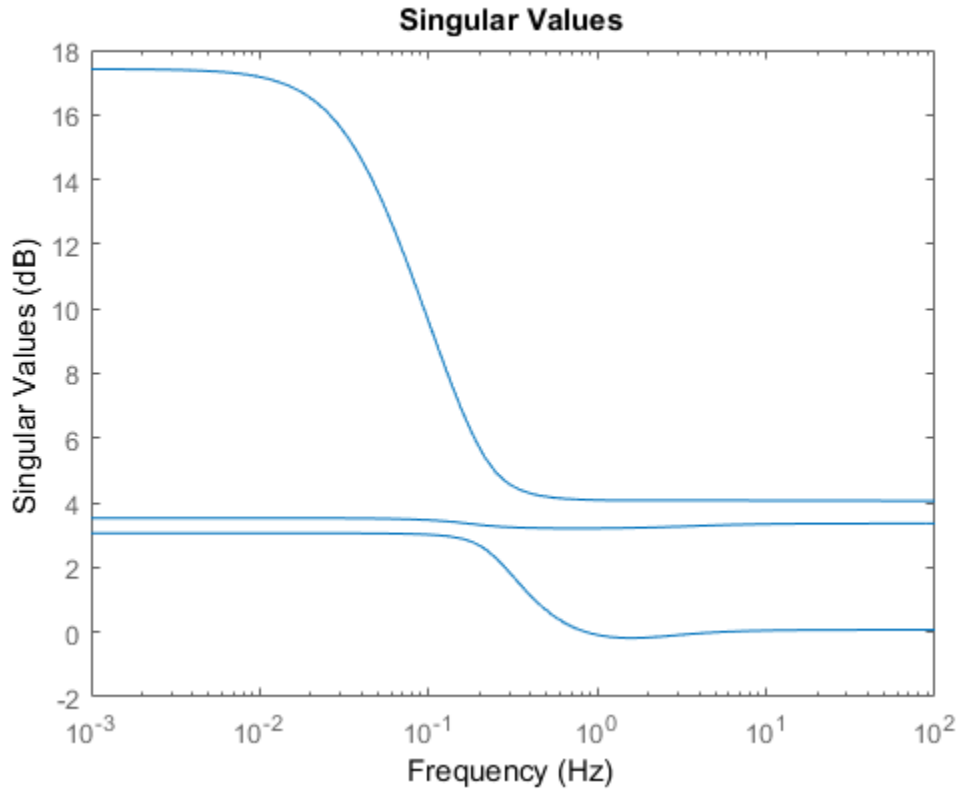
Plot the singular value responses of a dynamic system.

```
sys = rss(3,3,5);  
h = sigmaplot(sys);
```



Set properties of the plot handle `h` to customize the plot. For example, change the plot units to Hz.

```
setoptions(h, 'FreqUnits', 'Hz');
```



## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`getoptions` | `sigma` | `setoptions` | `sigmaoptions`

Introduced before R2006a

# sisoinit

Configure Control System Designer at startup

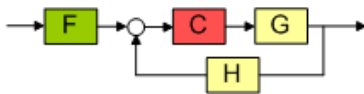
## Syntax

```
init_config = sisoinit(config)
```

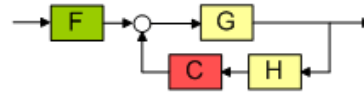
## Description

`init_config = sisoinit(config)` returns a template `init_config` for initializing the **Control System Designer** with one of the following control system configurations:

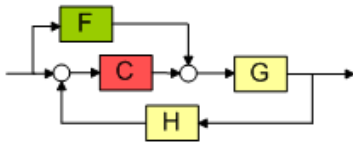
Configuration 1



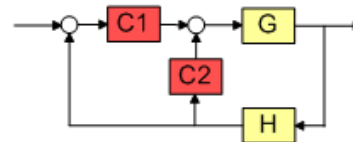
Configuration 2



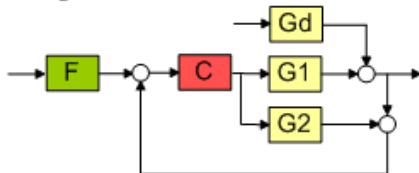
Configuration 3



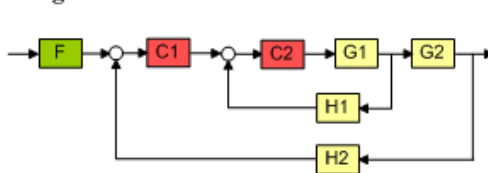
Configuration 4



Configuration 5



Configuration 6



For more information about the control system configurations supported by the **Control System Designer**, see “Feedback Control Architectures”.

For each configuration, you can specify the plant model  $G$  and the sensor dynamics  $H$ , initialize the compensator  $C$  and prefilter  $F$ , and configure the open-loop and closed-loop views by specifying the corresponding fields of the structure `init_config`. Then you can start the **Control System Designer** in the specified configuration using `controlSystemDesigner(init_config)`.

Output argument `init_config` is an object with properties. The following tables list the block and loop properties.

### Block Properties

Block	Properties	Values
F	Name	Character vector
	Description	Character vector
	Value	LTI object
G	Name	Character vector
	Value	<ul style="list-style-type: none"> <li>LTI object</li> <li>Row or column array of LTI objects. If the sensor <math>H</math> is also an array of LTI objects, the lengths of <math>G</math> and <math>H</math> must match.</li> </ul>
H	Name	Character vector
	Value	<ul style="list-style-type: none"> <li>LTI object</li> <li>Row or column array of LTI objects. If the plant <math>G</math> is also an array of LTI objects, the lengths of <math>H</math> and <math>G</math> must match.</li> </ul>
C	Name	Character vector
	Description	Character vector
	Value	LTI object

### Loop Properties

Loops	Properties	Values
OL1	Name	Character vector
	Description	Character vector



Loops	Properties	Values
	View	'rlocus' 'bode'
CL1	Name	Character vector
	Description	Character vector
	View	'bode'

## Examples

### Initialize Control System Designer

Create an initialization template for configuration 2, with the compensator in the feedback path.

```
T = sisoinit(2);
```

Specify the fixed plant model.

```
T.G.Value = tf(1, [1 1]);
```

Specify an initial compensator value.

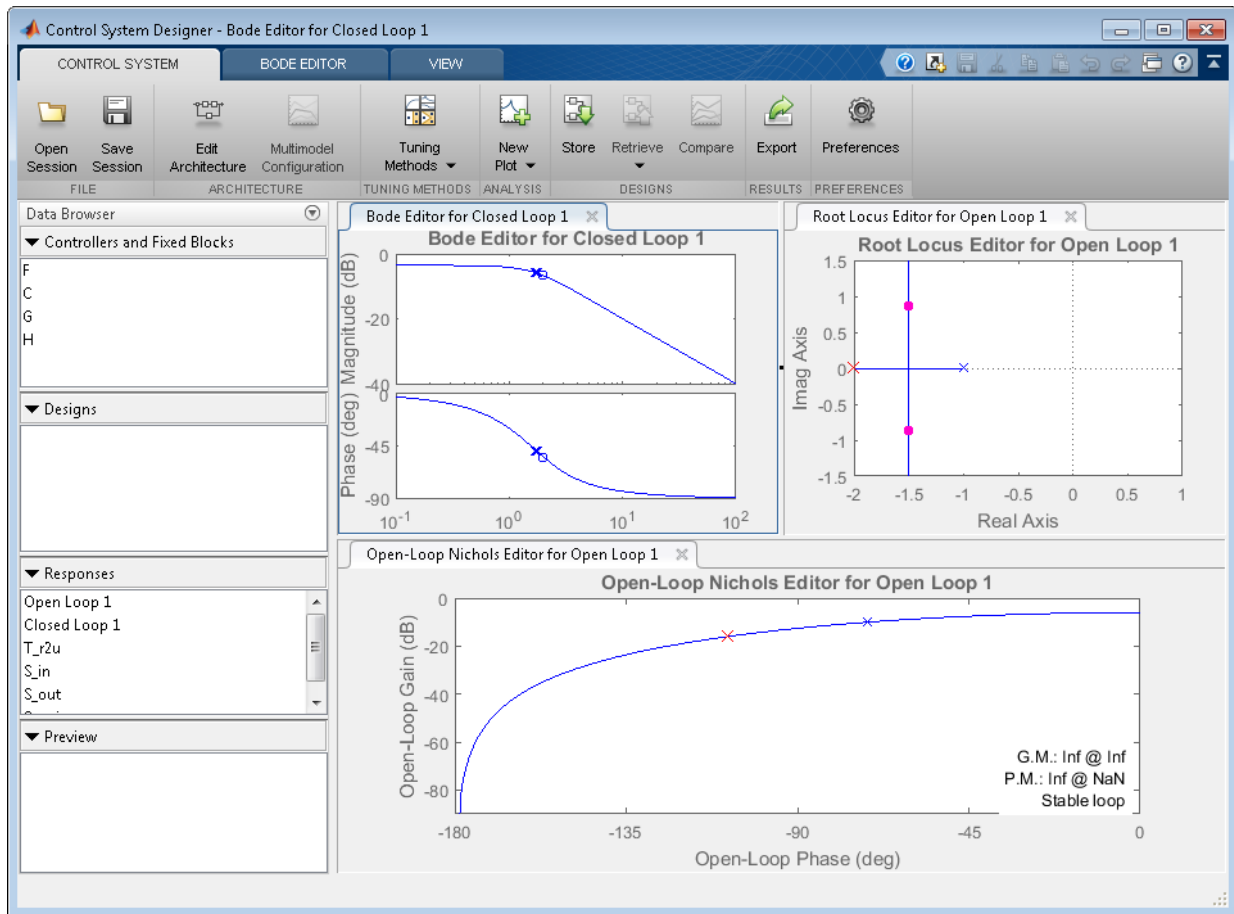
```
T.C.Value = tf(1,[1 2]);
```

Open a root locus Editor and Nichols editor for tuning the open-loop response.

```
T.OL1.View = {'rlocus', 'nichols'};
```

Open Control System Designer using the specified configuration settings.

```
controlSystemDesigner(T)
```



By default, the template for configuration 2 also opens a Bode editor for tuning the closed-loop response.

### Initialize Control System Designer Using Array of Plant Models

Specify a configuration template.

```
initconfig = sisoinit(2);
```

Specify model parameters.

```
m = 3;
```

```
b = 0.5;  
k = 8:1:10;  
T = 0.1:.05:.2;
```

Create an array of LTI objects to model variations in plant G.

```
for ct = 1:length(k);  
    G(:,:,ct) = tf(1,[m,b,k(ct)]);  
end
```

Assign G to the initial configuration.

```
initconfig.G.Value = G;
```

Specify initial compensator value.

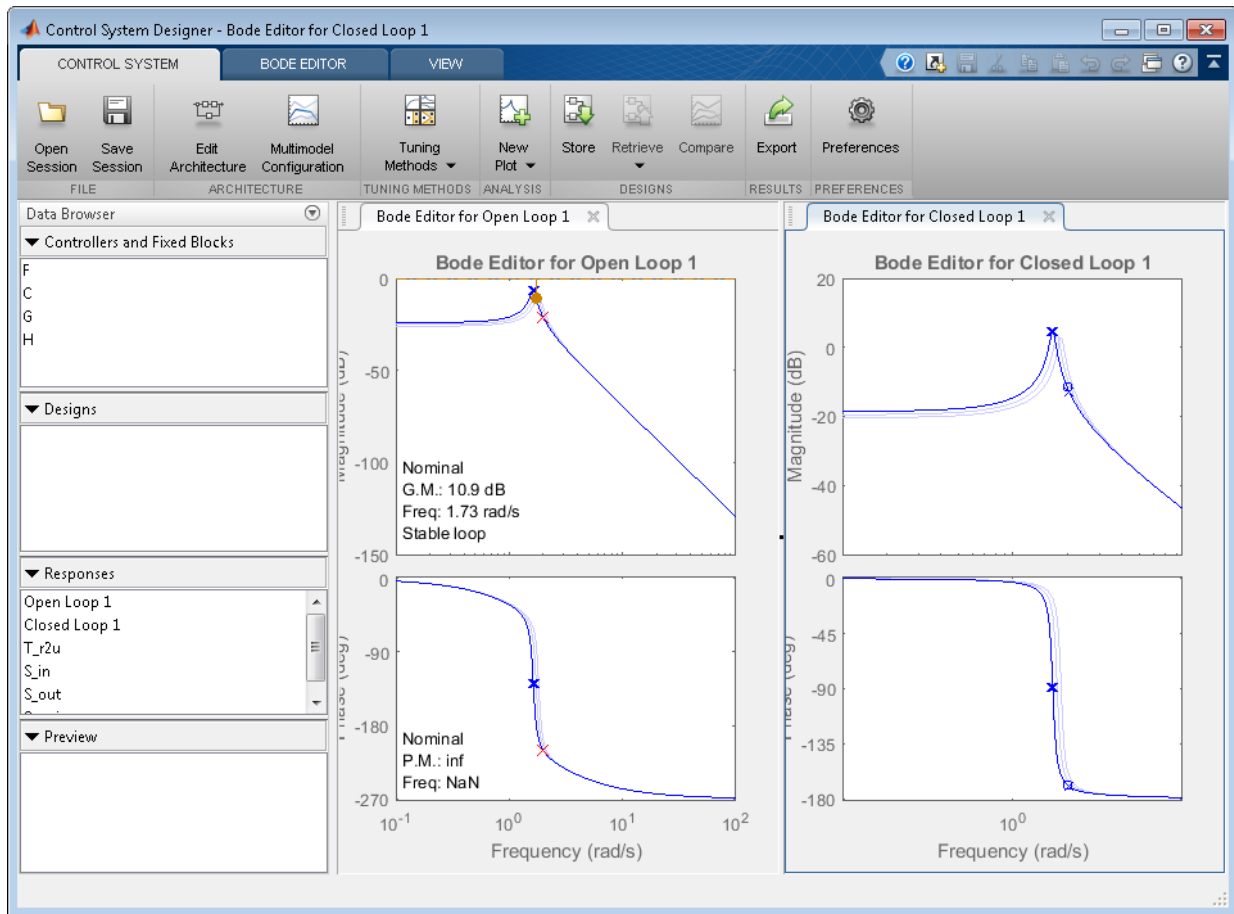
```
initconfig.C.Value = tf(1,[1 2]);
```

Use a graphical Bode editor to tune the open-loop response.

```
initconfig.OL1.View = {'bode'};
```

Open Control System Designer using the specified configuration settings.

```
controlSystemDesigner(initconfig)
```



By default, the template for configuration 2 also opens a Bode editor for tuning the closed-loop response.

- “Feedback Control Architectures”
- “Programmatically Initializing the Control System Designer”
- “Multimodel Control Design”

### See Also

Control System Designer

**Introduced in R2006a**

## size

Query output/input/array dimensions of input–output model and number of frequencies of FRD model

### Syntax

```
size(sys)
d = size(sys)
Ny = size(sys,1)
Nu = size(sys,2)
Sk = size(sys,2+k)
Nf = size(sys, 'frequency')
```

### Description

When invoked without output arguments, `size(sys)` returns a description of type and the input-output dimensions of `sys`. If `sys` is a model array, the array size is also described. For identified models, the number of free parameters is also displayed. The lengths of the array dimensions are also included in the response to `size` when `sys` is a model array.

`d = size(sys)` returns:

- The row vector `d = [Ny Nu]` for a single dynamic 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 dynamic 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 a model array.

`Nf = size(sys, 'frequency')` returns the number of frequencies when `sys` is a frequency response data model. This is the same as the length of `sys.frequency`.

## Examples

### Example 1

Consider the model array of random 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.
```

### Example 2

Consider the process model:

```
sys = idproc({'p1d', 'p2'; 'p3uz', 'p0'});
```

It's input-output dimensions and number of free parameters are obtained by typing:

```
size(sys)
```

Process model with 2 outputs, 2 inputs and 12 free parameters.

### See Also

`issiso` | `ndims` | `isempty`

**Introduced before R2006a**

## sminreal

Structural pole/zero cancellations

### 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`.

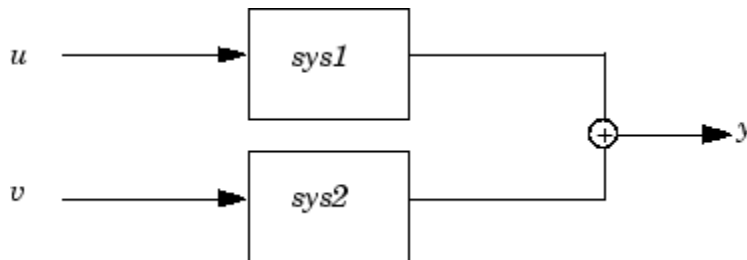
*sminreal* eliminates only structurally non minimal states, i.e., states that can be discarded by looking only at hard zero entries in the *A*, *B*, and *C* matrices. Such structurally nonminimal states arise, for example, when linearizing a Simulink model that includes some unconnected state-space or transfer function blocks.

### Examples

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))
```

## More About

### Tips

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.

### See Also

`minreal`

**Introduced before R2006a**

## spectralfact

Spectral factorization of linear systems

### Syntax

```
[G,S] = spectralfact(H)
[G,S] = spectralfact(F,R)
G = spectralfact(F,[])
```

### Description

`[G,S] = spectralfact(H)` computes the spectral factorization:  
 $H = G' * S * G$

of an LTI model satisfying  $H = H'$ . In this factorization, **S** is a symmetric matrix and **G** is a square, stable, and minimum-phase system with unit (identity) feedthrough. **G'** is the conjugate of **G**, which has transfer function  $G(-s)^T$  in continuous time, and  $G(1/z)^T$  in discrete time.

`[G,S] = spectralfact(F,R)` computes the spectral factorization:  
 $F' * R * F = G' * S * G$

without explicitly forming  $H = F' * R * F$ . As in the previous syntax, **S** is a symmetric matrix and **G** is a square, stable, and minimum-phase system with unit feedthrough.

`G = spectralfact(F,[])` computes a stable, minimum-phase system **G** such that:  
 $G' * G = F' * F$ .

### Examples

#### Spectral Factorization of System

Consider the following system.

```
G0 = ss(zpk([-1 -5 1+2i 1-2i],[-100 1+2i 1-2i -10],1e3));
H = G0'*G0;
```

$G_0$  has a mix of stable and unstable dynamics.  $H$  is a self-conjugate system whose dynamics consist of the poles and zeros of  $G_0$  and their reflections across the imaginary axis. Use spectral factorization to separate the stable poles and zeros into  $G$  and the unstable poles and zeros into  $G'$ .

```
[G,S] = spectralfact(H);
```

Confirm that  $G$  is stable and minimum phase, by checking that all its poles and zeros fall in the left half-plane ( $\text{Re}(s) < 0$ ).

```
p = pole(G)
z = zero(G)
```

```
p =
```

```
1.0e+02 *
-0.0100 + 0.0200i
-0.0100 - 0.0200i
-0.1000 + 0.0000i
-1.0000 + 0.0000i
```

```
z =
```

```
-1.0000 + 2.0000i
-1.0000 - 2.0000i
-1.0000 + 0.0000i
-5.0000 + 0.0000i
```

$G$  also has unit feedthrough.

```
G.D
```

```
ans =
```

```
1
```

Because  $H$  is SISO,  $S$  is a scalar. If  $H$  were MIMO, the dimensions of  $S$  would match the I/O dimensions of  $H$ .

S

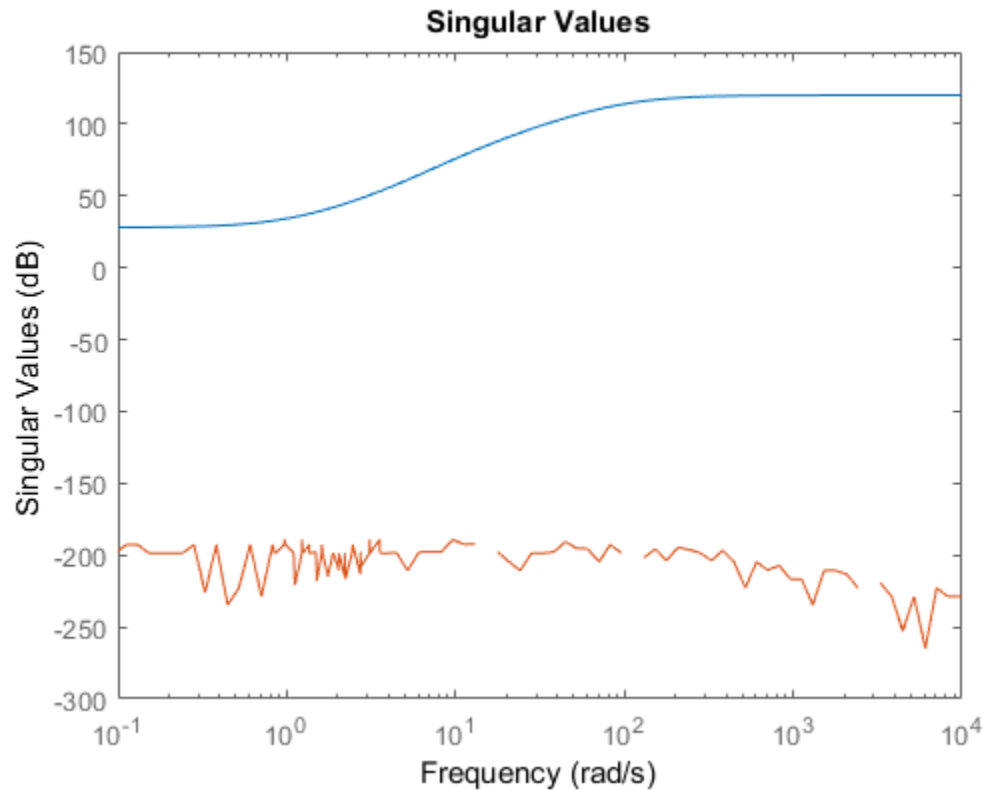
S =

1000000

Confirm that **G** and **S** satisfy  $H = G' * S * G$  by comparing the original system to the difference between the original and factored systems. `sigmaplot` throws a warning because the difference is very small.

```
Hf = G'*S*G;  
sigmaplot(H,H-Hf)
```

Warning: The frequency response has poor relative accuracy. This may be because the response is nearly zero or infinite at all frequencies, or because the state-space realization is ill conditioned. Use the "prescale" command to investigate further.



### Spectral Factorization from Factored Form

Suppose that you have the following 2-output, 2-input state-space model,  $F$ .

```
A = [-1.1  0.37;  
      0.37 -0.95];  
B = [0.72 0.71;  
      0   -0.20];  
C = [0.12 1.40  
      1.49 1.41];  
D = [0.67 0.7172;  
      -1.2  0];  
F = ss(A,B,C,D);
```

Suppose further that you have a symmetric 2-by-2 matrix, R.

$$R = \begin{bmatrix} 0.65 & 0.61 \\ 0.61 & -3.42 \end{bmatrix};$$

Compute the spectral factorization of the system given by  $H = F' * R * F$ , without explicitly computing H.

```
[G,S] = spectralfact(F,R);
```

G is a minimum-phase system with identity feedthrough.

G.D

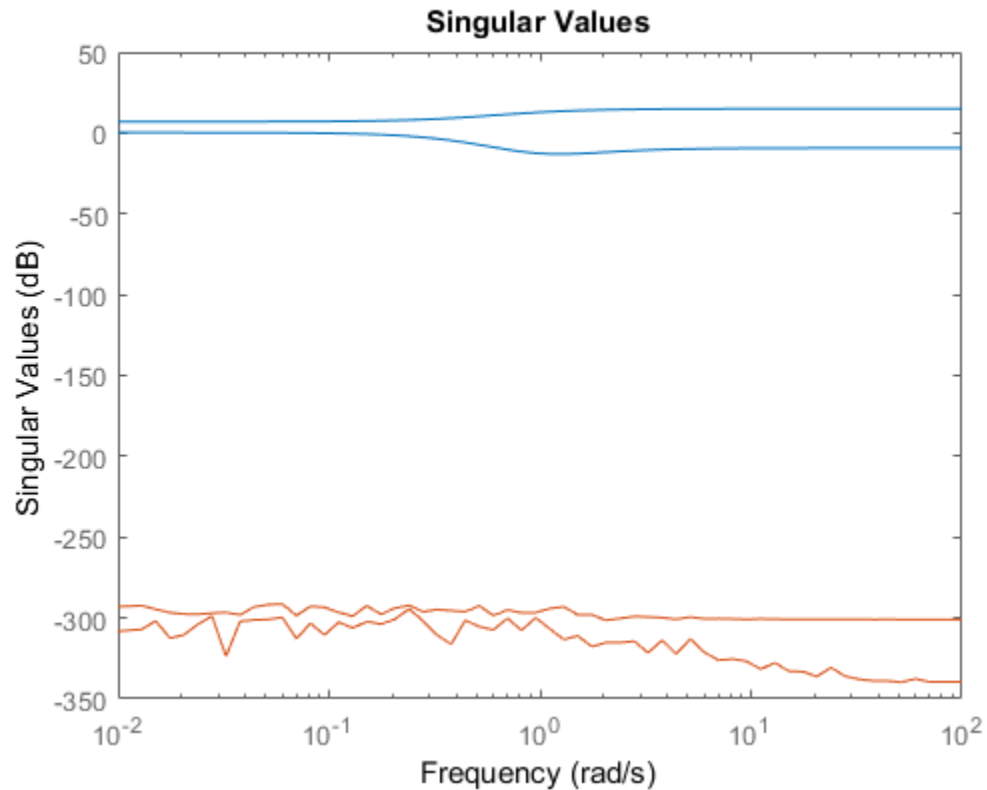
```
ans =
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Because F is has two inputs and two outputs, both R and S are 2-by-2 matrices.

Confirm that  $G' * S * G = F' * R * F$  by comparing the original factorization to the difference between the two factorizations. The singular values of the difference are far below those of the original system.

```
Ff = F' * R * F;  
Gf = G' * S * G;  
sigmaplot(Ff, Ff - Gf)
```



### Implicit Factorization

Consider the following discrete-time system.

```
F = zp(-1.76, [-1+i -1-i], -4, 0.002);
```

F has poles and zeros outside the unit circle. Use `spectralfact` to compute a system G with stable poles and zeros, such that  $G' * G = F' * F$ .

```
G = spectralfact(F, [])
```

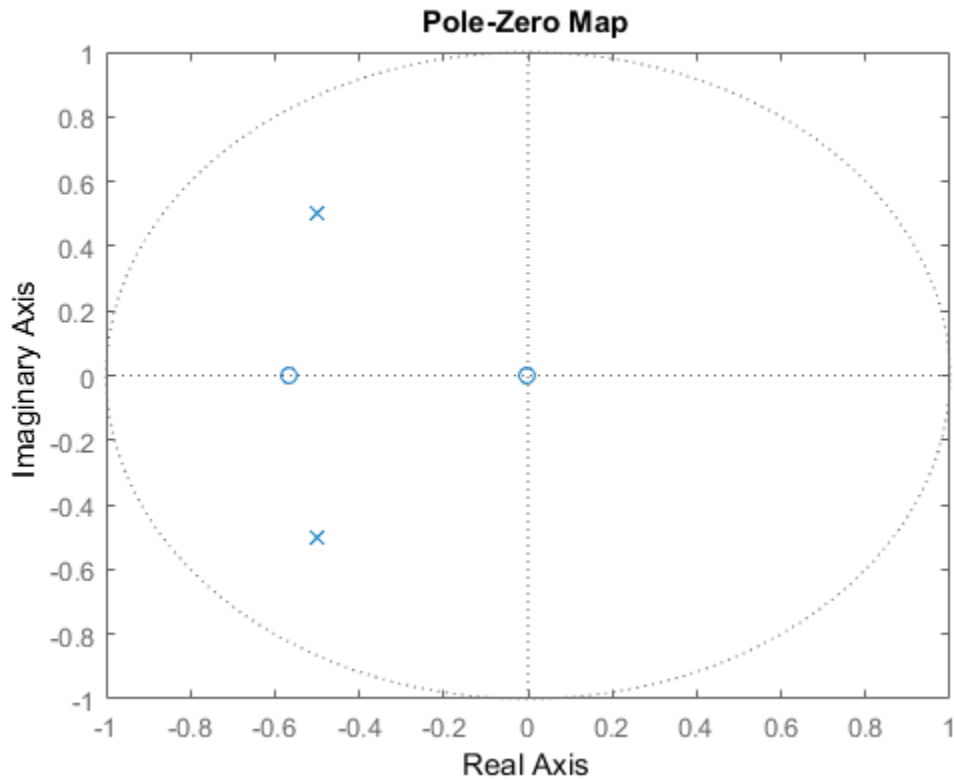
```
G =
```

$$\frac{-3.52 z (z+0.5682)}{(z^2 + z + 0.5)}$$

Sample time: 0.002 seconds  
Discrete-time zero/pole/gain model.

Unlike F, G has no poles or zeroes outside the unit circle. G does have an additional zero at  $z = 0$ , which is a reflection of the unstable zero at  $z = \text{Inf}$  in F.

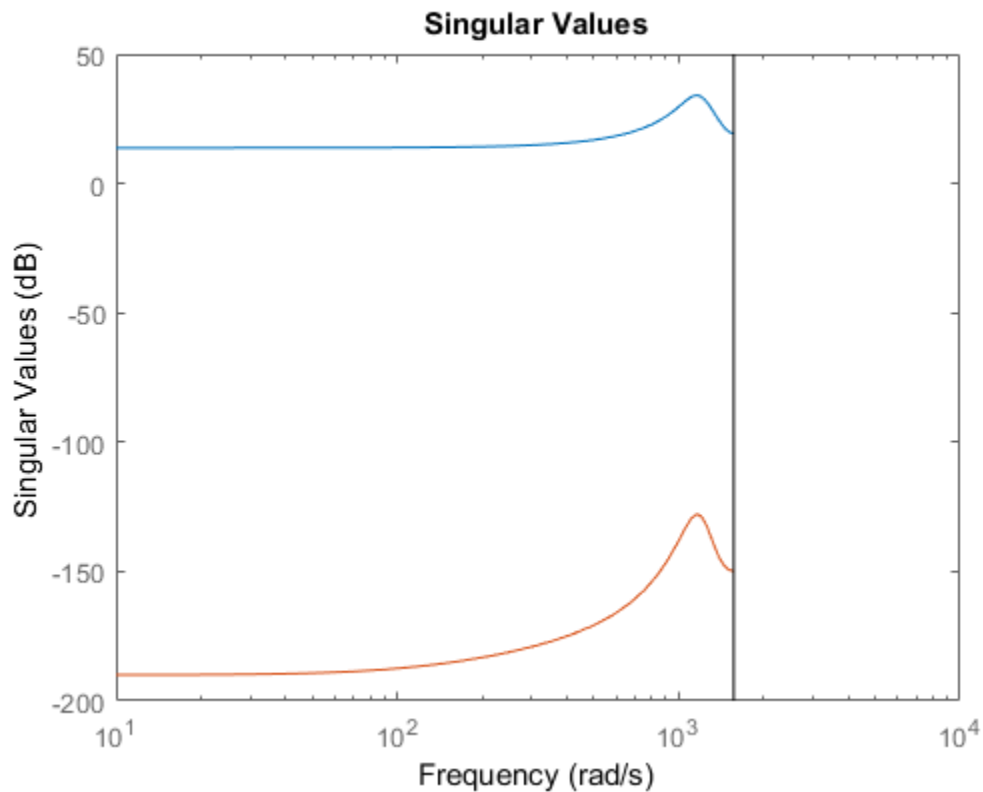
pzplot(G)





Confirm that  $G' * G = F' * F$  by comparing the original factorization to the difference between the two factorizations. The singular values of the difference are far below those of the original factorization.

```
Ff = F'*F;  
Gf = G'*G;  
sigmaplot(Ff,Ff-Gf)
```



## Input Arguments

**H** — Self-conjugate LTI model  
tf | zpk | ss

Self-conjugate LTI model, specified as a `tf`, `ss`, or `zpk` model. Self-conjugate means that is equal to its conjugate,  $H = H'$ . The conjugate  $H'$  is the transfer function  $H(-s)^T$  in continuous time and  $H(1/z)^T$  in discrete time.

H can be SISO or MIMO, provided it has as many outputs as inputs. H can be continuous or discrete with the following restrictions:

- In continuous time, H must be biproper with no poles or zeros at infinity or on the imaginary axis.
- In discrete time, H must have no poles or zeros on the unit circle.

**F — F factor**`tf | zpk | ss`

F factor of the factored form  $H = F' * R * F$ , specified as a `tf`, `ss`, or `zpk` model. F cannot have more inputs than outputs.

**R — R factor**`square matrix`

R factor of the factored form  $H = F' * R * F$ , specified as a symmetric square matrix with as many rows as there are outputs in F.

## Output Arguments

**G — LTI factor**`tf | zpk | ss`

LTI factor, returned as a `tf`, `ss`, or `zpk` model. G is a stable, minimum-phase system that satisfies:

- $H = G' * S * G$ , if you use the syntax `[G,S] = spectralfact(H)`.
- $G' * S * G = F' * R * F$ , if you use the syntax `[G,S] = spectralfact(F,R)`.
- $G' * G = F' * F$ , if you use the syntax `G = spectralfact(F,[])`.

**S — Numeric factor**`matrix`

Numeric factor, returned as a symmetric matrix that satisfies:

- $H = G' * S * G$ , if you use the syntax `[G,S] = spectralfact(H)`. The dimensions of  $S$  match the I/O dimensions of  $H$  and  $G$ .
- $G' * S * G = F' * R * F$ , if you use the syntax `[G,S] = spectralfact(F,R)`. The size of  $S$  along each dimension matches the number of outputs of  $F$ .

## More About

### Tips

- `spectralfact` assumes that  $H$  is self-conjugate. In some cases when  $H$  is not self-conjugate, `spectralfact` returns  $G$  and  $S$  that do not satisfy  $H = G' * S * G$ . Therefore, verify that your input model is in fact self-conjugate before using `spectralfact`. One way to verify  $H$  is to compare  $H$  to  $H - H'$  on a singular value plot.

```
sigmaplot(H,H-H')
```

If  $H$  is self-conjugate, the  $H - H'$  line on the plot lies far below the  $H$  line.

- “Arithmetic Operations”

### See Also

`modsep` | `stabsep`

Introduced in R2016a

## **ss**

Create state-space model, convert to state-space model

### **Syntax**

```
sys = ss(A,B,C,D)
sys = ss(A,B,C,D,Ts)
sys = ss(D)
sys = ss(A,B,C,D,Itisys)
sys_ss = ss(sys)
sys_ss = ss(sys, 'minimal')
sys_ss = ss(sys, 'explicit')
sys_ss = ss(sys, 'measured')
sys_ss = ss(sys, 'noise')
sys_ss = ss(sys, 'augmented')
```

### **Description**

Use **ss** to create state-space models (**ss** model objects) with real- or complex-valued matrices or to convert dynamic system models to state-space model form. You can also use **ss** to create Generalized state-space (**genss**) models.

### **Creation of State-Space Models**

`sys = ss(A,B,C,D)` creates a state-space model object representing the continuous-time state-space model

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

For a model with  $N_x$  states,  $N_y$  outputs, and  $N_u$  inputs:

- **A** is an  $N_x$ -by- $N_x$  real- or complex-valued matrix.
- **B** is an  $N_x$ -by- $N_u$  real- or complex-valued matrix.

- $C$  is an  $N_y$ -by- $N_x$  real- or complex-valued matrix.
- $D$  is an  $N_y$ -by- $N_u$  real- or complex-valued matrix.

To set  $D = 0$ , 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  $T_s$  (in seconds). Set  $T_s = -1$  or  $T_s = []$  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 properties inherited from the model `ltisys` (including the sample time).

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

`'PropertyName',PropertyValue`

Each pair specifies a particular property of the model, for example, the input names or some notes on the model history. See “Properties” on page 2-949 for more information about available `ss` model object properties.

The following expression:

`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 a dynamic system model `sys` to state-space form. The output `sys_ss` is an equivalent state-space model (`ss` model 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 state-space realization is equivalent to `sys_ss = minreal(ss(sys))`.

`sys_ss = ss(sys, 'explicit')` computes an explicit realization ( $E = I$ ) of the dynamic system model `sys`. If `sys` is improper, `ss` returns an error.

---

**Note:** Conversions to state space are not uniquely defined in the SISO case. They are also not guaranteed to produce a minimal realization in the MIMO case. For more information, see “Recommended Working Representation”.

---

## Conversion of Identified Models

An identified model is represented by an input-output equation of the form  $y(t) = Gu(t) + He(t)$ , where  $u(t)$  is the set of measured input channels and  $e(t)$  represents the noise channels. If  $\Lambda = LL'$  represents the covariance of noise  $e(t)$ , this equation can also be written as  $y(t) = Gu(t) + HLv(t)$ , where  $\text{cov}(v(t)) = I$ .

`sys_ss = ss(sys)` or `sys_ss = ss(sys, 'measured')` converts the measured component of an identified linear model into the state-space form. `sys` is a model of type `idss`, `idproc`, `idtf`, `idpoly`, or `idgrey`. `sys_ss` represents the relationship between  $u$  and  $y$ .

`sys_ss = ss(sys, 'noise')` converts the noise component of an identified linear model into the state space form. It represents the relationship between the noise input  $v(t)$  and output  $y_{noise} = HL v(t)$ . The noise input channels belong to the `InputGroup` 'Noise'. The names of the noise input channels are  $v@yname$ , where  $yname$  is the name of the corresponding output channel. `sys_ss` has as many inputs as outputs.

`sys_ss = ss(sys, 'augmented')` converts both the measured and noise dynamics into a state-space model. `sys_ss` has  $ny+nu$  inputs such that the first  $nu$  inputs represent the channels  $u(t)$  while the remaining by channels represent the noise channels  $v(t)$ . `sys_ss.InputGroup` contains 2 input groups- 'measured' and 'noise'. `sys_ss.InputGroup.Measured` is set to  $1:nu$  while `sys_ss.InputGroup.Noise` is set to  $nu+1:nu+ny$ . `sys_ss` represents the equation  $y(t) = [G \ HL] [u; v]$

---

**Tip** An identified nonlinear model cannot be converted into a state-space form. Use linear approximation functions such as `linearize` and `linapp`.

---

## Creation of Generalized State-Space Models

You can use the syntax:

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

to create a Generalized state-space (`genss`) model when one or more of the matrices `A`, `B`, `C`, `D` is a tunable `realp` or `genmat` model. For more information about Generalized state-space models, see “Models with Tunable Coefficients”.

## Properties

`ss` objects have the following properties:

### **A, B, C, D, E**

State-space matrices.

- **A** — State matrix *A*. Square real- or complex-valued matrix with as many rows as states.
- **B** — Input-to-state matrix *B*. Real- or complex-valued matrix with as many rows as states and as many columns as inputs.
- **C** — State-to-output matrix *C*. Real- or complex-valued matrix with as many rows as outputs and as many columns as states.
- **D** — Feedthrough matrix *D*. Real- or complex-valued matrix with as many rows as outputs and as many columns as inputs.
- **E** — *E* matrix for implicit (descriptor) state-space models. By default `e = []`, meaning that the state equation is explicit. To specify an implicit state equation  $E \frac{dx}{dt} = Ax + Bu$ , set this property to a square matrix of the same size as *A*. See `dss` for more information about creating descriptor state-space models.

### **Scaled**

Logical value indicating whether scaling is enabled or disabled.

When `Scaled = 0` (false), most numerical algorithms acting on the state-space model automatically rescale the state vector to improve numerical accuracy. You can disable such auto-scaling by setting `Scaled = 1` (true). For more information about scaling, see `prescale`.

**Default:** 0 (false)

### **StateName**

State names, specified as one of the following:

- Character vector — For first-order models, for example, `'velocity'`.
- Cell array of character vectors — For models with two or more states
- `''` — For unnamed states.

**Default:** `''` for all states

### **StateUnit**

State units, specified as one of the following:

- Character vector — For first-order models, for example, `'velocity'`.
- Cell array of character vectors — For models with two or more states
- `''` — For unnamed states.

Use `StateUnit` to keep track of the units each state is expressed in. `StateUnit` has no effect on system behavior.

**Default:** `''` for all states

### **InternalDelay**

Vector storing internal delays.

Internal delays arise, for example, when closing feedback loops on systems with delays, or when connecting delayed systems in series or parallel. For more information about internal delays, see “Closing Feedback Loops with Time Delays” in the *Control System Toolbox User's Guide*.

For continuous-time models, internal delays are expressed in the time unit specified by the `TimeUnit` property of the model. For discrete-time models, internal delays are



expressed as integer multiples of the sample time  $T_s$ . For example, `InternalDelay = 3` means a delay of three sampling periods.

You can modify the values of internal delays. However, the number of entries in `sys.InternalDelay` cannot change, because it is a structural property of the model.

### **InputDelay**

Input delay for each input channel, specified as a scalar value or numeric vector. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sample time  $T_s$ . For example, `InputDelay = 3` means a delay of three sample times.

For a system with  $N_u$  inputs, set `InputDelay` to an  $N_u$ -by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0

### **OutputDelay**

Output delays. `OutputDelay` is a numeric vector specifying a time delay for each output channel. For continuous-time systems, specify output delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify output delays in integer multiples of the sample time  $T_s$ . For example, `OutputDelay = 3` means a delay of three sampling periods.

For a system with  $N_y$  outputs, set `OutputDelay` to an  $N_y$ -by-1 vector, where each entry is a numerical value representing the output delay for the corresponding output channel. You can also set `OutputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all output channels

### **$T_s$**

Sample time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** `''` for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### **OutputName**

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, 'measurements'.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{ 'measurements(1)'; 'measurements(2) '}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all output channels

### **OutputUnit**

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, 'seconds'.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

## OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### Name

System name, specified as a character vector. For example, `'system_1'`.

**Default:** `''`

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** `{}`

### UserData

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** `[]`

### SamplingGrid

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This

information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta,w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:,:,1,1) [zeta=0.3, w=5] =
```

```
      25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```
      25
-----
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink

Control Design commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

**Default:** []

## Examples

### Create Discrete-Time State-Space Model

Create a state-space model with a sample time of 0.25 seconds and the following state-space matrices:

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

Specify the state-space matrices.

```
A = [0 1; -5 -2];  
B = [0;3];  
C = [0 1];  
D = 0;
```

Specify the sample time.

```
Ts = 0.25;
```

Create the state-space model.

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

### Specify State and Input Names for Discrete-Time State-Space Model

Create state-space matrices and specify sample time.

```
A = [0 1; -5 -2];  
B = [0;3];  
C = [0 1];  
D = 0;  
Ts = 0.05;
```

Create state-space model, specifying the state and input names.

```
sys = ss(A,B,C,D,Ts, 'StateName', {'Position' 'Velocity'}, ...  
        'InputName', 'Force');
```

The number of state and input names must be consistent with the dimensions of A, B, C, and D.

### Convert Transfer Function to State-Space Model

Compute the state-space model of the following transfer function:

$$H(s) = \left[ \begin{array}{c} \frac{s+1}{s^3+3s^2+3s+2} \\ \frac{s^2+3}{s^2+s+1} \end{array} \right]$$

Create the transfer function model.

```
H = [tf([1 1],[1 3 3 2]) ; tf([1 0 3],[1 1 1])];
```

Convert this model to a state-space model.

```
sys = ss(H);
```

Examine the size of the state-space model.

```
size(sys)
```

State-space model with 2 outputs, 1 inputs, and 5 states.

The number of states is equal to the cumulative order of the SISO entries in  $H(s)$ .

To obtain a minimal realization of  $H(s)$ , enter

```
sys = ss(H, 'minimal');  
size(sys)
```

State-space model with 2 outputs, 1 inputs, and 3 states.

The resulting model has an order of three, which is the minimum number of states needed to represent  $H(s)$ . To see this number of states, refactor  $H(s)$  as the product of a first-order system and a second-order system.



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

## Explicit Realization of Descriptor State-Space Model

Create a descriptor state-space model ( $E \neq I$ ).

```
a = [2 -4; 4 2];
b = [-1; 0.5];
c = [-0.5, -2];
d = [-1];
e = [1 0; -3 0.5];
sysd = dss(a,b,c,d,e);
```

Compute an explicit realization of the system ( $E = I$ ).

```
syse = ss(sysd, 'explicit')
```

```
syse =
```

```
A =
      x1    x2
x1      2   -4
x2     20  -20
```

```
B =
      u1
x1    -1
x2    -5
```

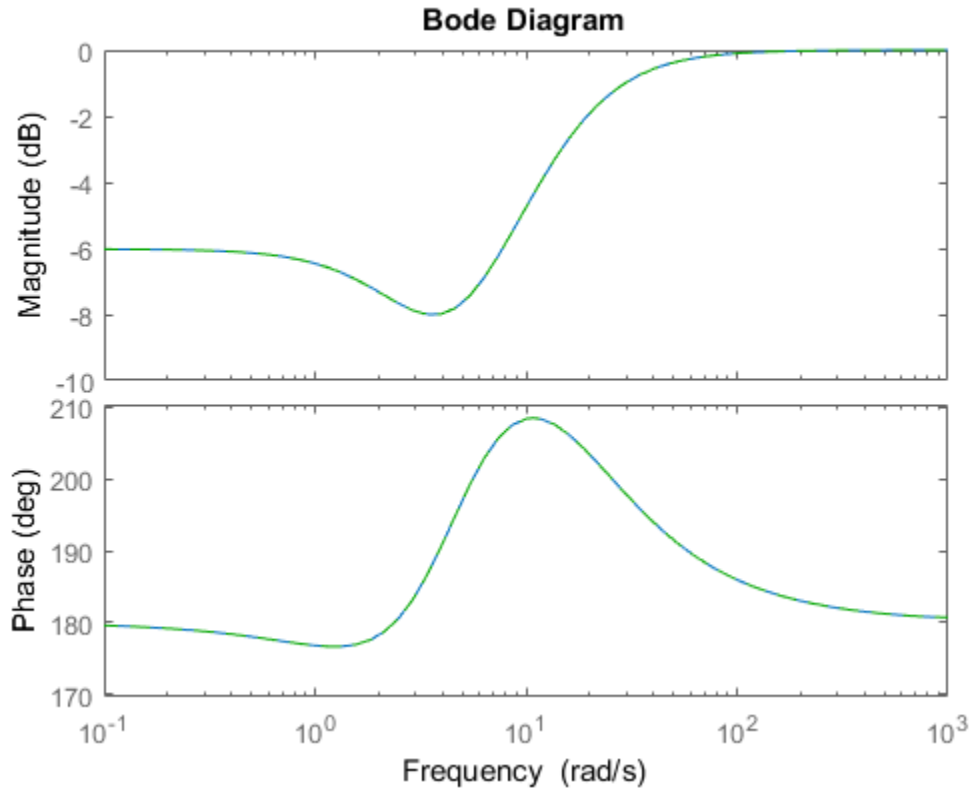
```
C =
      x1    x2
y1  -0.5   -2
```

```
D =
      u1
y1   -1
```

Continuous-time state-space model.

Confirm that the descriptor and explicit realizations have equivalent dynamics.

```
bodeplot(sysd, syse, 'g--')
```



### Create State-Space Model with Both Fixed and Tunable Parameters

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

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

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

Create the tunable parameters using `realp`.

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

Define a generalized matrix using algebraic expressions of **a** and **b**.

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

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

Create the fixed-value state-space matrices.

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

Use **ss** to create the state-space model.

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

```
sys =
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 2 states, and 0 direct terms.
a: Scalar parameter, 2 occurrences.
b: Scalar parameter, 2 occurrences.
```

Type "ss(sys)" to see the current value, "get(sys)" to see all properties, and "sys.Blocks" to see the blocks.

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

## Extract Components from Identified State-Space Model

Extract the measured and noise components of an identified polynomial model into two separate state-space models. The former (measured component) can serve as a plant model while the latter can serve as a disturbance model for control system design.

```
load icEngine
z = iddata(y,u,0.04);
sys = ssest(z,3);

sysMeas = ss(sys,'measured')
```

```
sysNoise = ss(sys, 'noise')
```

Alternatively, use `ss(sys)` to extract the measured component.

## More About

### Algorithms

For TF to SS model conversion, `ss(sys_tf)` returns a modified version of the controllable canonical form. It uses an algorithm similar to `tf2ss`, but further rescales the state vector to compress the numerical range in state matrix `A` and to improve numerics in subsequent computations.

For ZPK to SS conversion, `ss(sys_zpk)` uses direct form II structures, as defined in signal processing texts. See *Discrete-Time Signal Processing* by Oppenheim and Schaffer for details.

For example, in the following code, `A` and `sys.A` differ by a diagonal state transformation:

```
n=[1 1];
d=[1 1 10];
[A,B,C,D]=tf2ss(n,d);
sys=ss(tf(n,d));
A
```

```
A =
```

```
    -1    -10
     1     0
```

```
sys.A
```

```
ans =
    -1    -5
     2     0
```

For details, see `balance`.

- “What Are Model Objects?”
- “State-Space Models”
- “MIMO State-Space Models”

**See Also**

dss | frd | get | set | ssdata | tf | zpk

**Introduced before R2006a**

## ss2ss

State coordinate transformation for state-space model

### Syntax

```
sysT = ss2ss(sys, T)
```

### Description

Given a state-space model `sys` with equations

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

or the innovations form used by the identified state-space (IDSS) models:

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

(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.

$$\begin{aligned}\dot{\bar{x}} &= TAT^{-1}\bar{x} + TBu \\ y &= CT^{-1}\bar{x} + Du\end{aligned}$$

or, in the case of an IDSS model:

$$\begin{aligned}\dot{\bar{x}} &= TAT^{-1}\bar{x} + TBu + TKe \\ y &= CT^{-1}\bar{x} + Du + e\end{aligned}$$

(IDSS models require System Identification Toolbox software.)

`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.

## Examples

Perform a similarity transform to improve the conditioning of the  $A$  matrix.

```
T = balance(sys.A)
sysb = ss2ss(sys, inv(T))
```

## See Also

`balreal` | `canon`

**Introduced before R2006a**

## ssdata

Access 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 `ss` for more information on the format of state-space model data.

If *sys* appears in descriptor form (nonempty *E* matrix), an equivalent explicit form is first derived.

If *sys* has internal delays, *A*, *B*, *C*, *D* are obtained by first setting all internal delays to zero (creating a zero-order Padé approximation). For some systems, setting delays to zero creates singular algebraic loops, which result in either improper or ill-defined, zero-delay approximations. For these systems, `ssdata` cannot display the matrices and returns an error. This error does not imply a problem with the model *sys* itself.

`[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 | getdelaymodel | set | tfdata | zpkdata | get | ss

**Introduced before R2006a**

## stabsep

Stable-unstable decomposition

### Syntax

```
[GS,GNS]=stabsep(G)
[G1,GNS] = stabsep(G,'abstol',ATOL,'reltol',RTOL)
[G1,G2]=stabsep(G, ..., 'Mode', MODE, 'Offset', ALPHA)
[G1,G2] = stabsep(G, opts)
```

### Description

`[GS,GNS]=stabsep(G)` decomposes the LTI model `G` into its stable and unstable parts

$$G = GS + GNS$$

where `GS` contains all stable modes that can be separated from the unstable modes in a numerically stable way, and `GNS` contains the remaining modes. `GNS` is always strictly proper.

`[G1,GNS] = stabsep(G, 'abstol', ATOL, 'reltol', RTOL)` specifies absolute and relative error tolerances for the stable/unstable decomposition. The frequency responses of `G` and `GS + GNS` should differ by no more than  $ATOL + RTOL * \text{abs}(G)$ . Increasing these tolerances helps separate nearby stable and unstable modes at the expense of accuracy. The default values are  $ATOL=0$  and  $RTOL=1e-8$ .

`[G1,G2]=stabsep(G, ..., 'Mode', MODE, 'Offset', ALPHA)` produces a more general stable/unstable decomposition where `G1` includes all separable poles lying in the regions defined using offset `ALPHA`. This can be useful when there are numerical accuracy issues. For example, if you have a pair of poles close to, but slightly to the left of the  $j\omega$ -axis, you can decide not to include them in the stable part of the decomposition if numerical considerations lead you to believe that the poles may be in fact unstable

This table lists the stable/unstable boundaries as defined by the offset `ALPHA`.

Mode	Continuous Time Region	Discrete Time Region
1	$\text{Re}(s) < -\text{ALPHA} * \max(1,  \text{Im}(s) )$	$1 -  z  < 1 - \text{ALPHA}$

Mode	Continuous Time Region	Discrete Time Region
2	$\text{Re}(s) > \text{ALPHA} * \max(1,  \text{Im}(s) )$	$2  z  > 1 + \text{ALPHA}$

The default values are MODE=1 and ALPHA=0.

$[G1, G2] = \text{stabsep}(G, \text{opts})$  computes the stable/unstable decomposition of  $G$  using the options specified in the `stabsepOptions` object `opts`.

## Examples

Compute a stable/unstable decomposition with absolute error no larger than  $1e-5$  and an offset of 0.1:

```
h = zpk(1, [-2 -1 1 -0.001], 0.1)
[hs, hns] = stabsep(h, stabsepOptions('AbsTol', 1e-5, 'Offset', 0.1));
```

The stable part of the decomposition has poles at -1 and -2.

hs

```
Zero/pole/gain:
-0.050075 (s+2.999)
-----
      (s+1) (s+2)
```

The unstable part of the decomposition has poles at +1 and -0.001 (which is nominally stable).

hns

```
Zero/pole/gain:
0.050075 (s-1)
-----
      (s+0.001) (s-1)
```

## See Also

`stabsepOptions` | `modsep`

**Introduced before R2006a**

## stabsepOptions

Options for stable-unstable decomposition

### Syntax

```
opts = stabsepOptions  
opts = stabsepOptions('OptionName', OptionValue)
```

### Description

`opts = stabsepOptions` returns the default options for the `stabsep` command.

`opts = stabsepOptions('OptionName', OptionValue)` accepts one or more comma-separated name/value pairs. Specify *OptionName* inside single quotes.

### Input Arguments

#### Name-Value Pair Arguments

##### 'Focus'

Focus of decomposition. Specified as one of the following values:

- 'stable'            First output of `stabsep` contains only stable dynamics.
- 'unstable'        First output of `stabsep` contains only unstable dynamics.

Default: 'stable'

##### 'AbsTol, RelTol'

Absolute and relative error tolerance for stable/unstable decomposition. Positive scalar values. When decomposing a model  $G$ , `stabsep` ensures that the frequency responses of  $G$  and  $GS + GU$  differ by no more than  $\text{AbsTol} + \text{RelTol} * \text{abs}(G)$ . Increasing these tolerances helps separate nearby stable and unstable modes at the expense of accuracy. See `stabsep` for more information.

**Default:** AbsTol = 0; RelTol = 1e-8

### 'Offset'

Offset for the stable/unstable boundary. Positive scalar value. The first output of `stabsep` includes only poles satisfying:

Continuous time:

- $\text{Re}(s) < -\text{Offset} * \max(1, |\text{Im}(s)|)$  (Focus = 'stable')
- $\text{Re}(s) > \text{Offset} * \max(1, |\text{Im}(s)|)$  (Focus = 'unstable')

Discrete time:

- $|z| < 1 - \text{Offset}$  (Focus = 'stable')
- $|z| > 1 + \text{Offset}$  (Focus = 'unstable')

Increase the value of `Offset` to treat poles close to the stability boundary as unstable.

**Default:** 0

For additional information on the options and how to use them, see the `stabsep` reference page.

## Examples

Compute the stable/unstable decomposition of the system given by:

$$G(s) = \frac{10(s+0.5)}{(s+10^{-6})(s+2-5i)(s+2+5i)}$$

Use the `Offset` option to force `stabsep` to exclude the pole at  $s = 10^{-6}$  from the stable term of the stable/unstable decomposition.

```
G = zpk(-.5, [-1e-6 -2+5i -2-5i], 10);
opts = stabsepOptions('Offset', .001); % Create option set
[G1,G2] = stabsep(G,opts) % treats -1e-6 as unstable
```

These commands return the result:

```
Zero/pole/gain:  
-0.17241 (s-54)  
-----  
(s^2 + 4s + 29)
```

```
Zero/pole/gain:  
0.17241  
-----  
(s+1e-006)
```

The pole at  $s = 10^{-6}$  is in the second (unstable) output.

### See Also

stabsep

**Introduced in R2010a**

## stack

Build model array by stacking models or model arrays along array dimensions

### Syntax

```
sys = stack(arraydim,sys1,sys2,...)
```

### Description

`sys = stack(arraydim,sys1,sys2,...)` produces an array of dynamic system models `sys` by stacking (concatenating) the models (or 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. For more information about model arrays and array dimensions, see “Model Arrays”.

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`.

## Examples

### Example 1

If `sys1` and `sys2` are two models:

- `stack(1,sys1,sys2)` produces a 2-by-1 model array.
- `stack(2,sys1,sys2)` produces a 1-by-2 model array.
- `stack(3,sys1,sys2)` produces a 1-by-1-by-2 model array.

### Example 2

Stack identified state-space models derived from the same estimation data and compare their bode responses.

```
load iddata1 z1
sysc = cell(1,5);
opt = ssestOptions('Focus','simulation');
for i = 1:5
    sysc{i} = ssest(z1,i-1,opt);
end
sysArray = stack(1, sysc{:});
bode(sysArray);
```

**Introduced before R2006a**



# step

Step response plot of dynamic system; step response data

## Syntax

```
step(sys)
step(sys,Tfinal)
step(sys,t)
step(sys1,sys2,...,sysN)
step(sys1,sys2,...,sysN,Tfinal)
step(sys1,sys2,...,sysN,t)
y = step(sys,t)
[y,t] = step(sys)
[y,t] = step(sys,Tfinal)
[y,t,x] = step(sys)
[y,t,x,yzd] = step(sys)
[y,...] = step(sys,...,options)
```

## Description

`step` calculates the step response of a dynamic system. For the state-space case, zero initial state is assumed. When it is invoked with no output arguments, this function plots the step response on the screen.

`step(sys)` plots the step response of an arbitrary dynamic system model, `sys`. This model can be continuous- or discrete-time, 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,Tfinal)` simulates the step response from  $t = 0$  to the final time  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time ( $T_s = -1$ ), `step` interprets `Tfinal` as the number of sampling periods to simulate.

`step(sys,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time models,

$t$  should be of the form  $T_i:T_s:T_f$ , where  $T_s$  is the sample time. For continuous-time models,  $t$  should be of the form  $T_i:dt:T_f$ , where  $dt$  becomes the sample time of a discrete approximation to the continuous system (see “Algorithms” on page 2-985). The `step` command always applies the step input at  $t=0$ , regardless of  $T_i$ .

To plot the step response of several models `sys1`, ..., `sysN` on a single figure, use

```
step(sys1,sys2,...,sysN)
step(sys1,sys2,...,sysN,Tfinal)
step(sys1,sys2,...,sysN,t)
```

All of the systems plotted on a single plot must have the same number of inputs and outputs. You can, however, plot a mix of continuous- and discrete-time systems on a single plot. This syntax is useful to compare the step responses of multiple systems.

You can also specify a distinctive color, linestyle, marker, or all three 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 = step(sys,t)
[y,t] = step(sys)
[y,t] = step(sys,Tfinal)
[y,t,x] = step(sys)
```

`step` returns the output response  $y$ , the time vector  $t$  used for simulation (if not supplied as an input argument), and the state trajectories  $x$  (for state-space models only). No plot generates 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

*(length of t) × (number of outputs) × (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

*(length of t) × (number of states) × (number of inputs)*

For identified models (see `idlti` and `idn1model`) `[y,t,x,ysd] = step(sys)` also computes the standard deviation `ysd` of the response `y` (`ysd` is empty if `sys` does not contain parameter covariance information).

`[y,...] = step(sys,...,options)` specifies additional options for computing the step response, such as the step amplitude or input offset. Use `stepDataOptions` to create the option set `options`.

## Examples

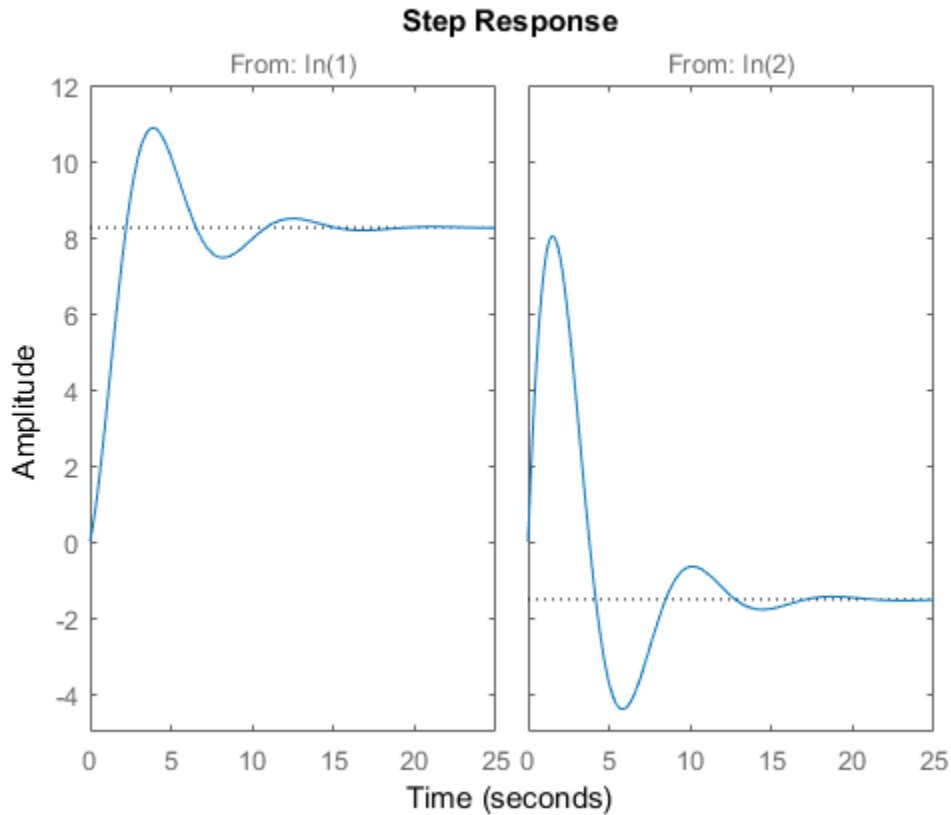
### Step Response Plot of Dynamic System

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)
```

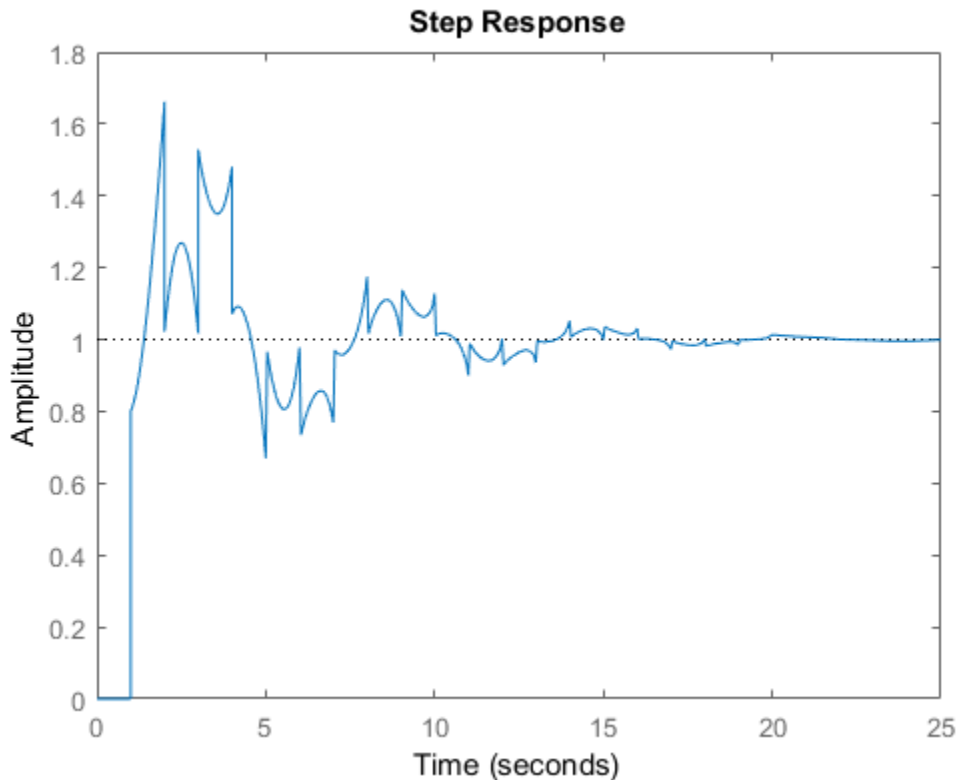


The left plot shows the step response of the first input channel, and the right plot shows the step response of the second input channel.

### Step Response Plot of Feedback Loop with Delay

Create a feedback loop with delay and plot its step response.

```
s = tf('s');
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(ss(G),1);
step(T)
```



The system step response displayed is chaotic. The step response of systems with internal delays may exhibit odd behavior, such as recurring jumps. Such behavior is a feature of the system and not software anomalies.

### Step Responses of Identified Models with Confidence Regions

Compare the step response of a parametric identified model to a non-parametric (empirical) model. Also view their  $3\sigma$  confidence regions.

Load the data.

```
load iddata1 z1
```

Estimate a parametric model.

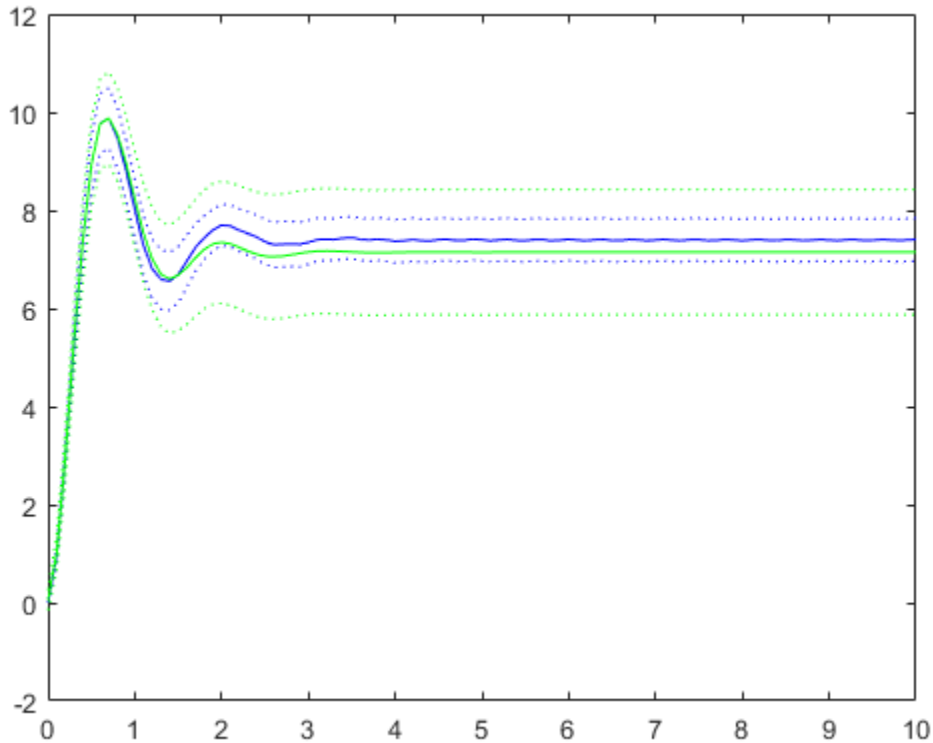
```
sys1 = ssest(z1,4);
```

Estimate a non-parametric model.

```
sys2 = impulseest(z1);
```

Plot the step responses for comparison.

```
t = (0:0.1:10)';  
[y1, ~, ~, ysd1] = step(sys1,t);  
[y2, ~, ~, ysd2] = step(sys2,t);  
plot(t, y1, 'b', t, y1+3*ysd1, 'b:', t, y1-3*ysd1, 'b:')  
hold on  
plot(t, y2, 'g', t, y2+3*ysd2, 'g:', t, y2-3*ysd2, 'g:')
```



### Validate Linearization of Identified Nonlinear ARX Model

Validate the linearization of a nonlinear ARX model by comparing the small amplitude step responses of the linear and nonlinear models.

Load the data.

```
load iddata2 z2;
```

Estimate a nonlinear ARX model.

```
nlsys = nlarx(z2,[4 3 10], 'tree', 'custom', {'sin(y1(t-2)*u1(t))+y1(t-2)*u1(t)+u1(t).*u1(t)'});
```

Determine an equilibrium operating point for `nlsys` corresponding to a steady-state input value of 1.

```
u0 = 1;  
[X,-,r] = findop(nlsys, 'steady', 1);  
y0 = r.SignalLevels.Output;
```

Obtain a linear approximation of `nlsys` at this operating point.

```
sys = linearize(nlsys,u0,X);
```

Validate the usefulness of `sys` by comparing its small-amplitude step response to that of `nlsys`.

The nonlinear system `nlsys` is operating at an equilibrium level dictated by (`u0`, `y0`). Introduce a step perturbation of size 0.1 about this steady-state and compute the corresponding response.

```
opt = stepDataOptions;  
opt.InputOffset = u0;  
opt.StepAmplitude = 0.1;  
t = (0:0.1:10)';  
ynl = step(nlsys, t, opt);
```

The linear system `sys` expresses the relationship between the perturbations in input to the corresponding perturbation in output. It is unaware of the nonlinear system's equilibrium values.

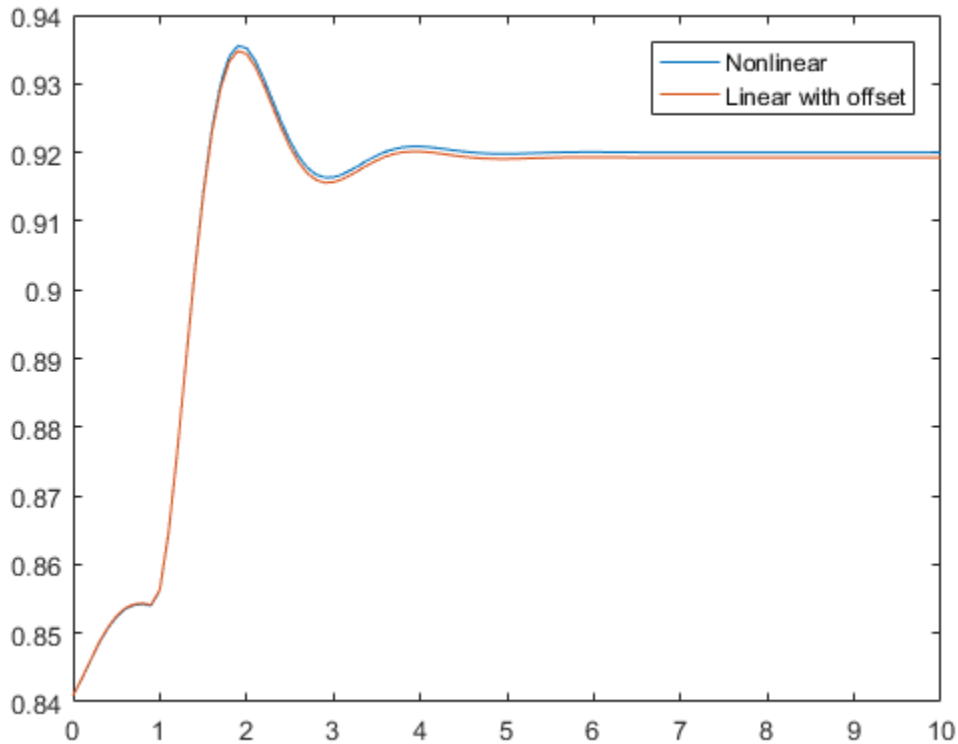
Plot the step response of the linear system.

```
opt = stepDataOptions;  
opt.StepAmplitude = 0.1;  
y1 = step(sys, t, opt);
```

Add the steady-state offset, `y0`, to the response of the linear system and plot the responses.

```
plot(t, ynl, t, y1+y0)  
legend('Nonlinear', 'Linear with offset')
```





### Step Response of Identified Time-Series Model

Compute the step response of an identified time-series model.

A time-series model, also called a signal model, is one without measured input signals. The step plot of this model uses its (unmeasured) noise channel as the input channel to which the step signal is applied.

Load the data.

```
load iddata9;
```

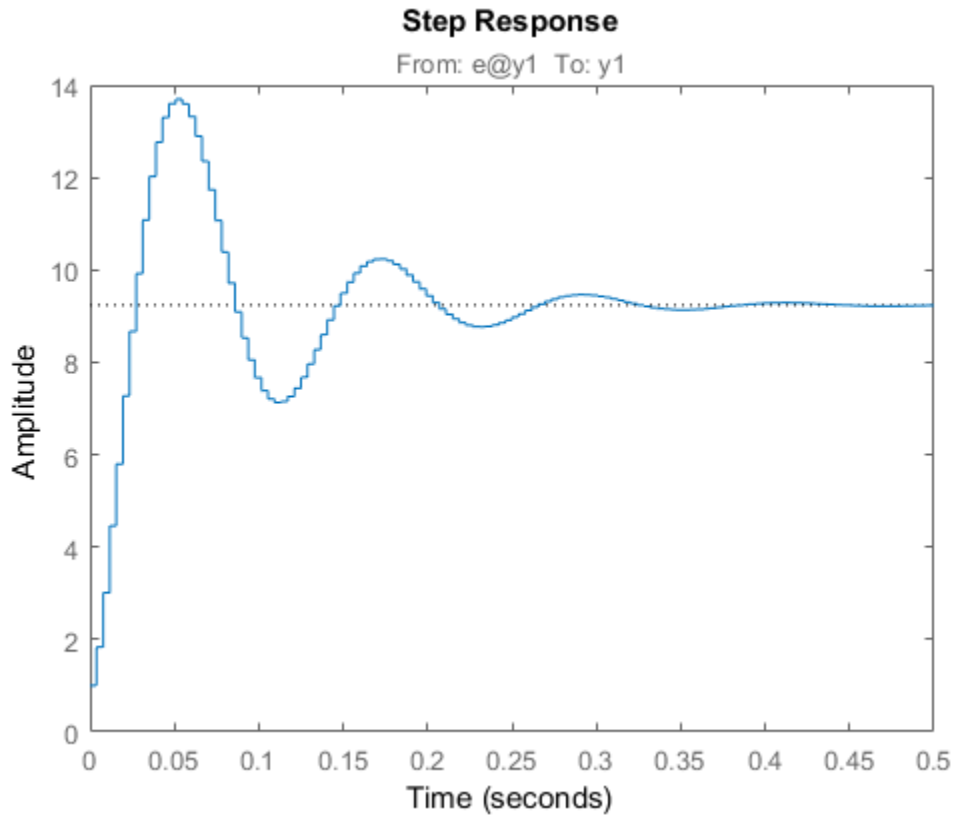
Estimate a time-series model.

```
sys = ar(z9, 4);
```

`ys` is a model of the form  $A y(t) = e(t)$ , where  $e(t)$  represents the noise channel. For computation of step response,  $e(t)$  is treated as an input channel, and is named `e@y1`.

Plot the step response.

```
step(sys)
```



## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### Algorithms

Continuous-time models without internal delays are converted to state space and discretized using zero-order hold on the inputs. The sample time 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). The resulting simulation time steps  $\mathbf{t}$  are equisampled with spacing  $\mathbf{dt}$ .

For systems with internal delays, Control System Toolbox software uses variable step solvers. As a result, the time steps  $\mathbf{t}$  are not equisampled.

## References

- [1] L.F. Shampine and P. Gahinet, "Delay-differential-algebraic equations in control theory," *Applied Numerical Mathematics*, Vol. 56, Issues 3–4, pp. 574–588.

## See Also

[impulse](#) | [initial](#) | [Linear System Analyzer](#) | [lsim](#) | [stepDataOptions](#)

**Introduced before R2006a**

# stepDataOptions

Options set for `step`

## Syntax

```
opt = stepDataOptions  
opt = stepDataOptions(Name, Value)
```

## Description

`opt = stepDataOptions` creates the default options for `step`.

`opt = stepDataOptions(Name, Value)` creates an options set with the options specified by one or more `Name, Value` pair arguments.

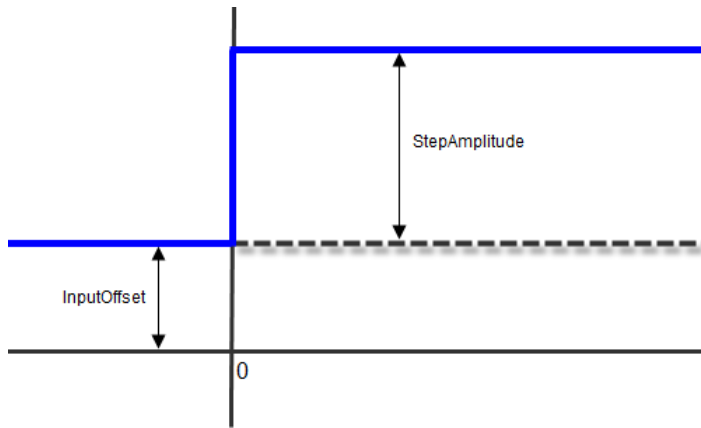
## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

**'InputOffset'**

Input signal level for all time  $t < 0$ , as shown in the next figure.



**Default:** 0

**'StepAmplitude'**

Change of input signal level which occurs at time  $t = 0$ , as shown in the previous figure.

**Default:** 1

## Output Arguments

**opt**

Option set containing the specified options for `step`.

## Examples

### Specify Input Offset and Step Amplitude Level for Step Response

Create a transfer function model.

```
sys = tf(1,[1,1]);
```

Create an option set for `step` to specify input offset and step amplitude level.

```
opt = stepDataOptions('InputOffset',-1,'StepAmplitude',2);
```

Calculate the step response using the specified options.

```
[y,t] = step(sys,opt);
```

## **See Also**

step

**Introduced in R2012a**

## stepinfo

Rise time, settling time, and other step response characteristics

### Syntax

```
S = stepinfo(y,t,yfinal)
S = stepinfo(y,t)
S = stepinfo(y)
S = stepinfo(sys)
S = stepinfo(...,'SettlingTimeThreshold',ST)
S = stepinfo(...,'RiseTimeLimits',RT)
```

### Description

`S = stepinfo(y,t,yfinal)` takes step response data (`t,y`) and a steady-state value `yfinal` and returns a structure `S` containing the following performance indicators:

- `RiseTime` — Rise time
- `SettlingTime` — Settling time
- `SettlingMin` — Minimum value of `y` once the response has risen
- `SettlingMax` — Maximum value of `y` once the response has risen
- `Overshoot` — Percentage overshoot (relative to `yfinal`)
- `Undershoot` — Percentage undershoot
- `Peak` — Peak absolute value of `y`
- `PeakTime` — Time at which this peak is reached

For SISO responses, `t` and `y` are vectors with the same length `NS`. For systems with `NU` inputs and `NY` outputs, you can specify `y` as an `NS`-by-`NY`-by-`NU` array (see `step`) and `yfinal` as an `NY`-by-`NU` array. `stepinfo` then returns a `NY`-by-`NU` structure array `S` of performance metrics for each I/O pair.

`S = stepinfo(y,t)` uses the last sample value of `y` as steady-state value `yfinal`. `S = stepinfo(y)` assumes `t = 1:ns`.



`S = stepinfo(sys)` computes the step response characteristics for an LTI model `sys` (see `tf`, `zpk`, or `ss` for details).

`S = stepinfo(..., 'SettlingTimeThreshold', ST)` lets you specify the threshold `ST` used in the settling time calculation. The response has settled when the error  $|y(t) - y_{\text{final}}|$  becomes smaller than a fraction `ST` of its peak value. The default value is `ST=0.02` (2%).

`S = stepinfo(..., 'RiseTimeLimits', RT)` lets you specify the lower and upper thresholds used in the rise time calculation. By default, the rise time is the time the response takes to rise from 10 to 90% of the steady-state value (`RT=[0.1 0.9]`). Note that `RT(2)` is also used to calculate `SettlingMin` and `SettlingMax`.

## Examples

### Step Response Characteristics of Fifth-Order System

Create a fifth order system and ascertain the response characteristics.

```
sys = tf([1 5],[1 2 5 7 2]);
S = stepinfo(sys, 'RiseTimeLimits', [0.05,0.95])
```

These commands return the following result:

```
S =
    RiseTime: 7.4454
    SettlingTime: 13.9378
    SettlingMin: 2.3737
    SettlingMax: 2.5201
    Overshoot: 0.8032
    Undershoot: 0
    Peak: 2.5201
    PeakTime: 15.1869
```

### See Also

`lsiminfo` | `step`

Introduced in R2006a

## stepplot

Plot step response and return plot handle

### Syntax

```
h = stepplot(sys)
stepplot(sys,Tfinal)
stepplot(sys,t)
stepplot(sys1,sys2,...,sysN)
stepplot(sys1,sys2,...,sysN,Tfinal)
stepplot(sys1,sys2,...,sysN,t)
stepplot(AX,...)
stepplot(..., plotoptions)
stepplot(..., dataoptions)
```

### Description

`h = stepplot(sys)` plots the step response of the dynamic system model `sys`. It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

For multiinput models, independent step commands are applied to each input channel. The time range and number of points are chosen automatically.

`stepplot(sys,Tfinal)` simulates the step response from `t = 0` to the final time `t = Tfinal`. Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sample time (`Ts = -1`), `stepplot` interprets `Tfinal` as the number of sampling intervals to simulate.

`stepplot(sys,t)` uses the user-supplied time vector `t` for simulation. Express `t` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time

models,  $t$  should be of the form  $T_i:T_s:T_f$ , where  $T_s$  is the sample time. For continuous-time models,  $t$  should be of the form  $T_i:dt:T_f$ , where  $dt$  becomes the sample time of a discrete approximation to the continuous system (see `step`). The `stepplot` command always applies the step input at  $t=0$ , regardless of  $T_i$ .

To plot the step responses of multiple models `sys1,sys2,...` on a single plot, use:

```
stepplot(sys1,sys2,...,sysN)
```

```
stepplot(sys1,sys2,...,sysN,Tfinal)
```

```
stepplot(sys1,sys2,...,sysN,t)
```

You can also specify a color, line style, and marker for each system, as in

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

`stepplot(AX,...)` plots into the axes with handle `AX`.

`stepplot(..., plotoptions)` customizes the plot appearance using the options set, `plotoptions`. Use `timeOptions` to create the options set.

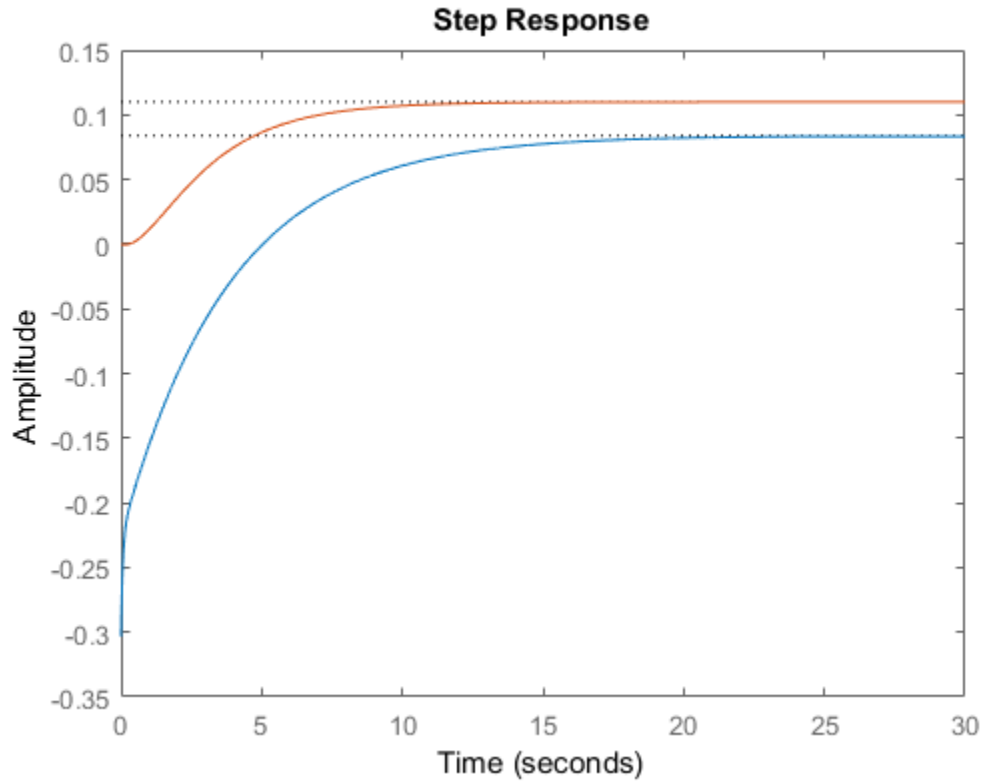
`stepplot(..., dataoptions)` specifies options such as the step amplitude and input offset using the options set, `dataoptions`. Use `stepDataOptions` to create the options set.

## Examples

### Normalized Response on Step Plot

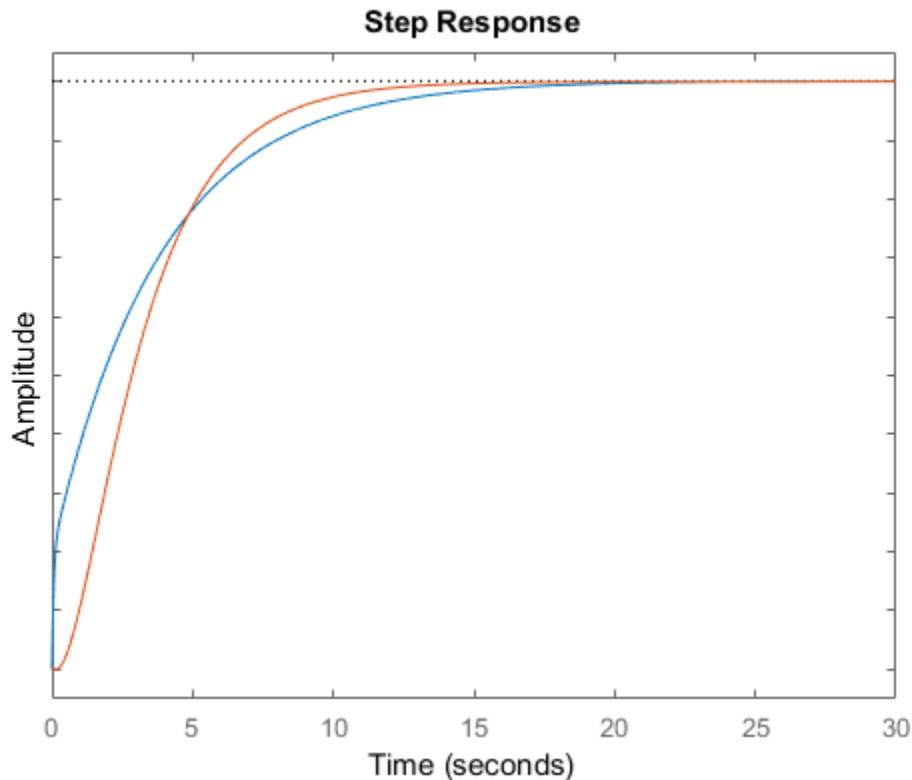
Generate a step response plot for two dynamic systems.

```
sys1 = rss(3);  
sys2 = rss(3);  
h = stepplot(sys1,sys2);
```



Each step response settles at a different steady-state value. Use the plot handle to normalize the plotted response.

```
setoptions(h, 'Normalize', 'on')
```



Now, the responses settle at the same value expressed in arbitrary units.

## Step Responses of Identified Models with Confidence Region

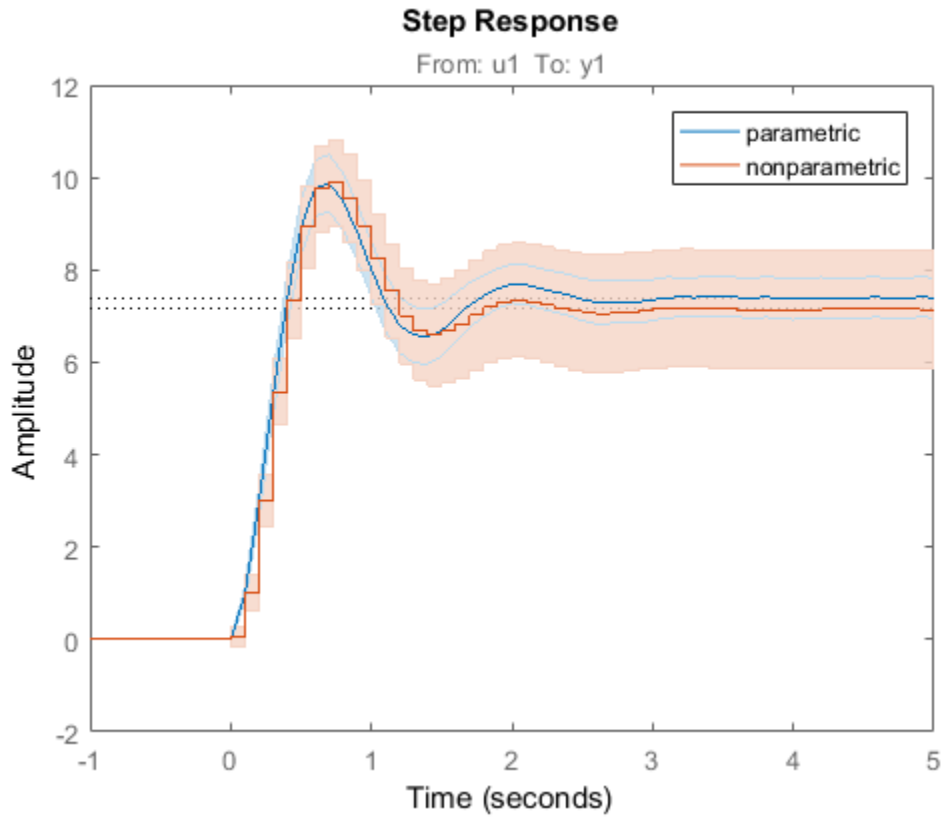
Compare the step response of a parametric identified model to a nonparametric (empirical) model, and view their 3- $\sigma$  confidence regions. (Identified models require System Identification Toolbox™ software.)

Identify a parametric and a nonparametric model from sample data.

```
load iddata1 z1
sys1 = ssest(z1,4);
sys2 = impulseest(z1);
```

Plot the step responses of both identified models. Use the plot handle to display the 3- $\sigma$  confidence regions.

```
t = -1:0.1:5;  
h = stepplot(sys1,sys2,t);  
showConfidence(h,3)  
legend('parametric','nonparametric')
```



The nonparametric model `sys2` shows higher uncertainty.

## Step Response of Nonlinear Model

Plot the step response of a nonlinear (Hammerstein-Wiener) model using a starting offset of 2 and step amplitude of 0.5. (Hammerstein-Weiner models require System Identification Toolbox software.)

```
load twotankdata
z = iddata(y, u, 0.2, 'Name', 'Two tank system');
sys = nlhw(z, [1 5 3], pwlinear, poly1d);

dataoptions = stepDataOptions('InputOffset', 2, 'StepAmplitude', 0.5);
stepplot(sys,60,dataoptions);
```

## More About

### Tips

You can change the properties of your plot, for example the units. For information on the ways to change properties of your plots, see “Ways to Customize Plots”.

### See Also

`setoptions` | `getoptions` | `step`

**Introduced before R2006a**

# strseq

Create sequence of indexed character vectors

## Syntax

```
txtarray = strseq(TXT,INDICES)
```

## Description

`txtarray = strseq(TXT,INDICES)` creates a sequence of indexed character vectors in the cell array `txtarray` by appending the integer values `INDICES` to the character vector `TXT`.

---

**Note:** You can use `strvec` to aid in system interconnection. For an example, see the [sumblk](#) reference page.

---

## Examples

### Create a Cell Array of Indexed Text

Index the text 'e' with the numbers 1, 2, and 4.

```
txtarray = strseq('e',[1 2 4])
```

```
txtarray =
```

```
3×1 cell array
```

```
'e1'
```

```
'e2'
```

```
'e4'
```

### See Also

[strcat](#) | [connect](#)



**Introduced in R2008b**

## sumblk

Summing junction for name-based interconnections

### Syntax

```
S = sumblk(formula)
S = sumblk(formula,signalsize)
S = sumblk(formula,signames1,signames2,...)
```

### Description

`S = sumblk(formula)` creates the transfer function, `S`, of the summing junction described by `formula`. The character vector `formula` specifies an equation that relates the scalar input and output signals of `S`.

`S = sumblk(formula,signalsize)` returns a vector-valued summing junction. The input and output signals are vectors with `signalsize` elements.

`S = sumblk(formula,signames1,signames2,...)` replaces aliases (signal names beginning with %) in `formula` by the signal names `signames`. The number of `signames` arguments must match the number of aliases in `formula`. The first alias in `formula` is replaced by `signames1`, the second by `signames2`, and so on.

### Input Arguments

#### **formula**

Equation that relates the input and output signals of the summing junction transfer function `S`, specified as a character vector. For example, the following command:

```
S = sumblk('e = r - y + d')
```

creates a summing junction with input names 'r', 'y', and 'd', output name 'e' and equation  $e = r - y + d$ .

If you specify a **signalsize** greater than 1, the inputs and outputs of **S** are vector-valued signals. **sumblk** automatically performs vector expansion of the signal names of **S**. For example, the following command:

```
S = sumblk('v = u + d',2)
```

specifies a summing junction with input names {'u(1)';'u(2)';'d(1)';'d(2)'} and output names {'v(1)';'v(2)'}. The formulas of this summing junction are  $v(1) = u(1) + d(1)$ ;  $v(2) = u(2) + d(2)$ .

You can use one or more aliases in **formula** to refer to signal names defined in a variable. An alias is a signal name that begins with **%**. When **formula** contains aliases, **sumblk** replaces each alias with the corresponding **signames** argument.

Aliases are useful when you want to name individual entries in a vector-valued signal. Aliases also allow you to use input or output names of existing models. For example, if **C** and **G** are dynamic system models with nonempty **InputName** and **OutputName** properties, respectively, you can create a summing junction using the following expression.

```
S = sumblk('%e = r - %y',C.InputName,G.OutputName)
```

**sumblk** uses the values of **C.InputName** and **G.OutputName** in place of **%e** and **%y**, respectively. The vector dimension of **C.InputName** and **G.OutputName** must match. **sumblk** assigns the signal **r** the same dimension.

### **signalsize**

Number of elements in each input and output signal of **S**. Setting **signalsize** greater than 1 lets you specify a summing junction that operates on vector-valued signals.

**Default:** 1

### **signames**

Signal names to replace one alias (signal name beginning with **%**) in the argument **formula**. You must provide one **signames** argument for each alias in **formula**.

Specify **signames** as:

- A cell array of signal names.
- The **InputName** or **OutputName** property of a model in the MATLAB workspace. For example:

```
S = sumblk('%e = r - y',C.InputName)
```

This command creates a summing junction whose outputs have the same name as the inputs of the model `C` in the MATLAB workspace.

## Output Arguments

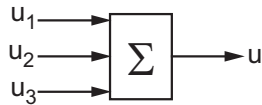
### **s**

Transfer function for the summing junction, represented as a MIMO `tf` model object.

## Examples

### Summing Junction with Scalar-Valued Signals

Create the summing junction of the following illustration. All signals are scalar-valued.



This summing junction has the formula  $u = u_1 + u_2 + u_3$ .

```
S = sumblk('u = u1+u2+u3');
```

`S` is the transfer function (`tf`) representation of the sum  $u = u_1 + u_2 + u_3$ . The transfer function `S` gets its input and output names from the formula.

```
S.OutputName,S.Inputname
```

```
ans =
```

```
    'u'
```

```
ans =
```

```
    'u1'
```

```
    'u2'
```

```
'u3'
```

### Summing Junction with Vector-Valued Signals

Create the summing junction  $v = u - d$  where  $u, d, v$  are vector-valued signals of length 2.

```
S = sumblk('v = u-d',2);
```

sumblk automatically performs vector expansion of the signal names of S.

```
S.OutputName,S.Inputname
```

```
ans =
```

```
'v(1)'  
'v(2)'
```

```
ans =
```

```
'u(1)'  
'u(2)'  
'd(1)'  
'd(2)'
```

### Summing Junction with Vector-Valued Signals That Have Specified Signal Names

Create the summing junction

$$e(1) = \text{setpoint}(1) - \text{alpha} + d(1)$$

$$e(2) = \text{setpoint}(2) - q + d(2)$$

The signals `alpha` and `q` have custom names that are not merely the vector expansion of a single signal name. Therefore, use an alias in the formula specifying the summing junction.

```
S = sumblk('e = setpoint - %y + d', {'alpha';'q'});
```

sumblk replaces the alias `%y` with the cell array `{'alpha';'q'}`.

```
S.OutputName,S.Inputname
```

```
ans =
```

```
    'e(1)'  
    'e(2)'
```

```
ans =
```

```
    'setpoint(1)'  
    'setpoint(2)'  
    'alpha'  
    'q'  
    'd(1)'  
    'd(2)'
```

## More About

### Tips

- Use `sumbk` in conjunction with `connect` to interconnect dynamic system models and derive aggregate models for block diagrams.
- “Multi-Loop Control System”
- “MIMO Control System”

### See Also

`connect` | `series` | `parallel` | `strseq`

**Introduced in R2008a**

## systeme

Tune fixed-structure control systems modeled in MATLAB

`systeme` tunes fixed-structure control systems subject to both soft and hard design goals. `systeme` can tune multiple fixed-order, fixed-structure control elements distributed over one or more feedback loops. For an overview of the tuning workflow, see “Automated Tuning Workflow”.

This command tunes control systems modeled in MATLAB. For tuning Simulink models, use `sITuner` to create an interface to your Simulink model. You can then tune the control system with `systeme` for `sITuner` (requires Simulink Control Design).

## Syntax

```
[CL,fSoft] = systeme(CLO,SoftReqs)
[CL,fSoft,gHard] = systeme(CLO,SoftReqs,HardReqs)
[CL,fSoft,gHard] = systeme(CLO,SoftReqs,HardReqs,options)
[CL,fSoft,gHard,info] = systeme( ___ )
```

## Description

`[CL,fSoft] = systeme(CLO,SoftReqs)` tunes the free parameters of the control system model, `CLO`, to best meet the soft tuning requirements. The best achieved soft constraint values are returned as `fSoft`. For robust tuning against real parameter uncertainty, use a control system model with uncertain real parameters. For robust tuning against a set of plant models, use an array of control system models `CLO`. (See “Input Arguments” on page 2-1016.)

`[CL,fSoft,gHard] = systeme(CLO,SoftReqs,HardReqs)` tunes the control system to best meet the soft tuning requirements subject to satisfying the hard tuning requirements (constraints). It returns the best achieved values for the soft and hard constraints.

`[CL,fSoft,gHard] = systeme(CLO,SoftReqs,HardReqs,options)` specifies options for the optimization.

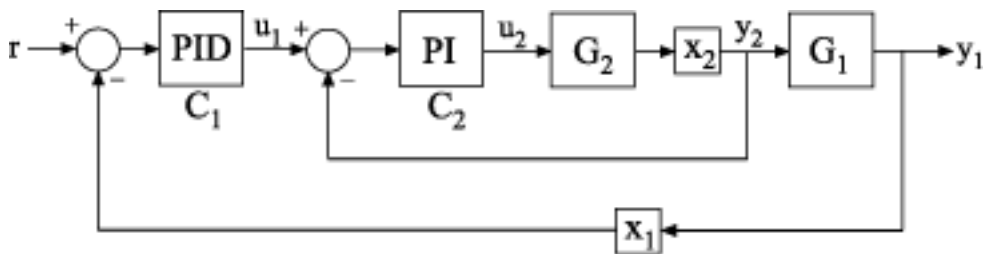
[CL, fSoft, gHard, info] = systune( \_\_\_ ) also returns detailed information about each optimization run. All input arguments described for the previous syntaxes also apply here.

## Examples

### Tune Control System to Soft Requirements

Tune a cascaded control system to meet requirements of reference tracking and disturbance rejection.

The cascaded control system of the following illustration includes two tunable controllers, the PI controller for the inner loop,  $C_2$ , and the PID controller for the outer loop,  $C_1$ .



The blocks  $x_1$  and  $x_2$  mark analysis-point locations. These are locations at which loops can be opened or signals injected for the purpose of specifying requirements for tuning the system.

Tune the free parameters of this control system to meet the following requirements:

- The output signal,  $y_1$ , tracks the reference signal,  $r$ , with a response time of 10 seconds and a steady-state error of 1%.
- A disturbance injected at  $x_2$  is suppressed at  $y_1$  by a factor of 10.

Create tunable Control Design Blocks to represent the controllers, and numeric LTI models to represent the plants. Also, create AnalysisPoint blocks to mark the points of interest in each feedback loop.

```
G2 = zpk([], -2, 3);
G1 = zpk([], [-1 -1 -1], 10);
```



```
C20 = tunablePID('C2','pi');
C10 = tunablePID('C1','pid');
```

```
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
```

Connect these components to build a model of the entire closed-loop control system.

```
InnerLoop = feedback(X2*G2*C20,1);
CLO = feedback(G1*InnerLoop*C10,X1);
CLO.InputName = 'r';
CLO.OutputName = 'y';
```

CLO is a tunable `genss` model. Specifying names for the input and output channels allows you to identify them when you specify tuning requirements for the system.

Specify tuning requirements for reference tracking and disturbance rejection.

```
Rtrack = TuningGoal.Tracking('r','y',10,0.01);
Rreject = TuningGoal.Gain('X2','y',0.1);
```

The `TuningGoal.Tracking` requirement specifies that the signal at 'y' track the signal at 'r' with a response time of 10 seconds and a tracking error of 1%.

The `TuningGoal.Gain` requirement limits the gain from the implicit input associated with the `AnalysisPoint` block, X2, to 'y'. (See `AnalysisPoint`.) Limiting this gain to a value less than 1 ensures that a disturbance injected at X2 is suppressed at the output.

Tune the control system.

```
[CL,fSoft] = systune(CLO,[Rtrack,Rreject]);
```

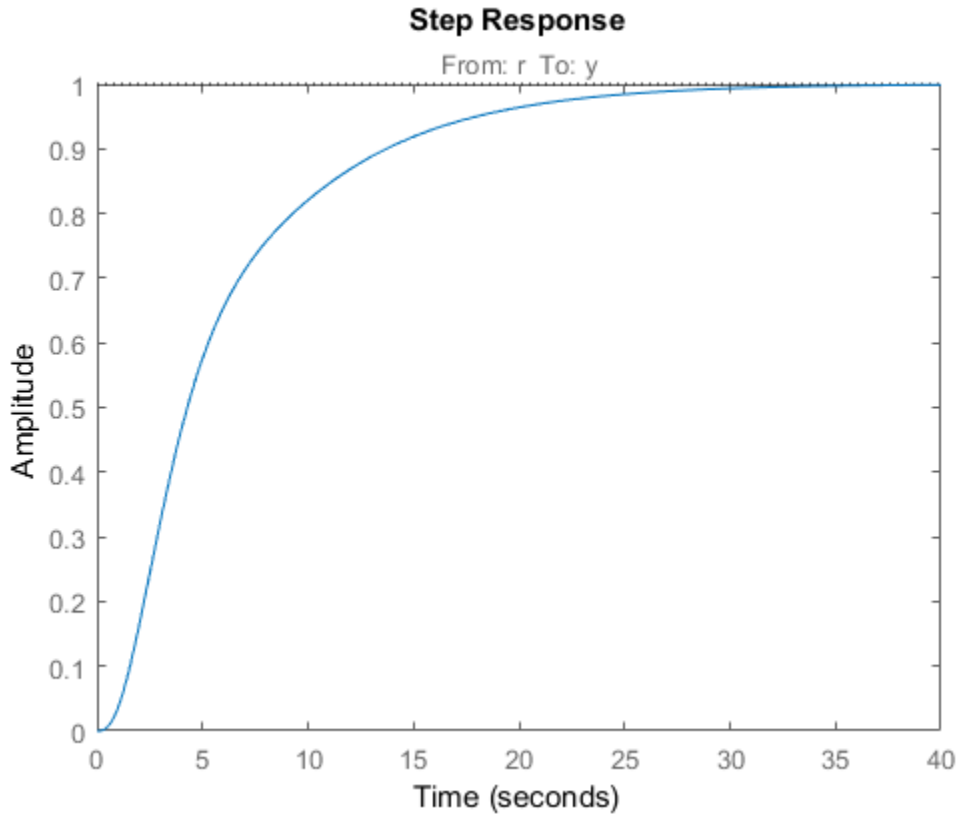
```
Final: Soft = 1.24, Hard = -Inf, Iterations = 87
```

`systune` converts each tuning requirement into a normalized scalar value,  $f$ . The command adjusts the tunable parameters of CLO to minimize the  $f$  values. For each requirement, the requirement is satisfied if  $f < 1$  and violated if  $f > 1$ . `fSoft` is the vector of minimized  $f$  values. The largest of the minimized  $f$  values is displayed as `Soft`.

The output model CL is the tuned version of CLO. CL contains the same Control Design Blocks as CLO, with current values equal to the tuned parameter values.

Validate that the tuned control system meets the tracking requirement by examining the step response from 'r' to 'y'.

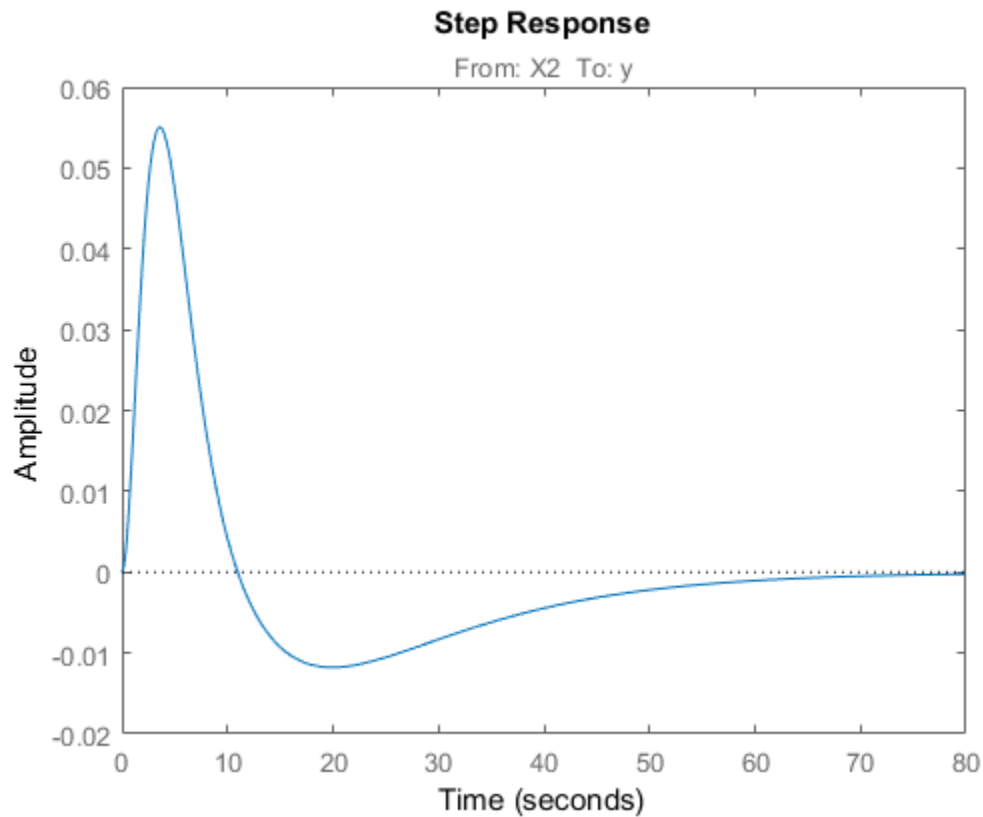
```
stepplot(CL)
```



The step plot shows that in the tuned control system, CL, the output tracks the input with approximately the desired response time.

Validate the tuned system against the disturbance rejection requirement by examining the closed-loop response to a signal injected at X2.

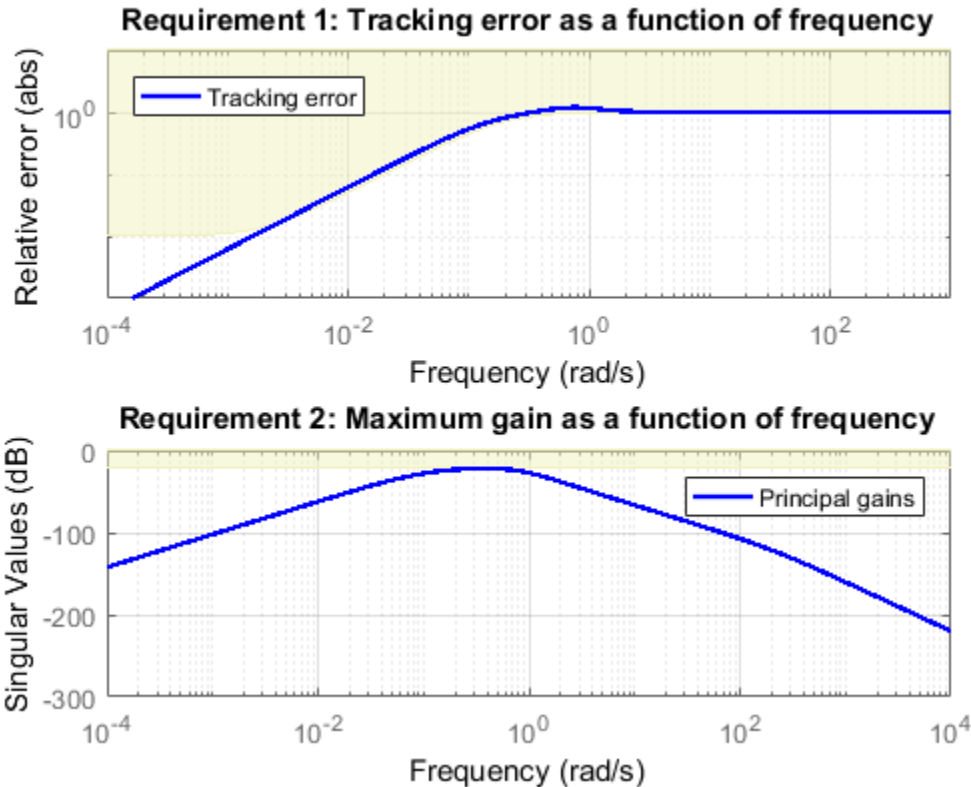
```
CLdist = getIOTransfer(CL, 'X2', 'y');  
stepplot(CLdist);
```



`getIOTransfer` extracts the closed-loop response from the specified inputs to outputs. In general, `getIOTransfer` and `getLoopTransfer` are useful for validating a control system tuned with `systune`.

You can also use `viewSpec` to compare the responses of the tuned control system directly against the tuning requirements, `Rtrack` and `Rreject`.

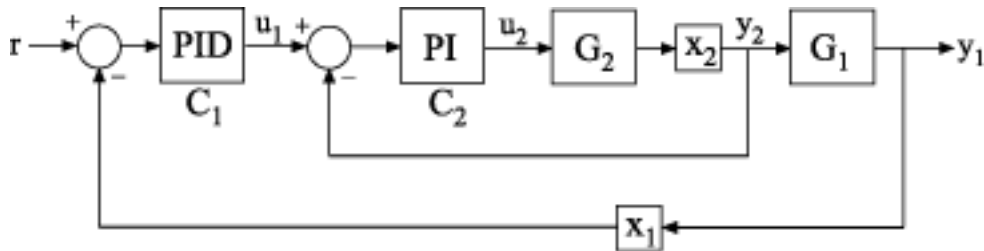
```
viewSpec([Rtrack,Rreject],CL)
```



### Tune Control System to Both Hard and Soft Requirements

Tune a cascaded control system to meet requirements of reference tracking and disturbance rejection. These requirements are subject to a hard constraint on the stability margins of the inner and outer loops.

The cascaded control system of the following illustration includes two tunable controllers, the PI controller for the inner loop,  $C_2$ , and the PID controller for the outer loop,  $C_1$ .



The blocks  $x_1$  and  $x_2$  mark analysis-point locations. These are locations at which you can open loops or inject signals for the purpose of specifying requirements for tuning the system.

Tune the free parameters of this control system to meet the following requirements:

- The output signal,  $y_1$ , tracks the reference signal at  $r$  with a response time of 5 seconds and a steady-state error of 1%.
- A disturbance injected at  $x_2$  is suppressed at the output,  $y_1$ , by a factor of 10.

Impose these tuning requirements subject to hard constraints on the stability margins of both loops.

Create tunable Control Design Blocks to represent the controllers and numeric LTI models to represent the plants. Also, create `AnalysisPoint` blocks to mark the points of interest in each feedback loop.

```
G2 = zpk([], -2, 3);
G1 = zpk([], [-1 -1 -1], 10);
```

```
C20 = tunablePID('C2', 'pi');
C10 = tunablePID('C1', 'pid');
```

```
X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
```

Connect these components to build a model of the entire closed-loop control system.

```
InnerLoop = feedback(X2*G2*C20, 1);
CLO = feedback(G1*InnerLoop*C10, X1);
CLO.InputName = 'r';
CLO.OutputName = 'y';
```

`CLO` is a tunable `genss` model. Specifying names for the input and output channels allows you to identify them when you specify tuning requirements for the system.

Specify tuning requirements for reference tracking and disturbance rejection.

```
Rtrack = TuningGoal.Tracking('r','y',5,0.01);  
Rreject = TuningGoal.Gain('X2','y',0.1);
```

The `TuningGoal.Tracking` requirement specifies that the signal at 'y' tracks the signal at 'r' with a response time of 5 seconds and a tracking error of 1%.

The `TuningGoal.Gain` requirement limits the gain from the implicit input associated with the `AnalysisPoint` block X2 to the output, 'y'. (See `AnalysisPoint`.) Limiting this gain to a value less than 1 ensures that a disturbance injected at X2 is suppressed at the output.

Specify tuning requirements for the gain and phase margins.

```
RmargOut = TuningGoal.Margins('X1',18,60);  
RmargIn = TuningGoal.Margins('X2',18,60);  
RmargIn.Openings = 'X1';
```

`RmargOut` imposes a minimum gain margin of 18 dB and a minimum phase margin of 60 degrees. Specifying X1 imposes that requirement on the outer loop. Similarly, `RmargIn` imposes the same requirements on the inner loop, identified by X2. To ensure that the inner-loop margins are evaluated with the outer loop open, include the outer-loop analysis-point location, X1, in `RmargIn.Openings`.

Tune the control system to meet the soft requirements of tracking and disturbance rejection, subject to the hard constraints of the stability margins.

```
SoftReqs = [Rtrack,Rreject];  
HardReqs = [RmargIn,RmargOut];  
[CL,fSoft,gHard] = systune(CLO,SoftReqs,HardReqs);
```

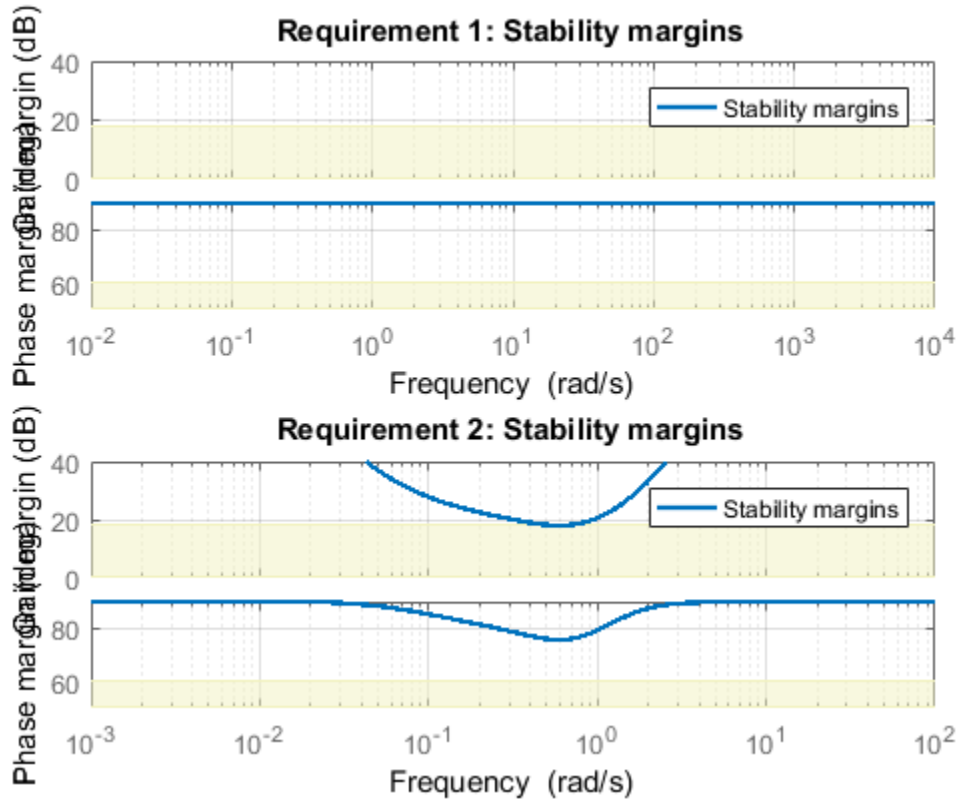
```
Final: Soft = 3.35, Hard = 0.99984, Iterations = 212
```

`systune` converts each tuning requirement into a normalized scalar value,  $f$  for the soft constraints and  $g$  for the hard constraints. The command adjusts the tunable parameters of CLO to minimize the  $f$  values, subject to the constraint that each  $g < 1$ .

The displayed value `Hard` is the largest of the minimized  $g$  values in `gHard`. This value is less than 1, indicating that both the hard constraints are satisfied.

Validate the tuned control system against the stability margin requirements.

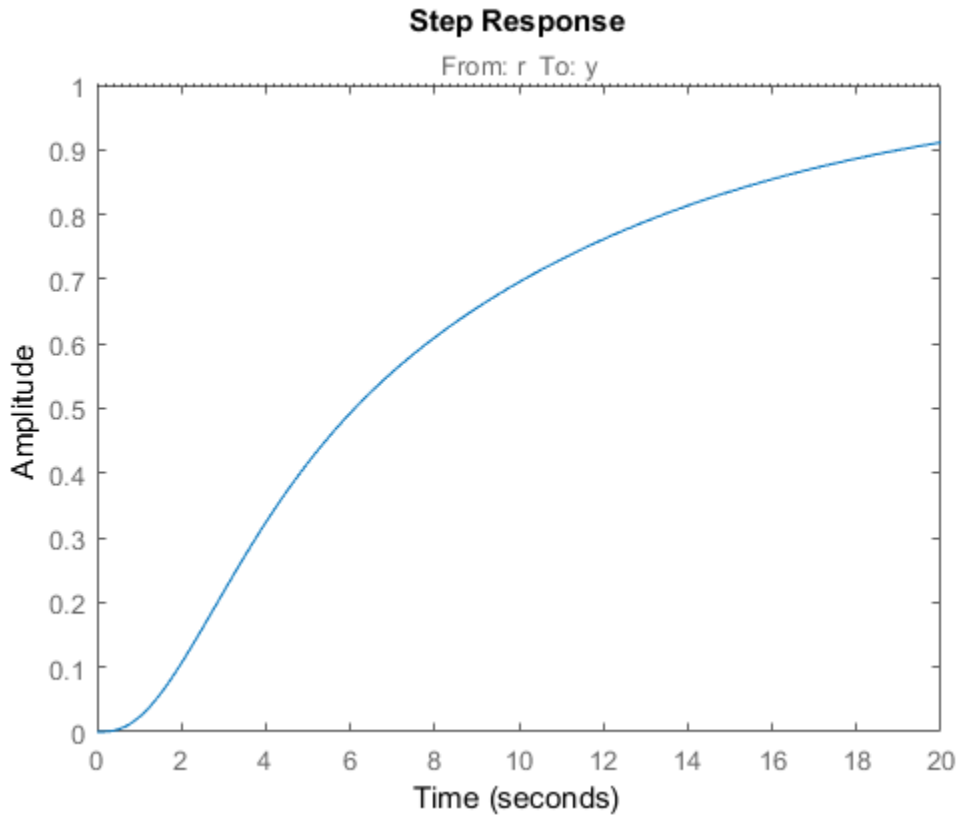
```
figure;
viewSpec(HardReqs,CL)
```



The `viewSpec` plot confirms that the stability margin requirements for both loops are satisfied by the tuned control system at all frequencies. The red lines represent the actual stability margins of the tuned system. The blue lines represent the margin used in the optimization calculation, which is an upper bound on the actual margin.

Examine whether the tuned control system meets the tracking requirement by examining the step response from 'r' to 'y'.

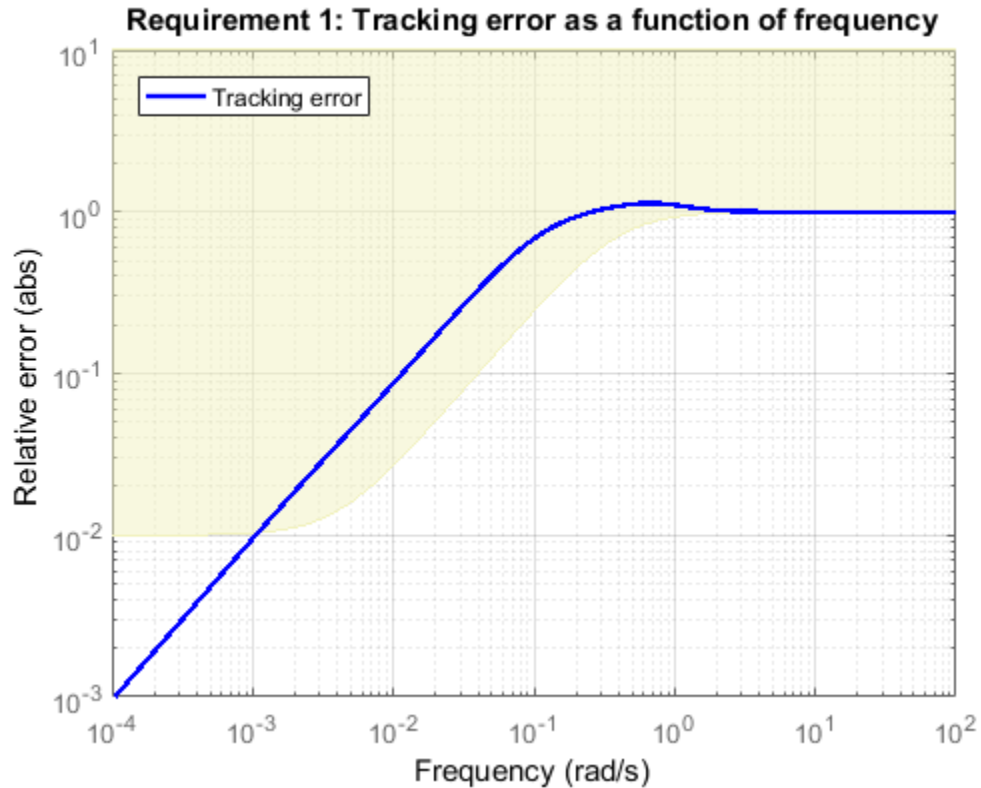
```
figure;
stepplot(CL,20)
```



The step plot shows that in the tuned control system, CL, the output tracks the input but the response is somewhat slower than desired and the tracking error may be larger than desired. For further information, examine the tracking requirement directly with `viewSpec`.

```
figure;  
viewSpec(Rtrack,CL)
```





The actual tracking error crosses into the shaded area between 1 and 10 rad/s, indicating that the requirement is not met in this regime. Thus, the tuned control system cannot meet the soft tracking requirement, time subject to the hard constraints of the stability margins. To achieve the desired performance, you may need to relax one of your requirements or convert one or more hard constraints to soft constraints.

- “Tuning Control Systems with SYSTUNE”
- “Building Tunable Models”

# Input Arguments

## **CLO** — Control system to tune

generalized state-space model | model array

Control system to tune, specified as a generalized state-space (**genss**) model or array of models with tunable parameters. To construct **CLO**:

- 1 Parameterize the tunable elements of your control system. You can use predefined structures, such as **tunablePID**, **tunableGain**, and **tunableTF**. Alternatively, you can create your own structure from elementary tunable parameters (**realp**).
- 2 Build a closed-loop model of the overall control system as an interconnection of fixed and tunable components. To do so, use model interconnection commands such as **feedback** and **connect**. Use **AnalysisPoint** blocks to mark additional signals of interest for specifying and assessing tuning requirements.

For more information about creating models to tune, see “Setup for Tuning Control System Modeled in MATLAB”.

For robust tuning of a control system against a set of plant models (requires Robust Control Toolbox), specify an array of tunable **genss** models that have the same tunable parameters. To make the controller robust against parameter uncertainty, use a model with uncertain real parameters defined with **ureal** or **uss**. In this case, **CLO** is a **genss** model that contains both tunable and uncertain control design blocks. For more information about robust tuning, see “Robust Tuning Approaches”.

## **SoftReqs** — Soft tuning goals (objectives)

vector of **TuningGoal** objects

Soft tuning goals (objectives) for tuning the control system, specified as a vector of **TuningGoal** objects. These objects capture your design requirements, such as **TuningGoal.Tracking**, **TuningGoal.StepTracking**, or **TuningGoal.Margins**.

**systemtune** tunes the tunable parameters of the control system to minimize the soft tuning goals. This tuning is subject to satisfying the hard tuning goals (if any).

For more information about available tuning goals, see “Tuning Goals”.

## **HardReqs** — Hard tuning goals (constraints)

[ ] (default) | vector of **TuningGoal** objects

Hard tuning goals (constraints) for tuning the control system, specified as a vector of `TuningGoal` objects. These objects capture your design requirements, such as `TuningGoal.Tracking`, `TuningGoal.StepTracking`, or `TuningGoal.Margins`.

`systeme` converts each hard tuning goal to a normalized scalar value. `systeme` then optimizes the free parameters to minimize those normalized values. A hard goal is satisfied if the normalized value is less than 1.

For more information about available tuning goals, see “Tuning Goals”.

### **options** — Options for tuning algorithm

`systemeOptions` object

Options for the tuning algorithm, specified as an options set you create with `systemeOptions`. Available options include:

- Number of additional optimizations to run. Each optimization starts from random initial values of the free parameters.
- Tolerance for terminating the optimization.
- Flag for using parallel processing.

See the `systemeOptions` reference page for more details about all available options.

## Output Arguments

### **CL** — Tuned control system

generalized state-space model

Tuned control system, returned as a generalized state-space (`genss`) model. This model has the same number and type of tunable elements (Control Design Blocks) as `CL0`. The current values of these elements are the tuned parameters. Use `getBlockValue` or `showTunable` to access values of the tuned elements.

If you provide an array of control system models to tune as the input argument, `CL0`, `systeme` tunes the parameters of all the models simultaneously. In this case, `CL` is an array of tuned `genss` models. For more information, see “Robust Tuning Approaches”.

### **fSoft** — Best achieved soft constraint values

vector

Best achieved soft constraint values, returned as a vector. `systeme` converts the soft requirements to a function of the free parameters of the control system. The command then tunes the parameters to minimize that function subject to the hard constraints. (See “Algorithms” on page 2-1021.) `fSoft` contains the best achieved value for each of the soft constraints. These values appear in `fSoft` in the same order that the constraints are specified in `SoftReqs`. `fSoft` values are meaningful only when the hard constraints are satisfied.

### **gHard — Best achieved hard constraint values**

vector

Best achieved hard constraint values, returned as a vector. `systeme` converts the hard requirements to a function of the free parameters of the control system. The command then tunes the parameters to drive those values below 1. (See “Algorithms” on page 2-1021.) `gHard` contains the best achieved value for each of the hard constraints. These values appear in `gHard` in the same order that the constraints are specified in `HardReqs`. If all values are less than 1, then the hard constraints are satisfied.

### **info — Detailed information about optimization runs**

structure

Detailed information about each optimization run, returned as a data structure. In addition to examining detailed results of the optimization, you can use `info` as an input to `viewSpec` or `evalSpec` to maintain consistency after modifying the closed-loop system with `usample`, `usubs`, or `setBlockValue`.

The fields of `info` are:

#### **Run — Run number**

scalar

Run number, returned as a scalar. If you use the `RandomStart` option of `systemeOptions` to perform multiple optimization runs, `info` is a struct array, and `info.Run` is the index.

#### **Iterations — Total number of iterations**

scalar

Total number of iterations performed during run, returned as a scalar. This value is the number of iterations performed in each run before the optimization terminates.

#### **fBest — Best overall soft constraint value**

scalar

Best overall soft constraint value, returned as a scalar. **systeme** converts the soft requirements to a function of the free parameters of the control system. The command then tunes the parameters to minimize that function subject to the hard constraints. (See “Algorithms” on page 2-1021.) `info.fBest` is the maximum soft constraint value at the final iteration. This value is meaningful only when the hard constraints are satisfied.

### **gBest — Best overall hard constraint value**

scalar

Best overall hard constraint value, returned as a scalar. **systeme** converts the hard requirements to a function of the free parameters of the control system. The command then tunes the parameters to drive those values below 1. (See “Algorithms” on page 2-1021.) `info.gBest` is the maximum hard constraint value at the final iteration. This value must be less than 1 for the hard constraints to be satisfied.

### **fSoft — Individual soft constraint values**

vector

Individual soft constraint values, returned as a vector. **systeme** converts each soft requirement to a normalized value that is a function of the free parameters of the control system. The command then tunes the parameters to minimize that value subject to the hard constraints. (See “Algorithms” on page 2-1021.) `info.fSoft` contains the individual values of the soft constraints at the end of each run. These values appear in `fSoft` in the same order that the constraints are specified in `SoftReqs`.

### **gHard — Individual hard constraint values**

vector

Individual hard constraint values, returned as a vector. **systeme** converts each hard requirement to a normalized value that is a function of the free parameters of the control system. The command then tunes the parameters to minimize those values. A hard requirement is satisfied if its value is less than 1. (See “Algorithms” on page 2-1021.) `info.gHard` contains the individual values of the hard constraints at the end of each run. These values appear in `gHard` in the same order that the constraints are specified in `HardReqs`.

### **MinDecay — Minimum decay rate of closed-loop poles**

vector

Minimum decay rate of closed-loop poles, returned as a vector.

By default, closed-loop pole locations of the tuned system are constrained to satisfy  $\text{Re}(p) < -10^{-7}$ . Use the `MinDecay` option of `systuneOptions` to change this constraint.

### **Blocks — Tuned values of tunable blocks and parameters**

structure

Tuned values of tunable blocks and parameters in the tuned control system, `CL`, returned as a structure. You can also use `getBlockValue` or `showBlockValue` to access the tuned parameter values.

### **LoopScaling — Optimal diagonal scaling for MIMO tuning requirements**

state-space model

Optimal diagonal scaling for evaluating MIMO tuning requirements, returned as a state-space model.

When applied to multiloop control systems, `TuningGoal.LoopShape` and `TuningGoal.Margins` can be sensitive to the scaling of the loop transfer functions to which they apply. This sensitivity can lead to poor optimization results. `systune` automatically corrects scaling issues and returns the optimal diagonal scaling matrix `d` as a state-space model in `info.LoopScaling`.

The loop channels associated with each diagonal entry of `D` are listed in `info.LoopScaling.InputName`. The scaled loop transfer is  $D \setminus L * D$ , where `L` is the open-loop transfer measured at the locations `info.LoopScaling.InputName`.

### **wcPert — Worst combinations of uncertain parameters**

structure array

Worst combinations of uncertain parameters, returned as a structure array. (Applies for robust tuning of control systems with uncertainty only.) Each structure contains one set of uncertain parameter values. The perturbations with the worst performance are listed first.

### **wcf — Worst objective value**

positive scalar

Largest soft goal value over the uncertainty range when using the tuned controller. (Applies for robust tuning of control systems with uncertainty only.)

### **wcg — Worst constraint value**

positive scalar

Largest hard goal value over the uncertainty range when using the tuned controller. (Applies for robust tuning of control systems with uncertainty only.)

**wcDecay — Worst decay rate**

scalar

Smallest closed-loop decay rate over the uncertainty range when using the tuned controller. (Applies for robust tuning of control systems with uncertainty only.) A positive value indicates robust stability. See MinDecay option in `systemeOptions` for details.

## Alternative Functionality

### App

The Control System Tuner app provides a graphical interface to control system tuning.

## More About

### Algorithms

$x$  is the vector of tunable parameters in the control system to tune. `systeme` converts each soft and hard tuning requirement `SoftReqs(i)` and `HardReqs(j)` into normalized values  $f_i(x)$  and  $g_j(x)$ , respectively. `systeme` then solves the constrained minimization problem:

$$\text{Minimize } \max_i f_i(x) \text{ subject to } \max_j g_j(x) < 1, \text{ for } x_{\min} < x < x_{\max}.$$

$x_{\min}$  and  $x_{\max}$  are the minimum and maximum values of the free parameters of the control system.

When you use both soft and hard tuning goals, the software approaches this optimization problem by solving a sequence of unconstrained subproblems of the form:

$$\min_x \max(\alpha f(x), g(x)).$$

The software adjusts the multiplier  $\alpha$  so that the solution of the subproblems converges to the solution of the original constrained optimization problem.

`systeme` returns the control system with parameters tuned to the values that best solve the minimization problem. `systeme` also returns the best achieved values of  $f_i(x)$  and  $g_j(x)$ , as `fSoft` and `gHard` respectively.

For information about the functions  $f_i(x)$  and  $g_j(x)$  for each type of constraint, see the reference pages for each `TuningGoal` requirement object.

`systeme` uses the nonsmooth optimization algorithms described in [1],[2],[3],[4]

`systeme` computes the  $H_\infty$  norm using the algorithm of [5] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see <http://slicot.org>.

- “Programmatic Tuning”
- “Generalized Models”
- “Robust Tuning Approaches”

## References

- [1] Apkarian, P. and D. Noll, "Nonsmooth H-infinity Synthesis," *IEEE Transactions on Automatic Control*, Vol. 51, No. 1, (2006), pp. 71–86.
- [2] Apkarian, P. and D. Noll, "Nonsmooth Optimization for Multiband Frequency-Domain Control Design," *Automatica*, 43 (2007), pp. 724–731.
- [3] Apkarian, P., P. Gahinet, and C. Buhr, "Multi-model, multi-objective tuning of fixed-structure controllers," *Proceedings ECC* (2014), pp. 856–861.
- [4] Apkarian, P., M.-N. Dao, and D. Noll, "Parametric Robust Structured Control Design," *IEEE Transactions on Automatic Control*, 2015.
- [5] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the  $H_\infty$ -Norm of a Transfer Function Matrix," *System Control Letters*, Vol. 14, No. 4 (1990), pp. 287–293.

## See Also

`TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Margins` | `AnalysisPoint` | `genss` | `looptune` | `looptune` (for `sITuner`) | `sITuner` | `systeme` (for `sITuner`) | `systemeOptions` | `viewSpec`



**Introduced in R2012b**

## systemeOptions

Set options for `systeme`

### Syntax

```
options = systemeOptions  
options = systemeOptions(Name,Value)
```

### Description

`options = systemeOptions` returns the default option set for the `systeme` command.

`options = systemeOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

### Input Arguments

#### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`systemeOptions` takes the following `Name` arguments:

#### 'Display'

Amount of information to display during `systeme` runs.

`Display` takes the following values:

- `'final'` — Display a one-line summary at the end of each optimization run. The display includes the best achieved values for the soft and hard constraints, `fSoft` and `gHard`. The display also includes the number of iterations for each run.

Example:

```
Final: Soft = 1.09, Hard = 0.68927, Iterations = 58
```

- 'sub' — Display the result of each optimization subproblem.

When you use both soft and hard tuning goals, the software solves the optimization as a sequence of subproblems of the form:

$$\min_x \max(\alpha f(x), g(x)).$$

Here,  $x$  is the vector of tunable parameters,  $f(x)$  is the largest normalized soft-constraint value, and  $g(x)$  is the largest normalized hard-constraint value. (See the “Algorithms” section of the `systeme` reference page for more information.) The software adjusts the multiplier  $\alpha$  so that the solution of the subproblems converges to the solution of the original constrained optimization problem. When you select 'sub', the report includes the results of each of these subproblems.

Example:

```
alpha=0.1: Soft = 3.97, Hard = 0.68927, Iterations = 8
alpha=0.5036: Soft = 1.36, Hard = 0.68927, Iterations = 8
alpha=1.47: Soft = 1.09, Hard = 0.68927, Iterations = 42
Final: Soft = 1.09, Hard = 0.68927, Iterations = 58
```

- 'iter' — Display optimization progress after each iteration. The display includes the value after each iteration of the objective parameter being minimized. The objective parameter is whichever is larger of  $\alpha f(x)$  and  $g(x)$ . The display also includes a progress value that indicates the percent change in the constraints from the previous iteration.

Example:

```
Iter 1: Objective = 4.664, Progress = 93%
Iter 2: Objective = 2.265, Progress = 51.4%
Iter 3: Objective = 0.7936, Progress = 65%
Iter 4: Objective = 0.7183, Progress = 9.48%
Iter 5: Objective = 0.6893, Progress = 4.04%
Iter 6: Objective = 0.6893, Progress = 0%
Iter 7: Objective = 0.6893, Progress = 0%
Iter 8: Objective = 0.6893, Progress = 0%
alpha=0.1: Soft = 3.97, Hard = 0.68927, Iterations = 8
```

```
Iter 1: Objective = 1.146, Progress = 42.7%
Iter 2: Objective = 1.01, Progress = 11.9%
...
alpha=1.47: Soft = 1.09, Hard = 0.68927, Iterations = 42
Final: Soft = 1.09, Hard = 0.68927, Iterations = 58
```

- `'off'` — Run in silent mode, displaying no information during or after the run.

**Default:** `'final'`

**'MaxIter'**

Maximum number of iterations in each optimization run, when the run does not converge to within tolerance.

**Default:** 300

**'RandomStart'**

Number of additional optimizations starting from random values of the free parameters in the controller.

If `RandomStart = 0`, `systeme` performs a single optimization run starting from the initial values of the tunable parameters. Setting `RandomStart = N > 0` runs  $N$  additional optimizations starting from  $N$  randomly generated parameter values.

`systeme` tunes by finding a local minimum of a gain minimization problem. To increase the likelihood of finding parameter values that meet your design requirements, set `RandomStart > 0`. You can then use the best design that results from the multiple optimization runs.

Use with `UseParallel = true` to distribute independent optimization runs among MATLAB workers (requires Parallel Computing Toolbox software).

**Default:** 0

**'UseParallel'**

Parallel processing flag.

Set to `true` to enable parallel processing by distributing randomized starts among workers in a parallel pool. If there is an available parallel pool, then the software

performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

- If **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, then the software starts a parallel pool using the settings in those preferences.
- If **Automatically create a parallel pool** is not selected in your preferences, then the software performs the optimization runs successively, without parallel processing.

If **Automatically create a parallel pool** is not selected in your preferences, you can manually start a parallel pool using `parpool` before running the tuning command.

Using parallel processing requires Parallel Computing Toolbox software.

**Default:** `false`

**'SoftTarget'**

Target value for soft constraints.

The optimization stops when the largest soft constraint value falls below the specified `SoftTarget` value. The default value `SoftTarget = 0` minimizes the soft constraints subject to satisfying the hard constraints.

**Default:** `0`

**'SoftTol'**

Relative tolerance for termination.

The optimization terminates when the relative decrease in the soft constraint value decreases by less than `SoftTol` over 10 consecutive iterations. Increasing `SoftTol` speeds up termination, and decreasing `SoftTol` yields tighter final values.

**Default:** `0.001`

**'SoftScale'**

A priori estimate of best soft constraint value.

For problems that mix soft and hard constraints, providing a rough estimate of the optimal value of the soft constraints (subject to the hard constraints) helps to speed up the optimization.

**Default:** 1

**'MinDecay'**

Minimum decay rate for stabilized dynamics.

Most tuning goals carry an implicit closed-loop stability or minimum-phase constraint. *Stabilized dynamics* refers to the poles and zeros affected by these constraints. The `MinDecay` option constrains all stabilized poles and zeros to satisfy:

- $\text{Re}(s) < -\text{MinDecay}$  (continuous time).
- $\log(|z|) < -\text{MinDecay}$  (discrete time).

Adjust the minimum value if the optimization fails to meet the default value, or if the default value conflicts with other requirements. Alternatively, use `TuningGoal.Poles` to control the decay rate of a specific feedback loop.

For more information about implicit constraints for a particular tuning goal, see the reference page for that tuning goal.

**Default:** 1e-7

**'MaxRadius'**

Maximum spectral radius for stabilized dynamics.

This option constrains all stabilized poles and zeros to satisfy  $|s| < \text{MaxRadius}$ . Stabilized dynamics are those poles and zeros affected by implicit stability or minimum-phase constraints of the tuning goals. The `MaxRadius` constraint is useful to prevent these poles and zeros from going to infinity as a result of algebraic loops becoming singular or control effort growing unbounded. Adjust the maximum radius if the optimization fails to meet the default value, or if the default value conflicts with other requirements.

`MaxRadius` is ignored for discrete-time tuning, where stability constraints already impose  $|z| < 1$ .

For more information about implicit constraints for a particular tuning goal, see the reference page for that tuning goal.

**Default:** 1e8

## Output Arguments

### options

Option set containing the specified options for the `systune` command.

## Examples

### Create Options Set for `systune`

Create an options set for a `systune` run using five random restarts. Also, set the display level to show the progress of each iteration, and increase the relative tolerance of the soft constraint value to 0.01.

```
options = systuneOptions('RandomStart',5,'Display','iter',...  
                        'SoftTol',0.01);
```

Alternatively, use dot notation to set the values of `options`.

```
options = systuneOptions;  
options.RandomStart = 5;  
options.Display = 'iter';  
options.SoftTol = 0.01;
```

### Configure Option Set for Parallel Optimization Runs

Configure an option set for a `systune` run using 20 random restarts. Execute these independent optimization runs concurrently on multiple workers in a parallel pool.

If you have the Parallel Computing Toolbox software installed, you can use parallel computing to speed up `systune` tuning of fixed-structure control systems. When you run multiple randomized `systune` optimization starts, parallel computing speeds up tuning by distributing the optimization runs among workers.

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences, manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a `systemOptions` set that specifies 20 random restarts to run in parallel.

```
options = systemOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the `systemOptions` set when you call `system`. For example, suppose you have already created a tunable control system model, `CLO`. For tuning this system, you have created vectors `SoftReqs` and `HardReqs` of `TuningGoal` requirements objects. These vectors represent your soft and hard constraints, respectively. In that case, the following command uses parallel computing to tune the control system of `CLO`.

```
[CL,fSoft,gHard] = system(CLO,SoftReqs,HardReqs,options);
```

### See Also

| `system` | `system` (for `slTuner`)

**Introduced in R2012b**



## tf

Create transfer function model, convert to transfer function model

### Syntax

```
sys = tf(Numerator,Denominator)
sys = tf(Numerator,Denominator,Ts)
sys = tf(M)
sys = tf(Numerator,Denominator,Itisys)
tfsys = tf(sys)
tfsys = tf(sys, 'measured')
tfsys = tf(sys, 'noise')
tfsys = tf(sys, 'augmented')
```

### Description

Use `tf` 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. You can also use `tf` to create generalized state-space (`genss`) models or uncertain state-space (`uss`) models.

### Creation of Transfer Functions

`sys = tf(Numerator,Denominator)` creates a continuous-time transfer function with numerator(s) and denominator(s) specified by `Numerator` and `Denominator`. The output `sys` is:

- A `tf` model object, when `Numerator` and `Denominator` are numeric arrays.
- A generalized state-space model (`genss`) when `Numerator` or `Denominator` include tunable parameters, such as `realp` parameters or generalized matrices (`genmat`).
- An uncertain state-space model (`uss`) when `Numerator` or `Denominator` are uncertain (requires Robust Control Toolbox software).

In the SISO case, `Numerator` and `Denominator` 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, using one of the following approaches:

- Concatenate SISO `tf` models.
- Use the `tf` command with cell array arguments. In this case, `Numerator` and `Denominator` are cell arrays of row vectors with as many rows as outputs and as many columns as inputs. The row vectors `Numerator{i,j}` and `Denominator{i,j}` specify the numerator and denominator of the transfer function from input `j` to output `i`.

For examples of creating MIMO transfer functions, see “Examples” on page 2-1034 and “MIMO Transfer Functions” in the *Control System Toolbox User Guide*.

If all SISO entries of a MIMO transfer function have the same denominator, you can set `denominator` to the row vector representation of this common denominator. See “Examples” for more details.

`sys = tf(Numerator,Denominator,Ts)` creates a discrete-time transfer function with sample time `Ts` (in seconds). Set `Ts = -1` to leave the sample time unspecified. The input arguments `Numerator` and `Denominator` 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(Numerator,Denominator,lthisys)` creates a transfer function with properties inherited from the dynamic system model `lthisys` (including the sample time).

There are several ways to create 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 “Model Arrays”.

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

`'Property',Value`

Each pair specifies a particular property of the model, for example, the input names or the transfer function variable. For information about the properties of `tf` objects, see “Properties” on page 2-1041. Note that

```
sys = tf(Numerator,Denominator,'Property1',Value1,...,'PropertyN',ValueN)
```

is a shortcut for

```
sys = tf(Numerator,Denominator)
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 the dynamic system model `sys` to transfer function form. The output `tfsys` is a `tf` model object representing `sys` expressed as a transfer function.

If `sys` is a model with tunable components, such as a `genss`, `genmat`, `tunableTF`, or `tunableSS` model, the resulting transfer function `tfsys` takes the current values of the tunable components.

## Conversion of Identified Models

An identified model is represented by an input-output equation of the form  $y(t) = Gu(t) + He(t)$ , where  $u(t)$  is the set of measured input channels and  $e(t)$  represents the noise channels. If  $A = LL'$  represents the covariance of noise  $e(t)$ , this equation can also be written as:  $y(t) = Gu(t) + HLv(t)$ , where  $\text{cov}(v(t)) = I$ .

`tfsys = tf(sys)`, or `tfsys = tf(sys, 'measured')` converts the measured component of an identified linear model into the transfer function form. `sys` is a model

of type `idss`, `idproc`, `idtf`, `idpoly`, or `idgrey`. `tfsys` represents the relationship between `u` and `y`.

`tfsys = tf(sys, 'noise')` converts the noise component of an identified linear model into the transfer function form. It represents the relationship between the noise input,  $v(t)$  and output,  $y_{\text{noise}} = HL v(t)$ . The noise input channels belong to the `InputGroup` 'Noise'. The names of the noise input channels are  $v@yname$ , where `yname` is the name of the corresponding output channel. `tfsys` has as many inputs as outputs.

`tfsys = tf(sys, 'augmented')` converts both the measured and noise dynamics into a transfer function. `tfsys` has  $ny+nu$  inputs such that the first  $nu$  inputs represent the channels  $u(t)$  while the remaining by channels represent the noise channels  $v(t)$ . `tfsys.InputGroup` contains 2 input groups- 'measured' and 'noise'. `tfsys.InputGroup.Measured` is set to  $1:nu$  while `tfsys.InputGroup.Noise` is set to  $nu+1:nu+ny$ . `tfsys` represents the equation  $y(t) = [G HL] [u; v]$ .

---

**Tip** An identified nonlinear model cannot be converted into a transfer function. Use linear approximation functions such as `linearize` and `linapp`.

---

## Creation of Generalized State-Space Models

You can use the syntax:

```
gensys = tf(Numerator,Denominator)
```

to create a Generalized state-space (`genss`) model when one or more of the entries `Numerator` and `Denominator` depends on a tunable `realp` or `genmat` model. For more information about Generalized state-space models, see “Models with Tunable Coefficients”.

## Examples

### Create Transfer Function with One Input and Two Outputs

Create the following transfer function model:

$$H(p) = \begin{bmatrix} \frac{p+1}{p^2+2p+2} \\ \frac{1}{p} \end{bmatrix}$$

The model has an input current and two outputs, torque and angular velocity.

Specify the numerator and denominator coefficients of the model:

```
Numerator = {[1 1] ; 1};
Denominator = {[1 2 2] ; [1 0]};
```

Create the transfer function model, specifying the input name, output names, and variable.

```
H = tf(Numerator,Denominator,'InputName','current',...
       'OutputName',{'torque' 'ang. velocity'},...
       'Variable','p')
```

H =

From input "current" to output...

```
      p + 1
torque: -----
      p^2 + 2 p + 2
```

```
      1
ang. velocity: -
              p
```

Continuous-time transfer function.

Setting the `Variable` property of the model to 'p' causes the result to display as a transfer function of the variable  $p$ .

## Create Transfer Function Model Using Rational Expression

To use a rational expression to create a SISO transfer function model, first specify `s` as a `tf` object.

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

Create a transfer function using `s` in a rational expression.

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

This method produces the same transfer function as:

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

## Second-Order Transfer Function from Damping and Natural Frequency

Create a `tf` model that represents a second-order system with known natural frequency and damping ratio.

The transfer function of a second-order system, expressed in terms of its damping ratio  $\zeta$  and natural frequency  $\omega_0$ , is:

$$H(s) = \frac{\omega_0^2}{s^2 + 2\zeta\omega_0s + \omega_0^2}$$

Represent this transfer function in MATLAB using the `tf` command. For example, suppose you have a system with  $\zeta = 0.25$  and  $\omega_0 = 3$  rad/s.

```
zeta = 0.25;
w0 = 3;
H = tf(w0^2,[1,2*zeta*w0,w0^2])
```

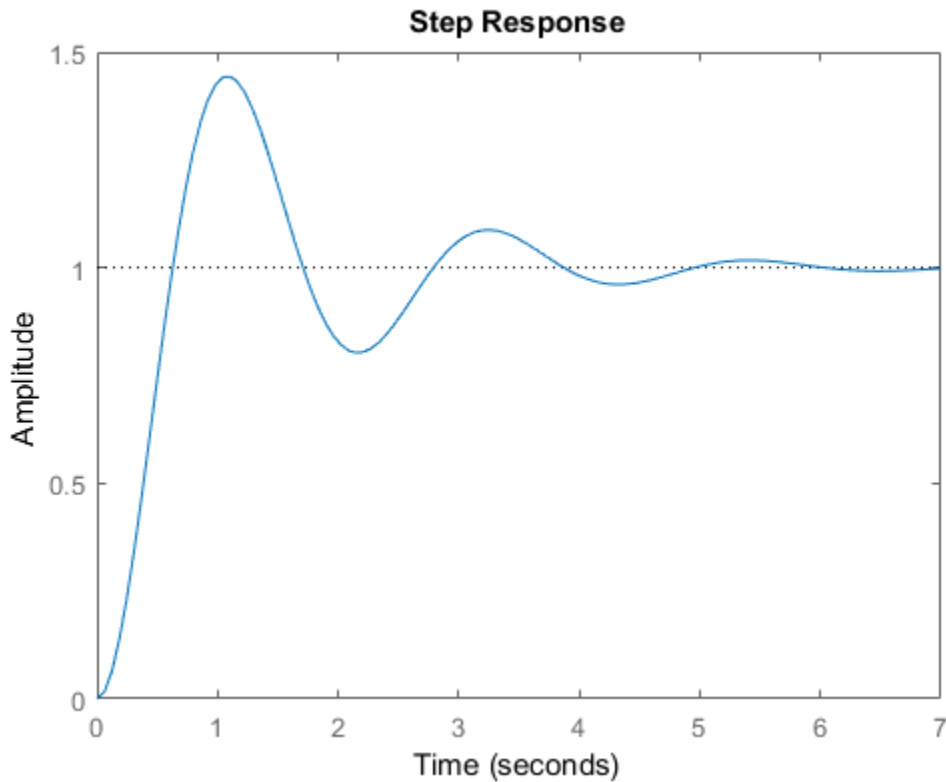
H =

$$\frac{9}{s^2 + 1.5 s + 9}$$

Continuous-time transfer function.

Examine the response of this transfer function to a step input.

```
stepplot(H)
```



The plot shows the ringdown expected of a second-order system with a low damping ratio.

### Create MIMO Transfer Function Model

Create a transfer function for the discrete-time, multi-input, multi-output model:

$$H(z) = \begin{bmatrix} \frac{1}{z+0.3} & \frac{z}{z+0.3} \\ \frac{-z+2}{z+0.3} & \frac{3}{z+0.3} \end{bmatrix}$$

with sample time  $T_s = 0.2$  seconds.

Specify the numerator coefficients as a 2-by-2 matrix.

```
Numerators = {1 [1 0];[-1 2] 3};
```

Specify the coefficients of the common denominator as a row vector.

```
Denominator = [1 0.3];
```

Create the discrete-time transfer function model.

```
Ts = 0.2;
```

```
H = tf(Numerators,Denominator,Ts);
```

### Convert State-Space Model to Transfer Function

Compute the transfer function of the following state-space model:

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

Specify the state-space model.

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

Convert this model to a transfer function.

```
tf(sys)
```

```
ans =
```

```
From input 1 to output:
```

```
s - 4.441e-16
```

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

```
From input 2 to output:
```

```
s^2 + 5 s + 8
```

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

Continuous-time transfer function.

### Create Array of Transfer Function Models

You can use a `for` loop to specify an array of SISO transfer function models.



Pre-allocate the array with zero transfer functions.

```
H = tf(zeros(1,1,10));
```

The first two indices represent the number of outputs and inputs for the models, while the third index is the number of models in the array.

Create the transfer function models.

```
s = tf('s');
for k = 1:10
    H(:,:,k) = k/(s^2+s+k);
end
```

## Create Tunable Low-Pass Filter

This example shows how to create a low-pass filter with one tunable parameter  $a$ :

$$F = \frac{a}{s + a}$$

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

Create a tunable real parameter with an initial value of 10.

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

Use `tf` to create the tunable filter  $F$ .

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

$F$  is a `genss` object which has the tunable parameter  $a$  in its `Blocks` property. You can connect  $F$  with other tunable or numeric models to create more complex control system models. For example, see “Control System with Tunable Components”.

## Extract Transfer Functions from Identified Model

Extract the measured and noise components of an identified polynomial model into two separate transfer functions (requires System Identification Toolbox). The measured component can serve as a plant model, while the noise component can serve as a disturbance model for control system design.

```
load icEngine;
z = iddata(y,u,0.04);
nb = 2; nf = 2; nc = 1; nd = 3; nk = 3;
sys = bj(z, [nb nc nd nf nk]);
```

`sys` is a model of the form:  $y(t) = B/F u(t) + C/D e(t)$ , where  $B/F$  represents the measured component and  $C/D$  the noise component.

```
sysMeas = tf(sys, 'measured')
sysNoise = tf(sys, 'noise')
```

Alternatively, you can simply use `tf(sys)` to extract the measured component.

## 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), 'q'	Use the row vector <code>[ak ... a1 a0]</code> to specify the polynomial $a_k z^k + \dots + a_1 z + a_0$ (coefficients ordered in <i>descending</i> powers of $z$ or $q$ ).

Variable	Convention
'z^-1'	Use the row vector [b0 b1 ... bk] to specify the polynomial $b_0 + b_1z^{-1} + \dots + b_kz^{-k}$ (coefficients in <i>ascending</i> powers of $z^{-1}$ ).

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

```
Numerator = [0 1 1];
Denominator = [1 2 3];
```

for `g` (the numerator is padded with zeros on the left) and the values

```
Numerator = [1 1 0];
Denominator = [1 2 3];
```

for `h` (the numerator is padded with zeros on the right).

## Properties

`tf` objects have the following properties:

### **Numerator**

Transfer function numerator coefficients.

For SISO transfer functions, **Numerator** is a row vector of polynomial coefficients in order of descending power (for **Variable** values **s**, **z**, **p**, or **q**) or in order of ascending power (for **Variable** values  $z^{-1}$  or  $q^{-1}$ ).

For MIMO transfer functions with **Ny** outputs and **Nu** inputs, **Numerator** is a **Ny**-by-**Nu** cell array of the numerator coefficients for each input/output pair.

### **Denominator**

Transfer function denominator coefficients.

For SISO transfer functions, **Denominator** is a row vector of polynomial coefficients in order of descending power (for **Variable** values **s**, **z**, **p**, or **q**) or in order of ascending power (for **Variable** values  $z^{-1}$  or  $q^{-1}$ ).

For MIMO transfer functions with **Ny** outputs and **Nu** inputs, **Denominator** is a **Ny**-by-**Nu** cell array of the denominator coefficients for each input/output pair.

### **Variable**

Transfer function display variable, specified as one of the following:

- 's' — Default for continuous-time models
- 'z' — Default for discrete-time models
- 'p' — Equivalent to 's'
- 'q' — Equivalent to 'z'
- 'z<sup>-1</sup>' — Inverse of 'z'
- 'q<sup>-1</sup>' — Equivalent to 'z<sup>-1</sup>'

The value of **Variable** is reflected in the display, and also affects the interpretation of the **Numerator** and **Denominator** coefficient vectors for discrete-time models. For **Variable** = 'z' or 'q', the coefficient vectors are ordered in descending powers of the variable. For **Variable** = 'z<sup>-1</sup>' or 'q<sup>-1</sup>', the coefficient vectors are ordered as ascending powers of the variable.

**Default:** 's'

## **IODelay**

Transport delays. **IODelay** is a numeric array specifying a separate transport delay for each input/output pair.

For continuous-time systems, specify transport delays in the time unit stored in the **TimeUnit** property. For discrete-time systems, specify transport delays in integer multiples of the sample time, **Ts**.

For a MIMO system with  $N_y$  outputs and  $N_u$  inputs, set **IODelay** to a  $N_y$ -by- $N_u$  array. Each entry of this array is a numerical value that represents the transport delay for the corresponding input/output pair. You can also set **IODelay** to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

## **InputDelay**

Input delay for each input channel, specified as a scalar value or numeric vector. For continuous-time systems, specify input delays in the time unit stored in the **TimeUnit** property. For discrete-time systems, specify input delays in integer multiples of the sample time **Ts**. For example, **InputDelay** = 3 means a delay of three sample times.

For a system with  $N_u$  inputs, set **InputDelay** to an  $N_u$ -by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

You can also set **InputDelay** to a scalar value to apply the same delay to all channels.

**Default:** 0

## **OutputDelay**

Output delays. **OutputDelay** is a numeric vector specifying a time delay for each output channel. For continuous-time systems, specify output delays in the time unit stored in the **TimeUnit** property. For discrete-time systems, specify output delays in integer multiples of the sample time **Ts**. For example, **OutputDelay** = 3 means a delay of three sampling periods.

For a system with  $N_y$  outputs, set **OutputDelay** to an  $N_y$ -by-1 vector, where each entry is a numerical value representing the output delay for the corresponding output channel. You can also set **OutputDelay** to a scalar value to apply the same delay to all channels.

**Default:** 0 for all output channels

### **Ts**

Sample time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time  $T_s$ , and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to {'controls(1)'; 'controls(2)' }.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, 'seconds'.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** '' for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];
```

```
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### OutputName

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all output channels

### OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.



Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

### **OutputGroup**

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### **Name**

System name, specified as a character vector. For example, `'system_1'`.

**Default:** ''

### **Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** {}

### **UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

### **SamplingGrid**

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the (`zeta,w`) values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

`M`

```
M(:,:,1,1) [zeta=0.3, w=5] =
```

```
      25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```
      25
-----
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the

variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sLinearizer` populate `SamplingGrid` in this way.

**Default:** `[]`

## More About

### Algorithms

`tf` uses the MATLAB function `poly` to convert zero-pole-gain models, and the functions `zero` and `pole` to convert state-space models.

- “What Are Model Objects?”
- “Transfer Functions”
- “Discrete-Time Numeric Models”
- “MIMO Transfer Functions”

### See Also

`filt` | `frd` | `genmat` | `genss` | `get` | `realp` | `set` | `ss` | `tfdata` | `tunableTF` | `zpk`

**Introduced before R2006a**

## tfdata

Access transfer function data

### Syntax

```
[num,den] = tfdata(sys)
[num,den,Ts] = tfdata(sys)
[num,den,Ts,sdnum,sdden]=tfdata(sys)
[num,den,Ts,...]=tfdata(sys,J1,...,Jn)
```

### 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`. For more information on the format of transfer function model data, see the `tf` reference page.

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`.

`[num,den,Ts,sdnum,sdden]=tfdata(sys)` also returns the uncertainties in the numerator and denominator coefficients of identified system `sys`. `sdnum{i,j}(k)` is the 1 standard uncertainty in the value `num{i,j}(k)` and `sdden{i,j}(k)` is the 1 standard uncertainty in the value `den{i,j}(k)`. If `sys` does not contain uncertainty information, `sdnum` and `sdden` are empty (`[]`).

`[num,den,Ts,...]=tfdata(sys,J1,...,Jn)` extracts the data for the `(J1,...,JN)` entry in the model array `sys`.

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

```
sys.Ts
sys.variable
```

## Examples

### Example 1

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)
```

This command 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
```

### Example 2

Extract the numerator, denominator and their standard deviations for a 2-input, 1 output identified transfer function.

```
load iddata7
```

```
transfer function model
```

```
sys1 = tfest(z7, 2, 1, 'InputDelay',[1 0]);
```

```
an equivalent process model
```

```
sys2 = procest(z7, {'P2UZ', 'P2UZ'}, 'InputDelay',[1 0]);
```

```
[num1, den1, ~, dnum1, dden1] = tfdata(sys1);
```

```
[num2, den2, ~, dnum2, dden2] = tfdata(sys2);
```

### See Also

`ssdata` | `zpkdata` | `get` | `tf`

**Introduced before R2006a**

# thiran

Generate fractional delay filter based on Thiran approximation

## Syntax

```
sys = thiran(tau, Ts)
```

## Description

`sys = thiran(tau, Ts)` discretizes the continuous-time delay `tau` using a Thiran filter to approximate the fractional part of the delay. `Ts` specifies the sample time.

## Input Arguments

### **tau**

Time delay to discretize.

### **Ts**

Sample time.

## Output Arguments

### **sys**

Discrete-time tf object.

## Examples

Approximate and discretize a time delay that is a noninteger multiple of the target sample time.



```
sys1 = thiran(2.4, 1)
```

Transfer function:

```
0.004159 z^3 - 0.04813 z^2 + 0.5294 z + 1
-----
z^3 + 0.5294 z^2 - 0.04813 z + 0.004159
```

Sample time: 1

The time delay is 2.4 s, and the sample time is 1 s. Therefore, `sys1` is a discrete-time transfer function of order 3.

Discretize a time delay that is an integer multiple of the target sample time.

```
sys2 = thiran(10, 1)
```

Transfer function:

```
1
----
z^10
```

Sample time: 1

## More About

### Tips

- If `tau` is an integer multiple of `Ts`, then `sys` represents the pure discrete delay  $z^{-N}$ , with  $N = \text{tau}/T_s$ . Otherwise, `sys` is a discrete-time, all-pass, infinite impulse response (IIR) filter of order `ceil(tau/Ts)`.
- `thiran` approximates and discretizes a pure time delay. To approximate a pure continuous-time time delay without discretizing, use `pade`. To discretize continuous-time models having time delays, use `c2d`.

### Algorithms

The Thiran fractional delay filter has the following form:

$$H(z) = \frac{a_N z^N + a_{N-1} z^{N-1} + \dots + a_1}{a_0 z^N + a_1 z^{N-1} + \dots + a_N}$$

The coefficients  $a_0, \dots, a_N$  are given by:

$$a_k = (-1)^k \binom{N}{k} \prod_{i=0}^N \frac{D - N + i}{D - N + k + i}, \quad \forall k : 1, 2, \dots, N$$
$$a_0 = 1$$

where  $D = \tau/T_s$  and  $N = \text{ceil}(D)$  is the filter order. See [1].

## References

- [1] T. Laakso, V. Valimäki, “Splitting the Unit Delay”, *IEEE Signal Processing Magazine*, Vol. 13, No. 1, p.30-60, 1996.

## See Also

c2d | pade | tf

Introduced in R2010a

# timeoptions

Create list of time plot options

## Syntax

`P = timeoptions`

`P = timeoptions('cstprefs')`

## Description

`P = timeoptions` returns a list of available options for time plots with default values set. You can use these options to customize the time value plot appearance from the command line.

`P = timeoptions('cstprefs')` initializes the plot options you selected in the Control System and System Identification Toolbox Preferences Editor. For more information about the editor, see “Toolbox Preferences Editor” in the User's Guide documentation.

This table summarizes the available time plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid	Show or hide the grid, specified as one of the following values: 'off'   'on' <b>Default:</b> 'off'
GridColor	Color of the grid lines, specified as one of the following: Vector of RGB values in the range [0,1]   character vector of color name   'none'. For example, for yellow color, specify as one of the following: [1 1 0], 'yellow', or 'y'. <b>Default:</b> [0.15,0.15,0.15]
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits

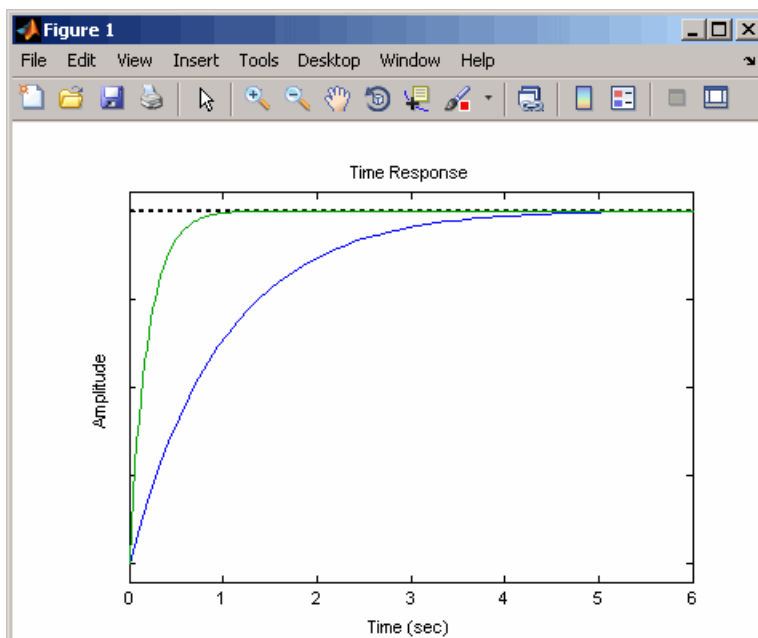
Option	Description
IOGrouping	Grouping of input-output pairs, specified as one of the following values: 'none'   'inputs'   'outputs'   'all' <b>Default:</b> 'none'
InputLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels
Normalize	Normalize responses, specified as one of the following values: 'on'   'off' <b>Default:</b> 'off'
SettleTimeThreshold	Settling time threshold
RiseTimeLimits	Rise time limits
TimeUnits	Time units, specified as one of the following values: <ul style="list-style-type: none"> <li>• 'nanoseconds'</li> <li>• 'microseconds'</li> <li>• 'milliseconds'</li> <li>• 'seconds'</li> <li>• 'minutes'</li> <li>• 'hours'</li> <li>• 'days'</li> <li>• 'weeks'</li> <li>• 'months'</li> <li>• 'years'</li> </ul> <b>Default:</b> 'seconds' <p>You can also specify 'auto' which uses time units specified in the <code>TimeUnit</code> property of the input system. For multiple systems with different time units, the units of the first system is used.</p>

## Examples

In this example, enable the normalized response option before creating a plot.

```
P = timeoptions;  
% Set normalize response to on in options  
P.Normalize = 'on';  
% Create plot with the options specified by P  
h = stepplot(tf(10,[1,1]),tf(5,[1,5]),P);
```

The following step plot is created with the responses normalized.



## See Also

[impzplot](#) | [lsimplot](#) | [setoptions](#) | [stepplot](#) | [getoptions](#) | [initialplot](#)

Introduced in R2008a

## totaldelay

Total combined I/O delays for 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 `ioDelayMatrix` 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`.

### Examples

```
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

`hasdelay` | `absorbDelay`

**Introduced before R2006a**

## tunableGain

Tunable static gain block

### Syntax

```
blk = tunableGain(name,Ny,Nu)
blk = tunableGain(name,G)
```

### Description

Model object for creating tunable static gains. `tunableGain` lets you parametrize tunable static gains for parameter studies or for automatic tuning with tuning commands such as `systune` or `looptune`.

`tunableGain` is part of the Control Design Block family of parametric models. Other Control Design Blocks include `tunablePID`, `tunableSS`, and `tunableTF`.

### Construction

`blk = tunableGain(name,Ny,Nu)` creates a parametric static gain block named `name`. This block has `Ny` outputs and `Nu` inputs. The tunable parameters are the gains across each of the `Ny`-by-`Nu` I/O channels.

`blk = tunableGain(name,G)` uses the double array `G` to dimension the block and initialize the tunable parameters.

### Input Arguments

#### **name**

Block Name, specified as a character vector such as 'K' or 'gain1'. (See “Properties” on page 2-1063.)

#### **Ny**

Non-negative integer specifying the number of outputs of the parametric static gain block `blk`.



**Nu**

Non-negative integer specifying the number of inputs of the parametric static gain block `blk`.

**G**

Double array of static gain values. The number of rows and columns of `G` determine the number of inputs and outputs of `blk`. The entries `G` are the initial values of the parametric gain block parameters.

## Properties

**Gain**

Parametrization of the tunable gain.

`blk.Gain` is a `param.Continuous` object. For general information about the properties of the `param.Continuous` object `blk.Gain`, see the `param.Continuous` object reference page.

The following fields of `blk.Gain` are used when you tune `blk` using `hinfstruct`:

Field	Description
Value	Current value of the gain matrix. For a block that has <code>Ny</code> outputs and <code>Nu</code> inputs, <code>blk.Gain.Value</code> is a <code>Ny</code> -by- <code>Nu</code> matrix. If you use the <code>G</code> input argument to create <code>blk</code> , <code>blk.Gain.Value</code> initializes to the values of <code>G</code> . Otherwise, all entries of <code>blk.Gain.Value</code> initialize to zero. <code>hinfstruct</code> tunes all entries in <code>blk.Gain.Value</code> except those whose values are fixed by <code>blk.Gain.Free</code> . Default: Array of zero values.
Free	Array of logical values determining whether the gain entries in <code>blk.Gain.Value</code> are fixed or free parameters.

Field	Description
	<ul style="list-style-type: none"> <li>• If <code>blk.Gain.Free(i,j) = 1</code>, then <code>blk.Gain.Value(i,j)</code> is a tunable parameter.</li> <li>• If <code>blk.Gain.Free(i,j) = 0</code>, then <code>blk.Gain.Value(i,j)</code> is fixed.</li> </ul> <p>Default: Array of 1 (<code>true</code>) values.</p>
Minimum	<p>Minimum value of the parameter. This property places a lower bound on the tuned value of the parameter. For example, setting <code>blk.Gain.Minimum = 1</code> ensures that all entries in the gain matrix have gain greater than 1.</p> <p>Default: <code>-Inf</code>.</p>
Maximum	<p>Maximum value of the parameter. This property places an upper bound on the tuned value of the parameter. For example, setting <code>blk.Gain.Maximum = 100</code> ensures that all entries in the gain matrix have gain less than 100.</p> <p>Default: <code>Inf</code>.</p>

### Ts

Sample time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### TimeUnit

Units for the time variable, the sample time  $T_s$ , and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to {'controls(1)'; 'controls(2)' }.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems

- Specifying connection points when interconnecting models

**Default:** '' for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, 'seconds'.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** '' for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### **OutputName**

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, 'measurements'.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2) '}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all output channels

### OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, 'seconds'.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

### OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement', :)
```

**Default:** Struct with no fields

**Name**

System name, specified as a character vector. For example, 'system\_1'.

**Default:** ''

**Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, 'System is MIMO'.

**Default:** {}

**UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

## Examples

Create a 2-by-2 parametric gain block of the form

$$\begin{bmatrix} g_1 & 0 \\ 0 & g_2 \end{bmatrix}$$

where  $g_1$  and  $g_2$  are tunable parameters, and the off-diagonal elements are fixed to zero.

```
blk = tunableGain('gainblock',2,2); % 2 outputs, 2 inputs  
blk.Gain.Free = [1 0; 0 1]; % fix off-diagonal entries to zero
```

All entries in `blk.Gain.Value` initialize to zero. Initialize the diagonal values to 1 as follows.

```
blk.Gain.Value = eye(2); % set diagonals to 1
```

Create a two-input, three-output parametric gain block and initialize all the parameter values to 1.

To do so, create a matrix to dimension the parametric gain block and initialize the parameter values.

```
G = ones(3,2);  
blk = tunableGain('gainblock',G);
```

Create a 2-by-2 parametric gain block and assign names to the inputs.

```
blk = tunableGain('gainblock',2,2) % 2 outputs, 2 inputs  
blk.InputName = {'Xerror','Yerror'} % assign input names
```

## More About

### Tips

- Use the `blk.Gain.Free` field of `blk` to specify additional structure or fix the values of specific entries in the block. To fix the gain value from input `i` to output `j`, set `blk.Gain.Free(i,j) = 0`. To allow `hinfstruct` to tune this gain value, set `blk.Gain.Free(i,j) = 1`.
- To convert a `tunableGain` parametric model to a numeric (non-tunable) model object, use model commands such as `tf`, `zpk`, or `ss`.
- “Control Design Blocks”
- “Models with Tunable Coefficients”

### See Also

`genss` | `hinfstruct` | `looptune` | `sysstune` | `tunablePID` | `tunablePID2` | `tunableSS` | `tunableTF`

**Introduced in R2011a**

## tunablePID

Tunable PID controller

### Syntax

```
blk = tunablePID(name,type)
blk = tunablePID(name,type,Ts)
blk = tunablePID(name,sys)
```

### Description

Model object for creating tunable one-degree-of-freedom PID controllers. `tunablePID` lets you parametrize a tunable SISO PID controller for parameter studies or for automatic tuning with tuning commands such as `sys tune`, `looptune`, or the Robust Control Toolbox command, `hinfstruct`.

`tunablePID` is part of the family of parametric Control Design Blocks. Other parametric Control Design Blocks include `tunableGain`, `tunableSS`, and `tunableTF`.

### Construction

`blk = tunablePID(name,type)` creates the one-degree-of-freedom continuous-time PID controller:

$$blk = K_p + \frac{K_i}{s} + \frac{K_d s}{1 + T_f s},$$

with tunable parameters `Kp`, `Ki`, `Kd`, and `Tf`. The `type` argument sets the controller type by fixing some of these values to zero (see “Input Arguments” on page 2-1071).

`blk = tunablePID(name,type,Ts)` creates a discrete-time PID controller with sample time `TS`:



$$blk = K_p + K_i IF(z) + \frac{K_d}{T_f + DF(z)},$$

where  $IF(z)$  and  $DF(z)$  are the discrete integrator formulas for the integral and derivative terms, respectively. The values of the `IFormula` and `DFormula` properties set the discrete integrator formulas (see “Properties” on page 2-1072).

`blk = tunablePID(name, sys)` uses the dynamic system model, `sys`, to set the sample time, `Ts`, and the initial values of the parameters `Kp`, `Ki`, `Kd`, and `Tf`.

## Input Arguments

### name

PID controller Name, specified as a character vector such as 'C' or 'PI1'. (See “Properties” on page 2-1072.)

### type

Controller type, specified as one of the values in the following table. Specifying a controller type fixes up to three of the PID controller parameters.

Value for type	Controller Type	Effect on PID Parameters
'P'	Proportional only	<code>Ki</code> and <code>Kd</code> are fixed to zero; <code>Tf</code> is fixed to 1; <code>Kp</code> is free
'PI'	Proportional-integral	<code>Kd</code> is fixed to zero; <code>Tf</code> is fixed to 1; <code>Kp</code> and <code>Ki</code> are free
'PD'	Proportional-derivative with first-order filter on derivative action	<code>Ki</code> is fixed to zero; <code>Kp</code> , <code>Kd</code> , and <code>Tf</code> are free
'PID'	Proportional-integral-derivative with first-order filter on derivative action	<code>Kp</code> , <code>Ki</code> , <code>Kd</code> , and <code>Tf</code> are free

### Ts

Sample time, specified as a scalar.

**sys**

Dynamic system model representing a PID controller.

## Properties

**Kp, Ki, Kd, Tf**

Parametrization of the PID gains Kp, Ki, Kd, and filter time constant Tf of the tunable PID controller blk.

The following fields of blk.Kp, blk.Ki, blk.Kd, and blk.Tf are used when you tune blk using a tuning command such as systune:

Field	Description
Value	Current value of the parameter.
Free	Logical value determining whether the parameter is fixed or tunable. For example, <ul style="list-style-type: none"> <li>• If blk.Kp.Free = 1, then blk.Kp.Value is tunable.</li> <li>• If blk.Kp.Free = 0, then blk.Kp.Value is fixed.</li> </ul>
Minimum	Minimum value of the parameter. This property places a lower bound on the tuned value of the parameter. For example, setting blk.Kp.Minimum = 0 ensures that Kp remains positive. blk.Tf.Minimum must always be positive.
Maximum	Maximum value of the parameter. This property places an upper bound on the tuned value of the parameter. For example, setting blk.Tf.Maximum = 100 ensures that the filter time constant does not exceed 100.

blk.Kp, blk.Ki, blk.Kd, and blk.Tf are param.Continuous objects. For general information about the properties of these param.Continuous objects, see the param.Continuous object reference page.

**IFormula, DFormula**

Discrete integrator formulas  $IF(z)$  and  $DF(z)$  for the integral and derivative terms, respectively, specified as one of the values in the following table.

Value	IF(z) or DF(z) Formula
'ForwardEuler'	$\frac{T_s}{z-1}$
'BackwardEuler'	$\frac{T_s z}{z-1}$
'Trapezoidal'	$\frac{T_s}{2} \frac{z+1}{z-1}$

**Default:** 'ForwardEuler'

**Ts**

Sample time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

**TimeUnit**

Units for the time variable, the sample time  $T_s$ , and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'

- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel name, specified as a character vector. Use this property to name the input channel of the controller model. For example, assign the name `error` to the input of a controller model `C` as follows.

```
C.InputName = 'error';
```

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `C.u` is equivalent to `C.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### **InputUnit**

Input channel units, specified as a character vector. Use this property to track input signal units. For example, assign the concentration units `mol/m^3` to the input of a controller model `C` as follows.

```
C.InputUnit = 'mol/m^3';
```

`InputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

### **InputGroup**

Input channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **OutputName**

Output channel name, specified as a character vector. Use this property to name the output channel of the controller model. For example, assign the name `control` to the output of a controller model `C` as follows.

```
C.OutputName = 'control';
```

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `C.y` is equivalent to `C.OutputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### **OutputUnit**

Output channel units, specified as a character vector. Use this property to track output signal units. For example, assign the unit `Volts` to the output of a controller model `C` as follows.

```
C.OutputUnit = 'Volts';
```

`OutputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

### **OutputGroup**

Output channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

**Name**

System name, specified as a character vector. For example, 'system\_1'.

**Default:** ''

**Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, 'System is MIMO'.

**Default:** {}

**UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

## Examples

### Tunable Controller with a Fixed Parameter

Create a tunable PD controller. Then, initialize the parameter values, and fix the filter time constant.

```
blk = tunablePID('pblock','PD');
blk.Kp.Value = 4;           % initialize Kp to 4
blk.Kd.Value = 0.7;        % initialize Kd to 0.7
blk.Tf.Value = 0.01;       % set parameter Tf to 0.01
blk.Tf.Free = false;       % fix parameter Tf to this value
blk
```

blk =

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

$$K_p + K_d * \frac{s}{T_f*s+1}$$

and tunable parameters Kp, Kd.

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

### Controller Initialized by Dynamic System Model

Create a tunable discrete-time PI controller. Use a pid object to initialize the parameters and other properties.

```
C = pid(5,2.2,'Ts',0.1,'IFormula','BackwardEuler');
blk = tunablePID('piblock',C)
```

```
blk =
```

```
Parametric discrete-time PID controller "piblock" with formula:
```

$$K_p + K_i * \frac{T_s * z}{z-1}$$

```
and tunable parameters Kp, Ki.
```

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

blk takes the value of properties, such as Ts and IFormula, from C.

### Controller with Named Input and Output

Create a tunable PID controller, and assign names to the input and output.

```
blk = tunablePID('pidblock','pid')
blk.InputName = {'error'} % assign input name
blk.OutputName = {'control'} % assign output name
```

## More About

### Tips

- You can modify the PID structure by fixing or freeing any of the parameters Kp, Ki, Kd, and Tf. For example, blk.Tf.Free = false fixes Tf to its current value.
- To convert a tunablePID parametric model to a numeric (nontunable) model object, use model commands such as pid, pidstd, tf, or ss. You can also use getValue to obtain the current value of a tunable model.

- “Control Design Blocks”
- “Models with Tunable Coefficients”

### **See Also**

`genss` | `hinfstruct` | `looptune` | `systune` | `tunableGain` | `tunablePID2` | `tunableSS` | `tunableTF`

**Introduced in R2011a**



# tunablePID2

Tunable two-degree-of-freedom PID controller

## Syntax

```
blk = tunablePID2(name,type)
blk = tunablePID2(name,type,Ts)
blk = tunablePID2(name,sys)
```

## Description

Model object for creating tunable two-degree-of-freedom PID controllers. `tunablePID2` lets you parametrize a tunable SISO two-degree-of-freedom PID controller. You can use this parametrized controller for parameter studies or for automatic tuning with tuning commands such as `systemtune`, `looptune`, or the Robust Control Toolbox command `hinfstruct`.

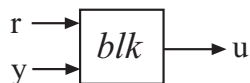
`tunablePID2` is part of the family of parametric Control Design Blocks. Other parametric Control Design Blocks include `tunableGain`, `tunableSS`, and `tunableTF`.

## Construction

`blk = tunablePID2(name,type)` creates the two-degree-of-freedom continuous-time PID controller described by the equation:

$$u = K_p (br - y) + \frac{K_i}{s} (r - y) + \frac{K_d s}{1 + T_f s} (cr - y).$$

$r$  is the setpoint command,  $y$  is the measured response to that setpoint, and  $u$  is the control signal, as shown in the following illustration.



The tunable parameters of the block are:

- Scalar gains  $K_p$ ,  $K_i$ , and  $K_d$
- Filter time constant  $T_f$
- Scalar weights  $b$  and  $c$

The **type** argument sets the controller type by fixing some of these values to zero (see “Input Arguments” on page 2-1080).

`blk = tunablePID2(name, type, Ts)` creates a discrete-time PID controller with sample time  $T_s$ . The equation describing this controller is:

$$u = K_p (br - y) + K_i IF(z)(r - y) + \frac{K_d}{T_f + DF(z)} (cr - y).$$

$IF(z)$  and  $DF(z)$  are the discrete integrator formulas for the integral and derivative terms, respectively. The values of the `IFormula` and `DFormula` properties set the discrete integrator formulas (see “Properties” on page 2-1081).

`blk = tunablePID2(name, sys)` uses the dynamic system model, `sys`, to set the sample time,  $T_s$ , and the initial values of all the tunable parameters. The model `sys` must be compatible with the equation of a two-degree-of-freedom PID controller.

## Input Arguments

### **name**

PID controller **Name**, specified as a character vector such as `'C'` or `'2DOFPID1'`. (See “Properties” on page 2-1081.)

### **type**

Controller type, specified as one of the values in the following table. Specifying a controller type fixes up to three of the PID controller parameters.

Value for type	Controller Type	Effect on PID Parameters
'P'	Proportional only	$K_i$ and $K_d$ are fixed to zero; $T_f$ is fixed to 1; $K_p$ is free
'PI'	Proportional-integral	$K_d$ is fixed to zero; $T_f$ is fixed to 1; $K_p$ and $K_i$ are free

Value for type	Controller Type	Effect on PID Parameters
'PD'	Proportional-derivative with first-order filter on derivative action	Ki is fixed to zero; Kp, Kd, and Tf are free
'PID'	Proportional-integral-derivative with first-order filter on derivative action	Kp, Ki, Kd, and Tf are free

**Ts**

Sample time, specified as a scalar.

**sys**

Dynamic system model representing a two-degree-of-freedom PID controller.

## Properties

**Kp, Ki, Kd, Tf, b, c**

Parametrization of the PID gains Kp, Ki, Kd, the filter time constant, Tf, and the scalar gains, b and c.

The following fields of `blk.Kp`, `blk.Ki`, `blk.Kd`, `blk.Tf`, `blk.b`, and `blk.c` are used when you tune `blk` using a tuning command such as `syntune`:

Field	Description
Value	Current value of the parameter. <code>blk.b.Value</code> , and <code>blk.c.Value</code> are always nonnegative.
Free	Logical value determining whether the parameter is fixed or tunable. For example, <ul style="list-style-type: none"> <li>If <code>blk.Kp.Free = 1</code>, then <code>blk.Kp.Value</code> is tunable.</li> <li>If <code>blk.Kp.Free = 0</code>, then <code>blk.Kp.Value</code> is fixed.</li> </ul>

Field	Description
Minimum	Minimum value of the parameter. This property places a lower bound on the tuned value of the parameter. For example, setting <code>blk.Kp.Minimum = 0</code> ensures that <code>Kp</code> remains positive. <code>blk.Tf.Minimum</code> must always be positive.
Maximum	Maximum value of the parameter. This property places an upper bound on the tuned value of the parameter. For example, setting <code>blk.c.Maximum = 1</code> ensures that <code>c</code> does not exceed unity.

`blk.Kp`, `blk.Ki`, `blk.Kd`, `blk.Tf`, `blk.b`, and `blk.c` are `param.Continuous` objects. For more information about the properties of these `param.Continuous` objects, see the `param.Continuous` object reference page.

### IFormula, DFormula

Discrete integrator formulas  $IF(z)$  and  $DF(z)$  for the integral and derivative terms, respectively, specified as one of the values in the following table.

Value	IF(z) or DF(z) Formula
'ForwardEuler'	$\frac{T_s}{z-1}$
'BackwardEuler'	$\frac{T_s z}{z-1}$
'Trapezoidal'	$\frac{T_s}{2} \frac{z+1}{z-1}$

**Default:** 'ForwardEuler'

### Ts

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit

specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel name, specified as a character vector or a 2-by-1 cell array of character vectors. Use this property to name the input channels of the controller model. For example, assign the names `setpoint` and `measurement` to the inputs of a 2-DOF PID controller model `C` as follows.

```
C.InputName = {'setpoint'; 'measurement'};
```

Alternatively, use automatic vector expansion to assign both input names. For example:

```
C.InputName = 'C-input';
```

The input names automatically expand to {'C-input(1)'; 'C-input(2)' }.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `C.u` is equivalent to `C.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** {' '; ' '}

### **InputUnit**

Input channel units, specified as a 2-by-1 cell array of character vectors. Use this property to track input signal units. For example, assign the units `Volts` to the reference input and the concentration units `mol/m^3` to the measurement input of a 2-DOF PID controller model `C` as follows.

```
C.InputUnit = {'Volts'; 'mol/m^3'};
```

`InputUnit` has no effect on system behavior.

**Default:** {' '; ' '}

### **InputGroup**

Input channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **OutputName**

Output channel name, specified as a character vector. Use this property to name the output channel of the controller model. For example, assign the name `control` to the output of a controller model `C` as follows.

```
C.OutputName = 'control';
```

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `C.y` is equivalent to `C.OutputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Specifying connection points when interconnecting models

**Default:** Empty character vector, ''

### **OutputUnit**

Output channel units, specified as a character vector. Use this property to track output signal units. For example, assign the unit `Volts` to the output of a controller model `C` as follows.

```
C.OutputUnit = 'Volts';
```

`OutputUnit` has no effect on system behavior.

**Default:** Empty character vector, ''

### **OutputGroup**

Output channel groups. This property is not needed for PID controller models.

**Default:** struct with no fields

### **Name**

System name, specified as a character vector. For example, `'system_1'`.

**Default:** ''

### **Notes**

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** {}

### **UserData**

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

## Examples

### Tunable Two-Degree-of-Freedom Controller with a Fixed Parameter

Create a tunable two-degree-of-freedom PD controller. Then, initialize the parameter values, and fix the filter time constant.

```
blk = tunablePID2('pdblck','PD');
blk.b.Value = 1;
blk.c.Value = 0.5;
blk.Tf.Value = 0.01;
blk.Tf.Free = false;
blk
```

```
blk =
```

```
Parametric continuous-time 2-DOF PID controller "pdblck" with equation:
```

$$u = K_p (b*r-y) + K_d \frac{s}{T_f*s+1} (c*r-y)$$

where r,y are the controller inputs and Kp, Kd, b, c are tunable gains.

Type "showBlockValue(blk)" to see the current value and "get(blk)" to see all properties.

### Controller Initialized by Dynamic System Model

Create a tunable two-degree-of-freedom PI controller. Use a two-input, one-output tf model to initialize the parameters and other properties.

```
s = tf('s');
Kp = 10;
Ki = 0.1;
b = 0.7;
sys = [(b*Kp + Ki/s), (-Kp - Ki/s)];
blk = tunablePID2('PI2dof',sys)
```

```
blk =
```

```
Parametric continuous-time 2-DOF PID controller "PI2dof" with equation:
```



$$u = K_p (b \cdot r - y) + K_i \frac{1}{s} (r - y)$$

where  $r, y$  are the controller inputs and  $K_p, K_i, b$  are tunable gains.

Type `"showBlockValue(blk)"` to see the current value and `"get(blk)"` to see all properties.

`blk` takes initial parameter values from `sys`.

If `sys` is a discrete-time system, `blk` takes the value of properties, such as `Ts` and `IFormula`, from `sys`.

### Controller with Named Inputs and Output

Create a tunable PID controller, and assign names to the inputs and output.

```
blk = tunablePID2('pidblock', 'pid');
blk.InputName = {'reference', 'measurement'};
blk.OutputName = {'control'};
```

`blk.InputName` is a cell array containing two names, because a two-degree-of-freedom PID controller has two inputs.

## More About

### Tips

- You can modify the PID structure by fixing or freeing any of the parameters. For example, `blk.Tf.Free = false` fixes `Tf` to its current value.
- To convert a `tunablePID2` parametric model to a numeric (nontunable) model object, use model commands such as `tf` or `ss`. You can also use `getValue` to obtain the current value of a tunable model.
- “Control Design Blocks”
- “Models with Tunable Coefficients”

### See Also

`genss` | `hinfstruct` | `looptune` | `systune` | `tunableGain` | `tunablePID` | `tunableSS` | `tunableTF`

**Introduced in R2012b**

# tunableSS

Tunable fixed-order state-space model

## Syntax

```
blk = tunableSS(name,Nx,Ny,Nu)
blk = tunableSS(name,Nx,Ny,Nu,Ts)
blk = tunableSS(name,sys)
blk = tunableSS(...,Astruct)
```

## Description

Model object for creating tunable fixed-order state-space models. `tunableSS` lets you parametrize a state-space model of a given order for parameter studies or for automatic tuning with tuning commands such as `system` or `looptune`.

`tunableSS` is part of the Control Design Block family of parametric models. Other Control Design Blocks include `tunablePID`, `tunableGain`, and `tunableTF`.

## Construction

`blk = tunableSS(name,Nx,Ny,Nu)` creates the continuous-time parametric state-space model named `name`. The state-space model `blk` has `Nx` states,`Ny` outputs, and `Nu` inputs. The tunable parameters are the entries in the *A*, *B*, *C*, and *D* matrices of the state-space model.

`blk = tunableSS(name,Nx,Ny,Nu,Ts)` creates a discrete-time parametric state-space model with sample time `Ts`.

`blk = tunableSS(name,sys)` uses the dynamic system `sys` to dimension the parametric state-space model, set its sample time, and initialize the tunable parameters.

`blk = tunableSS(...,Astruct)` creates a parametric state-space model whose *A* matrix is restricted to the structure specified in `Astruct`.

## Input Arguments

### **name**

Parametric state-space model **Name**, specified as a character vector such as 'C0'. (See “Properties” on page 2-1091.)

### **Nx**

Nonnegative integer specifying the number of states (order) of the parametric state-space model **blk**.

### **Ny**

Nonnegative integer specifying the number of outputs of the parametric state-space model **blk**.

### **Nu**

Nonnegative integer specifying the number of inputs of the parametric state-space model **blk**.

### **Ts**

Scalar sample time.

### **Astruct**

Constraints on the form of the **A** matrix of the parametric state-space model **blk**, specified as one of the following values:

Value for <b>Astruct</b>	Structure of <b>A</b> matrix
'tridiag'	<b>A</b> is tridiagonal. In tridiagonal form, <b>A</b> has free elements only in the main diagonal, the first diagonal below the main diagonal, and the first diagonal above the main diagonal. The remaining elements of <b>A</b> are fixed to zero.
'full'	<b>A</b> is full (every entry in <b>A</b> is a free parameter).
'companion'	<b>A</b> is in companion form. In companion form, the characteristic polynomial of the

Value for Astruct	Structure of A matrix
	system appears explicitly in the rightmost column of the A matrix. See <code>canon</code> for more information.

If you do not specify `Astruct`, `blk` defaults to 'tridiag' form.

### sys

Dynamic system model providing number of states, number of inputs and outputs, sample time, and initial values of the parameters of `blk`. To obtain the dimensions and initial parameter values, `tunableSS` converts `sys` to a state-space model with the structure specified in `Astruct`. If you omit `Astruct`, `tunableSS` converts `sys` into tridiagonal state-space form.

## Properties

### A, B, C, D

Parametrization of the state-space matrices  $A$ ,  $B$ ,  $C$ , and  $D$  of the tunable state-space model `blk`.

`blk.A`, `blk.B`, `blk.C`, and `blk.D` are `param.Continuous` objects. For general information about the properties of these `param.Continuous` objects, see the `param.Continuous` object reference page.

The following fields of `blk.A`, `blk.B`, `blk.C`, and `blk.D` are used when you tune `blk` using `hinfstruct`:

Field	Description
Value	Current values of the entries in the parametrized state-space matrix. For example, <code>blk.A.Value</code> contains the values of the A matrix of <code>blk</code> . <code>hinfstruct</code> tunes all entries in <code>blk.A.Value</code> , <code>blk.B.Value</code> , <code>blk.C.Value</code> , and <code>blk.D.Value</code> except those whose values are fixed by <code>blk.Gain.Free</code> .

Field	Description
Free	<p>2-D array of logical values determining whether the corresponding state-space matrix parameters are fixed or free parameters. For example:</p> <ul style="list-style-type: none"> <li>• If <code>blk.A.Free(i,j) = 1</code>, then <code>blk.A.Value(i,j)</code> is a tunable parameter.</li> <li>• If <code>blk.A.Free(i,j) = 0</code>, then <code>blk.A.Value(i,j)</code> is fixed.</li> </ul> <p>Defaults: By default, all entries in <code>B</code>, <code>C</code>, and <code>D</code> are tunable. The default free entries in <code>A</code> depend upon the value of <code>Astruct</code>:</p> <ul style="list-style-type: none"> <li>• <code>'tridiag'</code> — entries on the three diagonals of <code>blk.A.Free</code> are 1; the rest are 0.</li> <li>• <code>'full'</code> — all entries in <code>blk.A.Free</code> are 0.</li> <li>• <code>'companion'</code> — <code>blk.A.Free(1,:) = 1</code> and <code>blk.A.Free(j,j-1) = 1</code>; all other entries are 0.</li> </ul>
Minimum	<p>Minimum value of the parameter. This property places a lower bound on the tuned value of the parameter. For example, setting <code>blk.A.Minimum(1,1) = 0</code> ensures that the first entry in the <code>A</code> matrix remains positive. Default: <code>-Inf</code>.</p>
Maximum	<p>Maximum value of the parameter. This property places an upper bound on the tuned value of the parameter. For example, setting <code>blk.A.Maximum(1,1) = 0</code> ensures that the first entry in the <code>A</code> matrix remains negative. Default: <code>Inf</code>.</p>

## StateName

State names, specified as one of the following:

- Character vector — For first-order models, for example, 'velocity'.
- Cell array of character vectors — For models with two or more states
- '' — For unnamed states.

**Default:** '' for all states

## StateUnit

State units, specified as one of the following:

- Character vector — For first-order models, for example, 'velocity'.
- Cell array of character vectors — For models with two or more states
- '' — For unnamed states.

Use **StateUnit** to keep track of the units each state is expressed in. **StateUnit** has no effect on system behavior.

**Default:** '' for all states

## Ts

Sample time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period. This value is expressed in the unit specified by the **TimeUnit** property of the model. To denote a discrete-time model with unspecified sample time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use **c2d** and **d2c** to convert between continuous- and discrete-time representations. Use **d2d** to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

## TimeUnit

Units for the time variable, the sample time  $T_s$ , and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to {'controls(1)'; 'controls(2)' }.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems



- Specifying connection points when interconnecting models

**Default:** '' for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, 'seconds'.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** '' for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### **OutputName**

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, 'measurements'.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}.`

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all output channels

### OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, `'seconds'`.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** `''` for all output channels

### OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### Name

System name, specified as a character vector. For example, 'system\_1'.

**Default:** ''

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, 'System is MIMO'.

**Default:** {}

### UserData

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

## Examples

Create a parametrized 5th-order SISO model with zero D matrix.

```
blk = tunableSS('ssblock',5,1,1);
blk.D.Value = 0;      % set D = 0
blk.D.Free = false;  % fix D to zero
```

By default, the A matrix is in tridiagonal form. To parametrize the model in companion form, use the 'companion' input argument:

```
blk = tunableSS('ssblock',5,1,1,'companion');
blk.D.Value = 0;      % set D = 0
blk.D.Free = false;  % fix D to zero
```

Create a parametric state-space model, and assign names to the inputs.

```
blk = tunableSS('ssblock',5,2,2) % 5 states, 2 outputs, 2 inputs
blk.InputName = {'Xerror','Yerror'} % assign input names
```

## More About

### Tips

- Use the `Astruct` input argument to constrain the structure of the `A` matrix of the parametric state-space model. To impose additional structure constraints on the state-space matrices, use the fields `blk.A.Free`, `blk.B.Free`, `blk.C.Free`, and `blk.D.Free` to fix the values of specific entries in the parameter matrices.

For example, to fix the value of `blk.B(i,j)`, set `blk.B.Free(i,j) = 0`. To allow `hinfstruct` to tune `blk.B(i,j)`, set `blk.B.Free(i,j) = 1`.

- To convert a `tunableSS` parametric model to a numeric (non-tunable) model object, use model commands such as `ss`, `tf`, or `zpk`.
- “Control Design Blocks”
- “Models with Tunable Coefficients”

### See Also

`genss` | `hinfstruct` | `looptune` | `systune` | `tunableGain` | `tunablePID` | `tunablePID2` | `tunableTF`

**Introduced in R2011a**

# tunableSurface

Create tunable gain surface for gain scheduling

tunableSurface lets you parameterize and tune *gain schedules*, which are gains that vary as a function of one or more scheduling variables.

For tuning purposes, it is convenient to parameterize a variable gain as a smooth *gain surface* of the form:

$$K(n(\sigma)) = K_0 + K_1 F_1(n(\sigma)) + \dots + K_M F_M(n(\sigma)).$$

Here,  $\sigma$  is a vector of scheduling variables, and  $n(\sigma)$  is a normalization function that maps the range of each scheduling-variable value onto  $[-1, 1]$ .  $F_1, \dots, F_M$  are user-selected basis functions, and  $K_0, \dots, K_M$  are the coefficients to be tuned. You can use terms in a generic polynomial expansion as basis functions. Or, when the expected shape of  $K(\sigma)$  is known, you can use more specific functions. You can then use **systemtune** to tune the coefficients  $K_0, \dots, K_M$ , subject to your design requirements, over the range of scheduling-variable values.

## Syntax

`K = tunableSurface(name, K0init, domain, shapefcn)`

## Description

`K = tunableSurface(name, K0init, domain, shapefcn)` creates the tunable gain surface:

$$K(n(\sigma)) = K_0 + K_1 F_1(n(\sigma)) + \dots + K_M F_M(n(\sigma)).$$

The tunable surface `K` stores the basis functions specified by `shapefcn` and a discrete set of  $\sigma$  values (the *design points*) given by `domain`. The tunable gain surface has tunable coefficients  $K_0, \dots, K_M$ . The gain value is initialized to the constant gain `K0init`. You can combine `K` with other static or dynamic elements to construct a closed-loop model of your gain-scheduled control system. Or, use `K` to parameterize a lookup table in an `slTuner` interface to a Simulink model. Then, use **systemtune** to tune  $K_0, \dots, K_M$  so that the closed-loop system meets your design requirements at the selected design points.

## Examples

### Tunable Gain With One Scheduling Variable

Create a scalar gain  $K$  that varies as a quadratic function of  $t$ :

$$K(t) = K_0 + K_1 n(t) + K_2 (n(t))^2.$$

This gain surface can represent a gain that varies with time. The coefficients  $K_0$ ,  $K_1$ , and  $K_2$  are the tunable parameters of this time-varying gain. For this example, suppose that  $t$  varies from 0 to 40. In that case, the normalization function is  $n(t) = (t - 20)/20$ .

To represent the tunable gain surface  $K(t)$  in MATLAB®, first choose a vector of  $t$  values that are the design points of your system. For example, if your design points are linearization snapshots obtained at different time values, use these values for  $t$ . Create a sampling grid of these design points.

```
t = 0:5:40;  
domain = struct('t',t);
```

Specify a quadratic function for the variable gain.

```
shapefcn = @(x) [x,x^2];
```

`shapefcn` is the handle to an anonymous vector function. Each entry in the vector gives a term in the polynomial expansion that describes the variable gain. `tunableSurface` implicitly assumes the constant function  $f_0(t) = 1$ , so it need not be included in `shapefcn`.

Create the tunable gain surface  $K(t)$ .

```
K = tunableSurface('K',1,domain,shapefcn)
```

```
K =
```

```
Tunable surface "K" of scalar gains with:  
* Scheduling variables: t  
* Basis functions: t,t^2  
* Design points: 1x9 grid of t values
```

The display summarizes the characteristics of the gain surface, including the design points and the basis functions. Examine the properties of **K**.

```
get(K)
```

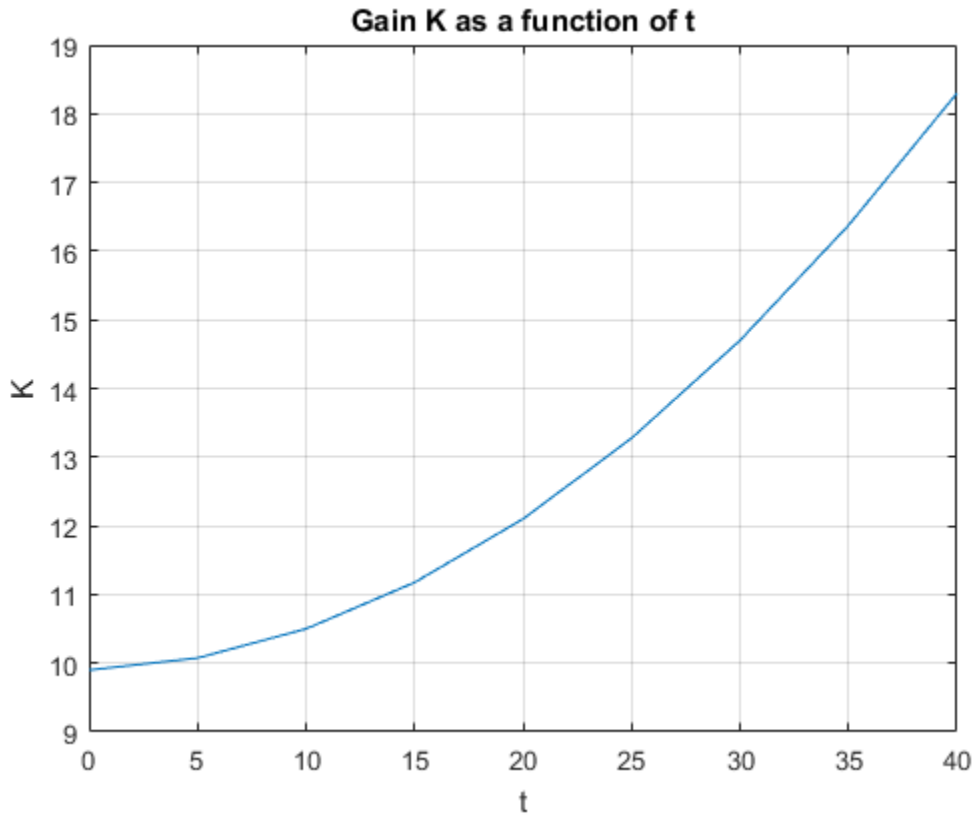
```
      Name: 'K'  
BasisFunctions: @(x)[x,x^2]  
Coefficients: [1×3 realp]  
SamplingGrid: [1×1 struct]
```

The **Coefficients** property of the tunable surface is the array of tunable coefficients,  $[K_0, K_1, K_2]$ , stored as an array-valued **realp** block.

You can now use the tunable surface in a control system model. For tuning in MATLAB, interconnect **K** with other control system elements just as you would use a Control Design Block to create a tunable control system model. For tuning in Simulink®, use **setBlockParam** to make **K** the parameterization of a tunable block in an **slTuner** interface. When you tune the model or **slTuner** interface using **systemtune**, the resulting model or interface contains tuned values for the coefficients  $K_0$ ,  $K_1$ , and  $K_2$ .

After you tune the coefficients, you can view the shape of the resulting gain curve using the **viewSurf** command. For this example, instead of tuning, manually set the coefficients to non-zero values. View the resulting gain as a function of time.

```
Ktuned = setData(K,[12.1,4.2,2]);  
viewSurf(Ktuned)
```



viewSurf displays the gain as a function of the scheduling variable, for the range of scheduling-variable values specified by domain and stored in the `SamplingGrid` property of the gain surface.

### Tunable Gain With Two Independent Scheduling Variables

This example shows how to model a scalar gain  $K$  with a bilinear dependence on two scheduling variables,  $\alpha$  and  $V$ , as follows:

$$K(\alpha, V) = K_0 + K_1x + K_2y + K_3xy.$$



Here,  $x$  and  $y$  are the normalized scheduling variables. Suppose that for this example,  $\alpha$  is an angle of incidence that ranges from 0 to 15 degrees, and  $V$  is a speed that ranges from 300 to 600 m/s. Then,  $x$  and  $y$  are given by:

$$x = \frac{\alpha - 7.5}{7.5}, \quad y = \frac{V - 450}{150}.$$

The coefficients  $K_0, \dots, K_3$  are the tunable parameters of this variable gain.

Create a grid of design points,  $(\alpha, V)$ , that are linearly spaced in  $\alpha$  and  $V$ . These design points are the scheduling-variable values used for tuning the gain-surface coefficients. They must correspond to parameter values at which you have sampled the plant.

```
[alpha,V] = ndgrid(0:3:15,300:50:600);
```

These arrays, `alpha` and `V`, represent the independent variation of the two scheduling variables, each across its full range. Put them into a structure to define the design points for the tunable surface.

```
domain = struct('alpha',alpha,'V',V);
```

Create the basis functions that describe the bilinear expansion.

```
shapefcn = @(x,y) [x,y,x*y]; % or use polyBasis('canonical',2,1)
```

In the array returned by `shapefcn`, the basis functions are:

$$\begin{aligned} F_1(x,y) &= x \\ F_2(x,y) &= y \\ F_3(x,y) &= xy. \end{aligned}$$

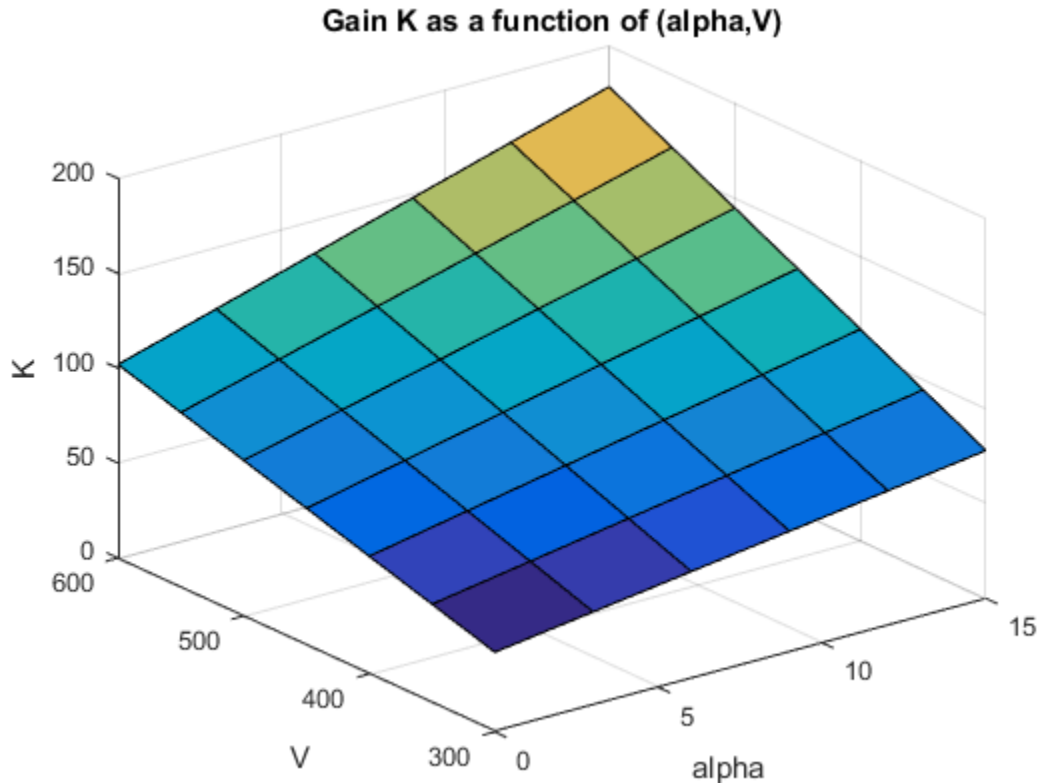
Create the tunable gain surface.

```
K = tunableSurface('K',1,domain,shapefcn);
```

You can use the tunable surface as the parameterization for a Lookup Table block in a Simulink model. Or, use model interconnection commands to incorporate it as a tunable element in a control system modeled in MATLAB. After you tune the coefficients, you can examine the resulting gain surface using the `viewSurf` command. For this example,

instead of tuning, manually set the coefficients to non-zero values and view the resulting gain.

```
Ktuned = setData(K,[100,28,40,10]);  
viewSurf(Ktuned)
```



`viewSurf` displays the gain surface as a function of the scheduling variables, for the ranges of values specified by `domain` and stored in the `SamplingGrid` property of the gain surface.

### Gain Surface Over Nonregular Grid

Create a gain surface using design points that do not form a regular grid in the operating domain. The gain surface varies as a bilinear function of variables  $\alpha$  and  $\beta$ :

$$K(\alpha, \beta) = K_0 + K_1\alpha + K_2\beta + K_3\alpha\beta.$$

Suppose that the values of interest of the scheduling variables are the following  $(\alpha, \beta)$  pairs.

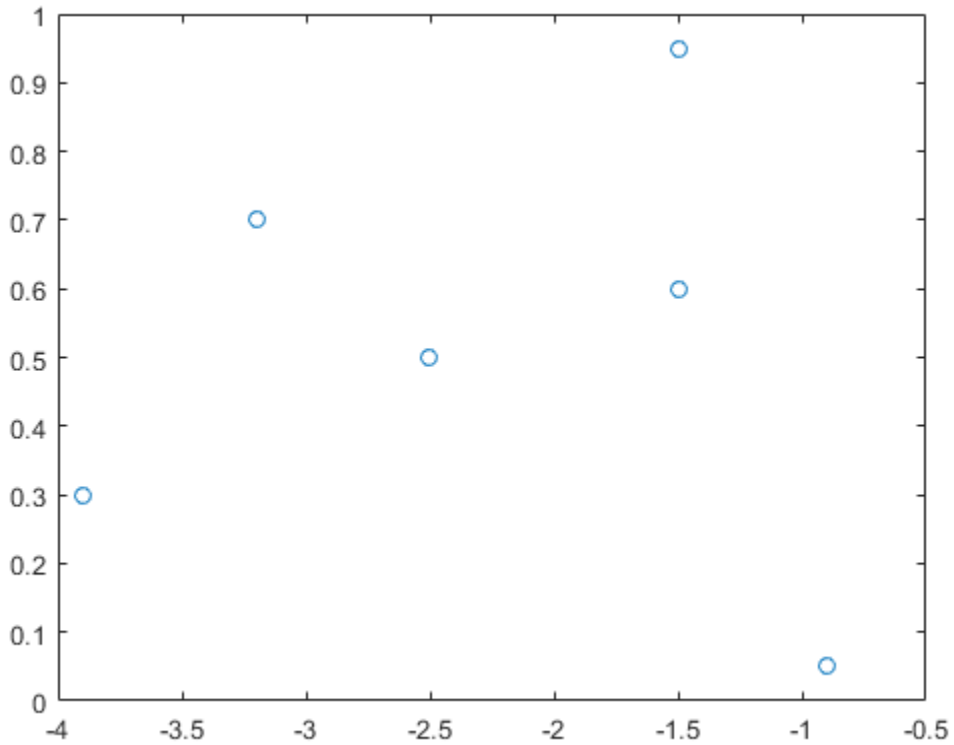
$$(\alpha, \beta) = \begin{cases} (-0.9, 0.05) \\ (-1.5, 0.6) \\ (-1.5, 0.95) \\ (-2.5, 0.5) \\ (-3.2, 0.7) \\ (-3.9, 0.3) \end{cases}.$$

Specify the  $(\alpha, \beta)$  sample values as vectors.

```
alpha = [-0.9;-1.5;-1.5;-2.5;-3.2;-3.9];  
beta = [0.05;0.6;0.95;0.5;0.7;0.3];  
domain = struct('alpha',alpha,'beta',beta);
```

Instead of a regular grid of  $(\alpha, \beta)$  values, here the system is sampled at irregularly spaced points on  $(\alpha, \beta)$ -space.

```
plot(alpha,beta,'o')
```



Specify the basis functions.

```
shapefcn = @(x,y) [x,y,x*y];
```

Create the tunable model of the gain surface using these sampled function values.

```
K = tunableSurface('K',1,domain,shapefcn)
```

```
K =
```

```
Tunable surface "K" of scalar gains with:  
* Scheduling variables: alpha,beta  
* Basis functions: alpha,beta,alpha*beta
```

\* Design points: 6x1 grid of (alpha,beta) values

The domain is the list of six  $(\alpha, \beta)$  pairs.

- “Tuning of Gain-Scheduled Three-Loop Autopilot”
- “Gain-Scheduled Control of a Chemical Reactor”

## Input Arguments

### **name** — Identifying label for the tunable gain

character vector

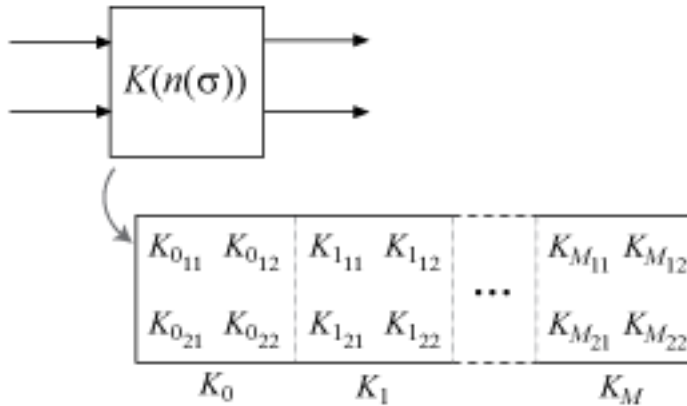
Identifying label for the tunable gain surface, specified as a character vector. `tunableSurface` uses this name for the `realp` block that represents the tunable coefficients of the surface. Therefore, you can use this name to refer to the tunable gain coefficients within a `genss` model of a control system or an `sITuner` interface.

### **K0init** — Initial value of constant term

scalar | array

Initial value of the constant term in the tunable gain surface, specified as a scalar or an array. The dimensions of `K0init` determine the I/O dimensions of the gain surface. For example, if the gain surface represents a two-input, two-output gain, you can set `K0init = ones(2)`. The remaining coefficients `K1, K2, . . .` always have the same size as `K0`. The tunable coefficients automatically expand so that the gains in each I/O channel are tuned independently.

For example, for a two-input, two-output surface, there is a set of expansion coefficients for each entry in the gain matrix.



Each entry  $K_{ij}$  in the tunable gain matrix  $K(n(o))$  is given by:

$$K_{ij}(n(\sigma)) = K_{ij_0} + K_{ij_1} F_1(n(\sigma)) + \dots + K_{ij_M} F_M(n(\sigma)).$$

**domain — Design points**

structure

Design points at which the gain surface is tuned, specified as a structure. The structure has fields containing the scheduling variables values at which you sample the plant for gain-scheduled tuning. For example, suppose that you want to tune a gain that varies as a function of two scheduling variables,  $\alpha$  and  $V$ . You linearize the plant at a grid of  $\alpha$  and  $V$  values, with  $\alpha = [0.5, 0.10, 0.15]$  and  $V = [700, 800, 900, 1000]$

. Specify the design points as follows:

```
[alpha,V] = ndgrid([0.5,0.10,0.15],[700,800,900,1000]);
domain = struct('alpha',alpha,'V',V);
```

The design points do not have to lie on a rectangular or regularly spaced grid (see “Gain Surface Over Nonregular Grid” on page 2-1104). However, for best results use design points that cover the full range of operating conditions. Since tuning only considers these design points, the validity of the tuned gain schedule is questionable at operating conditions far from the design points.

**shapefcn — Basis functions**

function handle

Basis functions used to model the gain surface in terms of the scheduling variables, specified as a function handle. The function associated with the handle takes normalized values of the scheduling variables as inputs and returns a vector of basis-function values. The basis functions always operate on the normalized range  $[-1,1]$ . `tunableSurface` implicitly normalizes the scheduling variables to this interval.

For example, consider the scheduling-variable values  $\alpha = [0.5,0.10,0.15]$  and  $V = [700,800,900,1000]$ . The following expression creates basis functions for a gain surface that is bilinear in these variables:

```
shapefcn = @(x,y) [x y x*y];
```

`shapefcn` is an anonymous function of two variables. The basis functions describe a parameterized gain  $G(\alpha, V) = G_0 + G_1x + G_2y + G_3xy$ . The normalized variables  $x(\alpha)$  and  $y(V)$  map the  $\alpha$  and  $V$  values of `domain` to the normalization interval  $[-1,1]$ . For example, the normalization of  $\alpha$  is:

$$x(\alpha) = \frac{2}{\Delta\alpha}(\alpha - \alpha_{mid}).$$

$\Delta\alpha$  is the difference between the maximum and minimum  $\alpha$  values in `domain`, and  $\alpha_{mid}$  is the midpoint between these values.

You can use anonymous functions to specify any set of basis functions that you need to describe the variable gain. Alternatively, you can use helper functions to generate basis functions automatically for commonly used expansions:

- `polyBasis` — Power series expansion and Chebyshev expansion.
- `fourierBasis` — Periodic Fourier series expansion. The basis functions generated by `fourierBasis` are periodic such that a gain surface  $K$  defined by those functions satisfies  $K(-1) = K(1)$ . When you create a gain surface using `tunableSurface`, the software normalizes the scheduling-variable range that you specify with `domain` to the interval  $[-1,1]$ . Therefore, if you use periodic basis functions, then the sampled range of the corresponding scheduling variable must be exactly one period. This restriction ensures that the periodicity of the basis function matches that of the scheduling variable. For example, if the periodically varying scheduling variable is an angle that ranges from 0 to  $2\pi$ , then the corresponding values in `domain` must also range from 0 to  $2\pi$ .

- **ndBasis** — Build multidimensional expansions from lower-dimensional expansions. This function is useful when you want to use different basis functions for different scheduling variables.

See the reference pages for those functions for more information about the basis functions they generate.

## Output Arguments

### **K** — Tunable gain surface

`tunableSurface` object

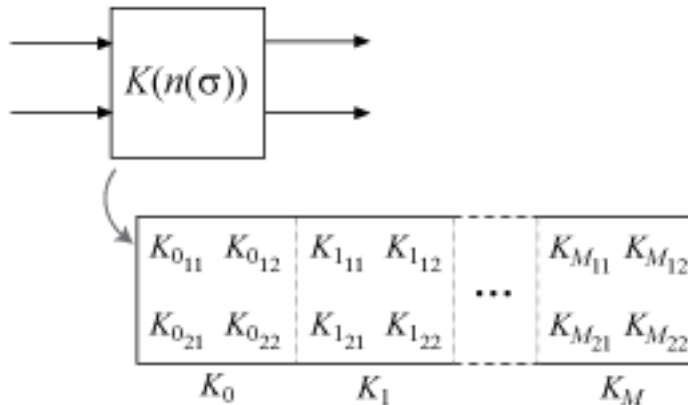
Tunable gain surface, returned as a `tunableSurface` object. This object has the following properties that store the coefficients, basis functions, and other information about the gain surface:

- **Name** — Name of the gain surface, specified as a character vector. When you create the gain surface, the `name` input argument sets the initial value of this property.
- **BasisFunctions** — Basis functions, specified as a function handle. When you create the gain surface, the `shapefcn` input argument sets the initial value of this property.
- **Coefficients** — Tunable coefficients of the gain surface, specified as an array-valued `realp` tunable parameter. The dimensions of `K0init` and the number of basis functions in `shapefcn` determine the dimensions of `K.Coefficients`.

For scalar gains, `K.Coefficients` has dimensions  $[1, M+1]$ , where  $M$  is the number of basis functions. The entries in `K.Coefficients` correspond to the tunable coefficients  $K_0, \dots, K_M$ .

For array-valued gains, each coefficient expands to the dimension of `K0init`. These expanded coefficients are concatenated horizontally in `K.Coefficients`. Therefore, for example, for a two-input, two-output gain surface, `K.Coefficients` has dimensions  $[2, 2(M+1)]$ .





Each entry  $K_{ij}$  in the tunable gain matrix  $K(n(\sigma))$  is given by:

$$K_{ij}(n(\sigma)) = K_{ij_0} + K_{ij_1} F_1(n(\sigma)) + \dots + K_{ij_M} F_M(n(\sigma)).$$

- **SamplingGrid** — Grid of design points, specified as a data structure. When you create the gain surface, the `domain` input argument sets the initial value of this property.

## More About

### Tips

- To tune a gain surface in a control system modeled in MATLAB: Connect the gain surface with an array of plant models corresponding to the design points in `domain`. For example, suppose `G` is such an array, and `K` represents a variable integration time. The following command builds a closed-loop model that you can tune with the `systeme` command.

```
C0 = tf(K, [1 0]);
T0 = feedback(C0*G, 1);
```

- To tune a gain surface in a control system modeled in Simulink: Use the gain surface to parameterize a Lookup Table or interpolation block in the Simulink model. For example, suppose `ST0` is an `sITuner` interface to a Simulink model, and `GainTable`

is the name of a tuned block in the interface. The following command sets the parameterization of `GainTable` to the tunable gain surface.

```
STO = setBlockParam(STO, 'GainTable',K);
```

- “Gain-Scheduled Control Systems”
- “Parametric Gain Surfaces”

### See Also

#### Functions

`evalSurf` | `fourierBasis` | `ndBasis` | `ndgrid` | `polyBasis` | `systune` | `viewSurf`

**Introduced in R2015b**

# tunableTF

Tunable transfer function with fixed number of poles and zeros

## Syntax

```
blk = tunableTF(name,Nz,Np)
blk = tunableTF(name,Nz,Np,Ts)
blk = tunableTF(name,sys)
```

## Description

Model object for creating tunable SISO transfer function models of fixed order. `tunableTF` lets you parametrize a transfer function of a given order for parameter studies or for automatic tuning with tuning commands such as `systemtune` or `looptune`.

`tunableTF` is part of the Control Design Block family of parametric models. Other Control Design Blocks include `tunablePID`, `tunableSS`, and `tunableGain`.

## Construction

`blk = tunableTF(name,Nz,Np)` creates the parametric SISO transfer function:

$$blk = \frac{a_m s^m + a_{m-1} s^{m-1} + \dots + a_1 s + a_0}{s^n + b_{n-1} s^{n-1} + \dots + b_1 s + b_0}.$$

$n = Np$  is the maximum number of poles of `blk`, and  $m = Nz$  is the maximum number of zeros. The tunable parameters are the numerator and denominator coefficients  $a_0, \dots, a_m$  and  $b_0, \dots, b_{n-1}$ . The leading coefficient of the denominator is fixed to 1.

`blk = tunableTF(name,Nz,Np,Ts)` creates a discrete-time parametric transfer function with sample time `Ts`.

`blk = tunableTF(name, sys)` uses the `tf` model `sys` to set the number of poles, number of zeros, sample time, and initial parameter values.

### Input Arguments

#### **name**

Parametric transfer function `Name`, specified as a character vector such as `'filt'` or `'DM'`. (See “Properties” on page 2-1114.)

#### **Nz**

Nonnegative integer specifying the number of zeros of the parametric transfer function `blk`.

#### **Np**

Nonnegative integer specifying the number of poles of the parametric transfer function `blk`.

#### **Ts**

Scalar sample time.

#### **sys**

`tf` model providing number of poles, number of zeros, sample time, and initial values of the parameters of `blk`.

### Properties

#### **Numerator, Denominator**

Parametrization of the numerator coefficients  $a_m, \dots, a_0$  and the denominator coefficients  $1, b_{n-1}, \dots, b_0$  of the tunable transfer function `blk`.

`blk.Numerator` and `blk.Denominator` are `param.Continuous` objects. For general information about the properties of these `param.Continuous` objects, see the `param.Continuous` object reference page.

The following fields of `blk.Numerator` and `blk.Denominator` are used when you tune `blk` using `hinfstruct`:

Field	Description
Value	<p>Array of current values of the numerator <math>a_m, \dots, a_0</math> or the denominator coefficients <math>1, b_{n-1}, \dots, b_0</math>. <code>blk.Numerator.Value</code> has length <math>Nz + 1</math>. <code>blk.Denominator.Value</code> has length <math>Np + 1</math>. The leading coefficient of the denominator (<code>blk.Denominator.Value(1)</code>) is always fixed to 1.</p> <p>By default, the coefficients initialize to values that yield a stable, strictly proper transfer function. Use the input <code>sys</code> to initialize the coefficients to different values.</p> <p><code>hinfstruct</code> tunes all values except those whose <code>Free</code> field is zero.</p>
Free	<p>Array of logical values determining whether the coefficients are fixed or tunable. For example,</p> <ul style="list-style-type: none"> <li>• If <code>blk.Numerator.Free(j) = 1</code>, then <code>blk.Numerator.Value(j)</code> is tunable.</li> <li>• If <code>blk.Numerator.Free(j) = 0</code>, then <code>blk.Numerator.Value(j)</code> is fixed.</li> </ul> <p>Default: <code>blk.Denominator.Free(1) = 0</code>; all other entries are 1.</p>
Minimum	<p>Minimum value of the parameter. This property places a lower bound on the tuned value of the parameter. For example, setting <code>blk.Numerator.Minimum(1) = 0</code> ensures that the leading coefficient of the numerator remains positive.</p> <p>Default: <code>-Inf</code>.</p>

Field	Description
Maximum	Maximum value of the parameter. This property places an upper bound on the tuned value of the parameter. For example, setting <code>blk.Numerator.Maximum(1) = 1</code> ensures that the leading coefficient of the numerator does not exceed 1. Default: <code>Inf</code> .

### Ts

Sample time. For continuous-time models,  $T_s = 0$ . For discrete-time models,  $T_s$  is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set  $T_s = -1$ .

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### TimeUnit

Units for the time variable, the sample time  $T_s$ , and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** `'seconds'`

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, `'controls'`.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, `'seconds'`.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** `''` for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### **OutputName**

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, `'measurements'`.
- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** `''` for all output channels



## OutputUnit

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, 'seconds'.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

## OutputGroup

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### Name

System name, specified as a character vector. For example, 'system\_1'.

**Default:** ''

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, 'System is MIMO'.

**Default:** {}

### UserData

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** []

## Examples

Create a parametric SISO transfer function with two zeros, four poles, and at least one integrator.

A transfer function with an integrator includes a factor of  $1/s$ . Therefore, to ensure that a parametrized transfer function has at least one integrator regardless of the parameter values, fix the lowest-order coefficient of the denominator to zero.

```
blk = tunableTF('tfblock',2,4); % two zeros, four poles
blk.Denominator.Value(end) = 0; % set last denominator entry to zero
blk.Denominator.Free(end) = 0; % fix it to zero
```

Create a parametric transfer function, and assign names to the input and output.

```
blk = tunableTF('tfblock',2,3);
blk.InputName = {'error'}; % assign input name
blk.OutputName = {'control'}; % assign output name
```

## More About

### Tips

- To convert a `tunableTF` parametric model to a numeric (non-tunable) model object, use model commands such as `tf`, `zpk`, or `ss`.
- “Control Design Blocks”
- “Models with Tunable Coefficients”

### See Also

`genss` | `hinfstruct` | `looptune` | `systune` | `tunableGain` | `tunablePID` | `tunablePID2` | `tunableSS`

**Introduced in R2011a**

## **tzero**

Invariant zeros of linear system

### **Syntax**

```
z = tzero(sys)
z = tzero(A,B,C,D,E)
z = tzero( ____,tol)
[z,nrank] = tzero( ____)
```

### **Description**

`z = tzero(sys)` returns the invariant zeros of the multi-input, multi-output (MIMO) dynamic system, `sys`. If `sys` is a minimal realization, the invariant zeros coincide with the transmission zeros of `sys`.

`z = tzero(A,B,C,D,E)` returns the invariant zeros of the state-space model

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

Omit `E` for an explicit state-space model ( $E = I$ ).

`z = tzero( ____,tol)` specifies the relative tolerance, `tol`, controlling rank decisions.

`[z,nrank] = tzero( ____)` also returns the normal rank of the transfer function of `sys` or of the transfer function  $H(s) = D + C(sE - A)^{-1}B$ .

### **Input Arguments**

#### **sys**

MIMO dynamic system model. If `sys` is not a state-space model, then `tzero` computes `tzero(ss(sys))`.

**A, B, C, D, E**

State-space matrices describing the linear system

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

$$y = Cx + Du.$$

tzero does not scale the state-space matrices when you use the syntax `z = tzero(A,B,C,D,E)`. Use `prescale` if you want to scale the matrices before using tzero.

Omit E to use  $E = I$ .

**tol**

Relative tolerance controlling rank decisions. Increasing tolerance helps detect nonminimal modes and eliminate very large zeros (near infinity). However, increased tolerance might artificially inflate the number of transmission zeros.

**Default:**  $\text{eps}^{(3/4)}$

## Output Arguments

**z**

Column vector containing the invariant zeros of `sys` or the state-space model described by `A, B, C, D, E`.

**nrank**

Normal rank of the transfer function of `sys` or of the transfer function  $H(s) = D + C(sE - A)^{-1}B$ . The *normal rank* is the rank for values of  $s$  other than the transmission zeros.

To obtain a meaningful result for `nrank`, the matrix  $s*E - A$  must be regular (invertible for most values of  $s$ ). In other words, `sys` or the system described by `A, B, C, D, E` must have a finite number of poles.

## Examples

### Find Transmission Zeros of MIMO Transfer Function

Create a MIMO transfer function, and locate its invariant zeros.

```
s = tf('s');  
H = [1/(s+1) 1/(s+2); 1/(s+3) 2/(s+4)];  
z = tzero(H)
```

```
z =  
  
-2.5000 + 1.3229i  
-2.5000 - 1.3229i
```

The output is a column vector listing the locations of the invariant zeros of  $H$ . This output shows that  $H$  has a complex pair of invariant zeros. Confirm that the invariant zeros coincide with the transmission zeros.

Check whether the first invariant zero is a transmission zero of  $H$ .

If  $z(1)$  is a transmission zero of  $H$ , then  $H$  drops rank at  $s = z(1)$ .

```
H1 = evalfr(H,z(1));  
svd(H1)
```

```
ans =  
  
1.5000  
0.0000
```

$H1$  is the transfer function,  $H$ , evaluated at  $s = z(1)$ .  $H1$  has a zero singular value, indicating that  $H$  drops rank at that value of  $s$ . Therefore,  $z(1)$  is a transmission zero of  $H$ .

A similar analysis shows that  $z(2)$  is also a transmission zero.

### Identify Unobservable and Uncontrollable Modes of MIMO Model

Obtain a MIMO model.

```
load ltiexamples gasf
size(gasf)
```

State-space model with 4 outputs, 6 inputs, and 25 states.

`gasf` is a MIMO model that might contain uncontrollable or unobservable states.

To identify the unobservable and uncontrollable modes of `gasf`, you need the state-space matrices `A`, `B`, `C`, and `D` of the model. `tzero` does not scale state-space matrices. Therefore, use `prescale` with `ssdata` to scale the state-space matrices of `gasf`.

```
[A,B,C,D] = ssdata(prescale(gasf));
```

Identify the uncontrollable states of `gasf`.

```
uncon = tzero(A,B,[],[])
```

```
uncon =
    -0.0568
    -0.0568
    -0.0568
    -0.0568
    -0.0568
    -0.0568
```

When you provide `A` and `B` matrices to `tzero`, but no `C` and `D` matrices, the command returns the eigenvalues of the uncontrollable modes of `gasf`. The output shows that there are six degenerate uncontrollable modes.

Identify the unobservable states of `gasf`.

```
unobs = tzero(A,[],C,[])
```

```
unobs =
    0×1 empty double column vector
```

When you provide **A** and **C** matrices, but no **B** and **D** matrices, the command returns the eigenvalues of the unobservable modes. The empty result shows that `gasf` contains no unobservable states.

## Alternatives

To calculate the zeros and gain of a single-input, single-output (SISO) system, use `zero`.

## More About

### Invariant zeros

For a MIMO state-space model

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

the *invariant zeros* are the complex values of  $s$  for which the rank of the system matrix

$$\begin{bmatrix} A - sE & B \\ C & D \end{bmatrix}$$

drops from its normal value. (For explicit state-space models,  $E = I$ ).

### Transmission zeros

For a MIMO state-space model

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

the *transmission zeros* are the complex values of  $s$  for which the rank of the equivalent transfer function  $H(s) = D + C(sE - A)^{-1}B$  drops from its normal value. (For explicit state-space models,  $E = I$ ).



Transmission zeros are a subset of the invariant zeros. For minimal realizations, the transmission zeros and invariant zeros are identical.

### Tips

- You can use the syntax `z = tzero(A,B,C,D,E)` to find the uncontrollable or unobservable modes of a state-space model. When `C` and `D` are empty or zero, `tzero` returns the uncontrollable modes of  $(A - sE, B)$ . Similarly, when `B` and `D` are empty or zero, `tzero` returns the unobservable modes of  $(C, A - sE)$ . See “Identify Unobservable and Uncontrollable Modes of MIMO Model” on page 2-1124 for an example.

### Algorithms

`tzero` is based on SLICOT routines AB08ND, AG08BD, and AB8NXX. `tzero` implements the algorithms in [1] and [2].

## References

- [1] Emami-Naeini, A. and P. Van Dooren, "Computation of Zeros of Linear Multivariable Systems," *Automatica*, 18 (1982), pp. 415–430.
- [2] Misra, P, P. Van Dooren, and A. Varga, “Computation of Structural Invariants of Generalized State-Space Systems,” *Automatica*, 30 (1994), pp. 1921-1936.

### See Also

pole | pzmap | zero

Introduced in R2012a

## unscentedKalmanFilter

Create unscented Kalman filter object for online state estimation

### Syntax

```
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,  
InitialState)  
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,  
InitialState,Name,Value)  
obj = unscentedKalmanFilter(Name,Value)
```

### Description

`obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState)` creates an unscented Kalman filter object for online state estimation of a discrete-time nonlinear system. `StateTransitionFcn` is a function that calculates the state of the system at time  $k$ , given the state vector at time  $k-1$ . `MeasurementFcn` is a function that calculates the output measurement of the system at time  $k$ , given the state at time  $k$ . `InitialState` specifies the initial value of the state estimates.

After creating the object, use the `correct` and `predict` commands to update state estimates and state estimation error covariance values using a discrete-time unscented Kalman filter algorithm and real-time data.

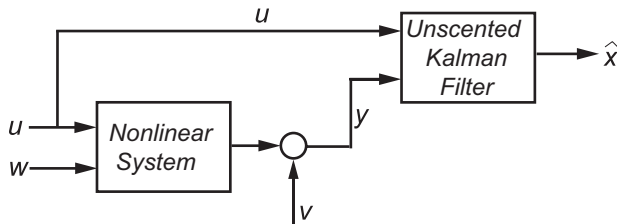
`obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState,Name,Value)` specifies additional attributes of the unscented Kalman filter object using one or more `Name,Value` pair arguments.

`obj = unscentedKalmanFilter(Name,Value)` creates an unscented Kalman filter object with properties specified using one or more `Name,Value` pair arguments. The properties that you do not specify retain their default value.

### Object Description

`unscentedKalmanFilter` creates an object for online state estimation of a discrete-time nonlinear system using the discrete-time unscented Kalman filter algorithm.

Consider a plant with states  $x$ , input  $u$ , output  $y$ , process noise  $w$ , and measurement noise  $v$ . Assume that you can represent the plant as a nonlinear system.



The algorithm computes the state estimates  $\hat{x}$  of the nonlinear system using state transition and measurement functions specified by you. The software lets you specify the noise in these functions as additive or nonadditive:

- **Additive Noise Terms** — The state transition and measurements equations have the following form:

$$x[k] = f(x[k-1], u_s[k-1]) + w[k-1]$$

$$y[k] = h(x[k], u_m[k]) + v[k]$$

Here  $f$  is a nonlinear state transition function that describes the evolution of states  $x$  from one time step to the next. The nonlinear measurement function  $h$  relates  $x$  to the measurements  $y$  at time step  $k$ .  $w$  and  $v$  are the zero-mean, uncorrelated process and measurement noises, respectively. These functions can also have additional input arguments that are denoted by  $u_s$  and  $u_m$  in the equations. For example, the additional arguments could be time step  $k$  or the inputs  $u$  to the nonlinear system. There can be multiple such arguments.

Note that the noise terms in both equations are additive. That is,  $x(k)$  is linearly related to the process noise  $w(k-1)$ , and  $y(k)$  is linearly related to the measurement noise  $v(k)$ .

- **Nonadditive Noise Terms** — The software also supports more complex state transition and measurement functions where the state  $x[k]$  and measurement  $y[k]$  are nonlinear functions of the process noise and measurement noise, respectively. When the noise terms are nonadditive, the state transition and measurements equation have the following form:

$$x[k] = f(x[k-1], w[k-1], u_s[k-1])$$

$$y[k] = h(x[k], v[k], u_m[k])$$

When you perform online state estimation, you first create the nonlinear state transition function  $f$  and measurement function  $h$ . You then construct the `unscentedKalmanFilter` object using these nonlinear functions, and specify whether the noise terms are additive or nonadditive. After you create the object, you use the `predict` command to predict state estimates at the next time step, and `correct` to correct state estimates using the unscented Kalman filter algorithm and real-time data. For information about the algorithm, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”.

You can use the following commands with `unscentedKalmanFilter` objects:

Command	Description
<code>correct</code>	Correct the state and state estimation error covariance at time step $k$ using measured data at time step $k$ .
<code>predict</code>	Predict the state and state estimation error covariance at time the next time step.
<code>clone</code>	Create another object with the same object property values.  Do not create additional objects using syntax <code>obj2 = obj</code> . Any changes made to the properties of the new object created in this way ( <code>obj2</code> ) also change the properties of the original object ( <code>obj</code> ).

For `unscentedKalmanFilter` object properties, see “Properties” on page 2-1136.

## Examples

### Create Unscented Kalman Filter Object for Online State Estimation

To define an unscented Kalman filter object for estimating the states of your system, you write and save the state transition function and measurement function for the system.

In this example, use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions describe a discrete-approximation to van der Pol oscillator with nonlinearity parameter, `mu`, equal to 1. The oscillator has two states.

Specify an initial guess for the two states. You specify the initial state guess as an `M`-element row or column vector, where `M` is the number of states.

```
initialStateGuess = [1;0];
```

Create the unscented Kalman filter object. Use function handles to provide the state transition and measurement functions to the object.

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,initialStateGuess);
```

The object has a default structure where the process and measurement noise are additive.

To estimate the states and state estimation error covariance from the constructed object, use the `correct` and `predict` commands and real-time data.

### Specify Process and Measurement Noise Covariances in Unscented Kalman Filter Object

Create an unscented Kalman filter object for a van der Pol oscillator with two states and one output. Use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. These functions are written for additive process and measurement noise terms. Specify the initial state values for the two states as `[2;0]`.

Since the system has two states and the process noise is additive, the process noise is a 2-element vector and the process noise covariance is a 2-by-2 matrix. Assume there is no cross-correlation between process noise terms, and both the terms have the same variance 0.01. You can specify the process noise covariance as a scalar. The software uses the scalar value to create a 2-by-2 diagonal matrix with 0.01 on the diagonals.

Specify the process noise covariance during object construction.

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0],...  
    'ProcessNoise',0.01);
```

Alternatively, you can specify noise covariances after object construction using dot notation. For example, specify the measurement noise covariance as 0.2.

```
obj.MeasurementNoise = 0.2;
```

Since the system has only one output, the measurement noise is a 1-element vector and the `MeasurementNoise` property denotes the variance of the measurement noise.

### Specify Nonadditive Measurement Noise in Unscented Kalman Filter Object

Create an unscented Kalman filter object for a van der Pol oscillator with two states and one output. Assume that the process noise terms in the state transition function are additive. That is, there is a linear relation between the state and process noise. Also assume that the measurement noise terms are nonadditive. That is, there is a nonlinear relation between the measurement and measurement noise.

```
obj = unscentedKalmanFilter('HasAdditiveMeasurementNoise', false);
```

Specify the state transition function and measurement functions. Use the previously written and saved functions, `vdpStateFcn.m` and `vdpMeasurementNonAdditiveNoiseFcn.m`.

The state transition function is written assuming the process noise is additive. The measurement function is written assuming the measurement noise is nonadditive.

```
obj.StateTransitionFcn = @vdpStateFcn;  
obj.StateTransitionFcn = @vdpMeasurementNonAdditiveNoiseFcn;
```

Specify the initial state values for the two states as `[2;0]`.

```
obj.State = [2;0];
```

You can now use the `correct` and `predict` commands to estimate the state and state estimation error covariance values from the constructed object.

### Specify Additional Inputs in State Transition and Measurement Functions

Consider a nonlinear system with input  $u$  whose state  $x$  and measurement  $y$  evolve according to the following state transition and measurement equations:

$$x[k] = \sqrt{x[k-1] + u[k-1]} + w[k-1]$$

$$y[k] = x[k] + 2 * u[k] + v[k]^2$$

The process noise  $w$  of the system is additive while the measurement noise  $v$  is nonadditive.

Create the state transition function and measurement function for the system. Specify the functions with an additional input  $u$ .

```
f = @(x,u)(sqrt(x+u));
h = @(x,v,u)(x+2*u+v^2);
```

$f$  and  $h$  are function handles to the anonymous functions that store the state transition and measurement functions, respectively. In the measurement function, because the measurement noise is nonadditive,  $v$  is also specified as an input. Note that  $v$  is specified as an input before the additional input  $u$ .

Create an unscented Kalman filter object for estimating the state of the nonlinear system using the specified functions. Specify the initial value of the state as 1, and the measurement noise as nonadditive.

```
obj = unscentedKalmanFilter(f,h,1,'HasAdditiveMeasurementNoise',false);
```

Specify the measurement noise covariance.

```
obj.MeasurementNoise = 0.01;
```

You can now estimate the state of the system using the `predict` and `correct` commands. You pass the values of  $u$  to `predict` and `correct`, which in turn pass them to the state transition and measurement functions, respectively.

Correct the state estimate with measurement  $y[k]=0.8$  and input  $u[k]=0.2$  at time step  $k$ .

```
correct(obj,0.8,0.2)
```

Predict the state at next time step, given  $u[k]=0.2$ .

```
predict(obj,0.2)
```

- “Nonlinear State Estimation Using Unscented Kalman Filter”
- “Generate Code for Online State Estimation in MATLAB”

## Input Arguments

**StateTransitionFcn** — State transition function  
function handle

State transition function  $f$ , specified as a function handle. The function calculates the  $M$ -element state vector of the system at time step  $k$ , given the state vector at time step  $k-1$ .  $M$  is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system, and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise  $w$  is additive, and the state transition function specifies how the states evolve as a function of state values at the previous time step:

$$x(k) = f(x(k-1), Us1, \dots, Usn)$$

Where  $x(k)$  is the estimated state at time  $k$ , and  $Us1, \dots, Usn$  are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.

- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

$$x(k) = f(x(k-1), w(k-1), Us1, \dots, Usn)$$

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.

### **MeasurementFcn — Measurement function**

function handle

Measurement function  $h$ , specified as a function handle. The function calculates the  $N$ -element output measurement vector of the nonlinear system at time step  $k$ , given the state vector at time step  $k$ .  $N$  is the number of measurements of the system. You write and save the measurement function, and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.



The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is true — The measurement noise  $v$  is additive, and the measurement function specifies how the measurements evolve as a function of state values:

$$y(k) = h(x(k), Um1, \dots, Umn)$$

Where  $y(k)$  and  $x(k)$  are the estimated output and estimated state at time  $k$ , and  $Um1, \dots, Umn$  are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command, which in turn passes them to the measurement function.

- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

$$y(k) = h(x(k), v(k), Um1, \dots, Umn)$$

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

### **InitialState** — Initial state estimates

vector

Initial state estimates, specified as an  $M$ -element vector, where  $M$  is the number of states in the system. Specify the initial state values based on your knowledge of the system.

The specified value is stored in the `State` property of the object. If you specify `InitialState` as a column vector then `State` is also a column vector, and `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned.

If you want a filter with single-precision floating-point variables, specify `InitialState` as a single-precision vector variable. For example, for a two-state system with state transition and measurement functions `vdpStateFcn.m` and `vdpMeasurementFcn.m`, create the unscented Kalman filter object with initial states `[1;2]` as follows:

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,single([1;2]))
```

Data Types: double | single

### Name-Value Pair Arguments

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside single quotes (' '). You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Use **Name**, **Value** arguments to specify properties of `unscentedKalmanFilter` object during object creation. For example, to create an unscented Kalman filter object and specify the process noise covariance as 0.01:

```
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState,'ProcessNoise',0.01);
```

### Properties

`unscentedKalmanFilter` object properties are of three types:

- Tunable properties that you can specify multiple times, either during object construction using **Name**, **Value** arguments, or any time afterwards during state estimation. After object creation, use dot notation to modify the tunable properties.

```
obj = unscentedKalmanFilter(StateTransitionFcn,MeasurementFcn,InitialState);  
obj.ProcessNoise = 0.01;
```

The tunable properties are **State**, **StateCovariance**, **ProcessNoise**, **MeasurementNoise**, **Alpha**, **Beta**, and **Kappa**.

- Nontunable properties that you can specify once, either during object construction or afterward using dot notation. Specify these properties before state estimation using **correct** and **predict**. The **StateTransitionFcn** and **MeasurementFcn** properties belong to this category.
- Nontunable properties that you must specify during object construction. The **HasAdditiveProcessNoise** and **HasAdditiveMeasurementNoise** properties belong to this category.

#### **Alpha** — Spread of sigma points

1e-3 (default) | scalar value between 0 and 1

Spread of sigma points around mean state value, specified as a scalar value between 0 and 1 ( $0 < \text{Alpha} \leq 1$ ).

The unscented Kalman filter algorithm treats the state of the system as a random variable with mean value `State` and variance `StateCovariance`. To compute the state and its statistical properties at the next time step, the algorithm first generates a set of state values distributed around the mean `State` value by using the unscented transformation. These generated state values are called sigma points. The algorithm uses each of the sigma points as an input to the state transition and measurement functions to get a new set of transformed state points and measurements. The transformed points are used to compute the state and state estimation error covariance value at the next time step.

The spread of the sigma points around the mean state value is controlled by two parameters `Alpha` and `Kappa`. A third parameter, `Beta`, impacts the weights of the transformed points during state and measurement covariance calculations:

- `Alpha` — Determines the spread of the sigma points around the mean state value. It is usually a small positive value. The spread of sigma points is proportional to `Alpha`. Smaller values correspond to sigma points closer to the mean state.
- `Kappa` — A second scaling parameter that is usually set to 0. Smaller values correspond to sigma points closer to the mean state. The spread is proportional to the square-root of `Kappa`.
- `Beta` — Incorporates prior knowledge of the distribution of the state. For Gaussian distributions, `Beta = 2` is optimal.

If you know the distribution of state and state covariance, you can adjust these parameters to capture the transformation of higher-order moments of the distribution. The algorithm can track only a single peak in the probability distribution of the state. If there are multiple peaks in the state distribution of your system, you can adjust these parameters so that the sigma points stay around a single peak. For example, choose a small `Alpha` to generate sigma points close to the mean state value.

For more information, see “Unscented Kalman Filter Algorithm”.

`Alpha` is a tunable property. You can change it using dot notation.

### **Beta** — Characterization of state distribution

2 (default) | scalar value greater than or equal to 0

Characterization of the state distribution that is used to adjust weights of transformed sigma points, specified as a scalar value greater than or equal to 0. For Gaussian distributions, `Beta = 2` is an optimal choice.

For more information, see the `Alpha` property description.

`Beta` is a tunable property. You can change it using dot notation.

### **HasAdditiveMeasurementNoise — Measurement noise characteristics**

`true` (default) | `false`

Measurement noise characteristics, specified as one of the following values:

- `true` — Measurement noise  $v$  is additive. The measurement function  $h$  that is specified in `MeasurementFcn` has the following form:

$$y(k) = h(x(k), Um1, \dots, Umn)$$

Where  $y(k)$  and  $x(k)$  are the estimated output and estimated state at time  $k$ , and  $Um1, \dots, Umn$  are any optional input arguments required by your measurement function.

- `false` — Measurement noise is nonadditive. The measurement function specifies how the output measurement evolves as a function of the state *and* measurement noise:

$$y(k) = h(x(k), v(k), Um1, \dots, Umn)$$

`HasAdditiveMeasurementNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

### **HasAdditiveProcessNoise — Process noise characteristics**

`true` (default) | `false`

Process noise characteristics, specified as one of the following values:

- `true` — Process noise  $w$  is additive. The state transition function  $f$  specified in `StateTransitionFcn` has the following form:

$$x(k) = f(x(k-1), Us1, \dots, Usn)$$

Where  $x(k)$  is the estimated state at time  $k$ , and  $Us1, \dots, Usn$  are any additional input arguments required by your state transition function.

- `false` — Process noise is nonadditive. The state transition function specifies how the states evolve as a function of the state *and* process noise at the previous time step:

$$x(k) = f(x(k-1), w(k-1), Us1, \dots, Usn)$$

`HasAdditiveProcessNoise` is a nontunable property, and you can specify it only during object construction. You cannot change it using dot notation.

### **Kappa — Spread of sigma points**

0 (default) | scalar value between 0 and 3

Spread of sigma points around mean state value, specified as a scalar value between 0 and 3 ( $0 \leq \text{Kappa} \leq 3$ ). `Kappa` is typically specified as 0. Smaller values correspond to sigma points closer to the mean state. The spread is proportional to the square-root of `Kappa`. For more information, see the `Alpha` property description.

`Kappa` is a tunable property. You can change it using dot notation.

### **MeasurementFcn — Measurement function**

[] (default) | function handle

Measurement function  $h$ , specified as a function handle. The function calculates the  $N$ -element output measurement vector of the nonlinear system at time step  $k$ , given the state vector at time step  $k$ .  $N$  is the number of measurements of the system. You write and save the measurement function and use it to construct the object. For example, if `vdpMeasurementFcn.m` is the measurement function, specify `MeasurementFcn` as `@vdpMeasurementFcn`. You can also specify `MeasurementFcn` as a function handle to an anonymous function.

The inputs to the function depend on whether you specify the measurement noise as additive or nonadditive in the `HasAdditiveMeasurementNoise` property of the object:

- `HasAdditiveMeasurementNoise` is true — The measurement noise  $v$  is additive, and the measurement function specifies how the measurements evolve as a function of state values:

$$y(k) = h(x(k), Um1, \dots, Umn)$$

Where  $y(k)$  and  $x(k)$  are the estimated output and estimated state at time  $k$ , and  $Um1, \dots, Umn$  are any optional input arguments required by your measurement function. For example, if you are using multiple sensors for tracking an object, an additional input could be the sensor position. During estimation, you pass these additional arguments to the `correct` command which in turn passes them to the measurement function.

- `HasAdditiveMeasurementNoise` is false — The measurement noise is nonadditive, and the measurement function also specifies how the output measurement evolves as a function of the measurement noise:

$$y(k) = h(x(k), v(k), Um1, \dots, Umn)$$

To see an example of a measurement function with additive process noise, type `edit vdpMeasurementFcn` at the command line. To see an example of a measurement function with nonadditive process noise, type `edit vdpMeasurementNonAdditiveNoiseFcn`.

`MeasurementFcn` is a nontunable property. You can specify it once before using the `correct` command either during object construction or using dot notation after object construction. You cannot change it after using the `correct` command.

### **MeasurementNoise — Measurement noise covariance**

1 (default) | scalar | matrix

Measurement noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveMeasurementNoise` property:

- `HasAdditiveMeasurementNoise` is true — Specify the covariance as a scalar or an  $N$ -by- $N$  matrix, where  $N$  is the number of measurements of the system. Specify a scalar if there is no cross-correlation between measurement noise terms and all the terms have the same variance. The software uses the scalar value to create an  $N$ -by- $N$  diagonal matrix.
- `HasAdditiveMeasurementNoise` is false — Specify the covariance as a  $V$ -by- $V$  matrix, where  $V$  is the number of measurement noise terms. `MeasurementNoise` must be specified before using `correct`. After you specify `MeasurementNoise` as a matrix for the first time, to then change `MeasurementNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the measurement noise terms and all the terms have the same variance. The software extends the scalar to a  $V$ -by- $V$  diagonal matrix with the scalar on the diagonals.

`MeasurementNoise` is a tunable property. You can change it using dot notation.

### **ProcessNoise — Process noise covariance**

1 (default) | scalar | matrix

Process noise covariance, specified as a scalar or matrix depending on the value of the `HasAdditiveProcessNoise` property:

- `HasAdditiveProcessNoise` is true — Specify the covariance as a scalar or an  $M$ -by- $M$  matrix, where  $M$  is the number of states of the system. Specify a scalar if there is no cross-correlation between process noise terms, and all the terms have the same variance. The software uses the scalar value to create an  $M$ -by- $M$  diagonal matrix.
- `HasAdditiveProcessNoise` is false — Specify the covariance as a  $W$ -by- $W$  matrix, where  $W$  is the number of process noise terms. `ProcessNoise` must be specified before using `predict`. After you specify `ProcessNoise` as a matrix for the first time, to then change `ProcessNoise` you can also specify it as a scalar. Specify as a scalar if there is no cross-correlation between the process noise terms and all the terms have the same variance. The software extends the scalar to a  $W$ -by- $W$  diagonal matrix.

`ProcessNoise` is a tunable property. You can change it using dot notation.

### **State — State of nonlinear system**

`[]` (default) | vector

State of the nonlinear system, specified as a vector of size  $M$ , where  $M$  is the number of states of the system.

When you use the `predict` command, `State` is updated with the predicted value at time step  $k$  using the state value at time step  $k-1$ . When you use the `correct` command, `State` is updated with the estimated value at time step  $k$  using measured data at time step  $k$ .

The initial value of `State` is the value you specify in the `InitialState` input argument during object creation. If you specify `InitialState` as a column vector, then `State` is also a column vector, and the `predict` and `correct` commands return state estimates as a column vector. Otherwise, a row vector is returned. If you want a filter with single-precision floating-point variables, you must specify `State` as a single-precision variable during object construction using the `InitialState` input argument.

`State` is a tunable property. You can change it using dot notation.

### **StateCovariance — State estimation error covariance**

1 (default) | scalar | matrix

State estimation error covariance, specified as a scalar or an  $M$ -by- $M$  matrix, where  $M$  is the number of states of the system. If you specify a scalar, the software uses the scalar value to create an  $M$ -by- $M$  diagonal matrix.

Specify a high value for the covariance when you do not have confidence in the initial state values that you specify in the `InitialState` input argument.

When you use the `predict` command, `StateCovariance` is updated with the predicted value at time step  $k$  using the state value at time step  $k-1$ . When you use the `correct` command, `StateCovariance` is updated with the estimated value at time step  $k$  using measured data at time step  $k$ .

`StateCovariance` is a tunable property. You can change it using dot notation after using the `correct` or `predict` commands.

### **StateTransitionFcn — State transition function**

[ ] (default) | function handle

State transition function  $f$ , specified as a function handle. The function calculates the  $M$ -element state vector of the system at time step  $k$ , given the state vector at time step  $k-1$ .  $M$  is the number of states of the nonlinear system.

You write and save the state transition function for your nonlinear system and use it to construct the object. For example, if `vdpStateFcn.m` is the state transition function, specify `StateTransitionFcn` as `@vdpStateFcn`. You can also specify `StateTransitionFcn` as a function handle to an anonymous function.

The inputs to the function you write depend on whether you specify the process noise as additive or nonadditive in the `HasAdditiveProcessNoise` property of the object:

- `HasAdditiveProcessNoise` is true — The process noise  $w$  is additive, and the state transition function specifies how the states evolve as a function of state values at previous time step:

$$x(k) = f(x(k-1), Us1, \dots, Usn)$$

Where  $x(k)$  is the estimated state at time  $k$ , and `Us1`, `...`, `Usn` are any additional input arguments required by your state transition function, such as system inputs or the sample time. During estimation, you pass these additional arguments to the `predict` command, which in turn passes them to the state transition function.

- `HasAdditiveProcessNoise` is false — The process noise is nonadditive, and the state transition function also specifies how the states evolve as a function of the process noise:

$$x(k) = f(x(k-1), w(k-1), Us1, \dots, Usn)$$

To see an example of a state transition function with additive process noise, type `edit vdpStateFcn` at the command line.



StateTransitionFcn is a nontunable property. You can specify it once before using the `predict` command either during object construction or using dot notation after object construction. You cannot change it after using the `predict` command.

## Output Arguments

### **obj** — unscented Kalman filter object for online state estimation

`unscentedKalmanFilter` object

Unscented Kalman filter object for online state estimation, returned as an `unscentedKalmanFilter` object. This object is created using the specified properties. Use the `correct` and `predict` commands to estimate the state and state estimation error covariance using the unscented Kalman filter algorithm.

When you use `predict`, `obj.State` and `obj.StateCovariance` are updated with the predicted value at time step  $k$  using the state value at time step  $k-1$ . When you use `correct`, `obj.State` and `obj.StateCovariance` are updated with the estimated values at time step  $k$  using measured data at time step  $k$ .

## More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation”
- “Validate Online State Estimation at the Command Line”
- “Troubleshoot Online State Estimation”

## See Also

### Functions

`clone` | `correct` | `extendedKalmanFilter` | `kalman` | `kalmd` | `predict`

### Blocks

Kalman Filter

Introduced in R2016b

## updateSystem

Update dynamic system data in a response plot

### Syntax

```
updateSystem(h,sys)  
updateSystem(h,sys,N)
```

### Description

`updateSystem(h,sys)` replaces the dynamic system used to compute a response plot with the dynamic system model or model array `sys`, and updates the plot. If the plot with handle `h` contains more than one system response, this syntax replaces the first response in the plot. `updateSystem` is useful, for example, to cause a plot in a GUI to update in response to interactive input. See “Build GUI With Interactive Response-Plot Updates”.

`updateSystem(h,sys,N)` replaces the data used to compute the Nth response in the plot.

### Examples

#### Update System Data in Response Plot

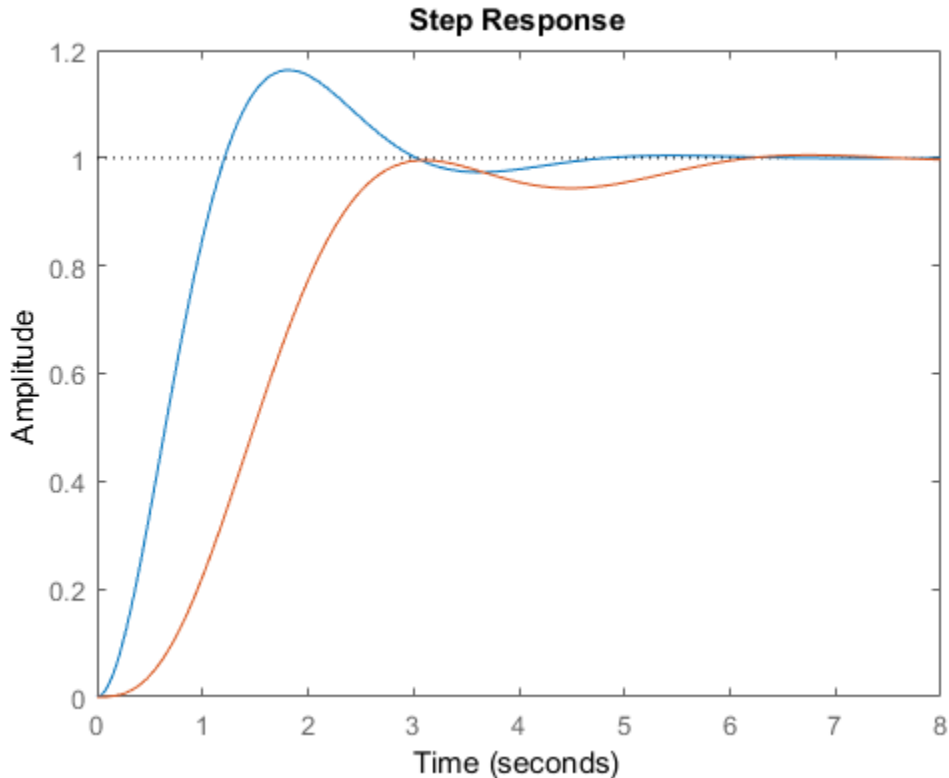
Replace step response data in an existing plot with data computed from a different dynamic system model.

Suppose you have a plant model and pure integrator controller that you designed for that plant. Plot the step responses of the plant and the closed-loop system.

```
w = 2;  
zeta = 0.5;  
G = tf(w^2,[1,2*zeta*w,w^2]);
```

```
C1 = pid(0,0.621);  
CL1 = feedback(G*C1,1);
```

```
h = stepplot(G,CL1);
```



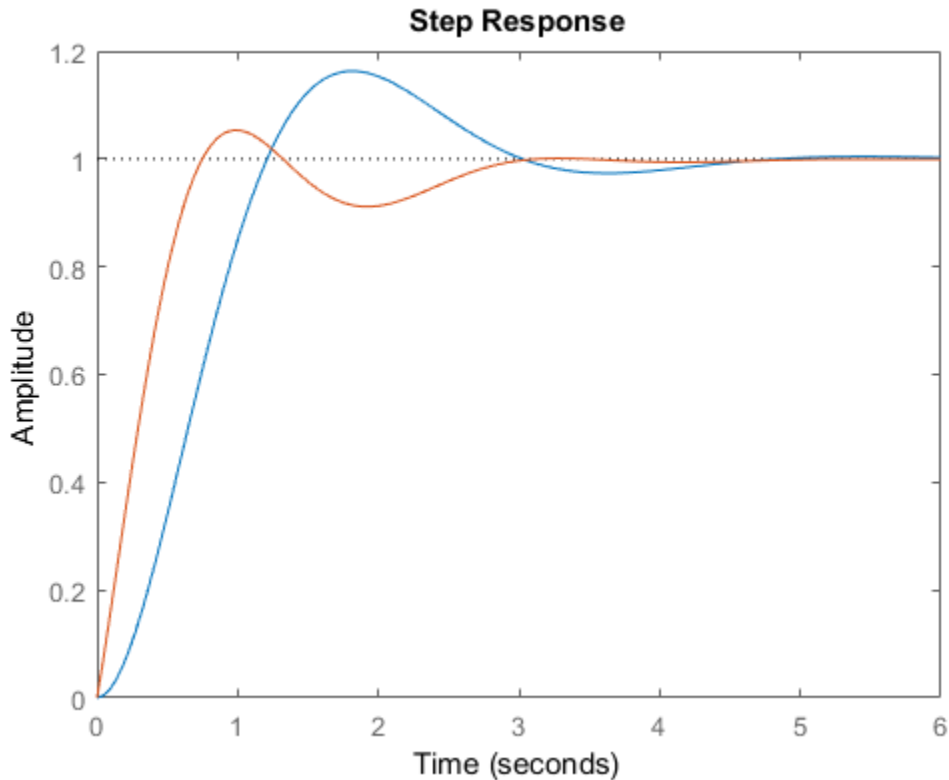
`h` is the plot handle that identifies the plot created by `stepplot`. In this figure, `G` is used to compute the first response, and `CL1` is used to compute the second response. This ordering corresponds to the order of inputs to `stepplot`.

Suppose you also have a PID controller design that you want to analyze. Create a model of the closed-loop system using this alternate controller.

```
C2 = pid(2,2.6,0.4,0.002);  
CL2 = feedback(G*C2,1);
```

Update the step plot to display the second closed-loop system instead of the first. The closed-loop system is the second response in the plot, so specify the index value 2.

```
updateSystem(h,CL2,2);
```



The `updateSystem` command replaces the system used to compute the second response displayed in the plot. Instead of displaying response data derived from `CL1`, the plot now shows data derived from `CL2`.

When you build a GUI that displays a response plot, use `updateSystem` in GUI control callbacks to cause those GUI controls to update the response plot. For an example showing how to implement such a GUI control, see “Build GUI With Interactive Response-Plot Updates”.

- “Build GUI With Interactive Response-Plot Updates”

## Input Arguments

### **h** — Plot to update

plot handle

Plot to update with new system data, specified as a plot handle. Typically, you obtain the plot handle as an output argument of a response plotting command such as `stepplot` or `bodeplot`. For example, the command `h = bodeplot(G)` returns a handle to a plot containing the Bode response of a dynamic system, `G`.

### **sys** — System for new response data

dynamic system model | model array

System from which to compute new response data for the response plot, specified as a dynamic system model or model array.

`sys` must match the plotted system that it replaces in both I/O dimensions and array dimensions. For example, suppose `h` refers to a plot that displays the step responses of a 5-element vector of 2-input, 2-output systems. In this case, `sys` must also be a 5-element vector of 2-input, 2-output systems. The number of states in the elements of `sys` need not match the number of states in the plotted systems.

### **N** — Index of system to replace

1 (default) | positive integer

Index of system to replace in the plot, specified as a positive integer. For example, suppose you create a plot using the following command.

```
h = impulseplot(G1,G2,G3,G4);
```

To replace the impulse data of `G3` with data from a new system, `sys`, use the following command.

```
updateSystem(h,sys,3);
```

**Introduced in R2013b**

## upsample

Upsample discrete-time models

### Syntax

```
sys1 = upsample(sys,L)
```

### Description

`sys1 = upsample(sys,L)` resamples the discrete-time dynamic system model `sys` at a sampling rate that is L-times faster than the sample time of `sys` ( $T_{s_0}$ ). L must be a positive integer. When `sys` is a TF model,  $H(z)$ , `upsample` returns `sys1` as  $H(z^L)$  with the sample time  $T_{s_0} / L$ .

The responses of models `sys` and `sys1` have the following similarities:

- The time responses of `sys` and `sys1` match at multiples of  $T_{s_0}$ .
- The frequency responses of `sys` and `sys1` match up to the Nyquist frequency  $\pi / T_{s_0}$ .

---

**Note:** `sys1` has L times as many states as `sys`.

---

### Examples

Create a transfer function with a sample time that is 14 times faster than that of the following transfer function:

```
sys = tf(0.75,[1 10 2],2.25)
```

Transfer function:

```
0.75  
-----  
z^2 + 10 z + 2
```

Sample time: 2.25

To create the upsampled transfer function `sys1`, type the following commands:

```
L=14;  
sys1 = upsample(sys,L)  
These commands return the result:
```

```
Transfer function:  
      0.75
```

```
-----  
z^28 + 10 z^14 + 2
```

```
Sample time: 0.16071
```

The sample time of `sys1` is 0.16071 seconds, which is 14 times faster than the 2.25 second sample time of `sys`.

## See Also

d2c | c2d | d2d

**Introduced in R2008b**

# viewSpec

View tuning requirements; validate design against requirements

## Syntax

```
viewSpec(Req)  
viewSpec(Req,T)  
viewSpec(Req,T,Info)
```

## Description

`viewSpec(Req)` displays a graphical view of a `TuningGoal` tuning requirement or vector of tuning requirements.

`viewSpec(Req,T)` plots the performance of a tuned control system against the tuning requirement. `viewSpec` applies the solver's loop scaling when evaluating MIMO open-loop requirements such as loop shapes or stability margins. This application ensures consistency with the tuning goal value computed by `sysTune` or `loopTune`.

`viewSpec(Req,T,Info)` uses the `Info` structure returned by `sysTune` to maintain consistency after modifying `T` with `usample`, `usubs`, or `setBlockValue`.

## Examples

### Visualize Tuning Requirement as Function of Frequency

Create a tuning requirement that constrains the response from a signal, 'd', to another signal, 'y', to roll off at 20 dB/decade at frequencies greater than 1. The requirement also imposes disturbance rejection (maximum gain of 1) in the frequency range [0,1].

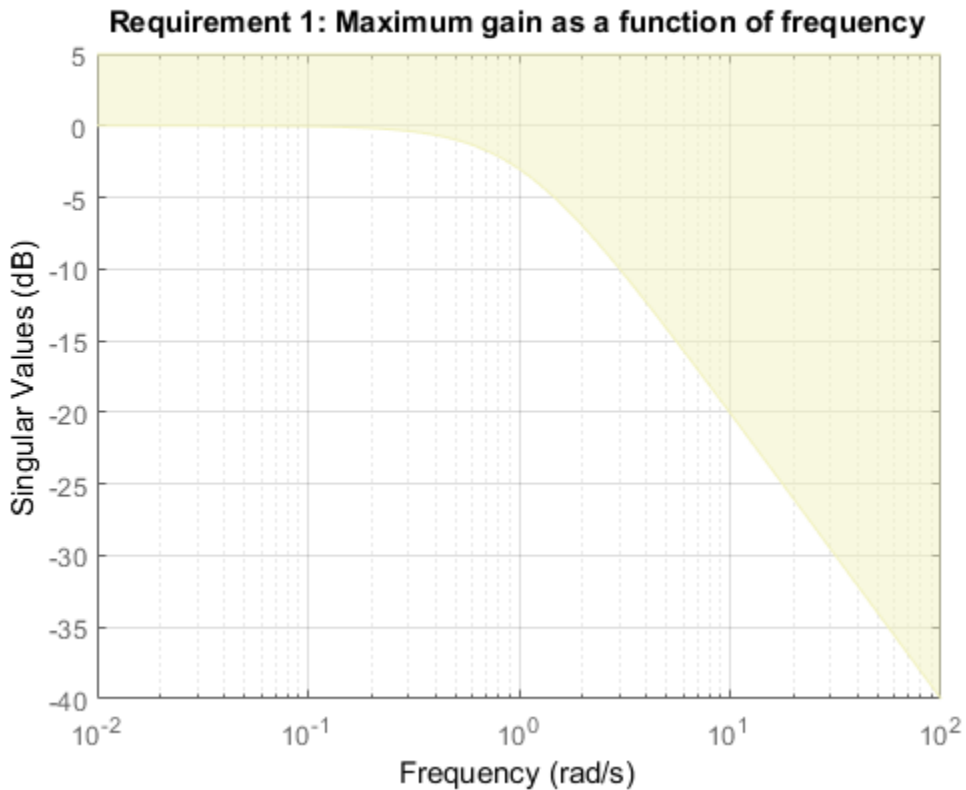
```
gmax = frd([1 1 0.01],[0 1 100]);  
Req = TuningGoal.MaxGain('du','u',gmax);
```

When you use a frequency response data (`frd`) model to sketch the bounds of a gain constraint or loop shape, the tuning requirement interpolates the constraint. This interpolation converts the constraint to a smooth function of frequency.



Examine the interpolated gain constraint using `viewSpec`.

```
viewSpec(Req)
```



The yellow region represents gain values that violate the tuning requirement.

### Validate Tuning Result Against Requirements

Validate a control system tuned with `systemtune` to determine whether small violations of tuning requirements are acceptable.

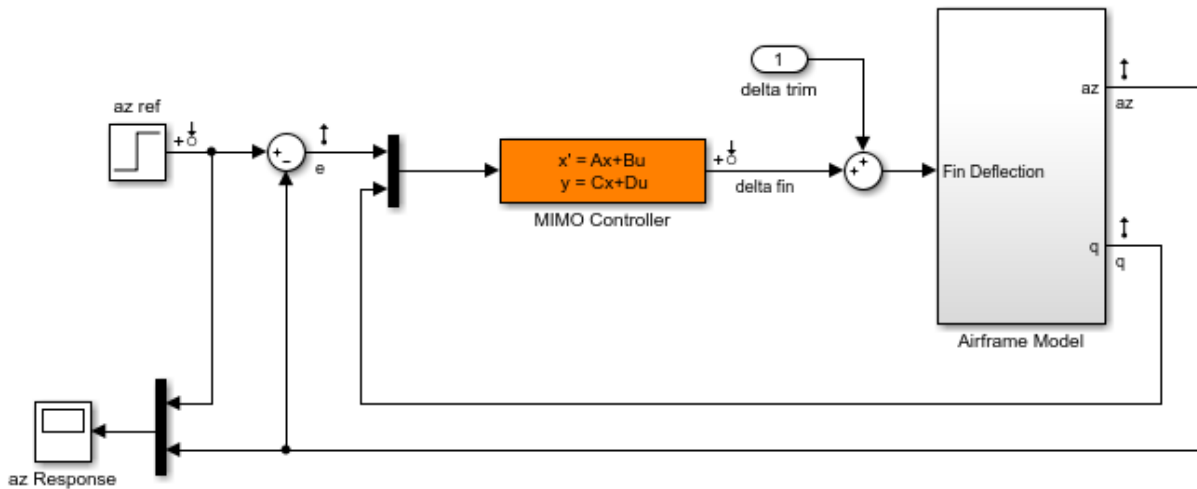
When you tune a control system using tuning commands such as `systemtune`, use `viewSpec` to compare the tuned result against the tuning requirements. Doing so can

help you determine whether the tuned system comes sufficiently close to meeting your soft requirements.

Open a Simulink® model that contains a control system you want to tune.

```
open_system('rct_airframe2')
```

### Two-loop autopilot for controlling the vertical acceleration of an airframe



Create requirements for tuning the control system. For this example, use tracking, roll-off, stability margin, and disturbance rejection requirements.

```
Req1 = TuningGoal.Tracking('az ref','az',1);
Req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0]));
Req3 = TuningGoal.Margins('delta fin',7,45);
MaxGain = frd([2 200 200],[0.02 2 200]);
Req4 = TuningGoal.Gain('delta fin','az',MaxGain);
```

Tune the model using these tuning requirements.

```
ST0 = s1Tuner('rct_airframe2','MIMO Controller');
addPoint(ST0,'delta fin');

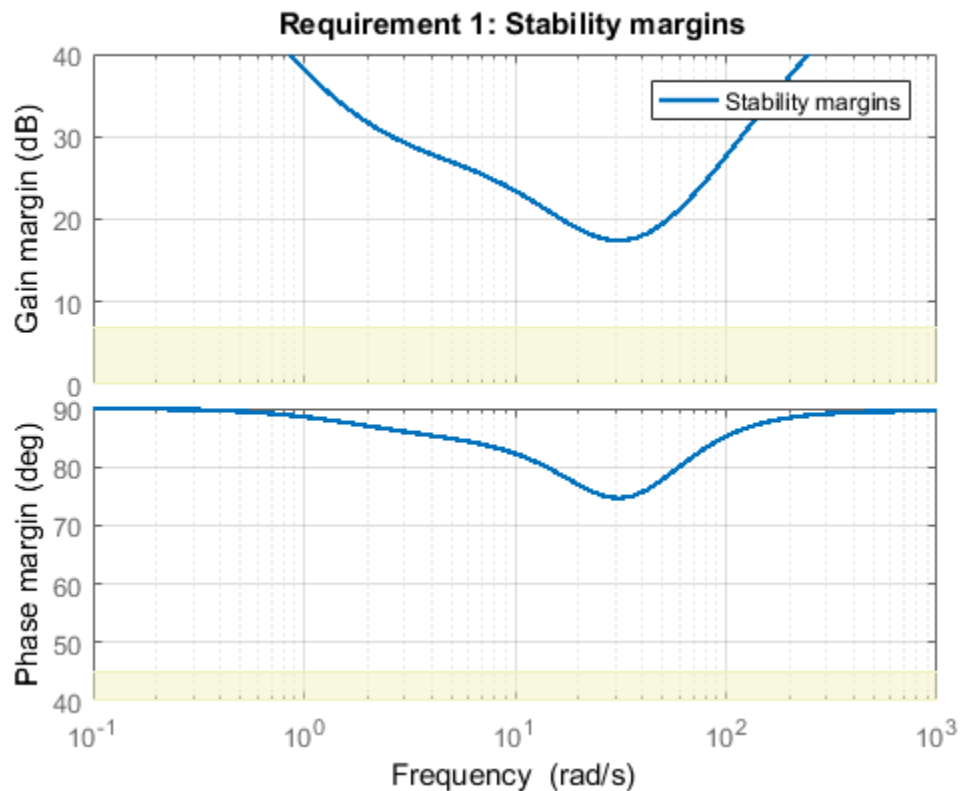
rng('default');
[ST1,fSoft] = systune(ST0,[Req1,Req2,Req3,Req4]);
```

Final: Soft = 1.15, Hard = -Inf, Iterations = 80

ST1 is a tuned version of the `sITuner` interface to the control system. ST1 contains the tuned values of the tunable parameters of the MIMO controller in the model.

Verify that the tuned system satisfies the margin requirement.

```
figure;
viewSpec(Req3,ST1)
```

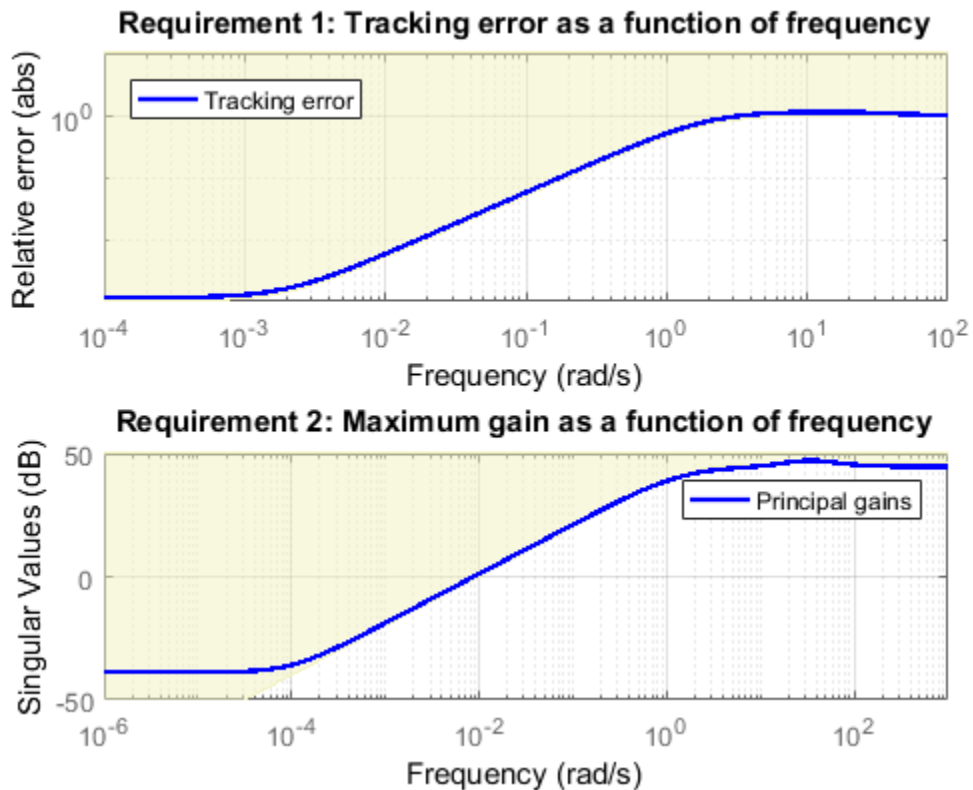


The yellow region denotes margins that do not satisfy the requirement. The red plot represents the actual stability margin of the tuned system, ST1. The blue plot represents the margin used in the optimization calculation, which is an upper bound on the actual

margin. For ST1, the plot indicates that the margin requirement is satisfied at all frequencies.

Validate the tracking and disturbance rejection requirements in the frequency domain.

```
figure
viewSpec([Req1,Req4],ST1)
```



When you provide a vector of requirements, `viewSpec` puts all the requirements into a single figure window.

The first plot shows that the tuned system very nearly meets the tracking requirement. The slight violation suggests that setpoint tracking will perform close to expectations.

The second plot shows that the disturbance rejection levels off in violation of the requirement at very low frequencies. A small bump near 35 rad/s suggests possible damped oscillations at this frequency.

Use `step` and `getIOTransfer` to examine setpoint tracking and disturbance rejection in the time domain.

## Input Arguments

### **Req** — Tuning requirement to view or validate

TuningGoal requirement object | vector of TuningGoal objects

Tuning requirement to view or validate, specified as a TuningGoal requirement object or vector of TuningGoal objects. TuningGoal requirement objects include:

- TuningGoal.Tracking
- TuningGoal.Gain
- TuningGoal.WeightedGain
- TuningGoal.Variance
- TuningGoal.WeightedVariance
- TuningGoal.LoopShape
- TuningGoal.Margins
- TuningGoal.Poles
- TuningGoal.ControllerPoles

### **T** — Tuned control system

generalized state-space model | sITuner interface object

Tuned control system, specified as a generalized state-space (`genss`) model or an `sITuner` interface to a Simulink model.

The control system, `T`, is typically the result of using the tuning requirement to tune control system parameters with `system`.

Example: `[T, fSoft, gHard, Info] = systune(T0, SoftReq, HardReq)`, where `T0` is a tunable `genss` model

Example: `[T, fSoft, gHard, Info] = systune(ST0, SoftReq, HardReq)`, where `ST0` is a `sITuner` interface object

### **Info — System information**

data structure returned by `sysstune`

System information, specified as the data structure returned by `sysstune` when you use that command to tune a control system. Use **Info** to maintain consistency after modifying `T` with `usample`, `usubs`, or `setBlockValue`.

## **More About**

- “Generalized Models”

## **See Also**

`TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Sensitivity` |  
`TuningGoal.Overshoot` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain`  
| `TuningGoal.Margins` | `TuningGoal.WeightedGain` | `TuningGoal.Variance` |  
`TuningGoal.WeightedVariance` | `TuningGoal.LoopShape` | `TuningGoal.Poles` |  
`TuningGoal.ControllerPoles` | `evalSpec` | `genss` | `sITuner` | `sysstune` | `sysstune`  
(for `sITuner`)

**Introduced in R2012b**

# viewSurf

Visualize gain surface as a function of scheduling variables

## Syntax

```
viewSurf(GS)
view(GS,xvar,xdata)
view(GS,xvar,xdata,yvar,ydata)
```

## Description

`viewSurf(GS)` plots the values of a 1-D or 2-D gain surface as a function of the scheduling variables. `GS` is a tunable gain surface that you create with `tunableSurface`. The plot uses the independent variable values specified in `GS.SamplingGrid`. For 2-D gain surfaces, the design points in `GS.SamplingGrid` must lie on a rectangular grid.

`view(GS,xvar,xdata)` plots the gain surface `GS` at the scheduling-variable values listed in `xdata`. The variable name `xvar` must match a scheduling variable name in `GS.SamplingGrid`. However, the values in `xdata` need not match design points in `GS.SamplingGrid`.

For a 2-D gain surface, the plot shows a parametric family of curves with one curve per value of the other scheduling variable. In the 2-D case, the design points in `GS.SamplingGrid` must lie on a rectangular grid.

`view(GS,xvar,xdata,yvar,ydata)` creates a surface plot of a 2-D gain surface evaluated over a grid of scheduling variable values given by `ndgrid(xdata,ydata)`. In this case, the design points of `GS` do not need to lie on a rectangular grid, and `xdata` and `ydata` do not need to match the design points.

## Examples

### View Gain Surface

Display a tunable gain surface that depends on two independent variables.

Model a scalar gain  $K$  with a bilinear dependence on two scheduling variables,  $\alpha$  and  $V$ , as follows:

$$K(\alpha, V) = K_0 + K_1x + K_2y + K_3xy.$$

Here,  $x$  and  $y$  are the normalized scheduling variables. Suppose that  $\alpha$  is an angle of incidence that ranges from 0 degrees to 15 degrees, and  $V$  is a speed that ranges from 300 m/s to 600 m/s. Then,  $x$  and  $y$  are given by:

$$x = \frac{\alpha - 7.5}{7.5}, \quad y = \frac{V - 450}{150}.$$

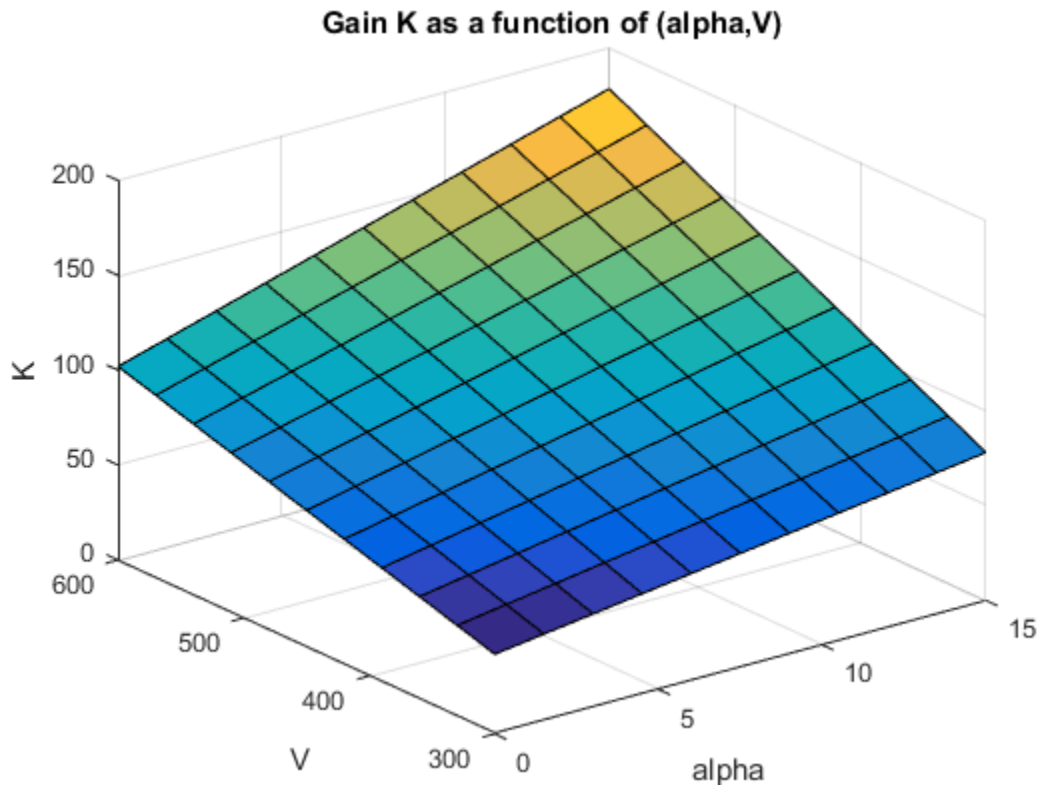
The coefficients  $K_0, \dots, K_3$  are the tunable parameters of this variable gain. Use `tunableSurface` to model this variable gain.

```
[alpha,V] = ndgrid(0:1.5:15,300:30:600);  
domain = struct('alpha',alpha,'V',V);  
shapefcn = @(x,y) [x,y,x*y];  
K = tunableSurface('K',1,domain,shapefcn);
```

Typically, you would tune the coefficients as part of a control system. You would then use `setBlockValue` or `setData` to write the tuned coefficients back to  $K$ , and view the tuned gain surface. For this example, instead of tuning, manually set the coefficients to non-zero values and view the resulting gain.

```
Ktuned = setData(K,[100,28,40,10]);  
viewSurf(Ktuned)
```





viewSurf displays the gain surface as a function of the scheduling variables, for the ranges of values specified by domain and stored in `Ktuned.SamplingGrid`.

### Plot Gain Surface for Specified Breakpoints

View a 1-D gain surface evaluated at different design points from the points specified in the gain surface.

When you create a gain surface using `tunableSurface`, you specify design points at which the gain coefficients are tuned. These points are typically the scheduling-variable values at which you have sampled or linearized the plant. However, you might want to implement the gain surface as a lookup table with breakpoints that are different from the specified design points. In this example, you create a gain surface with a set

of design points and then view the surface using a different set of scheduling variable values.

Create a scalar gain that varies as a quadratic function of one scheduling variable,  $t$ . Suppose that you have linearized your plant every five seconds from  $t = 0$  to  $t = 40$ .

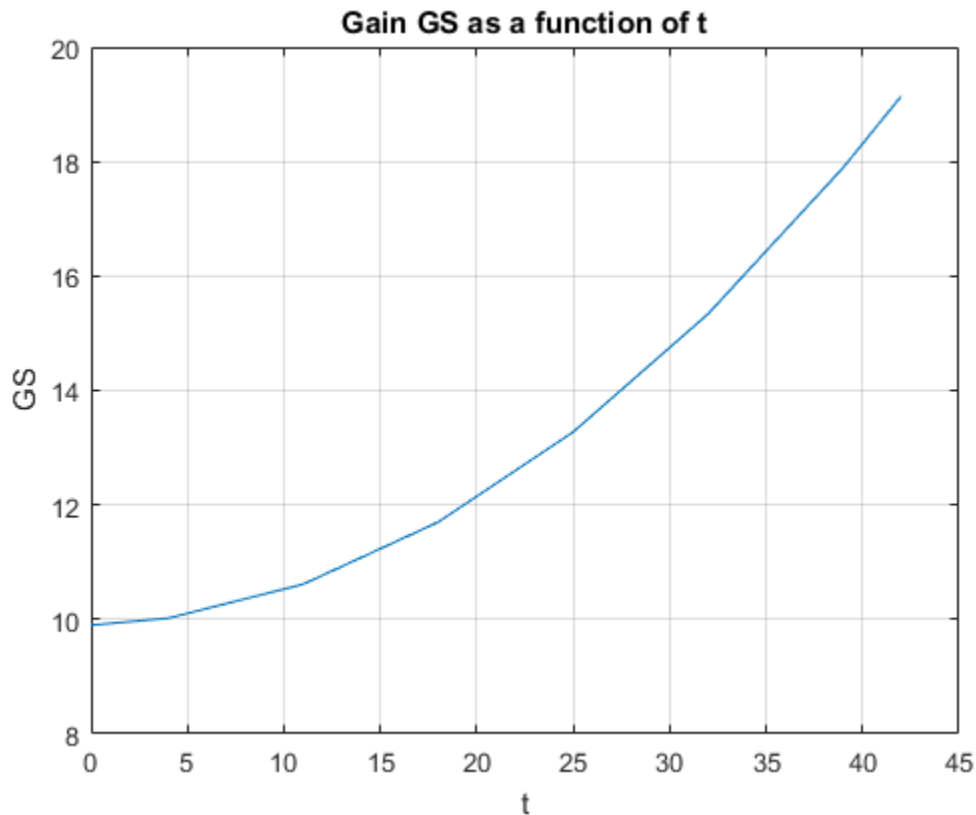
```
t = 0:5:40;
domain = struct('t',t);
shapefcn = @(x) [x,x^2];
GS = tunableSurface('GS',1,domain,shapefcn);
```

Typically, you would tune the coefficients as part of a control system. For this example, instead of tuning, manually set the coefficients to non-zero values.

```
GS = setData(GS,[12.1,4.2,2]);
```

Plot the gain surface evaluated at a different set of time values.

```
tvals = [0,4,11,18,25,32,39,42];
viewSurf(GS,'t',tvals)
```



The plot shows that the gain curve bends at the points specified in `tvals`, rather than the design points specified in `domain`. Also, `tvals` includes values outside of the scheduling-variable range of `domain`. If you attempt to extrapolate too far out of the range of values used for tuning, the software issues a warning.

### View 1-Dimensional Projections of 2-D Gain Surface

Plot gain surface values as a function of one independent variable, for a gain surface that depends on two independent variables.

Create a gain surface that is a bilinear function of two independent variables,  $\alpha$  and  $V$ .

```
[alpha,V] = ndgrid(0:1.5:15,300:30:600);
```

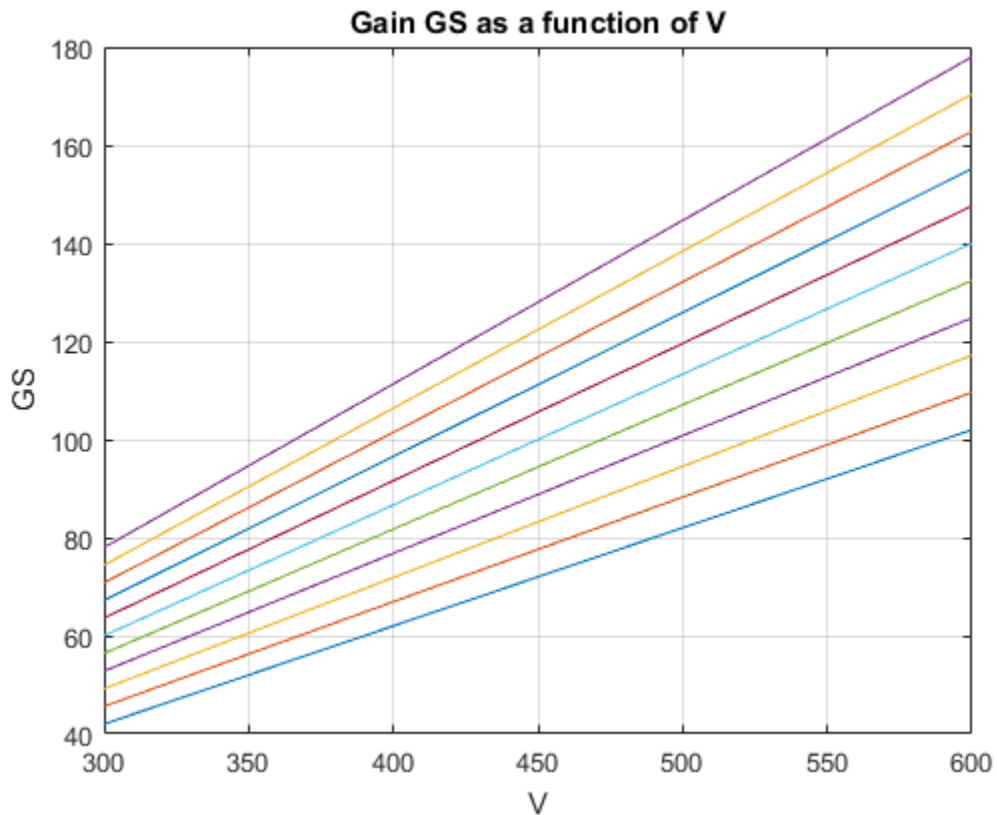
```
domain = struct('alpha',alpha,'V',V);  
shapefcn = @(x,y) [x,y,x*y];  
GS = tunableSurface('GS',1,domain,shapefcn);
```

Typically, you would tune the coefficients as part of a control system. For this example, instead of tuning, manually set the coefficients to non-zero values.

```
GS = setData(GS,[100,28,40,10]);
```

Plot the gain at selected values of  $V$ .

```
Vplot = [300:50:600];  
viewSurf(GS,'V',Vplot);
```



`viewSurf` evaluates the gain surface at the specified values of  $V$ , and plots the dependence on  $V$  for all values of  $\alpha$  in `domain`. Clicking any of the lines in the plot displays the corresponding  $\alpha$  value. This plot is useful to visualize the full range of gain variation due to one independent variable.

### Plot 2-D Gain Surface for Specified Breakpoints

View a 2-D gain surface evaluated at different scheduling-variable values from the design points specified in the gain surface.

When you create a gain surface using `tunableSurface`, you specify design points at which the gain coefficients are tuned. These points are typically the scheduling-variable values at which you have sampled or linearized the plant. However, you might want to implement the gain surface as a lookup table with breakpoints that are different from the specified design points. In this example, you create a gain surface with a set of design points and then view the surface using a different set of scheduling-variable values.

Create a gain surface that is a bilinear function of two independent variables,  $\alpha$  and  $V$ .

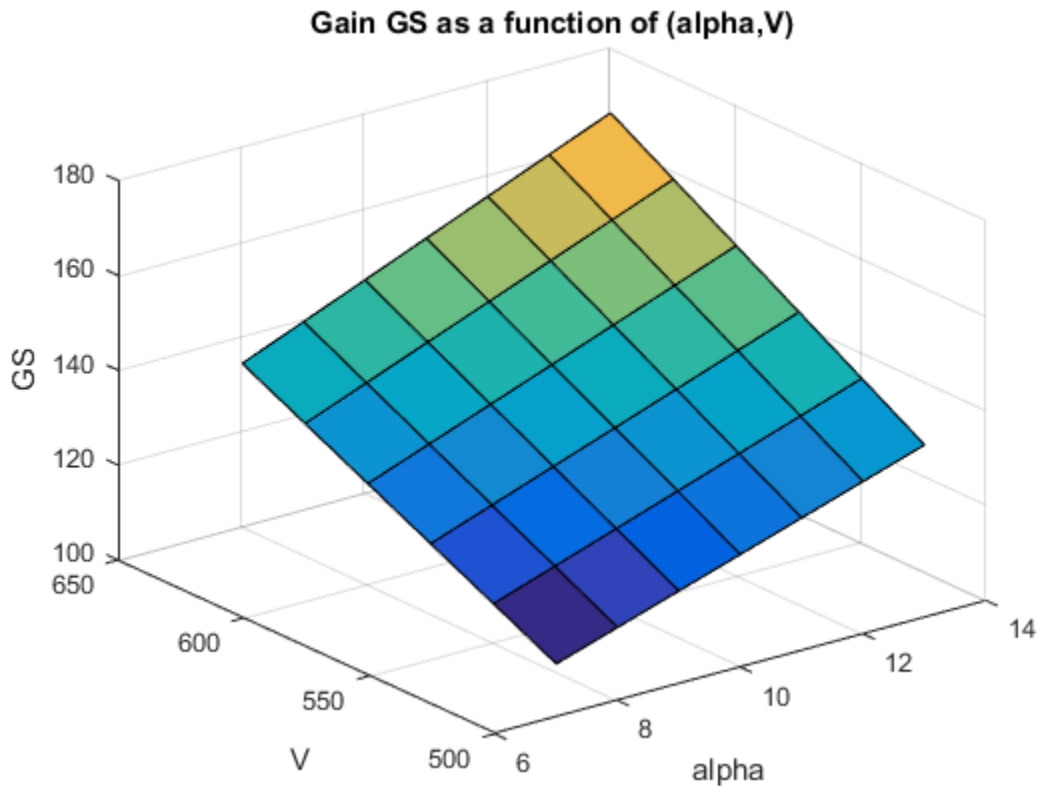
```
[alpha,V] = ndgrid(0:1.5:15,300:30:600);
domain = struct('alpha',alpha,'V',V);
shapefcn = @(x,y) [x,y,x*y];
GS = tunableSurface('GS',1,domain,shapefcn);
```

Typically, you would tune the coefficients as part of a control system. For this example, instead of tuning, manually set the coefficients to non-zero values.

```
GS = setData(GS,[100,28,40,10]);
```

Plot the gain at selected values of  $\alpha$  and  $V$ .

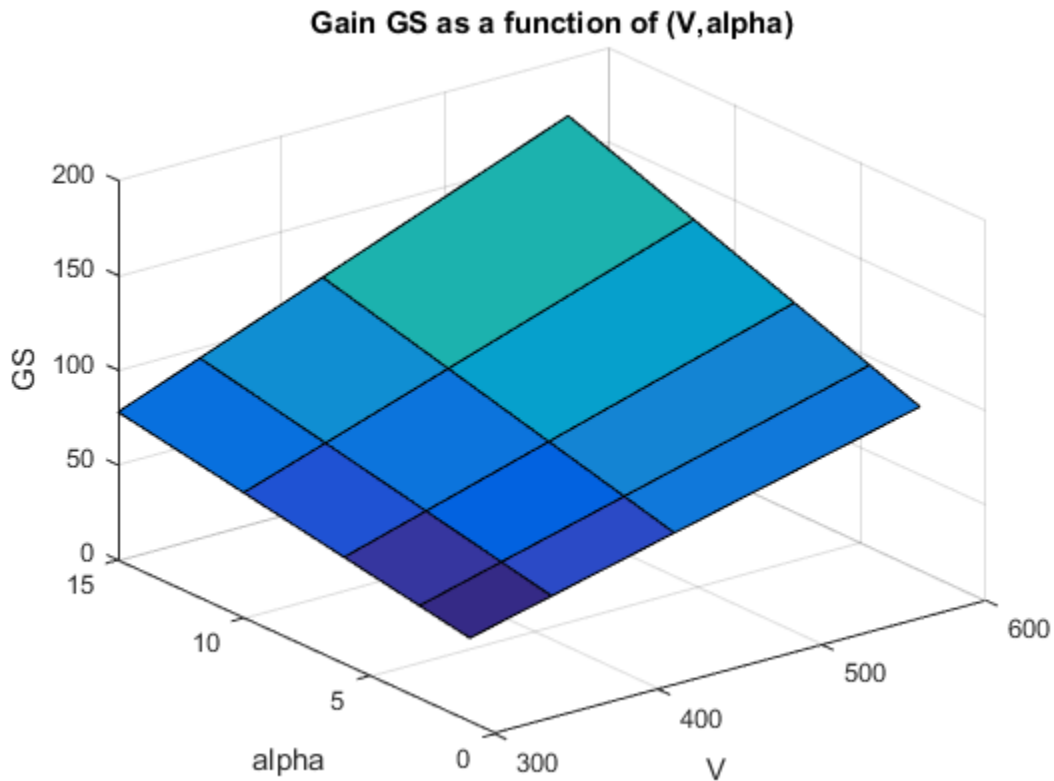
```
alpha_vec = [7:1:13];
V_vec = [500:25:625];
viewSurf(GS,'alpha',alpha_vec,'V',V_vec);
```



The breakpoints at which you evaluate the gain surface need not fall within the range specified by `domain`. However, if you attempt to evaluate the gain too far outside the range used for tuning, the software issues a warning.

The breakpoints also need not be regularly spaced. In addition, you can specify the scheduling variables in any order to get a different perspective on the shape of the surface. The variable that you specify first is used as the X-axis in the plot.

```
alpha_vec2 = [1,3,6,10,15];  
V_vec2 = [300,350,425,575];  
viewSurf(GS, 'V',V_vec2, 'alpha',alpha_vec2);
```



## Input Arguments

### **GS** — Gain surface

tunableSurface object

Gain surface to plot, specified as a `tunableSurface` object. GS can depend on one or two scheduling variables, and must be scalar-valued.

### **xvar** — X-axis variable

character vector

X-axis variable in the plot, specified as a character vector. The variable name `xvar` must match a scheduling variable name in `GS.SamplingGrid`.

### **xdata — X-axis-variable values**

numeric vector

X-axis-variable values at which to evaluate and plot the gain surface, specified as a numeric vector.

### **yvar — Y-axis variable**

character vector

Y-axis variable in the plot, specified as a character vector. The variable name `yvar` must match a scheduling variable name in `GS.SamplingGrid`.

### **ydata — Y-axis-variable values**

numeric vector

Y-axis-variable values at which to evaluate and plot the gain surface, specified as a numeric vector.

## **See Also**

`evalSurf` | `tunableSurface`

**Introduced in R2015b**



## xperm

Reorder states in state-space models

## Syntax

```
sys = xperm(sys,P)
```

## Description

`sys = xperm(sys,P)` reorders the states of the state-space model `sys` according to the permutation `P`. The vector `P` is a permutation of `1:NX`, where `NX` is the number of states in `sys`. For information about creating state-space models, see `ss` and `dss`.

## Examples

Order the states in the `ssF8` model in alphabetical order.

- 1 Load the `ssF8` model by typing the following commands:

```
load ltiexamples
ssF8
```

These commands return:

```
a =
      PitchRate  Velocity  AOA  PitchAngle
PitchRate      -0.7    -0.0458  -12.2      0
Velocity        0     -0.014   -0.2904  -0.562
AOA             1     -0.0057   -1.4      0
PitchAngle      1         0       0         0
```

```
b =
      Elevator  Flaperon
PitchRate     -19.1    -3.1
Velocity      -0.0119 -0.0096
AOA           -0.14   -0.72
PitchAngle    0       0
```

```
c =
      PitchRate  Velocity  AOA  PitchAngle
```

```
FlightPath      0      0      -1      1
Acceleration    0      0      0.733    0
```

```
d =
      Elevator  Flaperon
FlightPath      0      0
Acceleration  0.0768  0.1134
```

Continuous-time model.

- 2 Order the states in alphabetical order by typing the following commands:

```
[y,P]=sort(ssF8.StateName);
sys=xperm(ssF8,P)
```

These commands return:

```
a =
      AOA  PitchAngle  PitchRate  Velocity
AOA      -1.4         0           1      -0.0057
PitchAngle  0         0           1           0
PitchRate  -12.2      0          -0.7     -0.0458
Velocity   -0.2904   -0.562        0      -0.014
```

```
b =
      Elevator  Flaperon
AOA      -0.14   -0.72
PitchAngle  0     0
PitchRate  -19.1  -3.1
Velocity   -0.0119 -0.0096
```

```
c =
      AOA  PitchAngle  PitchRate  Velocity
FlightPath  -1         1           0           0
Acceleration 0.733     0           0           0
```

```
d =
      Elevator  Flaperon
FlightPath      0      0
Acceleration  0.0768  0.1134
```

Continuous-time model.

The states in `ssF8` now appear in alphabetical order.

## See Also

`ss` | `dss`

Introduced in R2008b

## zero

Zeros and gain of SISO dynamic system

### Syntax

```
z = zero(sys)
[z,gain] = zero(sys)
[z,gain] = zero(sysarr,J1,...,JN)
```

### Description

`z = zero(sys)` returns the zeros of the single-input, single-output (SISO) dynamic system model, `sys`.

`[z,gain] = zero(sys)` also returns the overall gain of `sys`.

`[z,gain] = zero(sysarr,J1,...,JN)` returns the zeros and gain of the model with subscripts `J1,...,JN` in the model array `sysarr`.

### Input Arguments

#### **sys**

SISO dynamic system model.

If `sys` has internal delays, `zero` sets all internal delays to zero, creating a zero-order Padé approximation. This approximation ensures that the system has a finite number of zeros. `zero` returns an error if setting internal delays to zero creates singular algebraic loops.

#### **sysarr**

Array of dynamic system models.

#### **J1,...,JN**

Indices identifying the model `sysarr(J1,...,JN)` in the array `sysarr`.

## Output Arguments

**z**

Column vector containing the locations of zeros in `sys`. The zero locations are expressed in the reciprocal of the time units of `sys`. For example, the zeros are in units of 1/minutes if the `TimeUnit` property of `sys` is `minutes`.

**gain**

Gain of `sys` (in the zero-pole-gain sense).

## Examples

### Calculate Zero Locations and Gain of Transfer Function

Create the following transfer function:

$$H(s) = \frac{4.2s^2 + 0.25s - 0.004}{s^2 + 9.6s + 17}$$

```
H = tf([4.2,0.25,-0.004],[1,9.6,17]);
```

Calculate the zero locations and overall gain of the transfer function.

```
[z, gain] = zero(H)
```

```
z =
```

```
-0.0726  
0.0131
```

```
gain =
```

```
4.2000
```

The zero locations are expressed in radians per second, because the time unit of the transfer function (`H.TimeUnit`) is seconds.

Change the model time units.

```
H = chgTimeUnit(H, 'minutes');
```

zero returns locations relative to the new unit.

```
[z,gain] = zero(H)
```

```
z =
```

```
-4.3581  
 0.7867
```

```
gain =
```

```
 4.2000
```

## Alternatives

To calculate the transmission zeros of a multi-input, multi-output system, use `tzero`.

### See Also

`pzmap` | `pole` | `tzero`

**Introduced before R2006a**

## zgrid

Generate z-plane grid of constant damping factors and natural frequencies

### Syntax

```
zgrid  
zgrid(z,wn)  
zgrid([],[])
```

### 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  $\pi$  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.

### Examples

#### Plot z-plane grid lines on the root locus

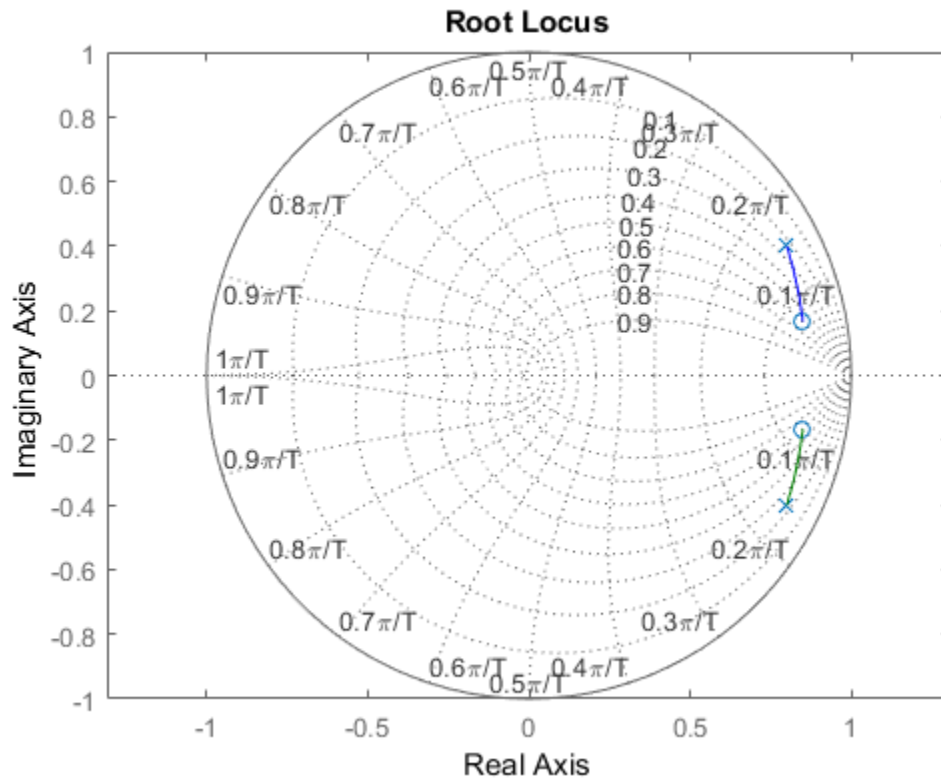
To see the z-plane grid on the root locus plot, type

```
H = tf([2 -3.4 1.5],[1 -1.6 0.8],-1)
rlocus(H)
zgrid
axis equal
```

H =

$$\frac{2z^2 - 3.4z + 1.5}{z^2 - 1.6z + 0.8}$$

Sample time: unspecified  
Discrete-time transfer function.



**See Also**

`sgrid` | `pzmap` | `rlocus`

Introduced before R2006a



# zpk

Create zero-pole-gain model; convert to zero-pole-gain model

## Syntax

```

sys = zpk(Z,P,K)
sys = zpk(Z,p,k,Ts)
sys = zpk(M)
sys = zpk(Z,p,k,ltisys)
s = zpk('s')
z = zpk('z',Ts)
zsys = zpk(sys)
zsys = zpk(sys, 'measured')
zsys = zpk(sys, 'noise')
zsys = zpk(sys, 'augmented')

```

## Description

Used `zpk` to create zero-pole-gain models (`zpk` model objects), or to convert dynamic systems 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` model object storing the model data.

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:

$$h(s) = k \frac{(s - z(1))(s - z(2)) \dots (s - z(m))}{(s - p(1))(s - p(2)) \dots (s - p(n))}$$

Set `Z` or `p` to `[]` for systems without zeros or poles. These two vectors need not have equal length and the model need not be proper (that is, have an excess of poles).

To create a MIMO zero-pole-gain model, specify the zeros, poles, and gain of each SISO entry of this model. In this case:

- $Z$  and  $P$  are cell arrays of vectors with as many rows as outputs and as many columns as inputs, and  $K$  is a matrix with as many rows as outputs and as many columns as inputs.
- The vectors  $Z\{i, j\}$  and  $P\{i, j\}$  specify the zeros and poles of the transfer function from input  $j$  to output  $i$ .
- $K(i, j)$  specifies the (scalar) gain of the transfer function from input  $j$  to output  $i$ .

See below for a MIMO example.

`sys = zpk(Z,p,k,Ts)` creates a discrete-time zero-pole-gain model with sample time  $Ts$  (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 properties inherited from the LTI model `ltisys` (including the sample time).

To create an array of `zpk` model objects, 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 property of the model, for example, the input names or the input delay time. For more information about the properties of `zpk` model objects, see “Properties” on page 2-1178. 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  $s = 0$ .

## Conversion of Identified Models

An identified model is represented by an input-output equation of the form  $y(t) = Gu(t) + He(t)$ , where  $u(t)$  is the set of measured input channels and  $e(t)$  represents the noise channels. If  $\Lambda = LL'$  represents the covariance of noise  $e(t)$ , this equation can also be written as  $y(t) = Gu(t) + HLv(t)$ , where  $\text{cov}(v(t)) = I$ .

`zsys = zpk(sys)`, or `zsys = zpk(sys, 'measured')` converts the measured component of an identified linear model into the ZPK form. `sys` is a model of type `idss`, `idproc`, `idtf`, `idpoly`, or `idgrey`. `zsys` represents the relationship between  $u$  and  $y$ .

`zsys = zpk(sys, 'noise')` converts the noise component of an identified linear model into the ZPK form. It represents the relationship between the noise input,  $v(t)$  and output,  $y_{\text{noise}} = HL v(t)$ . The noise input channels belong to the `InputGroup` `'Noise'`. The names of the noise input channels are `v@yname`, where `yname` is the name of the corresponding output channel. `zsys` has as many inputs as outputs.

`zsys = zpk(sys, 'augmented')` converts both the measured and noise dynamics into a ZPK model. `zsys` has  $n_y + n_u$  inputs such that the first  $n_u$  inputs represent the channels  $u(t)$  while the remaining by channels represent the noise channels

$v(t)$ . `zsys.InputGroup` contains 2 input groups, 'measured' and 'noise'.  
`zsys.InputGroup.Measured` is set to `1:nu` while `zsys.InputGroup.Noise` is set to `nu+1:nu+ny`. `zsys` represents the equation  $y(t) = [G \ HL] [u; v]$ .

---

**Tip** An identified nonlinear model cannot be converted into a ZPK system. Use linear approximation functions such as `linearize` and `linapp`.

---

## 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),  $z^{-1}$ ,  $q^{-1}$  (equivalent to  $z^{-1}$ ), or  $q$  (equivalent to  $z$ ) for discrete-time models. Reassign the 'Variable' property to override the defaults. Changing the variable affects only the display of zero-pole-gain models.

## Properties

zpk objects have the following properties:

### Z

System zeros.

The Z property stores the transfer function zeros (the numerator roots). For SISO models, Z is a vector containing the zeros. For MIMO models with  $N_y$  outputs and  $N_u$  inputs, Z is a  $N_y$ -by- $N_u$  cell array of vectors of the zeros for each input/output pair.

### P

System poles.

The P property stores the transfer function poles (the denominator roots). For SISO models, P is a vector containing the poles. For MIMO models with  $N_y$  outputs and  $N_u$  inputs, P is a  $N_y$ -by- $N_u$  cell array of vectors of the poles for each input/output pair.

### K

System gains.

The `K` property stores the transfer function gains. For SISO models, `K` is a scalar value. For MIMO models with `Ny` outputs and `Nu` inputs, `K` is a `Ny`-by-`Nu` matrix storing the gains for each input/output pair.

### DisplayFormat

Specifies how the numerator and denominator polynomials are factorized for display purposes.

The numerator and denominator polynomials are each displayed as a product of first- and second-order factors. `DisplayFormat` controls the display of those factors. `DisplayFormat` can take the following values:

- `'roots'` (default) — Display factors in terms of the location of the polynomial roots.
- `'frequency'` — Display factors in terms of root natural frequencies  $\omega_0$  and damping ratios  $\zeta$ .

The `'frequency'` display format is not available for discrete-time models with `Variable` value `'z^-1'` or `'q^-1'`.

- `'time constant'` — Display factors in terms of root time constants  $\tau$  and damping ratios  $\zeta$ .

The `'time constant'` display format is not available for discrete-time models with `Variable` value `'z^-1'` or `'q^-1'`.

For continuous-time models, the following table shows how the polynomial factors are written in each display format.

DisplayName Value	First-Order Factor (Real Root $R$ )	Second-Order Factor (Complex Root pair $R = a \pm jb$ )
<code>'roots'</code>	$(s - R)$	$(s^2 - as + \beta)$ , where $a = 2a$ , $\beta = a^2 + b^2$
<code>'frequency'</code>	$(1 - s/\omega_0)$ , where $\omega_0 = R$	$1 - 2\zeta(s/\omega_0) + (s/\omega_0)^2$ , where $\omega_0^2 = a^2 + b^2$ , $\zeta = a/\omega_0$
<code>'time constant'</code>	$(1 - \tau s)$ , where $\tau = 1/R$	$1 - 2\zeta(\tau s) + (\tau s)^2$ , where $\tau = 1/\omega_0$ , $\zeta = a\tau$

For discrete-time models, the polynomial factors are written as in continuous time, with the following variable substitutions:

$$s \rightarrow w = \frac{z-1}{T_s}; \quad R \rightarrow \frac{R-1}{T_s},$$

where  $T_s$  is the sample time. In discrete time,  $\tau$  and  $\omega_0$  closely match the time constant and natural frequency of the equivalent continuous-time root, provided  $|z-1| \ll T_s$  ( $\omega_0 \ll \pi/T_s = \text{Nyquist frequency}$ ).

**Default:** 'roots'

### Variable

Transfer function display variable, specified as one of the following:

- 's' — Default for continuous-time models
- 'z' — Default for discrete-time models
- 'p' — Equivalent to 's'
- 'q' — Equivalent to 'z'
- 'z^-1' — Inverse of 'z'
- 'q^-1' — Equivalent to 'z^-1'

The value of **Variable** only affects the display of **zpk** models.

**Default:** 's'

### IODelay

Transport delays. **IODelay** is a numeric array specifying a separate transport delay for each input/output pair.

For continuous-time systems, specify transport delays in the time unit stored in the **TimeUnit** property. For discrete-time systems, specify transport delays in integer multiples of the sample time, **Ts**.

For a MIMO system with **Ny** outputs and **Nu** inputs, set **IODelay** to a **Ny-by-Nu** array. Each entry of this array is a numerical value that represents the transport delay for the corresponding input/output pair. You can also set **IODelay** to a scalar value to apply the same delay to all input/output pairs.

**Default:** 0 for all input/output pairs

### **InputDelay**

Input delay for each input channel, specified as a scalar value or numeric vector. For continuous-time systems, specify input delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify input delays in integer multiples of the sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel.

You can also set `InputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0

### **OutputDelay**

Output delays. `OutputDelay` is a numeric vector specifying a time delay for each output channel. For continuous-time systems, specify output delays in the time unit stored in the `TimeUnit` property. For discrete-time systems, specify output delays in integer multiples of the sample time `Ts`. For example, `OutputDelay = 3` means a delay of three sampling periods.

For a system with `Ny` outputs, set `OutputDelay` to an `Ny`-by-1 vector, where each entry is a numerical value representing the output delay for the corresponding output channel. You can also set `OutputDelay` to a scalar value to apply the same delay to all channels.

**Default:** 0 for all output channels

### **Ts**

Sample time. For continuous-time models, `Ts = 0`. For discrete-time models, `Ts` is a positive scalar representing the sampling period. This value is expressed in the unit specified by the `TimeUnit` property of the model. To denote a discrete-time model with unspecified sample time, set `Ts = -1`.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous- and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

Units for the time variable, the sample time `Ts`, and any time delays in the model, specified as one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names, specified as one of the following:

- Character vector — For single-input models, for example, 'controls'.
- Cell array of character vectors — For multi-input models.

Alternatively, use automatic vector expansion to assign input names for multi-input models. For example, if `sys` is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to `{'controls(1)'; 'controls(2)'}`.

You can use the shorthand notation `u` to refer to the `InputName` property. For example, `sys.u` is equivalent to `sys.InputName`.



Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all input channels

### **InputUnit**

Input channel units, specified as one of the following:

- Character vector — For single-input models, for example, 'seconds'.
- Cell array of character vectors — For multi-input models.

Use `InputUnit` to keep track of input signal units. `InputUnit` has no effect on system behavior.

**Default:** '' for all input channels

### **InputGroup**

Input channel groups. The `InputGroup` property lets you assign the input channels of MIMO systems into groups and refer to each group by name. Specify input groups as a structure. In this structure, field names are the group names, and field values are the input channels belonging to each group. For example:

```
sys.InputGroup.controls = [1 2];  
sys.InputGroup.noise = [3 5];
```

creates input groups named `controls` and `noise` that include input channels 1, 2 and 3, 5, respectively. You can then extract the subsystem from the `controls` inputs to all outputs using:

```
sys(:, 'controls')
```

**Default:** Struct with no fields

### **OutputName**

Output channel names, specified as one of the following:

- Character vector — For single-output models. For example, 'measurements'.

- Cell array of character vectors — For multi-output models.

Alternatively, use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to `{'measurements(1)'; 'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

**Default:** '' for all output channels

### **OutputUnit**

Output channel units, specified as one of the following:

- Character vector — For single-output models. For example, 'seconds'.
- Cell array of character vectors — For multi-output models.

Use `OutputUnit` to keep track of output signal units. `OutputUnit` has no effect on system behavior.

**Default:** '' for all output channels

### **OutputGroup**

Output channel groups. The `OutputGroup` property lets you assign the output channels of MIMO systems into groups and refer to each group by name. Specify output groups as a structure. In this structure, field names are the group names, and field values are the output channels belonging to each group. For example:

```
sys.OutputGroup.temperature = [1];  
sys.InputGroup.measurement = [3 5];
```

creates output groups named `temperature` and `measurement` that include output channels 1, and 3, 5, respectively. You can then extract the subsystem from all inputs to the `measurement` outputs using:

```
sys('measurement',:)
```

**Default:** Struct with no fields

### Name

System name, specified as a character vector. For example, `'system_1'`.

**Default:** `''`

### Notes

Any text that you want to associate with the system, specified as a character vector or cell array of character vectors. For example, `'System is MIMO'`.

**Default:** `{}`

### UserData

Any type of data you want to associate with system, specified as any MATLAB data type.

**Default:** `[]`

### SamplingGrid

Sampling grid for model arrays, specified as a data structure.

For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, *M*, by independently sampling two variables, *zeta* and *w*. The following code attaches the (*zeta*, *w*) values to *M*.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)  
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display *M*, each entry in the array includes the corresponding *zeta* and *w* values.

*M*

```
M(:,:,1,1) [zeta=0.3, w=5] =
```

```
      25  
-----  
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```
      25  
-----  
s^2 + 3.5 s + 25
```

...

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `sLinearizer` populate `SamplingGrid` in this way.

**Default:** []

## Examples

### Example 1

Create the continuous-time SISO transfer function:

$$h(s) = \frac{-2s}{(s-1+j)(s-1-j)(s-2)}$$

Create  $h(s)$  as a `zpk` object using:

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

## Example 2

Specify the following one-input, two-output zero-pole-gain model:

$$H(z) = \begin{bmatrix} \frac{1}{z-0.3} \\ \frac{2(z+0.5)}{(z-0.1+j)(z-0.1-j)} \end{bmatrix}.$$

To do this, enter:

```
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 3

Convert the transfer function

```
h = tf([-10 20 0],[1 7 20 28 19 5]);
```

to zero-pole-gain form, using:

```
zpk(h)
```

This command returns the result:

```
Zero/pole/gain:
   -10 s (s-2)
-----
(s+1)^3 (s^2 + 4s + 5)
```

### Example 4

Create a discrete-time ZPK model from a rational expression in the variable  $z$ .

```
z = zpk('z',0.1);
H = (z+.1)*(z+.2)/(z^2+.6*z+.09)
```

This command returns the following result:

```
Zero/pole/gain:
(z+0.1) (z+0.2)
-----
      (z+0.3)^2
```

Sample time: 0.1

### Example 5

Create a MIMO zpk model using cell arrays of zeros and poles.

Create the two-input, two-output zero-pole-gain model

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

by entering:

```
Z = {[],-5;[1-i 1+i] []};
P = {0,[-1 -1];[1 2 3],[[]];
K = [-1 3;2 0];
H = zpk(Z,P,K);
```

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

## Example 6

Extract the measured and noise components of an identified polynomial model into two separate ZPK models. The former (measured component) can serve as a plant model while the latter can serve as a disturbance model for control system design.

```
load icEngine
z = iddata(y,u,0.04);
nb = 2; nf = 2; nc = 1; nd = 3; nk = 3;
sys = bj(z, [nb nc nd nf nk]);
```

sys is a model of the form,  $y(t) = B/F u(t) + C/D e(t)$ , where B/F represents the measured component and C/D the noise component.

```
sysMeas = zpk(sys, 'measured')
```

Alternatively, use can simply use `zpk(sys)` to extract the measured component.

```
sysNoise = zpk(sys, 'noise')
```

## More About

### Algorithms

zpk uses the MATLAB function `roots` to convert transfer functions and the functions `zero` and `pole` to convert state-space models.

### See Also

`frd` | `get` | `set` | `ss` | `tf` | `zpkdata`

Introduced before R2006a

## zpkdata

Access zero-pole-gain data

### Syntax

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

### Description

`[z,p,k] = zpkdata(sys)` returns the zeros **z**, poles **p**, and gain(s) **k** of the zero-pole-gain model **sys**. The outputs **z** and **p** are cell arrays with the following characteristics:

- **z** and **p** have as many rows as outputs and as many columns as inputs.
- The  $(i, j)$  entries `z{i,j}` and `p{i,j}` are the (column) vectors of zeros and poles of the transfer function from input **j** to output **i**.

The output **k** is a matrix with as many rows as outputs and as many columns as inputs such that  $k(i, j)$  is the gain of the transfer function from input **j** to output **i**. If **sys** is a transfer function or state-space model, it is first converted to zero-pole-gain form using `zpk`.

For SISO zero-pole-gain models, the 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] = zpkdata(sys)
```

 also returns the sample time **Ts**.

`[z,p,k,Ts,covz,covp,covk] = zpkdata(sys)` also returns the covariances of the zeros, poles and gain of the identified model **sys**. **COVZ** is a cell array such that `COVZ{ky,ku}` contains the covariance information about the zeros in the vector `z{ky,ku}`. `COVZ{ky,ku}` is a 3-D array of dimension 2-by-2-by-Nz, where Nz is the



length of  $z\{ky, ku\}$ , so that the (1,1) element is the variance of the real part, the (2,2) element is the variance of the imaginary part, and the (1,2) and (2,1) elements contain the covariance between the real and imaginary parts. `covp` has a similar relationship to `p.covk` is a matrix containing the variances of the elements of  $k$ .

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

```
sys.Ts
sys.inputname
```

## Examples

### Example 1

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...
```

```

      z
#1:  -----
      (z-0.3)

      2 (z+0.5)
#2:  -----
      (z^2 - 0.2z + 1.01)
```

Sample 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
```

### Example 2

Extract the ZPK matrices and their standard deviations for a 2-input, 1 output identified transfer function.

```
load iddata7
```

```
transfer function model
```

```
sys1 = tfest(z7, 2, 1, 'InputDelay',[1 0]);
```

```
an equivalent process model
```

```
sys2 = procest(z7, {'P2UZ', 'P2UZ'}, 'InputDelay',[1 0]);
```

```
1, p1, k1, ~, dz1, dp1, dk1] = zpkdata(sys1);
[z2, p2, k2, ~, dz2, dp2, dk2] = zpkdata(sys2);
```

Use `iopzplot` to visualize the pole-zero locations and their covariances

```
h = iopzplot(sys1, sys2);
showConfidence(h)
```

### See Also

`ssdata` | `tfdata` | `get` | `zpk`

**Introduced before R2006a**

# Block Reference

---

## Kalman Filter

Estimate states of discrete-time or continuous-time linear system



### Description

Use the Kalman Filter block to estimate states of a state-space plant model given process and measurement noise covariance data. The state-space model can be time-varying. A steady-state Kalman filter implementation is used if the state-space model and the noise covariance matrices are all time-invariant. A time-varying Kalman filter is used otherwise.

Kalman filter provides the optimal solution to the following continuous or discrete estimation problems:

#### Continuous-Time Estimation

Given the continuous plant

$$\dot{x}(t) = A(t)x(t) + B(t)u(t) + G(t)w(t) \quad (\text{state equation})$$

$$y(t) = C(t)x(t) + D(t)u(t) + H(t)w(t) + v(t) \quad (\text{measurement equation})$$

with known inputs  $u$ , white process noise  $w$ , and white measurement noise  $v$  satisfying:

$$E[w(t)] = E[v(t)] = 0$$

$$E[w(t)w^T(t)] = Q(t)$$

$$E[w(t)v^T(t)] = N(t)$$

$$E[v(t)v^T(t)] = R(t)$$

construct a state estimate  $\hat{x}$  that minimizes the state estimation error covariance

$$P(t) = E[(x - \hat{x})(x - \hat{x})^T].$$

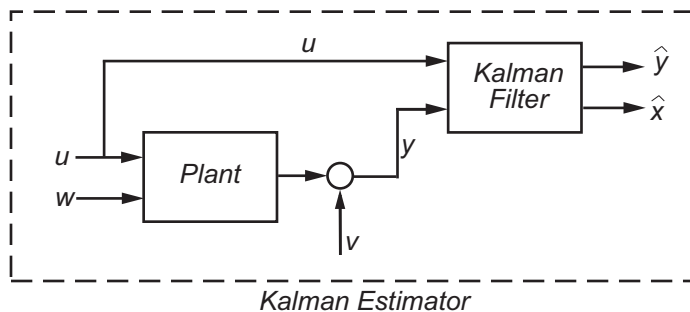
The optimal solution is the Kalman filter with equations

$$\begin{aligned} L(t) &= (P(t)C^T(t) + \bar{N}), \\ \dot{P}(t) &= A(t)P(t) + P(t)A^T(t) + \bar{Q}(t) - L(t)\bar{R}(t)L^T(t), \\ \dot{x}(t) &= A(t)x(t) + B(t)u(t) + L(t)(y(t) - C(t)x(t) - D(t)u(t)), \end{aligned}$$

where

$$\begin{aligned} \bar{Q}(t) &= G(t)Q(t)G^T(t), \\ \bar{R}(t) &= R(t) + H(t)N(t) + N^T(t)H^T(t) + H(t)Q(t)H^T(t), \\ \bar{N}(t) &= G(t)(Q(t)H^T(t) + N(t)). \end{aligned}$$

The Kalman filter uses the known inputs  $u$  and the measurements  $y$  to generate the state estimates  $\hat{x}$ . If you want, the block can also output the estimates of the true plant output  $\hat{y}$ .



The block implements the steady-state Kalman filter when the system matrices ( $A(t)$ ,  $B(t)$ ,  $C(t)$ ,  $D(t)$ ,  $G(t)$ ,  $H(t)$ ) and noise covariance matrices ( $Q(t)$ ,  $R(t)$ ,  $N(t)$ ) are constant (specified in the Block Parameters dialog box). The steady-state Kalman filter uses a constant matrix  $P$  that minimizes the steady-state estimation error covariance and solves the associated continuous-time algebraic Riccati equation:

$$P = \lim_{t \rightarrow \infty} E[(x - \hat{x})(x - \hat{x})^T].$$

### Discrete-Time Estimation

Given the discrete plant

$$\begin{aligned}x[n+1] &= A[n] x[n] + B[n] u[n] + G[n] w[n], \\y[n] &= C[n] x[n] + D[n] u[n] + H[n] w[n] + v[n],\end{aligned}$$

with known inputs  $u$ , white process noise  $w$  and white measurement noise  $v$  satisfying

$$\begin{aligned}E[u[n]] &= E[v[n]] = 0, \\E[u[n]w^T[n]] &= Q[n], \\E[u[n]v^T[n]] &= R[n], \\E[w[n]v^T[n]] &= N[n].\end{aligned}$$

The estimator has the following state equation

$$\hat{x}[n+1 | n] = A[n] \hat{x}[n | n-1] + B[n] u[n] + L[n](y[n] - C[n] \hat{x}[n | n-1] - D[n] u[n]),$$

where the gain  $L[n]$  is calculated through the discrete Riccati equation:

$$\begin{aligned}L[n] &= (A[n]P[n]C^T[n] + \bar{N}[n])(C[n]P[n]C^T[n] + \bar{R}[n])^{-1}, \\M[n] &= P[n]C^T[n](C[n]P[n]C^T[n] + \bar{R}[n])^{-1}, \\Z[n] &= (I - M[n]C[n])P[n](I - M[n]C[n])^T + M[n]\bar{R}[n]M^T[n], \\P[n+1] &= (A[n] - \bar{N}[n]\bar{R}^{-1}[n]C[n])Z[A[n] - \bar{N}[n]\bar{R}^{-1}[n]C[n]]^T + \bar{Q}[n] - N[n]\bar{R}^{-1}[n]N^T[n],\end{aligned}$$

where  $I$  is the identity matrix of appropriate size and

$$\begin{aligned}\bar{Q}[n] &= G[n]Q[n]G^T[n], \\\bar{R}[n] &= R[n] + H[n]N[n] + N^T[n]H^T[n] + H[n]Q[n]H^T[n], \\\bar{N}[n] &= G[n](Q[n]H^T[n] + N[n]),\end{aligned}$$

and

$$\begin{aligned}P[n] &= E[(x - \hat{x}[n | n-1])(x - \hat{x}[n | n-1])^T], \\Z[n] &= E[(x - \hat{x}[n | n])(x - \hat{x}[n | n])^T],\end{aligned}$$

The steady-state Kalman filter uses a constant matrix  $P$  that minimizes the steady-state estimation error covariance and solves the associated discrete-time algebraic Riccati equation.

There are two variants of discrete-time Kalman filters:

- The current estimator generates the state estimates  $\hat{x}[n | n]$  using all measurement available, including  $y[n]$ . The filter updates  $\hat{x}[n | n - 1]$  with  $y[n]$  and outputs:

$$\begin{aligned}\hat{x}[n | n] &= \hat{x}[n | n - 1] + M[n](y[n] - C[n]\hat{x}[n | n - 1] - D[n]u[n]), \\ \hat{y}[n | n] &= C[n]\hat{x}[n | n] + D[n]u[n].\end{aligned}$$

- The delayed estimator generates the state estimates  $\hat{x}[n | n - 1]$  using measurements up to  $y[n - 1]$ . The filter outputs  $\hat{x}[n | n - 1]$  as defined previously, along with the optional output  $\hat{y}[n | n - 1]$

$$\hat{y}[n | n - 1] = C[n]\hat{x}[n | n - 1] + D[n]u[n]$$

The current estimator has better estimation accuracy compared to the delayed estimator, which is important for slow sample times. However, it has higher computational cost, making it harder to implement inside control loops. More specifically, it has direct feedthrough. This leads to an algebraic loop if the Kalman filter is used in a feedback loop that does not contain any delays (the feedback loop itself also has direct feedthrough). The algebraic loop can impact the speed of simulation. You cannot generate code if your model contains algebraic loops.

The Kalman Filter block differs from the `kalman` command in the following ways:

- When calling `kalman(sys, ...)`, `sys` includes the `G` and `H` matrices. Specifically, `sys.B` has `[B G]` and `sys.D` has `[D H]`. When you provide a LTI variable to the Kalman Filter block, it does not assume that the LTI variable provided contains `G` and `H`. They are optional and separate.
- The `kalman` command outputs `[yhat; xhat]` by default. The block only outputs `xhat` by default.

## Parameters

The following table summarizes the Kalman Filter block parameters, accessible via the Block Parameter dialog box.

Task	Parameters
Specify filter settings	<ul style="list-style-type: none"> <li>• <b>Time domain</b></li> </ul>

Task	Parameters
	<ul style="list-style-type: none"> <li>• Use the current measurement <math>y[n]</math> to improve <math>\hat{x}[n]</math></li> </ul>
Specify the system model	<b>Model source</b> in <b>Model Parameters</b> tab
Specify initial state estimates	<b>Source</b> in <b>Model Parameters</b> tab
Specify noise characteristics	In <b>Model Parameters</b> tab: <ul style="list-style-type: none"> <li>• Use <b>G</b> and <b>H</b> matrices (default <b>G=I</b> and <b>H=0</b>)</li> <li>• <b>Q</b>, Time-invariant <b>Q</b></li> <li>• <b>R</b>, Time-invariant <b>R</b></li> <li>• <b>N</b>, Time-invariant <b>N</b></li> </ul>
Specify additional inports	In <b>Options</b> tab: <ul style="list-style-type: none"> <li>• Add input port <b>u</b></li> <li>• Add input port <b>Enable to control measurement updates</b></li> <li>• <b>External reset</b></li> </ul>
Specify additional outports	In <b>Options</b> tab: <ul style="list-style-type: none"> <li>• <b>Output estimated model output y</b></li> <li>• <b>Output state estimation error covariance Z</b></li> </ul>

## Time domain

Specify whether to estimate continuous-time or discrete-time states:

- **Discrete-Time (Default)** — Block estimates discrete-time states
- **Continuous-Time** — Block estimates continuous-time states

When the Kalman Filter block is in a model with synchronous state control (see the **State Control** block), you cannot select **Continuous-time**.



## Use the current measurement $y[n]$ to improve $\hat{x}[n]$

Use the current estimator variant of the discrete-time Kalman filter. When not selected, the delayed estimator (variant) is used.

This option is available only when **Time Domain** is **Discrete-Time**.

## Model source

Specify how the A, B, C, D matrices are provided to the block. Must be one of the following:

- **Dialog: LTI State-Space Variable** — Use the values specified in the LTI state-space variable. You must also specify the variable name in **Variable**. The sample time of the model must match the setting in the **Time domain** option, i.e. the model must be discrete-time if the **Time domain** is discrete-time.
- **Dialog: Individual A, B, C, D matrices** — Specify values in the following block parameters:
  - **A** — Specify the A matrix. It must be real and square.
  - **B** — Specify the B matrix. It must be real and have as many rows as the A matrix. This option is available only when **Add input port u** is selected in the **Options** tab.
  - **C** — Specify the C matrix. It must be real and have as many columns as the A matrix.
  - **D** — Specify the D matrix. It must be real. It must have as many rows as the C matrix and as many columns as the B matrix. This option is available only when **Add input port u** is selected in the **Options** tab.
- **External** — Specify the A, B, C, D matrices as input signals to the Kalman Filter block. If you select this option, the block includes additional input ports A, B, C and D. You must also specify the following in the block parameters:
  - **Number of states** — Number of states to be estimated, specified as a positive integer. The default value is 2.
  - **Number of inputs** — Number of known inputs in the model, specified as a positive integer. The default value is 2. This option is only available when **Add input port u** is selected.
  - **Number of outputs** — Number of measured outputs in the model, specified as a positive integer. The default value is 2.

## Sample Time

Block sample time, specified as -1 or a positive scalar.

This option is available only when **Time Domain** is **Discrete Time** and **Model Source** is **Dialog: Individual A, B, C, D matrices** or **External**. The sample time is obtained from the LTI state-space variable if the Model Source is **Dialog: LTI State-Space Variable**.

The default value is -1, which implies that the block inherits its sample time based on the context of the block within the model. All block input ports must have the same sample time.

## Source

Specify how to enter the initial state estimates and initial state estimation error covariance:

- **Dialog** — Specify the values directly in the dialog box. You must also specify the following parameters:
  - **Initial states  $\mathbf{x}[0]$**  — Specify the initial state estimate as a real scalar or vector. If you specify a scalar, all initial state estimates are set to this scalar. If you specify a vector, the length of the vector must match with the number of states in the model.
  - **State estimation error covariance  $\mathbf{P}[0]$**  (only when time-varying Kalman filter is used) — Specify the initial state estimation error covariance  $\mathbf{P}[0]$  for discrete-time Kalman filter or  $\mathbf{P}(0)$  for continuous-time Kalman filter. Must be specified as one of the following:
    - Real nonnegative scalar.  $\mathbf{P}$  is an  $N_s$ -by- $N_s$  diagonal matrix with the scalar on the diagonals.  $N_s$  is the number of states in the model.
    - Vector of real nonnegative scalars.  $\mathbf{P}$  is an  $N_s$ -by- $N_s$  diagonal matrix with the elements of the vector on the diagonals of  $\mathbf{P}$ .
    - $N_s$ -by- $N_s$  positive semi-definite matrix.
- **External** — Inherit the values from input ports. The block includes an additional input port  $X0$ . A second additional input port  $P0$  is added when time-varying Kalman filter is used.  $X0$  and  $P0$  must satisfy the same conditions described previously when you specify them in the dialog box.

## Use the Kalman Gain K from the model variable

Specify whether to use the pre-identified Kalman Gain contained in the state-space plant model. This option is available only when:

- **Model Source** is **Dialog: LTI State-Space Variable** and **Variable** is an identified state-space model (`idss`) with a nonzero K matrix.
- **Time Invariant Q**, **Time Invariant R** and **Time Invariant N** options are selected.

If the **Use G and H matrices (default G=I and H=0)** option is selected, **Time Invariant G** and **Time Invariant H** options must also be selected.

## Use G and H matrices (default G=I and H=0)

Specify whether to use non-default values for the G and H matrices. If you select this option, you must specify:

- **G** — Specify the G matrix. It must be a real matrix with as many rows as the A matrix. The default value is 1.
- **Time-invariant G** — Specify if the G matrix is time invariant. If you unselect this option, the block includes an additional input port G.
- **H** — Specify the H matrix. It must be a real matrix with as many rows as the C matrix and as many columns as the G matrix. The default value is 0.
- **Time-invariant H** — Specify if the H matrix is time invariant. If you unselect this option, the block includes an additional input port G.
- **Number of process noise inputs** — Specify the number of process noise inputs in the model. The default value is 1.

This option is available only when **Time-invariant G** and **Time-invariant H** are unselected. Otherwise, this information is inferred from the G or H matrix.

## Q

Process noise covariance matrix, specified as one of the following:

- Real nonnegative scalar. **Q** is an  $N_w$ -by- $N_w$  diagonal matrix with the scalar on the diagonals.  $N_w$  is the number of process noise inputs in the model.
- Vector of real nonnegative scalars. **Q** is an  $N_w$ -by- $N_w$  diagonal matrix with the elements of the vector on the diagonals of **Q**.

- $N_w$ -by- $N_w$  positive semi-definite matrix.

## Time Invariant Q

Specify if the Q matrix is time invariant. If you unselect this option, the block includes an additional input port Q.

## R

Measurement noise covariance matrix, specified as one of the following:

- Real positive scalar. R is an  $N_y$ -by- $N_y$  diagonal matrix with the scalar on the diagonals.  $N_y$  is the number of measured outputs in the model.
- Vector of real positive scalars. R is an  $N_y$ -by- $N_y$  diagonal matrix with the elements of the vector on the diagonals of R.
- $N_y$ -by- $N_y$  positive-definite matrix.

## Time Invariant R

Specify if the R matrix is time invariant. If you unselect this option, the block includes an additional input port R.

## N

Process and measurement noise cross-covariance matrix. Specify it as a  $N_w$ -by- $N_y$  matrix. The matrix  $\begin{bmatrix} Q & N \\ N^T & R \end{bmatrix}$  must be positive definite.

## Time Invariant N

Specify if the N matrix is time invariant. If you unselect this option, the block includes an additional input port N.

## Add input port u

Select this option if your model contains known inputs  $u(t)$  or  $u[k]$ . The option is selected by default. Unselecting this option removes the input port u from the block and removes the **B**, **D** and **Number of inputs** parameters from the block dialog box.

## Add input port Enable to control measurement updates

Select this option if you want to control the measurement updates. The block includes an additional input port **Enable**. The **Enable** input port takes a scalar signal. This option is unselected by default.

By default the block does measurement updates at each time step to improve the state and output estimates  $\hat{x}$  and  $\hat{y}$  based on measured outputs. The measurement update is skipped for the current sample time when the signal in the **Enable** port is 0. Concretely, the equation for state estimates become  $\dot{\hat{x}}(t) = A(t)\hat{x}(t) + B(t)u(t)$  for continuous-time Kalman filter and  $\hat{x}[n+1|n] = A[n]\hat{x}[n|n-1] + B[n]u[n]$  for discrete-time.

## External Reset

Option to reset estimated states and parameter covariance matrix using specified initial values.

Suppose you reset the block at a time step,  $\mathbf{t}$ . If the block is enabled at  $\mathbf{t}$ , the software uses the initial parameter values specified either in the block dialog or the input ports **P0** and **X0** to estimate the states. In other words, at  $\mathbf{t}$ , the block performs a time update and if it is enabled, a measurement update after the reset. The block outputs these updated estimates.

Specify one of the following:

- **None (Default)** — Estimated states  $\hat{x}$  and state estimation error covariance matrix **P** values are not reset.
- **Rising** — Triggers a reset when the control signal rises from a negative or zero value to a positive value. If the initial value is negative, rising to zero triggers a reset.
- **Falling** — Triggers a reset when the control signal falls from a positive or a zero value to a negative value. If the initial value is positive, falling to zero triggers a reset.
- **Either** — Triggers a reset when the control signal is either rising or falling.
- **Level** — Triggers a reset in either of these cases:
  - The control signal is nonzero at the current time step.
  - The control signal changes from nonzero at the previous time step to zero at the current time step.

- **Level hold** — Triggers reset when the control signal is nonzero at the current time step.

When you choose an option other than **None**, a **Reset** input port is added to the block to provide the reset control input signal.

## Output estimated model output $\hat{y}$

Add  $\hat{y}$  output port to the block to output the estimated model outputs. The option is unselected by default.

## Output estimated model output P or Z

Add P output port or Z output port to the block. The Z matrix is provided only when **Time Domain** is **Discrete Time** and the **Use the current measurement  $y[n]$  to improve  $\hat{x}[n]$**  is selected. Otherwise, the P matrix, as described in the “Description” on page 3-2 section previously, is provided.

The option is unselected by default.

## Ports

Port Name	Port Type (In/ Out)	Description
u (Optional)	In	Known inputs, specified as a real scalar or vector.
y	In	Measured outputs, specified as a real scalar or vector.
xhat	Out	Estimated states, returned as a real scalar or vector.
yhat (Optional)	Out	Estimated outputs, returned as a real scalar or vector.
P or Z (Optional)	Out	State estimation error covariance, returned as a matrix.
A (Optional)	In	A matrix, specified as a real matrix.

Port Name	Port Type (In/ Out)	Description
B (Optional)	In	B matrix, specified as a real matrix.
C (Optional)	In	C matrix, specified as a real matrix.
D (Optional)	In	D matrix, specified as a real matrix.
G (Optional)	In	G matrix, specified as a real matrix.
H (Optional)	In	H matrix, specified as a real matrix.
Q (Optional)	In	Q matrix, specified as a real scalar, vector or matrix.
R (Optional)	In	R matrix, specified as a real scalar, vector or matrix.
N (Optional)	In	N matrix, specified as a real matrix.
P0 (Optional)	In	P matrix at initial time, specified as a real scalar, vector, or matrix.
X0 (Optional)	In	Initial state estimates, specified as a real scalar or vector.
Enable (Optional)	In	Control signal to enable measurement updates, specified as a real scalar.
Reset (Optional)	In	Control signal to reset state estimates, specified as a real scalar.

## Supported Data Types

- Double-precision floating point
- Single-precision floating point (for discrete-time Kalman filter only)

---

### Note:

- All input ports except **Enable** and **Reset** must have the same data type (single or double).
  - **Enable** and **Reset** ports support `single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and `boolean` data types.
-

## 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

$$\bar{Q} = GQG^T$$

$$\bar{R} = R + HN + N^T H^T + HQH^T$$

$$\bar{N} = G(QH^T + N)$$

- The continuous-time Kalman filter cannot be used in Function-Call Subsystems or Triggered Subsystems.

## References

- [1] Franklin, G.F., J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems*, Second Edition, Addison-Wesley, 1990.
- [2] Lewis, F., *Optimal Estimation*, John Wiley & Sons, Inc, 1986.

## See Also

kalman

## Related Examples

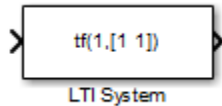
- “State Estimation Using Time-Varying Kalman Filter”

**Introduced in R2014b**



## LTI System

Use linear system model object in Simulink



## Description

The LTI System block imports linear system model objects into the Simulink environment.

The imported system must be proper. 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.

## Parameters

### 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. (When the block is in a model with synchronous state control (see the **State Control** block), you must specify a discrete-time model.)

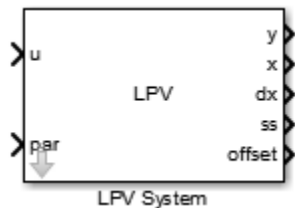
### Initial states (state-space only)

If your model is in state-space format, you can specify the initial states in vector format. The default is zero for all states.

**Introduced before R2006a**

## LPV System

Simulate Linear Parameter-Varying (LPV) systems



## Description

Represent and simulate Linear Parameter-Varying (LPV) systems in Simulink. The block also supports code generation.

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

Mathematically, an LPV system is represented as:

$$dx(t) = A(p)x(t) + B(p)u(t)$$

$$y(t) = C(p)x(t) + D(p)u(t)$$

$$x(0) = x_0$$

where

- $u(t)$  are the inputs
- $y(t)$  the outputs
- $x(t)$  are the model states with initial value  $x_0$
- $dx(t)$  is the state derivative vector  $\dot{x}$  for continuous-time systems and the state update vector  $x(t + \Delta T)$  for discrete-time systems.  $\Delta T$  is the sample time.
- $A(p)$ ,  $B(p)$ ,  $C(p)$  and  $D(p)$  are the state-space matrices parameterized by the scheduling parameter vector  $p$ .

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

The block implements a grid-based representation of the LPV system. You pick a grid of values for the scheduling parameters. At each value  $\mathbf{p} = \mathbf{p}^*$ , you specify the corresponding linear system as a state-space (`ss` or `idss`) model object. You use the generated array of state-space models to configure the LPV System block.

The block accepts an array of state-space models with operating point information. The information on the scheduling variables is extracted from the `SamplingGrid` property of the LTI array. The scheduling variables define the grid of the LPV models. They are scalar-valued quantities that can be functions of time, inputs and states, or constants. They are used to pick the local dynamics in the operating space. The software interpolates the values of these variables. The block uses this array with data interpolation and extrapolation techniques for simulation.

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

$$\begin{aligned} \mathbf{dx}(t) &= \mathbf{A}(p)\mathbf{x}(t) + \mathbf{B}(p)\mathbf{u}(t) + (\overline{\mathbf{dx}}(p) - \mathbf{A}(p)\overline{\mathbf{x}}(p) - \mathbf{B}(p)\overline{\mathbf{u}}(p)) \\ \mathbf{y}(t) &= \mathbf{C}(p)\mathbf{x}(t) + \mathbf{D}(p)\mathbf{u}(t) + (\overline{\mathbf{y}}(p) - \mathbf{C}(p)\overline{\mathbf{x}}(p) - \mathbf{D}(p)\overline{\mathbf{u}}(p)) \\ \mathbf{x}(0) &= \mathbf{x}_0 \end{aligned}$$

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

To obtain such representations of the linear system array, linearize a Simulink model over a batch of operating points (see “Batch Linearization” in Simulink Control Design documentation.) The offsets correspond to the operating points at which you linearized the model.

You can obtain the offsets by returning additional linearization information when calling functions such as `linearize` or `getIOTransfer`. You can then extract the offsets using `getOffsetsForLPV`. For an example, see “LPV Approximation of a Boost Converter Model”.

The following limitations apply to the LPV System block:

- Internal delays cannot be extrapolated to be less than their minimum value in the state-space model array.
- When using an irregular grid of linear models to define the LPV system, only the nearest neighbor interpolation scheme is used. This may reduce the accuracy of simulation results. It is recommended to work with regular grids. To learn more about regular and irregular grids, see “Regular vs. Irregular Grids”.

## Data Type Support

Single and double data. You must convert any other data type for input signals or model properties to these data types.

## Parameters

The LPV System Block Parameter dialog box contains five tabs for specifying the system data, scheduling algorithm and output ports. The following table summarizes the block parameters.

Task	Parameters
Specify an array of state-space models and initial states	In <b>LPV Model</b> tab: <ul style="list-style-type: none"> <li>• <b>State-space array</b></li> <li>• <b>Initial state</b></li> </ul>
Specify operating point offsets	In <b>LPV Model</b> tab: <ul style="list-style-type: none"> <li>• <b>Input offset</b></li> <li>• <b>Output offset</b></li> <li>• <b>State offset</b></li> </ul>
Specify offsets in state derivative or update variable	In the <b>LPV Model</b> tab: <ul style="list-style-type: none"> <li>• <b>State derivative/update offset</b></li> </ul>
Specify which model matrices are fixed and their nominal values to override entries in model data.	In the <b>Fixed Entries</b> tab: <ul style="list-style-type: none"> <li>• <b>Nominal Model</b></li> </ul>

Task	Parameters
In some situations, you may want to replace a parameter-dependent matrix such as $A(\rho)$ with a fixed value $A^*$ for simulation. For example, $A^*$ may represent an average value over the scheduling range.	<ul style="list-style-type: none"> <li>• <b>Fixed Coefficient Indices</b></li> </ul>
Specify options for interpolation and extrapolation	<p>In the <b>Scheduling</b> tab:</p> <ul style="list-style-type: none"> <li>• <b>Interpolation method</b></li> <li>• <b>Extrapolation method</b></li> <li>• <b>Index search method</b></li> <li>• <b>Begin index search using previous index result</b></li> </ul>
Specify additional outputs for the block	<p>In the <b>Outputs</b> tab:</p> <ul style="list-style-type: none"> <li>• <b>Output states</b></li> <li>• <b>Output state derivatives (continuous-time) or updates (discrete-time)</b></li> <li>• <b>Output interpolated state-space data</b></li> <li>• <b>Output interpolated offsets</b></li> </ul>
Specify code generation settings	<p>In the <b>Code Generation</b> tab:</p> <ul style="list-style-type: none"> <li>• <b>Block data type (discrete-time case only)</b></li> <li>• <b>Initial buffer size for delays</b></li> <li>• <b>Use fixed buffer size</b></li> </ul>

## State-space array

An array of state-space (**ss** or **idss**) models. All the models in the array must use the same definition of states. Use the **SamplingGrid** property of the state-space object to specify scheduling parameters for the model. See the **ss** or **idss** model reference page for more information on the **SamplingGrid** property.

When the block is in a model with synchronous state control (see the `State Control` block), you must specify an array of discrete-time models.

## Initial state

Initial conditions to use with the local model to start the simulation, specified one of the following:

- **0 (Default)**
- Double vector of length equal to the number of model states

## Input offset

Offsets in input  $u(\tau)$ , specified as one of the following:

- **0 (Default)** — Use when there are no input offsets ( $\bar{u}(p) = 0 \forall p$ ).
- Double vector of length equal to the number of inputs — Use when input offset is the same across the scheduling space.
- Double array of size `[nu 1 sysArraySize]` — Use when offsets are present and they vary across the scheduling space. Here, `nu` = number of inputs, `sysArraySize` = array size of state-space array. Use `size` to determine the array size.

You can obtain offsets during linearization and convert them to the format supported by the LPV System block. For more information, see “Approximating Nonlinear Behavior using an Array of LTI Systems” and `getOffsetsForLPV`.

## Output offset

Offsets in output  $y(\tau)$ , specified as one of the following:

- **0 (Default)** — Use when there are no output offsets ( $\bar{y}(p) = 0 \forall p$ ).
- Double vector of length equal to the number of outputs. Use when output offsets are the same across the scheduling space.
- Double array of size `[ny 1 sysArraySize]`. Use when offsets are present and they vary across the scheduling space. Here, `ny` = number of outputs, `sysArraySize` = array size of state-space array. Use `size` to determine the array size.

You can obtain offsets during linearization and convert them to the format supported by the LPV System block. For more information, see “Approximating Nonlinear Behavior using an Array of LTI Systems” and `getOffsetsForLPV`.

## State offset

Offsets in states  $x(t)$ , specified as one of the following:

- **0 (Default)** — Use when there are no state offsets  $\bar{x}(p) = 0 \forall p$ .
- Double vector of length equal to the number of states. Use when the state offsets are the same across the scheduling space.
- Double array of size `[nx 1 sysArraySize]`, where `nx` = number of states, `sysArraySize` = array size of state-space array. Use when offsets are present and they vary across the scheduling space. Here, `nx` = number of states, `sysArraySize` = array size of state-space array. Use `size` to determine the array size.

You can obtain offsets during linearization and convert them to the format supported by the LPV System block. For more information, see “Approximating Nonlinear Behavior using an Array of LTI Systems” and `getOffsetsForLPV`.

## State derivative/update offset

Offsets in state derivative or update variable  $dx(t)$ , specified as one of the following:

- If you obtained the linear system array by linearization under equilibrium conditions, select the **Assume equilibrium conditions** option. This option corresponds to an offset of  $\overline{dx}(p) = 0$  for a continuous-time system and  $\overline{dx}(p) = \bar{x}(p)$  for a discrete-time system. This option is selected by default.
- If the linear system contains at least one system that you obtained under non-equilibrium conditions, clear the **Assume equilibrium conditions** option. Specify one of the following in the **Offset value** field:
  - If the `dx` offset values are the same across the scheduling space, specify as a double vector of length equal to the number of states.
  - If the `dx` offsets are present and they vary across the scheduling space, specify as a double array of size `[nx 1 sysArraySize]`, where `nx` = number of states, and `sysArraySize` = array size of state-space array.

You can obtain offsets during linearization and convert them to the format supported by the LPV System block. For more information, see “Approximating Nonlinear Behavior using an Array of LTI Systems” and `getOffsetsForLPV`.

## Nominal Model

State-space model that provides the values of the fixed coefficients, specified as one of the following:

- **Use the first model in state-space array (Default):** — The first model in the state-space array is used to represent the LPV model. In the following example, the state-space array is specified by object `sys` and the fixed coefficients are taken from model `sys(:, :, 1)`.

```
% Specify a 4-by-5 array of state-space models.
sys = rss(4,2,3,4,5);
a = 1:4;
b = 10:10:50;
[av,bv] = ndgrid(a,b);
% Use “alpha” and “beta” variables as scheduling parameters.
sys.SamplingGrid = struct('alpha',av,'beta',bv);
```

Fixed coefficients are taken from the model `sysFixed = sys(:, :, 1)`, which corresponds to `[alpha=1, beta=10]`. If the (2,1) entry of A matrix is forced to be fixed, its value used during the simulation is `sysFixed.A(2, 1)`.

- **Custom value** — Specify a different state-space model for fixed entries. Specify a variable for the fixed model in the **State space model** field. The fixed model must use the same state basis as the state-space array in the LPV model.

## Fixed Coefficient Indices

Specify which coefficients of the state-space matrices and delay vectors are fixed.

Specify one of the following:

- **Scalar Boolean (true or false)**, if all entries of a matrix are to be treated the same way.

The default value is **false** for the state-space matrices and delay vectors, which means that they are treated as free.



- Logical matrix of a size compatible with the size of the corresponding matrix:

State-space matrix	Size of fixed entry matrix
<b>A matrix</b>	$n_x$ -by- $n_x$
<b>B matrix</b>	$n_x$ -by- $n_u$
<b>C matrix</b>	$n_y$ -by- $n_x$
<b>D matrix</b>	$n_y$ -by- $n_u$
<b>Input delay</b>	$n_u$ -by-1
<b>Output delay</b>	$n_y$ -by-1
<b>Internal delay</b>	$n_i$ -by-1

where,  $n_u$  = number of inputs,  $n_y$  = number of outputs,  $n_x$  = number of states,  $n_i$  = length of internal delay vector.

- Numerical indices to specify the location of fixed entries. See `sub2ind` reference page for more information on how to generate numerical indices corresponding to a given subscript ( $i, j$ ) for an element of a matrix.

## Interpolation method

Interpolation method. Defines how the state-space data must be computed for scheduling parameter values that are located away from their grid locations.

Specify one of the following options:

- **Flat** — Choose the state-space data at the grid point closest, but not larger than, the current point. The *current point* is the value of the scheduling parameters at current time.
- **Nearest** — Choose the state-space data at the closest grid point in the scheduling space.
- **Linear** — Obtain state-space data by linear interpolation of the nearest 2d neighbors in the scheduling space, where  $d$  = number of scheduling parameters.

The default interpolation scheme is **Linear** for regular grids of scheduling parameter values. For irregular grids, the **Nearest** interpolation scheme is always used regardless of the choice made. to learn more about regular and irregular grids, see “Regular vs. Irregular Grids”.

The **Linear** method provides the highest accuracy but takes longer to compute. The **Flat** and **Nearest** methods are good for models that have mode-switching dynamics.

### Extrapolation method

Extrapolation method. Defines how to compute the state-space data for scheduling parameter values that fall outside the range over which the state-space array has been provided (as specified in the **SamplingGrid** property).

Specify one of the following options:

- **Clip (Default):** — Disables extrapolation and returns the data corresponding to the last available scheduling grid point that is closest to the current point.
- **Linear** — Fits a line between the first or last pair of values for each scheduling parameter, depending upon whether the current value is less than the first or greater than the last grid point value, respectively. This method returns the point on that line corresponding to the current value. Linear extrapolation requires that the interpolation scheme be linear too.

### Index search method

The location of the current scheduling parameter values in the scheduling space is determined by a prelookup algorithm. Select **Linear search** or **Binary search**. Each search method has speed advantages in different situations. For more information on this parameter, see the Prelookup block reference page in Simulink documentation.

### Begin index search using previous index result

Select this check box when you want the block to start its search using the index found at the previous time step. For more information on this parameter, see the Prelookup block reference page in Simulink documentation.

### Output states

Add **x** port to the block to output state values. This option is selected by default.

### Output state derivatives (continuous-time) or updates (discrete-time)

Add **dx** port to the block to output state derivative values or update the values. This option is selected by default.

## Output interpolated state-space data

Add `ss` port to the block to output state-space data as a structure. This option is selected by default.

The fields of the generated structure are:

- State-space matrices `A`, `B`, `C`, `D`.
- Delays `InputDelay`, `OutputDelay`, and `InternalDelay`. The `InternalDelay` field is available only when the model has internal delay.

## Output interpolated offsets

Add `offset` port to the block to output LPV model offsets  $(\bar{u}(p), \bar{y}(p), \bar{x}(p), \bar{dx}(p))$ .

The fields of the structure are:

- `InputOffset`, `OutputOffset`, `StateOffset`, and `StateDerivativeOffset` in continuous-time.
- `InputOffset`, `OutputOffset`, `StateOffset`, and `StateUpdateOffset` in discrete-time.

## Block data type (discrete-time case only)

Supported data type. Use this option only for discrete-time state-space models. Specify `double` or `single`.

## Initial buffer size for delays

Initial memory allocation for the number of input points to store for models that contain delays. If the number of input points exceeds the initial buffer size, the block allocates additional memory. The default size is 1024.

When you run the model in Accelerator mode or build the model, make sure the initial buffer size is large enough to handle maximum anticipated delay in the model.

## Use fixed buffer size

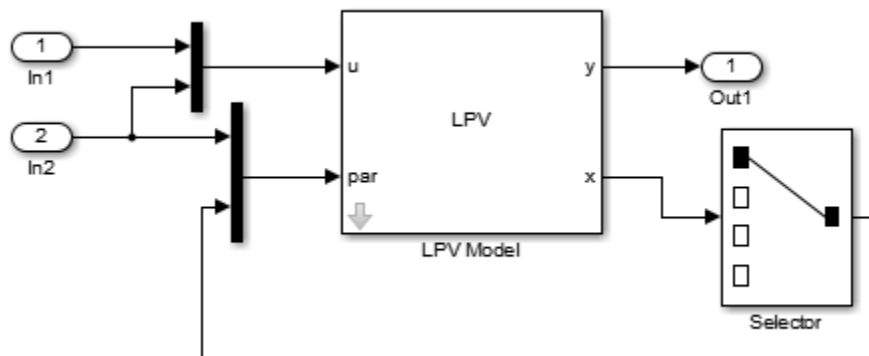
Specify whether to use a fixed buffer size to save delayed input and output data from previous time steps. Use this option for continuous-time LPV systems that contain input

or output delays. If the buffer is full, new data replaces data already in the buffer. The software uses linear extrapolation to estimate output values that are not in the buffer.

## Examples

### Configure the Scheduling Parameter Input Port

Consider a 2-input, 3-output, 4-state LPV model. Use input  $u(2)$  and state  $x(1)$  as scheduling parameters. Configure the Simulink model as shown in the following figure.



### Simulate a Linear Parameter-Varying System

Consider a linear mass-spring-damper system whose mass changes as a function of an external load command. The governing equation is:

$$m(u)\ddot{y} + c\dot{y} + k(y)y = F(t)$$

where  $m(u)$  is the mass dependent upon the external command  $u$ ,  $c$  is the damping ratio,  $k$  is the stiffness of the spring and  $F(t)$  is the forcing input.  $y(t)$  is position of the mass at a given time  $t$ . For a fixed value of  $u$ , the system is linear and expressed as:

$$A = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{c}{m} \end{bmatrix}, B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}, C = [1 \ 0]$$

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

where  $x = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}$  is the state vector and  $m$  is the value of the mass for a given value of  $u$ .

In this example, you want to study the model behavior over a range of input values from 1 to 10 Volts. For each value of  $u$ , measure the mass and compute the linear representation of the system. Suppose, mass is related to the input by the relationship:

$m(u) = 10u + 0.1u^2$ . For values of  $u$  ranging from 1:10 results in the following array of linear systems.

```
% Specify damping coefficient.
c = 5;
% Specify stiffness.
k = 300;
% Specify load command.
u = 1:10;
% Specify mass.
m = 10*u + 0.1*u.^2;
% Compute linear system at a given mass value.
for i = 1:length(u)
    A = [0 1; -k/m(i), -c/m(i)];
    B = [0; -1/m(i)];
    C = [1 0];
    sys(:,:,i) = ss(A,B,C,0);
end
```

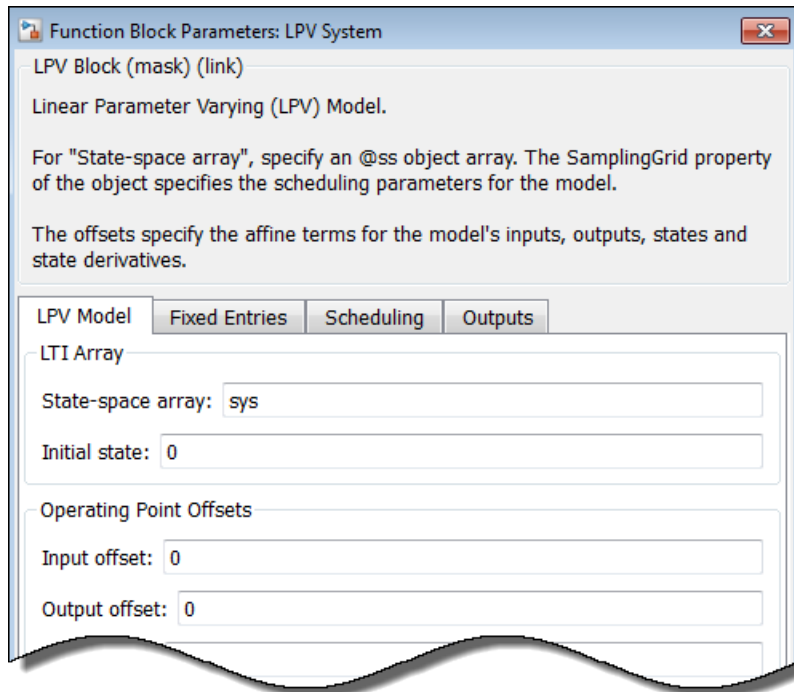
The variable  $u$  is the scheduling input. Add this information to the model.

```
sys.SamplingGrid = struct('LoadCommand',u);
```

Configure the LPV System block:

- Type `sys` in the **State-space array** field.
- Connect the input port `par` to a one-dimensional source signal that generates the values of the load command. If the source provides values between 1 and 10,

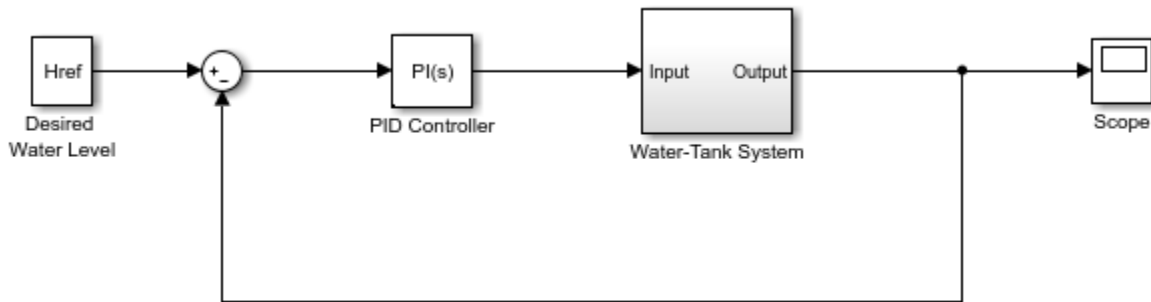
interpolation is used to compute the linear model at a given time instance. Otherwise, extrapolation is used.



## Extract LPV Offsets from Linearization Results

Open the Simulink model.

```
model = 'watertank';  
open_system(model)
```



Copyright 2004-2012 The MathWorks, Inc.

Specify linearization I/Os.

```
io(1) = linio('watertank/Desired Water Level',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

Vary plant parameters A and b, and create a 3-by-4 parameter grid.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),linspace(0.9*b,1.1*b,4));
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a linearization option set, setting the `StoreOffsets` option to `true`.

```
opt = linearizeOptions('StoreOffsets',true);
```

Linearize the model using the specified parameter grid, and return the linearization offsets in the `info` structure.

```
[sys,op,info] = linearize('watertank',io,params,opt);
```

Extract the linearization offsets.

```
offsets = getOffsetsForLPV(info)
```

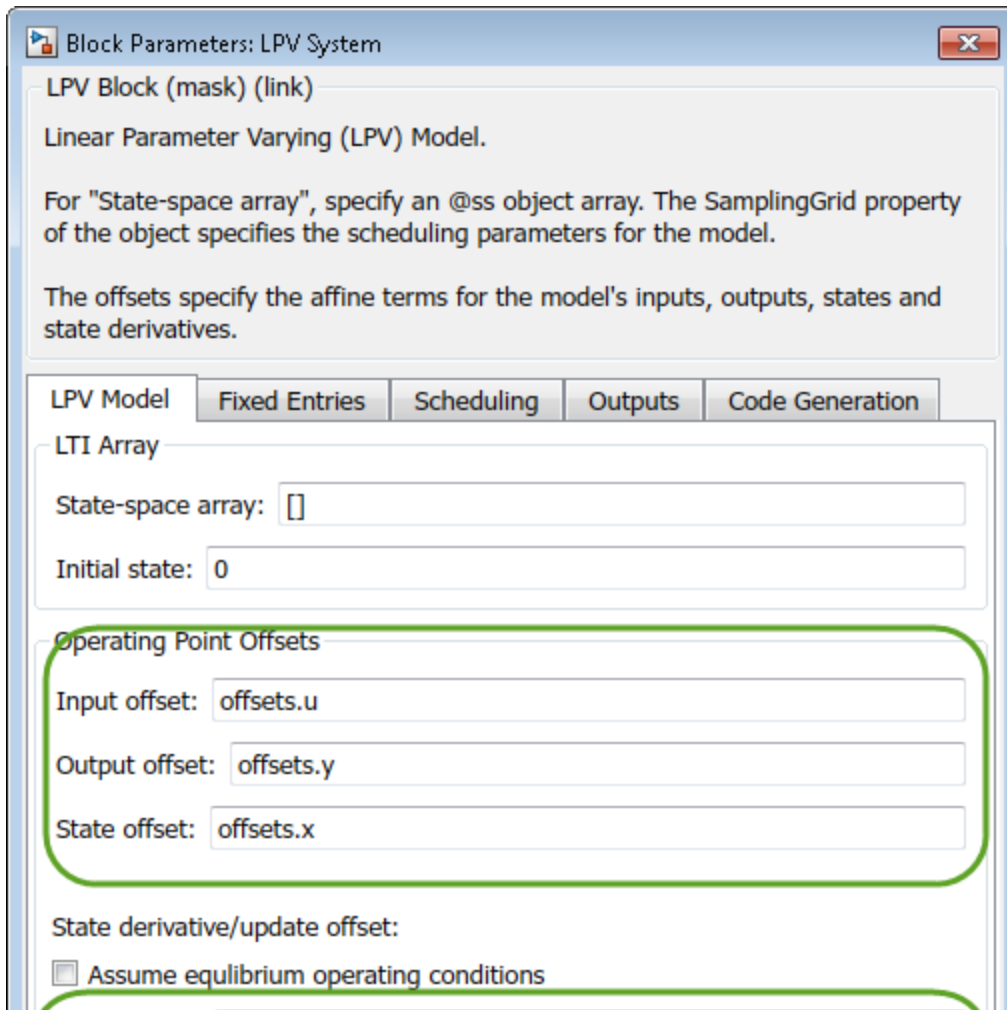
```
offsets =
```

struct with fields:

```
x: [2×1×3×4 double]
y: [1×1×3×4 double]
u: [1×1×3×4 double]
dx: [2×1×3×4 double]
```

To configure an LPV System block, use the fields from `offsets` directly.





Port Name	Port Type (In/ Out)	Description
u	In	Input signal $u(t)$ in Equation 3-2 described previously. In multi-input case, this port accepts a signal of the dimension of the input.

Port Name	Port Type (In/ Out)	Description
par	In	Provides the signals for variables defining the scheduling space (“sampling grid” variables). The scheduling variables can be functions of time, inputs and states, or constants. The required dependence can be achieved by preparing a scheduling signal using clock input (for time), input signal (u), and the outputs signals (x, dx/dt, y) of the LPV block, as required.
y	Out	Model output
x	Out	Values of the model states
xdot	Out	Values of the state derivatives. The state derivatives are sometimes used to define the scheduling parameters.
ss	Out	Local state-space model at the major simulation time steps
offset	Out	LPV model offsets

### See Also

getOffsetsForLPV

### More About

- “Linear Parameter-Varying Models”
- “Using LTI Arrays for Simulating Multi-Mode Dynamics”
- “Approximating Nonlinear Behavior using an Array of LTI Systems”
- “LPV Approximation of a Boost Converter Model”

**Introduced in R2014b**