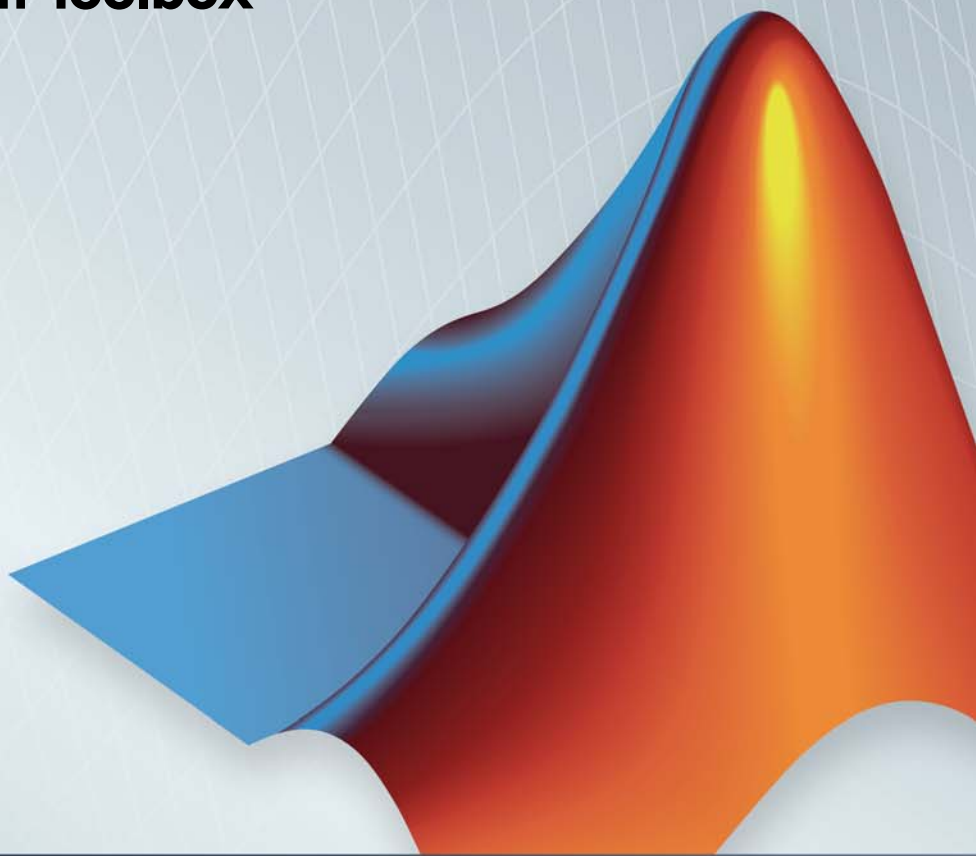


# Control System Toolbox™

## Reference

R2013b



# MATLAB®



## How to Contact MathWorks



[www.mathworks.com](http://www.mathworks.com) Web  
[comp.soft-sys.matlab](mailto:comp.soft-sys.matlab) Newsgroup  
[www.mathworks.com/contact\\_TS.html](http://www.mathworks.com/contact_TS.html) Technical Support



[suggest@mathworks.com](mailto:suggest@mathworks.com) Product enhancement suggestions  
[bugs@mathworks.com](mailto:bugs@mathworks.com) Bug reports  
[doc@mathworks.com](mailto:doc@mathworks.com) Documentation error reports  
[service@mathworks.com](mailto:service@mathworks.com) Order status, license renewals, passcodes  
[info@mathworks.com](mailto:info@mathworks.com) Sales, pricing, and general information



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

*Control System Toolbox™ Reference*

© COPYRIGHT 2001–2013 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)



## Functions — Alphabetical List

**1**

## Block Reference

**2**

## Index



# Functions — Alphabetical List

---

# abs

---

**Purpose** 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`



**Purpose** 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 **Example 1**

Create a discrete-time transfer function that has a time delay and absorb the time delay into the system dynamics as poles at  $z = 0$ .

```
z = tf('z',-1);  
sysd = (-.4*z - .1)/(z^2 + 1.05*z + .08);  
sysd.InputDelay = 3
```

These commands produce the result:

Transfer function:  
$$z^{-3} * \frac{-0.4 z - 0.1}{z^2 + 1.05 z + 0.08}$$

# absorbDelay

---

Sampling time: unspecified

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 delay into the system dynamics.

```
sysnd = absorbDelay(sysd)
```

The display of `sysnd` shows that the factor of  $z^{-3}$  has been absorbed as additional poles in the denominator.

```
Transfer function:
      -0.4 z - 0.1
-----
z^5 + 1.05 z^4 + 0.08 z^3
```

Sampling time: unspecified

Additionally, `sysnd` has no input delay:

```
sysnd.InputDelay
```

```
ans =
```

```
0
```

## Example 2

Convert "nk" into regular coefficients of a polynomial model.

Consider the discrete-time polynomial model:

```
m = idpoly(1,[0 0 0 2 3]);
```

The value of the B polynomial, `m.b`, has 3 leading zeros. Two of these zeros are treated as input-output delays. Consequently:

```
sys = tf(m)
```

creates a transfer function such that the numerator is [0 2 3] and the IO delay is 2. In order to treat the leading zeros as regular B coefficients, use absorbDelay:

```
m2 = absorbDelay(m);  
sys2 = tf(m2);
```

sys2's numerator is [0 0 0 2 3] and IO delay is 0. The model m2 treats the leading zeros as regular coefficients by freeing their values. m2.Structure.b.Free(1:2) is TRUE while m.Structure.b.Free(1:2) is FALSE.

## See Also

hasdelay | pade | totaldelay

# allmargin

---

**Purpose** Gain margin, phase margin, delay margin and crossover frequencies

**Syntax**  
`S = allmargin(sys)`  
`S = allmargin(mag,phase,w,ts)`

**Description** `S = allmargin(sys)`

`allmargin` 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 sampling 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

`ltiview` | `margin`

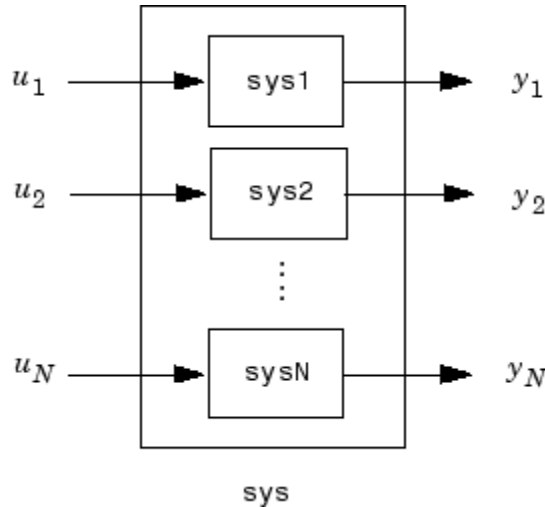
# append

**Purpose** Group models by appending their inputs and outputs

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

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

`append` 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 `s` 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

The commands

```
sys1 = tf(1,[1 0]);
sys2 = ss(1,2,3,4);
sys = append(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
```

# append

---

```
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

`connect` | `feedback` | `parallel` | `series`



**Purpose** Append state vector to output vector

**Syntax** `asys = augstate(sys)`

**Description** `asys = augstate(sys)`  
Given a state-space model `sys` with equations

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

(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`

# balreal

---

**Purpose** Gramian-based input/output balancing of state-space realizations

**Syntax**

```
[sysb, g] = balreal(sys)
[sysb, g] =
balreal(sys, 'AbsTol', ATOL, 'RelTol', RTOL, 'Offset',
          ALPHA)
[sysb, g] = balreal(sys, condmax)
[sysb, g, T, Ti] = balreal(sys)
[sysb, g] = 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] =
balreal(sys, 'AbsTol', ATOL, 'RelTol', RTOL, 'Offset', ALPHA)
```

specifies additional options for the stable/unstable decomposition. See the `stabsep` reference page for more information about these options. The default values are `ATOL = 0`, `RTOL = 1e-8`, and `ALPHA = 1e-8`.

`[sysb, g] = balreal(sys, condmax)` controls the condition number of the stable/unstable decomposition. Increasing `condmax` helps separate close by stable and unstable modes at the expense of accuracy. By default `condmax=1e8`.

`[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  $T^{-1}$ .

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.

`[sysb, g] = balreal(sys, opts)` computes the balanced realization using the options specified in the `hsvdOptions` object `opts`.

## Examples

### Example 1

Consider the zero-pole-gain model

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

Zero/pole/gain:

```
(s+10) (s+20.01)
-----
(s+5) (s+9.9) (s+20.1)
```

A state-space realization with balanced gramians is obtained by

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

The diagonal entries of the joint gramian are

```
g'
```

```
ans =
```

```
0.1006    0.0001    0.0000
```

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

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

to obtain the following first-order approximation of the original system.

```
zpk(sysr)
```

Zero/pole/gain:

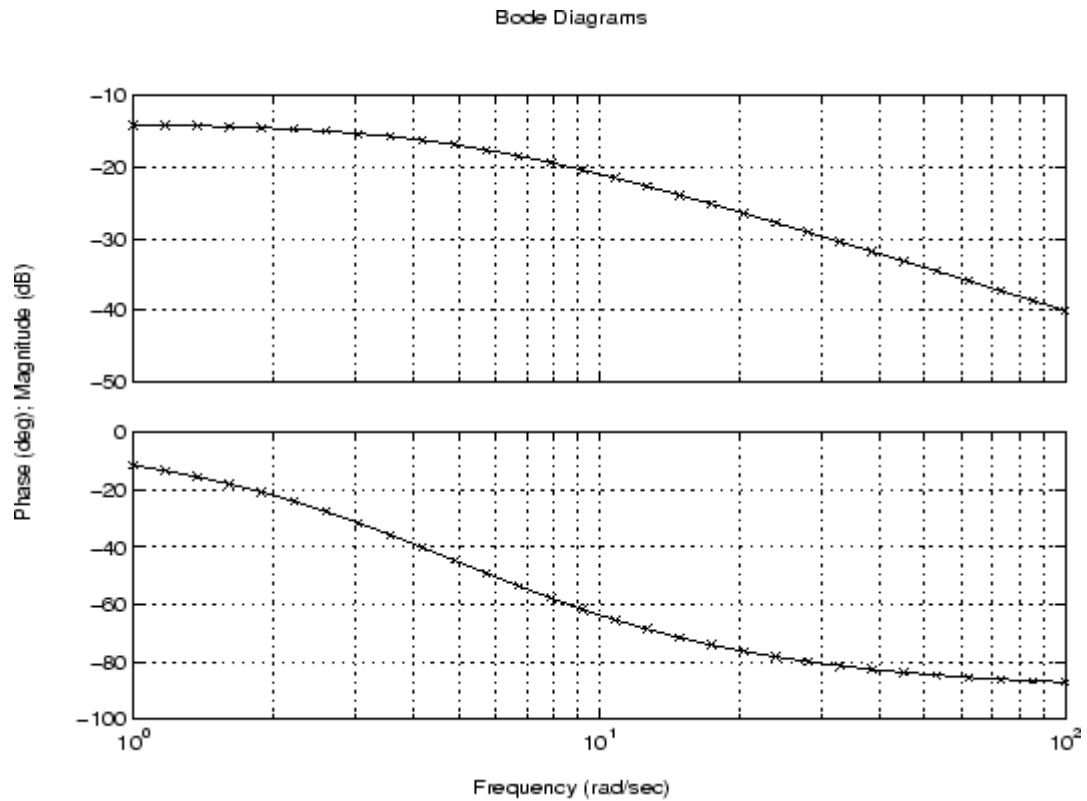
```
1.0001
```

```
-----
```

```
(s+4.97)
```

Compare the Bode responses of the original and reduced-order models.

```
bode(sys, '-', sysr, 'x')
```



### Example 2

Create this unstable system:

```
sys1=tf(1,[1 0 -1])
```

Transfer function:

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

Apply `balreal` to create a balanced gramian realization.

# balreal

---

```
[sysb,g]=balreal(sys1)
```

```
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 model.

```
g =  
      Inf  
0.2500
```

The unstable pole shows up as Inf in vector g.

## Algorithms

Consider the model

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

with controllability and observability gramians  $W_c$  and  $W_o$ . The state coordinate transformation  $\bar{x} = T x$  produces the equivalent model

$$\begin{aligned}\dot{\bar{x}} &= T A T^{-1} \bar{x} + T B u \\ y &= C T^{-1} \bar{x} + D u\end{aligned}$$

and transforms the gramians to

$$\bar{W}_c = T W_c T^T, \quad \bar{W}_o = T^{-T} W_o T^{-1}$$

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.

## References

- [1] Laub, A.J., M.T. Heath, C.C. Paige, and R.C. Ward, "Computation of System Balancing Transformations and Other Applications of Simultaneous Diagonalization Algorithms," *IEEE® Trans. Automatic Control*, AC-32 (1987), pp. 115-122.
- [2] Moore, B., "Principal Component Analysis in Linear Systems: Controllability, Observability, and Model Reduction," *IEEE Transactions on Automatic Control*, AC-26 (1981), pp. 17-31.
- [3] Laub, A.J., "Computation of Balancing Transformations," *Proc. ACC*, San Francisco, Vol.1, paper FA8-E, 1980.

## See Also

`hsvdOptions` | `gram` | `modred` | `ss`

# balred

---

**Purpose** 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.

*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 the specified options for the stable/unstable decomposition and state elimination method. Use the *balredOptions* command to create the option *setopts*.

---

**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)

---



**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` | `order` | `minreal` | `sminreal`

**Related Examples**

- "Approximate Model with Lower-Order Model"
- "Approximate Model with Unstable or Near-Unstable Pole"

**Concepts**

- "Why Simplify Models?"

# balredOptions

---

**Purpose** 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

#### '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:

- 'MatchDC' Discards the specified states and alters the remaining states to preserve the DC gain.
- 'Truncate' 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

$\text{AbsTol} + \text{RelTol} * \text{abs}(G)$ . Increasing these tolerances helps separate nearby stable and unstable modes at the expense of accuracy. See `stabsep` for more information.

**Default:**  $\text{AbsTol} = 0$ ;  $\text{RelTol} = 1\text{e-}8$

## '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:**  $1\text{e-}8$

For additional information on the options and how to use them, see the `balred` reference page.

## Examples

Compute a reduced-order approximation of the system given by:

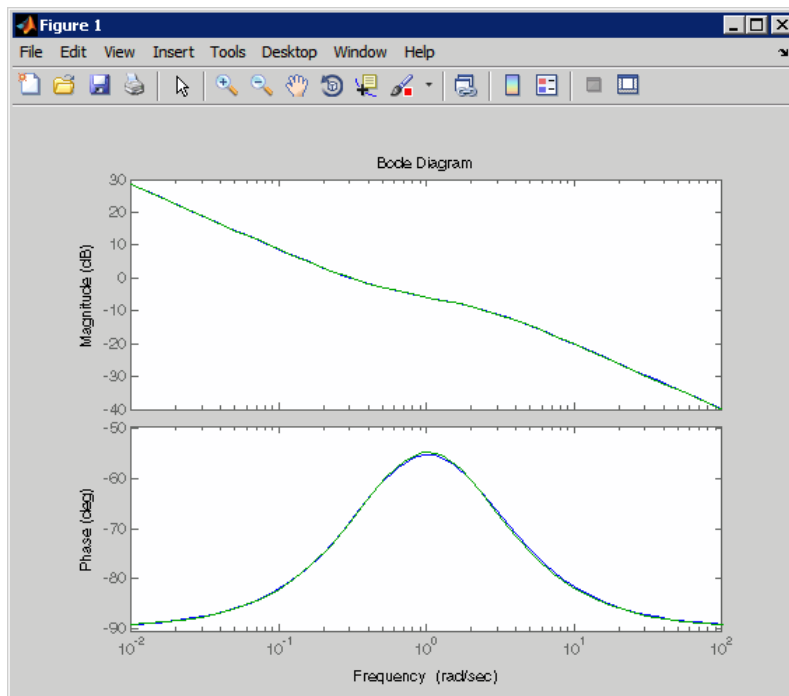
$$G(s) = \frac{(s+0.5)(s+1.1)(s+2.9)}{(s+10^{-6})(s+1)(s+2)(s+3)}$$

Use the `Offset` option to exclude the pole at  $s = 10^{-6}$  from the stable term of the stable/unstable decomposition.

```
sys = zpk([-0.5 -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 original and reduced-order models with `bode`:

`bode(sys,rsys)`



**See Also**      `balred` | `stabsep`

**Purpose** 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)` further specifies the critical gain drop in dB. The default value is -3 dB, or a 70.79 percent drop.

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

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

**See Also** `dcgain` | `issiso`

# bdschur

---

**Purpose** 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,  $CONDMAX = 1/\sqrt{\text{eps}}$ . 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`

**Purpose** Block-diagonal concatenation of models

**Syntax** `sys = blkdiag(sys1,sys2,...,sysN)`

**Description** `sys = blkdiag(sys1,sys2,...,sysN)` produces the aggregate system

$$\begin{bmatrix} \text{sys1} & 0 & \dots & 0 \\ 0 & \text{sys2} & \dots & \vdots \\ \vdots & \dots & \dots & 0 \\ 0 & \dots & 0 & \text{sysN} \end{bmatrix}$$

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](#)



**Purpose** Bode plot of frequency response, magnitude and phase of frequency response

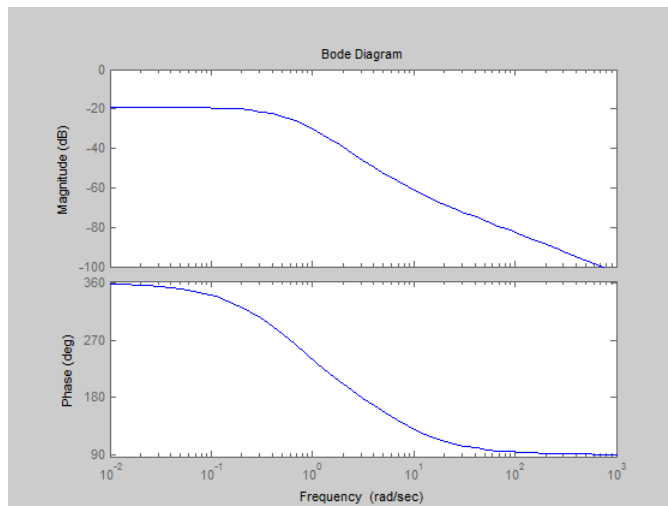
**Syntax**

```
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

---

`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)` plots system responses using the color, linestyle, and markers specified by the `PlotStyle` strings.

`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 one-, two-, or three-part string enclosed in single quotes ( ' ' ). The elements of the string can appear in any order. The string can specify only the line style, the marker, or the color.

For more information about configuring the `PlotStyle` string, see “Colors, Line Styles, 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:

$$\text{magdb} = 20 \cdot \log_{10}(\text{mag})$$

## **phase**

Phase of the system response in degrees, returned as a 3-D array with dimensions are (number of outputs)  $\times$  (number of inputs)  $\times$  (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**

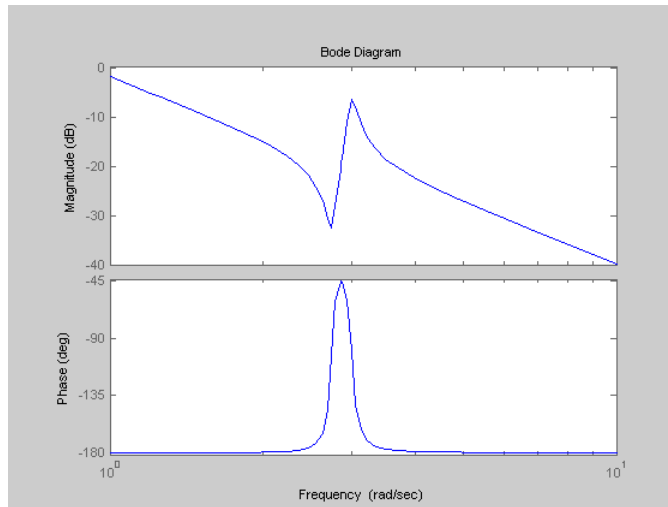
### **Bode Plot of Dynamic System**

Create Bode plot of the dynamic system:

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

$H(s)$  is a continuous-time SISO system.

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



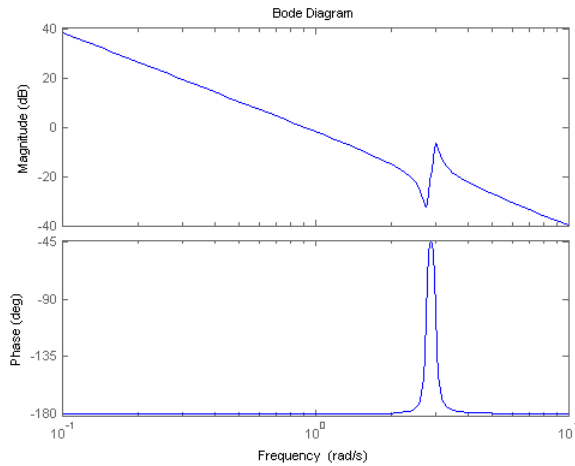
bode automatically selects the plot range based on the system dynamics.

### Bode Plot at Specified Frequencies

Create 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.



Alternatively, you can specify a vector of frequencies to use for evaluating and plotting the frequency response.

```
w = logspace(-1,1,50);  
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

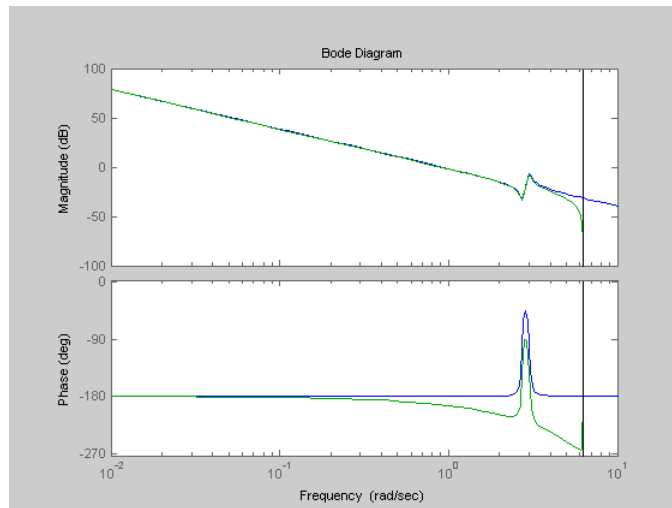
Compare the frequency response of a continuous-time system to an equivalent discretized system on the same Bode plot.

**1** 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');
```

**2** Create Bode plot that includes both systems.

`bode(H,Hd)`



### Bode Plot with Specified Line and Marker Attributes

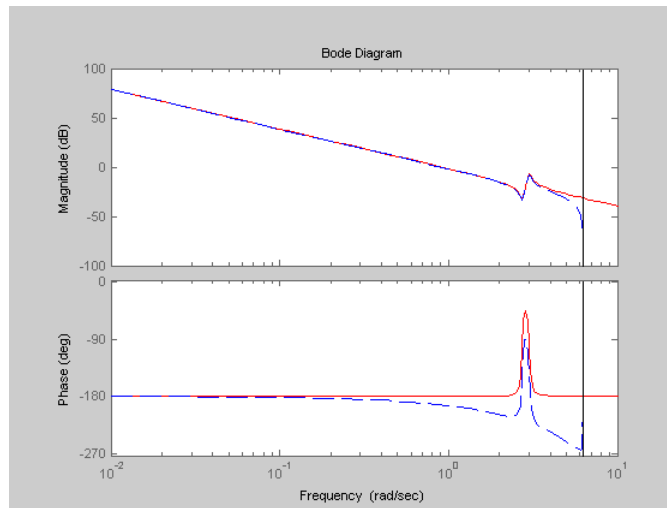
Specify the color, linestyle, or marker for each system in a Bode plot using the `PlotStyle` input arguments.

```
H = tf([1 0.1 7.5],[1 0.12 9 0 0]);
Hd = c2d(H,0.5,'zoh');
```

H and Hd are two different systems.

```
bode(H,'r',Hd,'b--')
```

The string `'r'` specifies a solid red line for the response of H. The string `'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 non-parametric model identified using the same data.

1 Identify parametric and non-parametric models based on data.

```
load iddata2 z2;
```



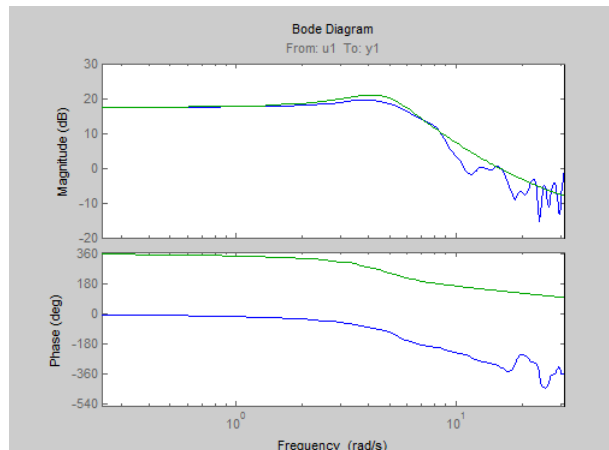
```
w = linspace(0,10*pi,128);  
sys_np = spa(z2,[],w);  
sys_p = tfest(z2,2);
```

`spa` and `tfest` require System Identification Toolbox™ software.

`sys_np` is a non-parametric identified model. `sys_p` is a parametric identified model.

- 2 Create a Bode plot that includes both systems.

```
bode(sys_np,sys_p,w);
```



### 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.

- 1 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);
```

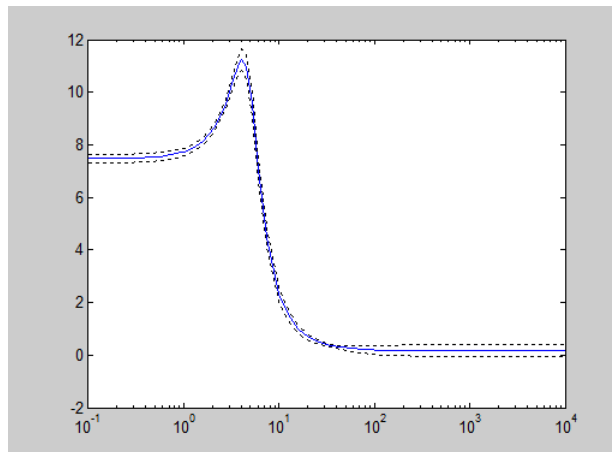
tfest requires System Identification Toolbox software.

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.

- 2 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:');
```



## 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 sampling 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 sampling time, bode uses  $T_s = 1$ .

## Alternatives

Use bodeplot when you need additional plot customization options.

## See Also

bodeplot | freqresp | nichols | nyquist

## How To

- “Dynamic System Models”

# bodemag

---

**Purpose** 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` | `ltiview`

**Purpose** Create list of Bode plot options

**Syntax**  
 P = bodeoptions  
 P = bodeoptions('cstprefs')

**Description** P = bodeoptions returns a list of available options for Bode plots with default values set. You can use these options to customize the Bode plot appearance using the command line.

P = bodeoptions('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.

The following table summarizes the Bode 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 strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

# bodeoptions

---

Option	Description
ConfidenceRegionNumber	Number of standard deviations to use to plotting the response confidence region (identified models only). <b>Default:</b> 1.
FreqUnits	Frequency units, specified as one of the following strings: <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></ul>

Option	Description
	<ul style="list-style-type: none"> <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 multiple systems with different time units, the units of the first system are used.</p>
FreqScale	Frequency scale Specified as one of the following strings: 'linear'   'log' <b>Default:</b> 'log'
MagUnits	Magnitude units Specified as one of the following strings: 'dB'   'abs' <b>Default:</b> 'dB'
MagScale	Magnitude scale Specified as one of the following strings: 'linear'   'log' <b>Default:</b> 'linear'
MagVisible	Magnitude plot visibility Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'on'
MagLowerLimMode	Enables a lower magnitude limit Specified as one of the following strings: 'auto'   'manual' <b>Default:</b> 'auto'
MagLowerLim	Specifies the lower magnitude limit
PhaseUnits	Phase units Specified as one of the following strings: 'deg'   'rad' <b>Default:</b> 'deg'

# bodeoptions

---

Option	Description
PhaseVisible	Phase plot visibility Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'on'
PhaseWrapping	Enables phase wrapping Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'
PhaseMatching	Enables phase matching Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'
PhaseMatchingFreq	Frequency for matching phase
PhaseMatchingValue	The value to which phase responses are matched closely

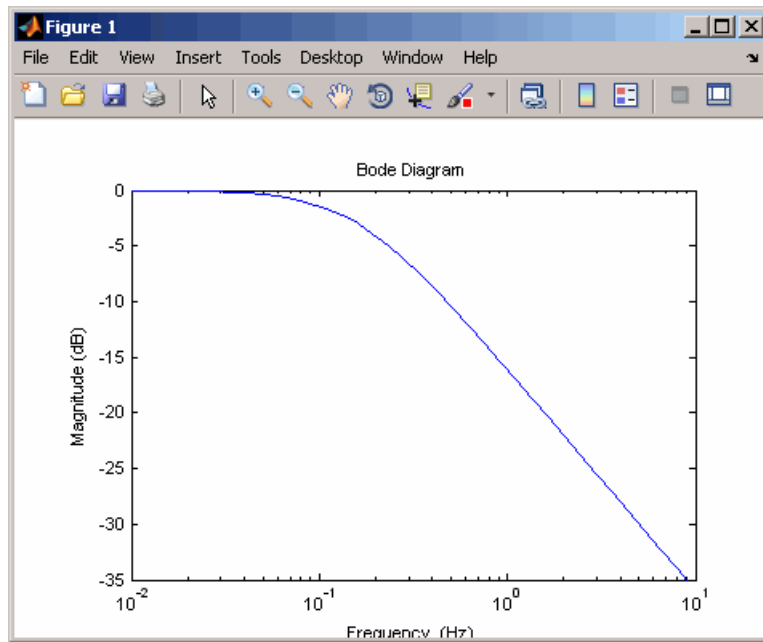
## Examples

In this example, set phase visibility and frequency units in the Bode plot options.

```
P = bodeoptions; % Set phase visibility to off and frequency units to Hz in options
P.PhaseVisible = 'off';
P.FreqUnits = 'Hz'; % Create plot with the options specified by P
h = bodeplot(tf(1,[1,1]),P);
```

The following plot is created, with the phase plot visibility turned off and the frequency units in Hz.





**See Also** `bode` | `bodeplot` | `getoptions` | `setoptions`

# bodeplot

---

**Purpose** 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 “Example 2” on page 1-45 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.

## **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”.

## **Examples**

### **Example 1**

Use the plot handle to change options in a Bode plot.

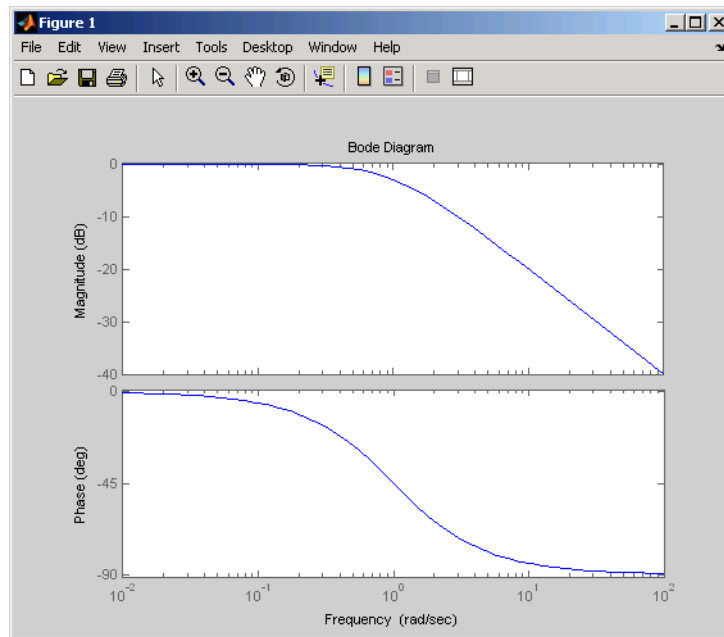
```
sys = rss(5);  
h = bodeplot(sys);  
% Change units to Hz and make phase plot invisible  
setoptions(h,'FreqUnits','Hz','PhaseVisible','off');
```

### **Example 2**

The properties `PhaseMatchingFreq` and `PhaseMatchingValue` are parameters you can use to specify the phase at a specified frequency. For example, enter the following commands.

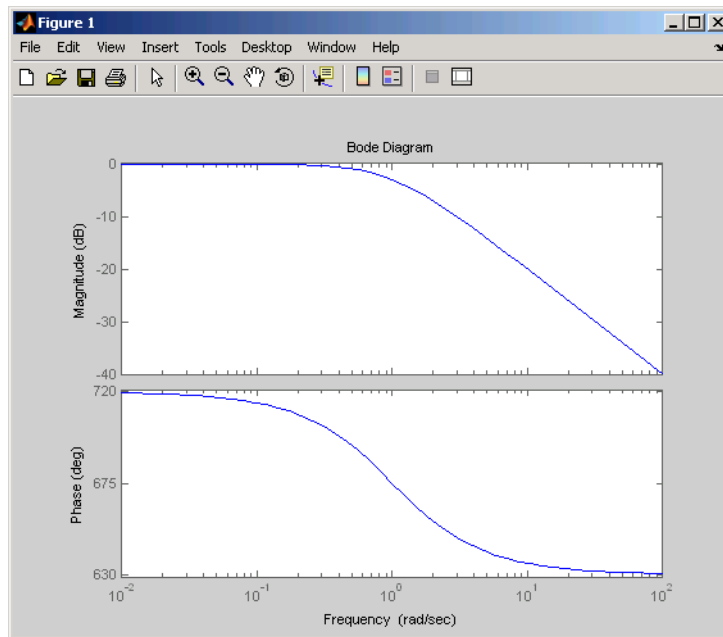
```
sys = tf(1,[1 1]);  
h = bodeplot(sys) % This displays a Bode plot.
```

# bodeplot



Use this code to match a phase of 750 degrees to 1 rad/s.

```
p = getoptions(h);  
p.PhaseMatching = 'on';  
p.PhaseMatchingFreq = 1;  
p.PhaseMatchingValue = 750; % Set the phase to 750 degrees at 1  
    % rad/s.  
setoptions(h,p); % Update the Bode plot.
```



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  $2 \cdot 360 - 45 = 675$ ).

### Example 3

Compare the frequency responses of identified state-space models of order 2 and 6 along with their 2 std confidence regions.

```
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:

# bodeplot

---

```
w = linspace(8,10*pi,256);  
h = bodeplot(sys1,sys2,w);  
setoptions(h, 'PhaseMatching', 'on', 'ConfidenceRegionNumberSD', 2);
```

Use the context menu by right-clicking **Characteristics > Confidence Region** to turn on the confidence region characteristic.

## Example 4

Compare the frequency response of a parametric model, identified from input/output data, to a nonparametric model identified using the same data.

- 1 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 non-parametric identified model. `sys_p` is a parametric identified model.

- 2 Create a Bode plot that includes both systems.

```
opt = bodeoptions; opt.PhaseMatching = 'on';  
bodeplot(sys_np,sys_p,w, opt);
```

## See Also

[bode](#) | [bodeoptions](#) | [getoptions](#) | [setoptions](#)

**Purpose**

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)`.

**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`.

**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 with `FcnType` 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**

String specifying a discretization method:

- `'zoh'` — Zero-order hold (default). Assumes the control inputs are piecewise constant over the sampling period `Ts`.
- `'foh'` — Triangle approximation (modified first-order hold). Assumes the control inputs are piecewise linear over the sampling period `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 the continuous-time transfer function:

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

with input delay  $T_d = 0.35$  second. To discretize this system using the triangle (first-order hold) approximation with sample time  $T_s = 0.1$  second, type

```
H = tf([1 -1], [1 4 5], 'inputdelay', 0.35);
Hd = c2d(H, 0.1, 'foh'); % discretize with FOH method and
                        % 0.1 second sample time
```

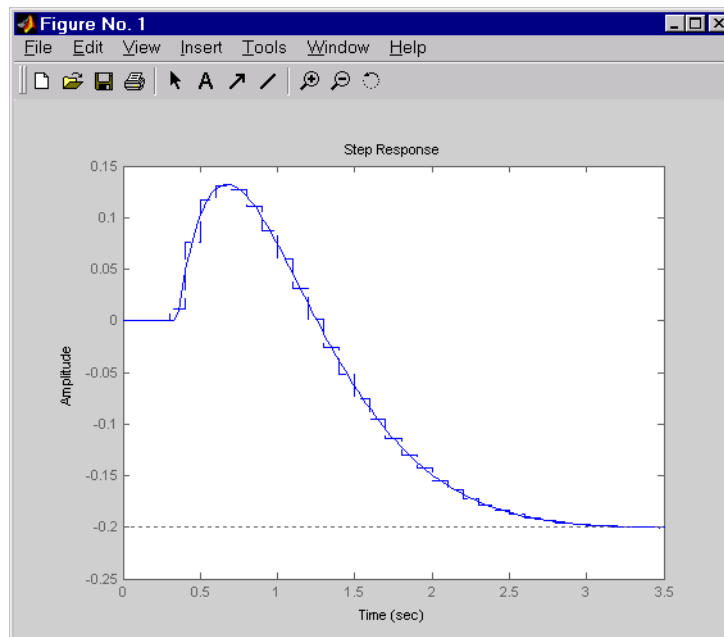
Transfer function:

$$\frac{0.0115 z^3 + 0.0456 z^2 - 0.0562 z - 0.009104}{z^6 - 1.629 z^5 + 0.6703 z^4}$$

Sampling time: 0.1

The next command compares the continuous and discretized step responses.

```
step(H, '-', Hd, '--')
```



Discretize the delayed transfer function

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

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

```
h = tf(10,[1 3 10],'iodelay',0.25); % create transfer function
hd = c2d(h, 0.1) % zoh is the default method
```

These commands produce the discrete-time transfer function

Transfer function:

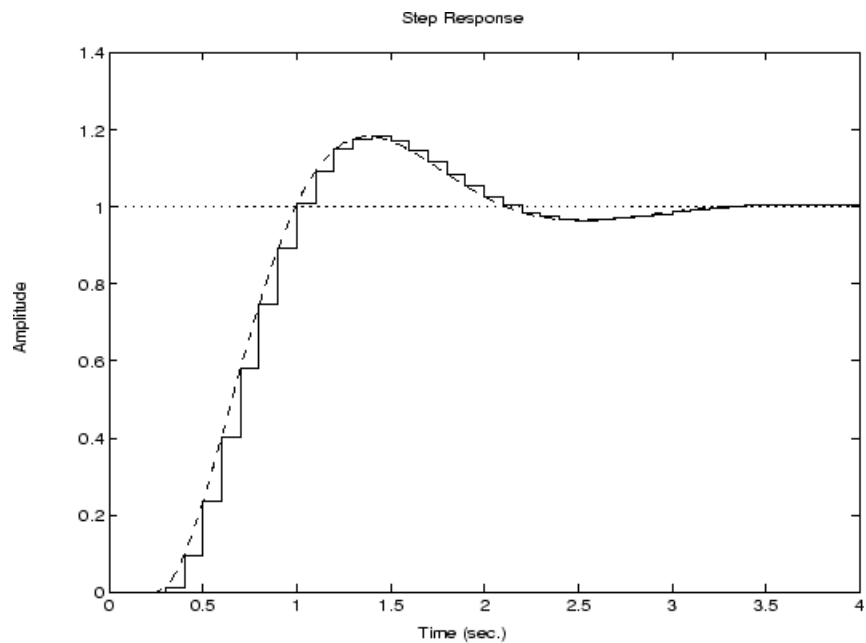
$$z^{(-3)} * \frac{0.01187 z^2 + 0.06408 z + 0.009721}{z^2 - 1.655 z + 0.7408}$$

Sampling time: 0.1

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 and discretized models using

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



Discretize a state-space model with time delay, using a Thiran filter to model fractional delays:

```
sys = ss(tf([1, 2], [1, 4, 2])); % create a state-space model
sys.InputDelay = 2.7           % add input delay
```

This command creates a continuous-time state-space model with two states, as the output shows:

```
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 (listed by channel): 2.7

Continuous-time model.

Use `c2dOptions` to create a set of discretization options, and discretize the model. This example uses the Tustin discretization method.

```

opt = c2dOptions('Method', 'tustin', 'FractDelayApproxOrder', 3);
sysd1 = c2d(sys, 1, opt)    % 1s sampling time

```

These commands yield the result

```

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
```

```
Sampling time: 1
Discrete-time model.
```

The discretized model now contains three additional states  $x_3$ ,  $x_4$ , and  $x_5$  corresponding to a third-order Thiran filter. Since the time delay divided by the sampling time is 2.7, the third-order Thiran filter (`FractDelayApproxOrder = 3`) can approximate the entire time delay.

---

Discretize an identified, continuous-time transfer function and compare its performance against a directly estimated discrete-time model

Estimate a continuous-time transfer function and discretize it.

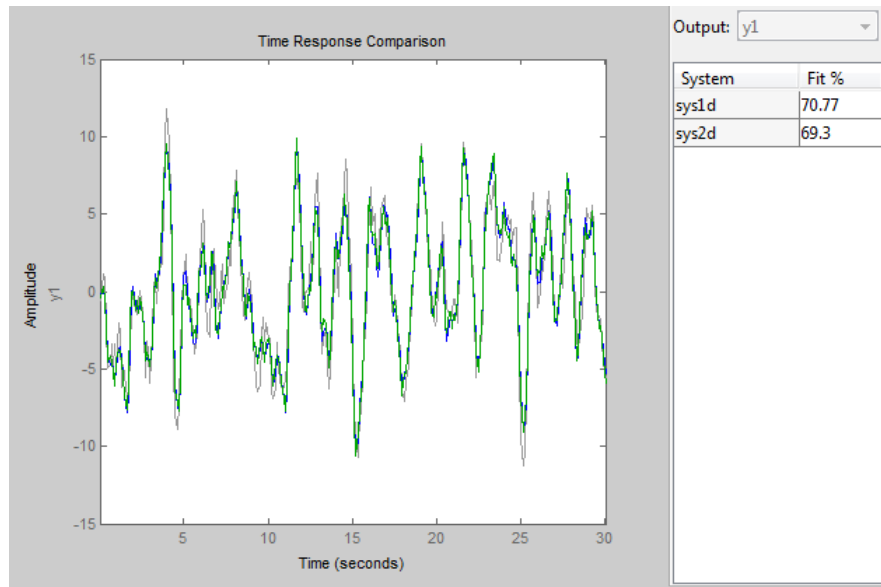
```
load iddata1
sys1c = tfest(z1, 2);
sys1d = c2d(sys1c, 0.1, 'zoh');
```

Estimate a second order discrete-time transfer function.

```
sys2d = tfest(z1, 2, 'Ts', 0.1);
```

Compare the two models.

```
compare(z1, sys1d, sys2d)
```



The two systems are virtually identical.

Discretize an identified state-space model to build a one-step ahead predictor of its response.

```
load iddata2
sysc = ssest(z2, 4);
sysd = c2d(sysc, 0.1, 'zoh');
[A,B,C,D,K] = idssdata(sysd);
Predictor = ss(A-K*C, [K B-K*D], C, [0 D], 0.1);
```

The 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`.

## Algorithms

For information about the algorithms for each `c2d` conversion method, see “Continuous-Discrete Conversion Methods”.

**See Also**

[c2dOptions](#) | [d2c](#) | [d2d](#) | [thiran](#) | [translatecov](#)

**How To**

- “Dynamic System Models”
- “Discretize a Compensator”
- “Continuous-Discrete Conversion Methods”



**Purpose** 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 sampling period $T_s$ .
'foh'	Triangle approximation (modified first-order hold), where <code>c2d</code> assumes the control inputs are piecewise linear over the sampling period $T_s$ . (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, <code>c2d</code> rounds any fractional time delays to the nearest multiple of the sample time. To approximate fractional time delays, use the <code>FractDelayApproxOrder</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 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 identical discretization options.

```
% generate two arbitrary continuous-time state-space models
sys1 = rss(3, 2, 2);
sys2 = rss(4, 4, 1);
```

Use c2dOptions to create a set of discretization options.

```
opt = c2dOptions('Method', 'tustin', 'PrewarpFrequency', 3.4);
```

Then, discretize both models using the option set.

```
dsys1 = c2d(sys1, 0.1, opt); % 0.1s sampling time
dsys2 = c2d(sys2, 0.2, opt); % 0.2s sampling time
```

The c2dOptions option set does not include the sampling time Ts. You can use the same discretization options to discretize systems using a different sampling time.

## 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

**Purpose** 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**  
String specifying the type of canonical form of `csys`. `type` can take one of the two 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.

**Definitions**

**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.

## Examples

This example uses `canon` to convert a system having doubled poles and clusters of close poles to modal canonical form.

Consider the system  $G$  having the following transfer function:

$$G(s) = 100 \frac{(s-1)(s+1)}{s(s+10)(s+10.0001)(s-(1+i))^2(s-(1-i))^2}$$

To create a linear model of this system and convert it to modal canonical form, enter:

```
G = zpke([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. To see this, enter the following command:

```
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

```

To separate the two poles near  $s = -10$ , you can increase the value of `condt`. For example:

```
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      0
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 maximum condition number of  $A$  is that the  $B$  matrix includes some large values.

```
format shortE
Gc2.B
```

ans =

```

3.2000e-001
-6.5691e-003
5.4046e-002
-1.9502e-001

```

```
1.0637e+000  
3.2533e+005  
3.2533e+005
```

---

This example estimates a state-space model that is freely parameterized and convert to companion form after estimation.

```
load icEngine.mat  
z = iddata(y,u,0.04);  
FreeModel = n4sid(z,4,'InputDelay',2);  
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)
```

the bounds are identical.

## 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`



**Purpose** Continuous-time algebraic Riccati equation solution

**Syntax**  
 $[X,L,G] = \text{care}(A,B,Q)$   
 $[X,L,G] = \text{care}(A,B,Q,R,S,E)$   
 $[X,L,G,\text{report}] = \text{care}(A,B,Q,\dots)$   
 $[X1,X2,D,L] = \text{care}(A,B,Q,\dots, \text{'factor'})$

**Description**  $[X,L,G] = \text{care}(A,B,Q)$  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,  $G = R^{-1}B^T XE$ .

$[X,L,G] = \text{care}(A,B,Q,R,S,E)$  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

$$L = \text{eig}(A - B*G, E)$$

$[X,L,G,\text{report}] = \text{care}(A,B,Q,\dots)$  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.

$[X1,X2,D,L] = \text{care}(A,B,Q,\dots, \text{'factor'})$  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 `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

$$A^T X + XA + X(\gamma^{-2} B_1 B_1^T - B_2 B_2^T) X + C^T C = 0$$

rewrite it in the care format as

$$A^T X + XA - X \underbrace{[B_1, B_2]}_B \underbrace{\begin{bmatrix} -\gamma^2 I & 0 \\ 0 & I \end{bmatrix}}_R^{-1} \underbrace{\begin{bmatrix} B_1^T \\ B_2^T \end{bmatrix}}_C X + C^T C = 0$$

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)
```

## 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.

## 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

$$\begin{bmatrix} Q & S \\ S^T & R \end{bmatrix} > 0$$

## 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

<b>Purpose</b>	Change frequency units of frequency-response data model
<b>Syntax</b>	<code>sys_new = chgFreqUnit(sys,newfrequnits)</code>
<b>Description</b>	<code>sys_new = chgFreqUnit(sys,newfrequnits)</code> changes units of the frequency points in <code>sys</code> to <code>newfrequnits</code> . Both <code>Frequency</code> and <code>FrequencyUnit</code> properties of <code>sys</code> adjust so that the frequency responses of <code>sys</code> and <code>sys_new</code> match.
<b>Tips</b>	<ul style="list-style-type: none"><li>• Use <code>chgFreqUnit</code> to change the units of frequency points without modifying system behavior.</li></ul>
<b>Input Arguments</b>	<p><b>sys</b> Frequency-response data (<code>frd</code>, <code>idfrd</code>, or <code>genfrd</code>) model</p> <p><b>newfrequnits</b> New units of frequency points, specified as one of the following strings:</p> <ul style="list-style-type: none"><li>• 'rad/TimeUnit'</li><li>• 'cycles/TimeUnit'</li><li>• 'rad/s'</li><li>• 'Hz'</li><li>• 'kHz'</li><li>• 'MHz'</li><li>• 'GHz'</li><li>• 'rpm'</li></ul> <p><code>rad/TimeUnit</code> and <code>cycles/TimeUnit</code> express frequency units relative to the system time units specified in the <code>TimeUnit</code> property.</p> <p><b>Default:</b> 'rad/TimeUnit'</p>

# chgFreqUnit

---

## 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

This example shows how to change units of the frequency points in a frequency-response data model.

- 1 Create a frequency-response data model.

```
load 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.

- 2 Change the frequency units.

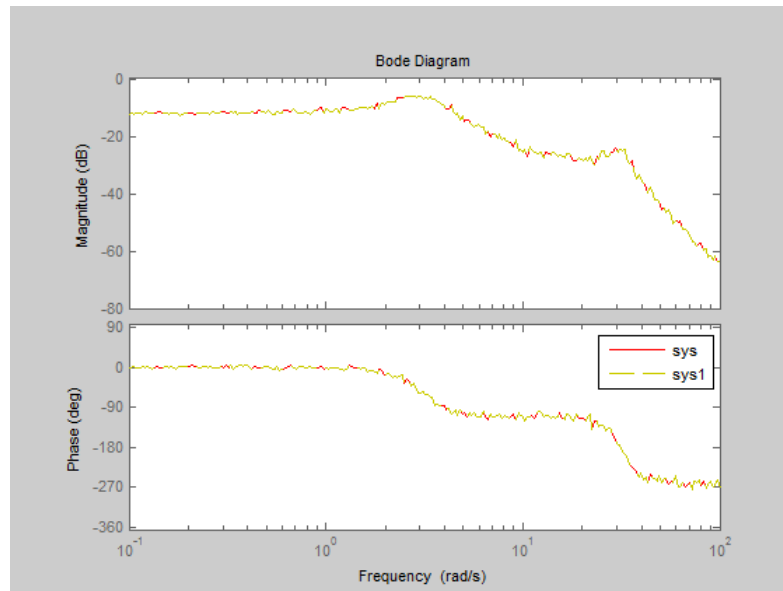
```
sys1 = chgFreqUnit(sys, 'rpm');
```

The `FrequencyUnit` property of `sys1` is `rpm`.

- 3 Compare the Bode responses of `sys` and `sys1`.

```
bode(sys, 'r', sys1, 'y--');  
legend('sys', 'sys1')
```

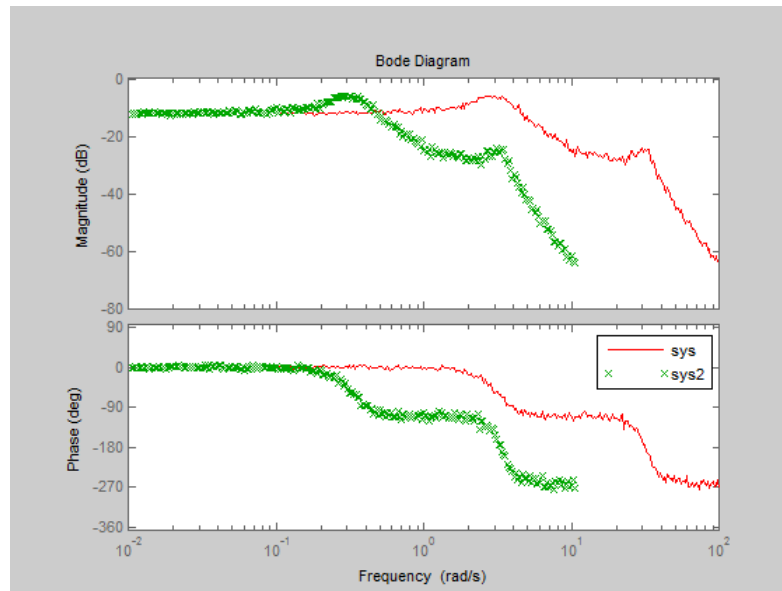
The magnitude and phase of `sys` and `sys1` match.



- 4** (Optional) Change the `FrequencyUnit` property of `sys` to compare the Bode response with the original system.

```
sys2=sys;
sys2.FrequencyUnit = 'rpm';
bode(sys, 'r', sys2, 'gx');
legend('sys', 'sys2');
```

Changing the `FrequencyUnit` property changes the original system. Therefore, the Bode responses of `sys` and `sys2` do not match. For example, the original corner frequency at 2 rad/s changes to 2 rpm (or 0.2 rad/s).



## See Also

`chgTimeUnit` | `frd`

## Tutorials

- “Specify Frequency Units of Frequency-Response Data Model”<sup>1</sup>

1.



<b>Purpose</b>	Change time units of dynamic system
<b>Syntax</b>	<code>sys_new = chgTimeUnit(sys,newtimeunits)</code>
<b>Description</b>	<code>sys_new = chgTimeUnit(sys,newtimeunits)</code> changes the time units of <code>sys</code> to <code>newtimeunits</code> . The time- and frequency-domain characteristics of <code>sys</code> and <code>sys_new</code> match.
<b>Tips</b>	<ul style="list-style-type: none"><li>• Use <code>chgTimeUnit</code> to change the time units without modifying system behavior.</li></ul>
<b>Input Arguments</b>	<p><b>sys</b> Dynamic system model</p> <p><b>newtimeunits</b> New time units, specified as one of the following strings:</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>

# chgTimeUnit

---

## 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

This example shows how to change the time units of a transfer function model.

- 1 Create a transfer function model.

```
num = [4 2];  
den = [1 3 10];  
sys = tf(num,den);
```

The default time units of **sys** is seconds.

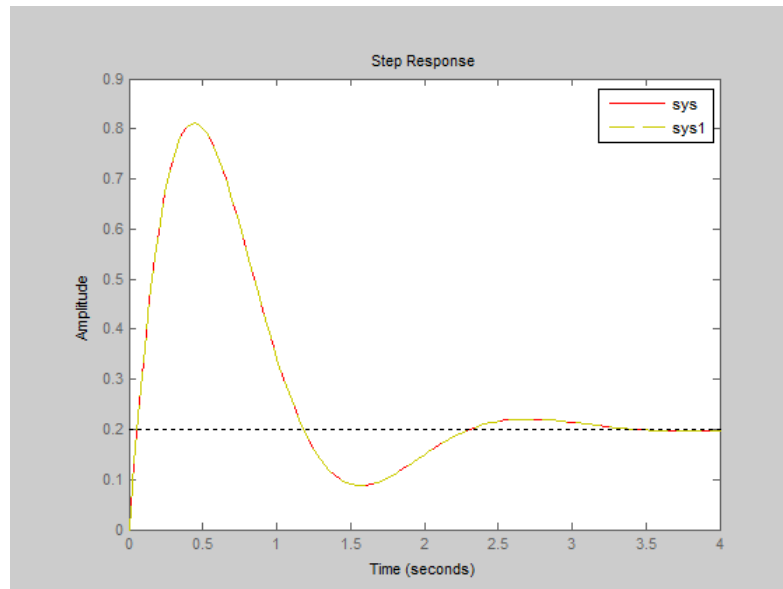
- 2 Change the time units.

```
sys1 = chgTimeUnit(sys,'minutes');
```

The `TimeUnit` property of **sys1** is milliseconds.

- 3 Compare the step responses of **sys** and **sys1**.

```
step(sys,'r',sys1,'y--');  
legend('sys','sys1');
```

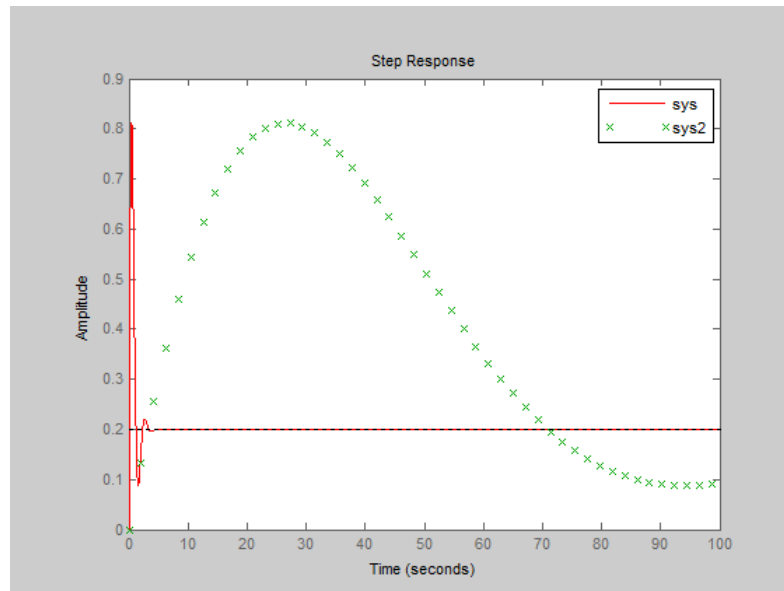


The step responses of `sys` and `sys1` match.

- 4 (Optional) Change the `TimeUnit` property of `sys`, and compare the step response with the original system.

```
sys2=sys;
sys2.TimeUnit = 'minutes';
step(sys, 'r', sys2, 'gx');
legend('sys', 'sys2');
```

Changing the `TimeUnit` property changes the original system. Therefore, 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.



## See Also

[chgFreqUnit](#) | [tf](#) | [zpk](#) | [ss](#) | [frd](#) | [pid](#)

## Tutorials

- “Specify Model Time Units”

**Purpose** 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`

# connect

---

**Purpose** Block diagram interconnections of dynamic systems

**Syntax**

```
sysc = connect(sys1,...,sysN,inputs,outputs)
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 you create using `sumblek`. The `connect` command interconnects the block diagram elements by matching the input and output signals 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(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 is not recommended.

`sysc = connect( __ ,opts)` builds the interconnected model using additional options. You can use `opts` with the input arguments of either of the previous syntaxes.

## Input Arguments

### **sys1,...,sysN**

Dynamic system models corresponding to the elements of your block diagram. For example, the elements of your block diagram can include one or more `tf` or `ss` model representing plant dynamics. Block diagram elements can also include a `pid` or `ltiblock.pid` model representing a controller. You can also include one or more summing junction you create using `sumblek`. Provide multiple arguments `sys1,...,sysN` to represent all of the block diagram elements and summing junctions.

**inputs**

For name-based interconnection, a string or cell array of strings specifying the inputs of the aggregate model `sysc`. The strings 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 string or cell array of strings specifying the outputs of the aggregate model `sysc`. The strings in `outputs` must correspond to entries in the `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 specifying 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.

# connect

## Output Arguments

### **opts**

Additional options for interconnection, specified as an options set you create with `connectOptions`.

### **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 an `ltiblock.pid` Control Design Block returns a `genss`.
- Interconnecting any model with frequency-response data model returns a frequency response data model.

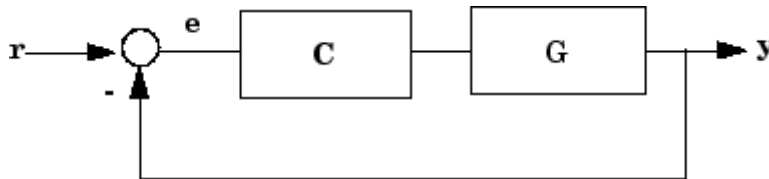
By default, `connect` automatically discards states that do not contribute to the I/O transfer function from the specified inputs to the specified outputs of the interconnected model. To retain the unconnected states, set the `Simplify` option of `connectOptions` to `false`. For example:

```
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 =
```

# connect

---

```
'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
```

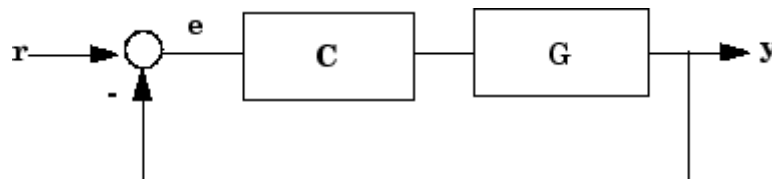
```
ans =
```

```
'r(1)'  
'r(2)'
```

---

## 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 = zpke([], [-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`. `1` specifies that the external input connects to  $u(1)$ . The last argument, `2`, specifies that the external output connects from  $w(2)$ .

## See Also

`sumblk` | `append` | `feedback` | `parallel` | `series` | `lft` | `connectOptions`

## How To

- “Multi-Loop Control System”
- “MIMO Control System”
- “MIMO Feedback Loop”

# connectOptions

---

<b>Purpose</b>	Options for the connect command
<b>Syntax</b>	<code>opt = connectOptions</code> <code>opt = connectOptions(Name,Value)</code>
<b>Description</b>	<code>opt = connectOptions</code> returns the default options for connect.  <code>opt = connectOptions(Name,Value)</code> returns an options set with the options specified by one or more <code>Name,Value</code> pair arguments.
<b>Input Arguments</b>	<b>Name-Value Pair Arguments</b> Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> .  <b>Example:</b> <code>'Simplify',false</code>  <b>'Simplify' - Automatic elimination of unconnected states</b> <code>true (default)   false</code> Automatic elimination of unconnected states, specified as either <code>true</code> or <code>false</code> . <ul style="list-style-type: none"><li>• <code>true</code> — <code>connect</code> 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.</li><li>• <code>false</code> — <code>connect</code> 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.</li></ul> <b>Data Types</b> 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.

## 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);
```

**See Also** connect

**Purpose** Output and state covariance of system driven by white noise

**Syntax**  
 $P = \text{covar}(\text{sys}, W)$   
 $[P, Q] = \text{covar}(\text{sys}, W)$

**Description** covar calculates the stationary covariance of the output  $y$  of an LTI model  $\text{sys}$  driven by Gaussian white noise inputs  $w$ . This function handles both continuous- and discrete-time cases.

$P = \text{covar}(\text{sys}, W)$  returns the steady-state output response covariance

$$P = E(yy^T)$$

given the noise intensity

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

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

$[P, Q] = \text{covar}(\text{sys}, W)$  also returns the steady-state state covariance

$$Q = E(xx^T)$$

when  $\text{sys}$  is a state-space model (otherwise  $Q$  is set to  $[\ ]$ ).

When applied to an  $N$ -dimensional LTI array  $\text{sys}$ , covar returns multidimensional arrays  $P$ ,  $Q$  such that

$P(:, :, i1, \dots, iN)$  and  $Q(:, :, i1, \dots, iN)$  are the covariance matrices for the model  $\text{sys}(:, :, i1, \dots, iN)$ .

**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.

## 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.

### References

[1] Bryson, A.E. and Y.C. Ho, *Applied Optimal Control*, Hemisphere Publishing, 1975, pp. 458-459.

### See Also

dlyap | lyap



**Purpose** Controllability matrix

**Syntax** `Co = ctrb(sys)`

**Description** `ctrb` computes the controllability matrix for state-space systems. For an  $n$ -by- $n$  matrix  $A$  and an  $n$ -by- $m$  matrix  $B$ , `ctrb(A,B)` returns the controllability matrix

$$Co = [B \ AB \ A^2B \ \dots \ A^{n-1}B] \quad (1-1)$$

where  $Co$  has  $n$  rows and  $nm$  columns.

`Co = ctrb(sys)` calculates the controllability matrix of the state-space LTI object `sys`. This syntax is equivalent to executing

```
Co = ctrb(sys.A,sys.B)
```

The system is controllable if  $Co$  has full rank  $n$ .

## Examples

Check if the system with the following data

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

```
B =
     1    -1
     1    -1
```

is controllable. Type

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

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

These commands produce the following result.

unco =  
1

## 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.

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

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`

**Purpose**

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,  $Abar$ ,  $Bbar$ , and  $Cbar$ , described above.  $T$  is the similarity transformation matrix and  $k$  is a vector of length  $n$ , where  $n$  is the order of the system represented by  $A$ . Each entry of  $k$  represents the number of controllable states factored out during each step of the transformation matrix calculation. The number of nonzero elements in  $k$  indicates how many iterations were necessary to calculate  $T$ , and  $\text{sum}(k)$  is the number of states in  $A_c$ , the controllable portion of  $Abar$ .

`ctrbf(A,B,C,tol)` uses the tolerance `tol` when calculating the controllable/uncontrollable subspaces. When the tolerance is not specified, it defaults to  $10 * n * \text{norm}(A, 1) * \text{eps}$ .

**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  $Abar$  shows an uncontrollable mode located at  $-3$  and a controllable mode located at  $2$ .

## 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

# ctrlpref

---

**Purpose** 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** `sisotool` | `ltiview`

**Purpose**

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`.

**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.

**Input Arguments****sysd**

Discrete-time dynamic system model

You cannot directly use an `idgrey` model with `FcnType='d'` with `d2c`. Convert the model into `idss` form first.

**method**

String specifying a discrete-to-continuous time conversion method:

- `'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  $T_s*\lambda$ .
- 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

### Example 1

Consider the discrete-time model with transfer function

$$H(z) = \frac{z-1}{z^2+z+0.3}$$

and sample time  $T_s = 0.1$  s. You can derive a continuous-time zero-order-hold equivalent model by typing

```
Hc = d2c(H)
```

Discretizing the resulting model `Hc` with the default zero-order hold method and sampling time  $T_s = 0.1$ s returns the original discrete model  $H(z)$ :

```
c2d(Hc,0.1)
```

To use the Tustin approximation instead of zero-order hold, type

```
Hc = d2c(H,'tustin')
```

As with zero-order hold, the inverse discretization operation

```
c2d(Hc,0.1,'tustin')
```

gives back the original  $H(z)$ .

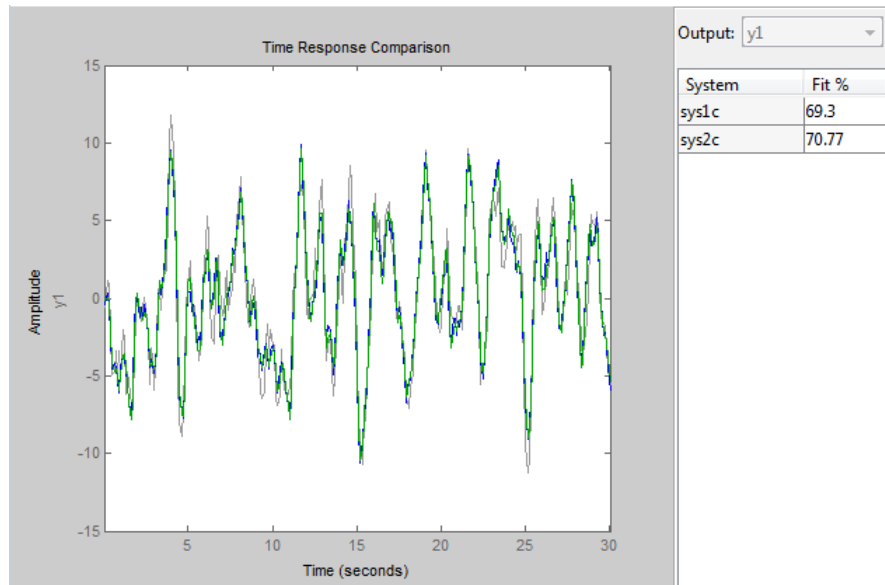
## Example 2

Convert an identified transfer function and compare its performance against a directly estimated continuous-time model.

```
load iddata1
sys1d = tfest(z1, 2, 'Ts', 0.1);
sys1c = d2c(sys1d, 'zoh');
sys2c = tfest(z1, 2);
```

```
compare(z1, sys1c, sys2c)
```

The two systems are virtually identical.



## Example 3

Analyze the effect of parameter uncertainty on frequency response across d2c operation on an identified model.

```
load iddata1
```

```
sysd = tfest(z1, 2, 'Ts', 0.1);  
sysc = d2c(sysd, 'zoh');
```

`sys1c` has no covariance information. Regenerate it using a zero iteration update with the same estimation command and estimation data:

```
opt = tfestOptions;  
opt.SearchOption.MaxIter = 0;  
sys1c = tfest(z1, sysc, opt);  
  
h = bodeplot(sysd, sysc);  
showConfidence(h)
```

The uncertainties of `sysc` and `sysd` are comparable up to the Nyquist frequency. However, `sysc` 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 `translatecov` which is a Gauss-approximation formula based translation of covariance across model type conversion operations.

## 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.

## 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)
```

Sampling time: 0.1

---

This discrete model coincides with  $H(z)$  after canceling the pole/zero pair at  $z = -0.5$ .

## 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

d2cOptions | c2d | d2d | translatecov | logm

# d2cOptions

---

**Purpose** 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:

- 'zoh' Zero-order hold, where `d2c` assumes the control inputs are piecewise constant over the sampling period `Ts`.
- '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. By default, `d2c` converts with no prewarp. To include prewarp, use the `PrewarpFrequency` option.
- 'matched' 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 sampling 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`

**Purpose** 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`.

- 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`.

## 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 = zpkm(0.7,0.5,1,0.1)
```



```
H2 = d2d(H,0.05)
Zero/pole/gain:
(z-0.8243)
-----
(z-0.7071)
```

Sampling time: 0.05

The inverse resampling operation, performed by typing `d2d(H2,0.1)`, yields back the initial model  $H(z)$ .

```
Zero/pole/gain:
(z-0.7)
-----
(z-0.5)
```

Sampling time: 0.1

## Example 2

Suppose you estimate 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)
```

## See Also

`d2dOptions` | `c2d` | `d2c` | `upsample` | `translatecov`

# d2dOptions

---

**Purpose** 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 sampling period $T_s$ .                                 |
| '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 sampling time
```

To resample to a different sampling 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`

# damp

---

**Purpose** Natural frequency; damping ratio

**Syntax**  
`damp(sys)`  
`[Wn,zeta] = damp(sys)`  
`[Wn,zeta,P] = damp(sys)`

**Description** `damp(sys)` calculates the damping ratio (also called damping factor) and natural frequency of the poles of the linear model `sys`. When invoked without output arguments, `damp` displays a table of the eigenvalues of `sys`, along with the corresponding damping ratios and natural frequencies. For discrete-time `sys`, the table includes the magnitude of each pole and the damping ratio and frequencies of equivalent continuous-time poles (see “Algorithms” on page 1-112). Frequencies are expressed in units of the reciprocal of the `TimeUnit` property of `sys`.

`[Wn,zeta] = damp(sys)` returns vectors `Wn` and `zeta` containing the natural frequencies  $\omega_n$  and damping ratios  $\zeta$  of the poles of `sys`.

`[Wn,zeta,P] = damp(sys)` also returns a vector `P` containing 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 sampling time, `Wn` contains the natural frequencies of the equivalent continuous-time poles (see “Algorithms” on page 1-112). If `sys` has unspecified sampling time (`Ts = -1`), `Wn` is empty.

**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 sampling time, `zeta` contains the damping ratios of the equivalent continuous-time poles (see “Algorithms” on page 1-112). If `sys` has unspecified sampling time (`Ts = -1`), `zeta` is empty.

### **P**

Vector containing the poles of `sys`, in order of increasing natural frequency. `P` is the same as the output of `pole(sys)`, up to ordering.

## **Examples**

### **Natural Frequency, Damping Ratio, and Poles of a Continuous-Time Transfer Function**

Compute the natural frequency, damping ratio and poles of a continuous-time transfer function.

Create the 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, and poles of `H`.

```
damp(H)
```

Eigenvalue	Damping	Frequency
-1.00e+000 + 1.41e+000i	5.77e-001	1.73e+000
-1.00e+000 - 1.41e+000i	5.77e-001	1.73e+000

(Frequencies expressed in rad/seconds)

The system eigenvalues are the pole locations.

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

---

## Natural Frequency, Damping Ratio and Poles of a Discrete-Time Transfer Function

Compute the natural frequency, damping ratio and poles of 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)
```

Eigenvalue	Magnitude	Damping	Frequency
-3.02e-001 + 8.06e-001i	8.61e-001	7.74e-002	1.93e+002
-3.02e-001 - 8.06e-001i	8.61e-001	7.74e-002	1.93e+002
-5.40e+000	5.40e+000	-4.73e-001	3.57e+002

(Frequencies expressed in rad/seconds)

The system eigenvalues are the pole locations.

Obtain vectors containing the natural frequencies and damping ratios of the poles.

```
[Wn,zeta] = damp(H);
```

## Algorithms

For a continuous-time linear system  $G(s)$ , the natural frequency  $\omega_n$  of a pole at  $s = R$  is given by:

$$\omega_n = |R|.$$

For a discrete-time linear system  $G(z)$  with a pole at  $z = R$ , `damp` returns the natural frequencies and damping ratios of equivalent continuous time poles. The locations of the equivalent poles are given by

$$s = \frac{\ln(R)}{T_s}$$

$T_s$  is the sampling time.

The natural frequency, time constant, and damping ratio of the system poles are defined as follows.

	Continuous Time	Discrete Time
<b>Location of Pole</b>	Real or complex eigenvalue at $s = R$	Real or complex eigenvalue at $z = R$
<b>Natural Frequency</b>	$\omega_n = \text{abs}(R)$	$\omega_n = \text{abs}(\log(R)) / T_s$
<b>Damping Ratio</b>	$\zeta = -\cos(\text{angle}(R))$	$\zeta = -\cos(\text{angle}(\log(R)))$
<b>Time Constant</b>	<ul style="list-style-type: none"> <li>• <math>\tau = 1 / (\zeta * \omega_n)</math> for <math>\zeta &gt; 0</math></li> <li>• Inf otherwise</li> </ul>	<ul style="list-style-type: none"> <li>• <math>\tau = 1 / (\zeta * \omega_n)</math> for <math>\zeta &gt; 0</math></li> <li>• Inf otherwise</li> </ul>

**See Also**

`eig` | `esort` | `dsort` | `pole` | `pzmap` | `zero`

**Purpose** 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.

## 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.

## 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

$$\begin{bmatrix} Q & S \\ S^T & R \end{bmatrix} > 0$$

## 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`

# db2mag

---

**Purpose** 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`

**Purpose** Low-frequency (DC) gain of LTI system

**Syntax** `k = dcgain(sys)`

**Description** `k = dcgain(sys)` computes the DC gain `k` of the LTI model `sys`.

**Continuous Time**

The continuous-time DC gain is the transfer function value at the frequency  $s = 0$ . For state-space models with matrices  $(A, B, C, D)$ , this value is

$$K = D - CA^{-1}B$$

**Discrete Time**

The discrete-time DC gain is the transfer function value at  $z = 1$ . For state-space models with matrices  $(A, B, C, D)$ , this value is

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

**Tips** The DC gain is infinite for systems with integrators.

**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:

# dcgain

---

```
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`.

## See Also

[evalfr](#) | [norm](#)

## **Purpose**

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.

---

# delays

---

**Purpose** 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 sampling period `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 = delays(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

**Purpose** 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  $e = \text{eig}(A-B*K)$ . 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

# dlyap

---

<b>Purpose</b>	Solve discrete-time Lyapunov equations
<b>Syntax</b>	$X = \text{dlyap}(A, Q)$ $X = \text{dlyap}(A, B, C)$ $X = \text{dlyap}(A, Q, [], E)$
<b>Description</b>	<p><math>X = \text{dlyap}(A, Q)</math> solves the discrete-time Lyapunov equation <math>AXA^T - X + Q = 0</math>,</p> <p>where <math>A</math> and <math>Q</math> are <math>n</math>-by-<math>n</math> matrices.</p> <p>The solution <math>X</math> is symmetric when <math>Q</math> is symmetric, and positive definite when <math>Q</math> is positive definite and <math>A</math> has all its eigenvalues inside the unit disk.</p> <p><math>X = \text{dlyap}(A, B, C)</math> solves the Sylvester equation <math>AXB - X + C = 0</math>,</p> <p>where <math>A</math>, <math>B</math>, and <math>C</math> must have compatible dimensions but need not be square.</p> <p><math>X = \text{dlyap}(A, Q, [], E)</math> solves the generalized discrete-time Lyapunov equation <math>AXA^T - EXE^T + Q = 0</math>,</p> <p>where <math>Q</math> is a symmetric matrix. The empty square brackets, <math>[\ ]</math>, are mandatory. If you place any values inside them, the function will error out.</p>
<b>Algorithms</b>	dlyap uses SLICOT routines SB03MD and SG03AD for Lyapunov equations and SB04QD (SLICOT) for Sylvester equations.
<b>Diagnostics</b>	<p>The discrete-time Lyapunov equation has a (unique) solution if the eigenvalues <math>a_1, a_2, \dots, a_N</math> of <math>A</math> satisfy <math>a_i a_j \neq 1</math> for all <math>(i, j)</math>.</p> <p>If this condition is violated, dlyap produces the error message</p> <p>Solution does not exist or is not unique.</p>
<b>References</b>	[1] Barraud, A.Y., "A numerical algorithm to solve $A X A - X = Q$ ," <i>IEEE Trans. Auto. Contr.</i> , AC-22, pp. 883-885, 1977.

- [2] Bartels, R.H. and G.W. Stewart, "Solution of the Matrix Equation  $AX + XB = C$ ," *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  $AX + XB = C$ ," *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

# dlyapchol

---

**Purpose** Square-root solver for discrete-time Lyapunov equations

**Syntax**  
 $R = \text{dlyapchol}(A,B)$   
 $X = \text{dlyapchol}(A,B,E)$

**Description**  $R = \text{dlyapchol}(A,B)$  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.

$X = \text{dlyapchol}(A,B,E)$  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.

**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`

**Purpose** 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 sampling 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
```

Sampling time: unspecified  
Discrete-time model.

## See Also

[rss](#) | [tf](#) | [zpk](#)

<b>Purpose</b>	Sort discrete-time poles by magnitude
<b>Syntax</b>	<pre>dsort [s,ndx] = dsort(p)</pre>
<b>Description</b>	<p><code>dsort</code> sorts the discrete-time poles contained in the vector <code>p</code> in descending order by magnitude. Unstable poles appear first.</p> <p>When called with one lefthand argument, <code>dsort</code> returns the sorted poles in <code>s</code>.</p> <p><code>[s,ndx] = dsort(p)</code> also returns the vector <code>ndx</code> containing the indices used in the sort.</p>
<b>Examples</b>	<p>Sort the following discrete poles.</p> <pre>p = -0.2410 + 0.5573i -0.2410 - 0.5573i 0.1503 -0.0972 -0.2590</pre> <pre>s = dsort(p)</pre> <pre>s = -0.2410 + 0.5573i -0.2410 - 0.5573i -0.2590 0.1503 -0.0972</pre>
<b>Limitations</b>	The poles in the vector <code>p</code> must appear in complex conjugate pairs.
<b>See Also</b>	<code>eig</code>   <code>esort</code>   <code>sort</code>   <code>pole</code>   <code>pzmap</code>   <code>zero</code>

**Purpose** 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  $D = \mathbf{0}$ , 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

# dssdata

---

**Purpose** 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`

**Purpose** Sort continuous-time poles by real part

**Syntax** `s = esort(p)`  
`[s,ndx] = esort(p)`

**Description** `esort` sorts the continuous-time poles contained in the vector `p` by real part. Unstable eigenvalues appear first and the remaining poles are ordered by decreasing real parts.

When called with one left-hand argument, `s = esort(p)` returns the sorted eigenvalues in `s`.

`[s,ndx] = esort(p)` returns the additional argument `ndx`, a vector containing the indices used in the sort.

**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`

# estim

---

**Purpose** Form state estimator given estimator gain

**Syntax**  
`est = estim(sys,L)`  
`est = estim(sys,L,sensors,known)`

**Description** `est = estim(sys,L)` produces a state/output estimator `est` given the plant state-space model `sys` and the estimator gain `L`. All inputs  $w$  of `sys` are assumed stochastic (process and/or measurement noise), and all outputs  $y$  are measured. The estimator `est` is returned in state-space form (SS object).

For a continuous-time plant `sys` with equations

$$\dot{x} = Ax + Bw$$

$$y = Cx + Dw$$

`estim` 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:

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

$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} \hat{x}$$

For a discrete-time plant `sys` with the following equations:

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

$$y[n] = Cx[n] + Dw[n]$$

`estim` uses estimator equations similar to those for continuous-time to generate a plant output estimate  $y[n | n-1]$  and a state estimate  $x[n | n-1]$ , 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  $z$ .

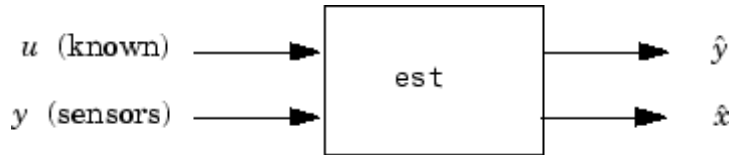
$$\dot{x} = Ax + B_1w + B_2u$$

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

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.

$$\dot{\hat{x}} = A\hat{x} + B_2u + L(y - C_2\hat{x} - D_{22}u)$$

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



**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.

**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];
```

## estim

---

```
known = [1,4,3];  
est = estim(sys,L,sensors,known)
```

See the function `kalman` for direct Kalman estimator design.

### See Also

`kalman` | `place` | `reg` | `kalmd` | `lqgreg` | `ss` | `ssest` | `predict`

**Purpose** 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  $\omega = 0.1$  rad/s, type:

```
sys = idtf(1,[1 2 1]);
```

# evalfr

---

```
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

`bode` | `freqresp` | `sigma`



**Purpose** 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`

# fcats

---

**Purpose** 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** `fdel` | `fselect` | `interp` | `frd`

<b>Purpose</b>	Delete specified data from frequency response data (FRD) models								
<b>Syntax</b>	<code>sysout = fdel(sys, freq)</code>								
<b>Description</b>	<code>sysout = fdel(sys, freq)</code> removes from the frd model <code>sys</code> the data nearest to the frequency values specified in the vector <code>freq</code> .								
<b>Tips</b>	<ul style="list-style-type: none"> <li>• Use <code>fdel</code> to remove unwanted data (for example, outlier points) at specified frequencies.</li> <li>• Use <code>fdel</code> to remove data at intersecting frequencies from <code>frd</code> models before merging them with <code>fcats</code>. <code>fcats</code> produces an error when you attempt to merge <code>frd</code> models that have intersecting frequency data.</li> <li>• To remove data from an <code>frd</code> model within a range of frequencies, use <code>fselect</code>.</li> </ul>								
<b>Input Arguments</b>	<p><b>sys</b> frd model.</p> <p><b>freq</b> Vector of frequency values.</p>								
<b>Output Arguments</b>	<p><b>sysout</b> frd model containing the data remaining in <code>sys</code> after removing the frequency points closest to the entries of <code>freq</code>.</p>								
<b>Examples</b>	<p>Remove selected data from a frd model. In this example, first obtain an frd model:</p> <pre>sys = frd(tf([1],[1 1]), logspace(0,1,10))</pre> <table> <thead> <tr> <th>Frequency (rad/s)</th> <th>Response</th> </tr> <tr> <th>-----</th> <th>-----</th> </tr> </thead> <tbody> <tr> <td>1.0000</td> <td>0.5000 - 0.5000i</td> </tr> <tr> <td>1.2915</td> <td>0.3748 - 0.4841i</td> </tr> </tbody> </table>	Frequency (rad/s)	Response	-----	-----	1.0000	0.5000 - 0.5000i	1.2915	0.3748 - 0.4841i
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.

## See Also

`fcats` | `fselect` | `frd`

**Purpose**

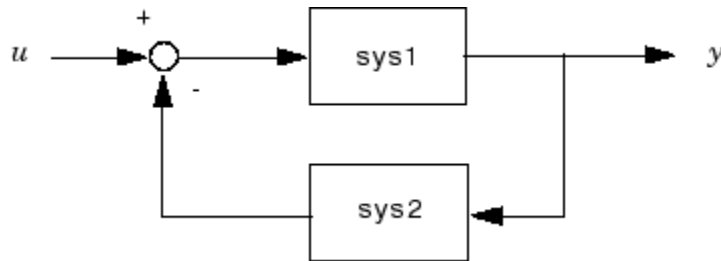
Feedback connection of two models

**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)
```

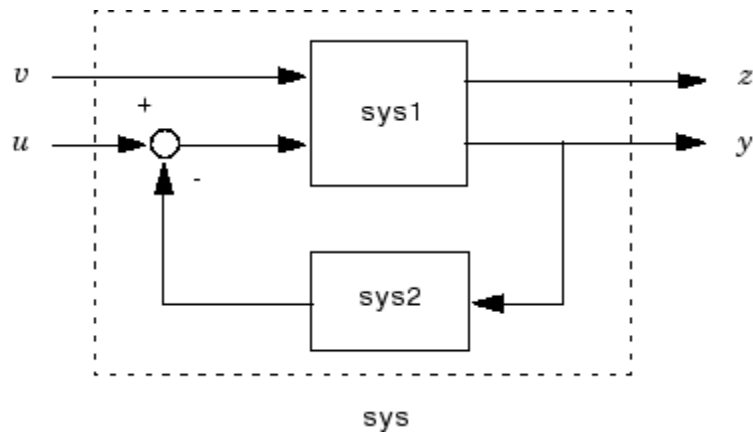
By default, `feedback(sys1,sys2)` assumes negative feedback and is equivalent to `feedback(sys1,sys2,-1)`.

Finally,

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

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

# feedback



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

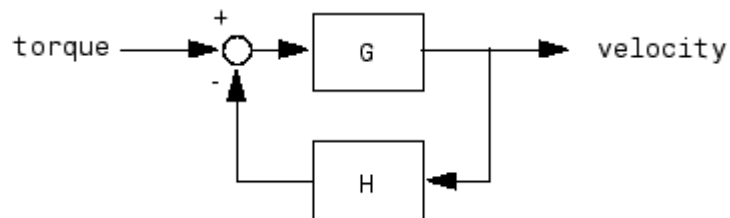
```
sys = feedback(sys1,sys2,feedin,feedout,+1)
```

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

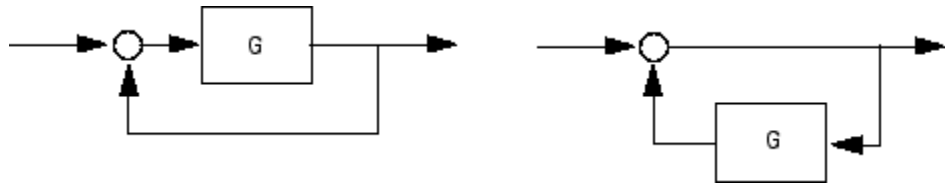
```
feedin = [4 2];
feedout = [1 3 4];
Cloop = feedback(P, K, feedin, feedout)
```

### Example 3

You can form the following negative-feedback loops

# feedback

---



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`



**Purpose**

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.

## Tips

`filt` behaves as `tf` with the `Variable` property set to `'z^-1'`. See `tf` entry below for details.

## 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
```

```
Sampling time: unspecified
```

**See Also**

tf | zpk | ss

# fnorm

---

**Purpose** Pointwise peak gain of FRD model

**Syntax** `fnm = fnorm(sys)`  
`fnm = fnorm(sys, ntype)`

**Description** `fnm = 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 `fnm` is an FRD object containing the peak gain across frequencies.

`fnm = 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`

**Purpose**

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 on page 1-152 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 1-152.

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 the string `units`. For a list of values for the string `units`, see the `FrequencyUnit` property in “Properties” on page 1-152.

## 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-by-Nu-by-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-by-...-by-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 is a string that 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  $[N_y \ N_u \ N_w]$ , where  $N_w$  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 sampling period,  $T_s$ .

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 delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. 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 sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

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 for all input channels

### **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 sampling period  $T_s$ . 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

### **Ts**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'

- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

### **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input 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. Set Name to a string to label the system.

**Default:** ''

**Notes**

Any text that you want to associate with the system. Set Notes to a string or a cell array of strings.

**Default:** {}

**UserData**

Any type of data you wish to associate with system. Set UserData to 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
```

...

**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);
```

**See Also**

[chgTimeUnit](#) | [chgFreqUnit](#) | [frdata](#) | [set](#) | [ss](#) | [tf](#) | [zpk](#) | [idfrd](#)

**Tutorials**

- “Frequency-Response Model”
- “MIMO Frequency Response Data Model”

**How To**

- “What Are Model Objects?”
- “Frequency Response Data (FRD) Models”

**Purpose** 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  $N_y$  outputs and  $N_u$  inputs at  $N_f$  frequencies:

- `response` is an  $N_y$ -by- $N_u$ -by- $N_f$  multidimensional array where the  $(i,j)$  entry specifies the response from input  $j$  to output  $i$ .
- `freq` is a column vector of length  $N_f$  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 and extract the frequency response data.

Create a frequency response data 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`

# freqresp

---

**Purpose** 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 the string `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**

String specifying the units of the frequencies in the input frequency vector `w`. Units can take 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 `Ny` outputs and `Nu` inputs, `H` is a 3D array with dimensions `Ny`-by-`Nu`-by-`Nw`, where `Nw` 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 `[Ny Nu S1 ... Sn]`, `H` is an array with dimensions `Ny`-by-`Nu`-by-`Nw`-by-`S1`-by-...-by-`Sn` 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.

## **Definitions**

### **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  $T_s = 1$ .

## **Examples**

### **Frequency Response**

Compute the frequency response of the 2-input, 2-output system

$$\text{sys} = \begin{bmatrix} 0 & \frac{1}{s+1} \\ \frac{s-1}{s+2} & 1 \end{bmatrix}$$

```

sys11 = 0;
sys22 = 1;
sys12 = tf(1,[1 1]);
sys21 = tf([1 -1],[1 2]);
sys = [sys11,sys12;sys21,sys22];

```

```
[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`.

### Response on Specified Frequency Grid

Compute the frequency response of the 2-input, 2-output system

$$\text{sys} = \begin{bmatrix} 0 & \frac{1}{s+1} \\ \frac{s-1}{s+2} & 1 \end{bmatrix}$$

on a logarithmically-spaced grid of 200 frequency points between 10 and 100 radians per second.

```

sys11 = 0;
sys22 = 1;
sys12 = tf(1,[1 1]);
sys21 = tf([1 -1],[1 2]);
sys = [sys11,sys12;sys21,sys22];

```

```
w = logspace(1,2,200);
```

```
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)
```

## 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.

## 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.

## See Also

`evalfr` | `bode` | `nyquist` | `nichols` | `sigma` | `ltiview` | `interp` | `spectrum`

# fselect

---

**Purpose** 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** `interp` | `fcats` | `fdel` | `frd`



**Purpose** 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  $j\omega$ -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`

**Purpose** 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`

<b>Purpose</b>	Generalized frequency response data (FRD) model
<b>Description</b>	Generalized FRD ( <code>genfrd</code> ) models arise when you combine numeric FRD models with models containing tunable components (Control Design Blocks). <code>genfrd</code> models keep track of how the tunable blocks interact with the tunable components. For more information about Control Design Blocks, see “Generalized Models”.
<b>Construction</b>	<p>To construct a <code>genfrd</code> model, use <code>series</code>, <code>parallel</code>, <code>lft</code>, or <code>connect</code>, or the arithmetic operators <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>\</code>, and <code>^</code>, to combine a numeric FRD model with control design blocks.</p> <p>You can also convert any numeric LTI model or control design block <code>sys</code> to <code>genfrd</code> form.</p> <p><code>frdsys = genfrd(sys, freqs, frequits)</code> converts any static model or dynamic system <code>sys</code> to a generalized FRD model. If <code>sys</code> is not an <code>frd</code> model object, <code>genfrd</code> computes the frequency response of each frequency point in the vector <code>freqs</code>. The frequencies <code>freqs</code> are in the units specified by the optional argument <code>frequits</code>. If <code>frequits</code> is omitted, the units of <code>freqs</code> are 'rad/TimeUnit'.</p> <p><code>frdsys = genfrd(sys, freqs, frequits, timeunits)</code> further specifies the time units for converting <code>sys</code> to <code>genfrd</code> form.</p> <p>For more information about time and frequency units of <code>genfrd</code> models, see “Properties” on page 1-175.</p>

### Input Arguments

#### **sys**

A static model or dynamic system model object.

#### **freqs**

Vector of frequency points. Express frequencies in the unit specified in `frequits`.

#### **frequits**

String specifying the frequency units of the `genfrd` model. Set `frequnits` to one of the following values:

- 'rad/TimeUnit'
- 'cycles/TimeUnit'
- 'rad/s'
- 'Hz'
- 'kHz'
- 'MHz'
- 'GHz'
- 'rpm'

**Default:** 'rad/TimeUnit'

## **timeunits**

String specifying the time units of the `genfrd` model. Set `timeunits` to one of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

**Default:** 'seconds'

## 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.

## 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` is a string that 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 delays. InputDelay is a numeric vector specifying a time delay for each input channel. 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 sampling period Ts. For example, InputDelay = 3 means a delay of three sampling periods.

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 for all input channels

### **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 sampling period 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**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'

- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

## **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input



model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string '' for all input channels

### **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set Name to a string to label the system.

**Default:** ''

**Notes**

Any text that you want to associate with the system. Set Notes to a string or a cell array of strings.

**Default:** {}

**UserData**

Any type of data you wish to associate with system. Set UserData to 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
```

...

**Default:** []

## See Also

frd | genss | getValue | chgFreqUnit

## How To

- “Models with Tunable Coefficients”
- “Generalized Models”

<b>Purpose</b>	Generalized matrix with tunable parameters
<b>Description</b>	Generalized matrices ( <code>genmat</code> ) are matrices that depend on tunable parameters (see <code>realp</code> ). You can use generalized matrices for parameter studies. You can also use generalized matrices for building generalized LTI models (see <code>genss</code> ) that represent control systems having a mixture of fixed and tunable components.
<b>Construction</b>	<p>Generalized matrices arise when you combine numeric values with static blocks such as <code>realp</code> objects. You create such combinations using any of the arithmetic operators <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>\</code>, and <code>^</code>. For example, if <code>a</code> and <code>b</code> are tunable parameters, the expression <code>M = a + b</code> is represented as a generalized matrix.</p> <p>A generalized matrix can represent a tunable gain surface for constructing gain-scheduled controllers. Use the Robust Control Toolbox™ command <code>gainsurf</code> to create such a tunable gain surface.</p> <p>The internal data structure of the <code>genmat</code> object <code>M</code> keeps track of how <code>M</code> depends on the parameters <code>a</code> and <code>b</code>. The <code>Blocks</code> property of <code>M</code> lists the parameters <code>a</code> and <code>b</code>.</p> <p><code>M = genmat(A)</code> converts the numeric array or tunable parameter <code>A</code> into a <code>genmat</code> object.</p>

### 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] =
```

```

          25
-----
s^2 + 3 s + 25
```

```
M(:,:,2,1) [zeta=0.35, w=5] =
```

```

          25
-----
s^2 + 3.5 s + 25
```

...

**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  $M = \begin{bmatrix} 1 & 0 \\ 0 & -9 \end{bmatrix}$ .

## See Also

`realp` | `genss` | `getValue` | `gainsurf`

## How To

- “Models with Tunable Coefficients”
- “Dynamic System Models”



**Purpose**

Generate test input signals for `lsim`

**Syntax**

```
[u,t] = gensig(type,tau)
[u,t] = gensig(type,tau,Tf,Ts)
```

**Description**

`[u,t] = gensig(type,tau)` generates a scalar signal `u` of class `type` and with period `tau` (in seconds). The following types of signals are available.

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

`gensig` returns a vector `t` of time samples and the vector `u` of signal values at these samples. All generated signals have unit amplitude.

`[u,t] = gensig(type,tau,Tf,Ts)` also specifies the time duration `Tf` of the signal and the spacing `Ts` between the time samples `t`.

You can feed the outputs `u` and `t` directly to `lsim` and simulate the response of a single-input linear system to the specified signal. Since `t` is uniquely determined by `Tf` and `Ts`, you can also generate inputs for multi-input systems by repeated calls to `gensig`.

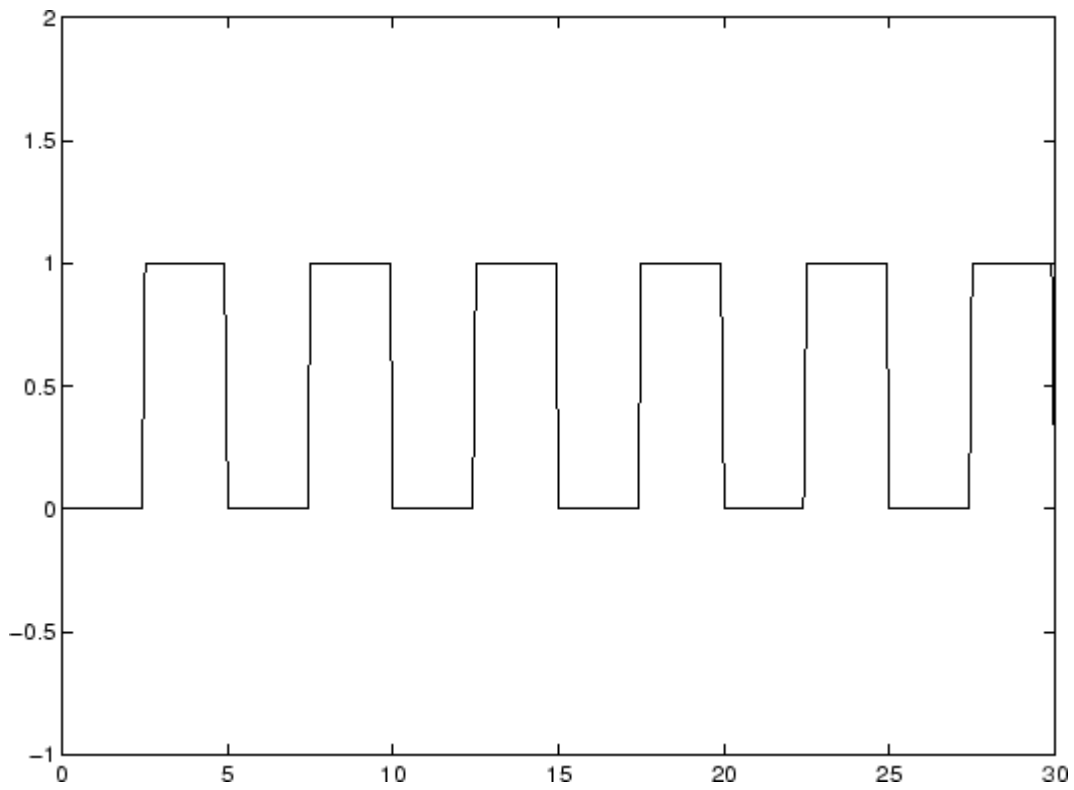
**Examples**

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`

---

<b>Purpose</b>	Generalized state-space model
<b>Description</b>	<p>Generalized state-space (<b>genss</b>) models are state-space models that include tunable parameters or components. <b>genss</b> 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”.</p> <p>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 <b>hinfstruct</b> (requires Robust Control Toolbox).</p>
<b>Construction</b>	<p>To construct a <b>genss</b> model:</p> <ul style="list-style-type: none"><li>• Use <b>series</b>, <b>parallel</b>, <b>lft</b>, or <b>connect</b>, or the arithmetic operators <b>+</b>, <b>-</b>, <b>*</b>, <b>/</b>, <b>\</b>, and <b>^</b>, to combine numeric LTI models with control design blocks.</li><li>• Use <b>tf</b> or <b>ss</b> with one or more input arguments that is a generalized matrix (<b>genmat</b>) instead of a numeric array</li><li>• Cast any numeric LTI model or control design block <b>sys</b> to <b>genss</b> form using: <pre>gensys = genss(sys)</pre></li></ul>
<b>Tips</b>	<ul style="list-style-type: none"><li>• You can manipulate <b>genss</b> models as ordinary <b>ss</b> models. Analysis commands such as <b>bode</b> and <b>step</b> evaluate the model by replacing each tunable parameter with its current value.</li></ul>
<b>Properties</b>	<p><b>Blocks</b></p> <p>Structure containing the control design blocks included in the generalized LTI model or generalized matrix. The field names of <b>Blocks</b> are the <b>Name</b> property of each control design block.</p>

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 sampling period `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 delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. 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 sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

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 for all input channels

### **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 sampling period  $T_s$ . 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

### **Ts**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

## OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input 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. Set `Name` to a string to label the system.

**Default:** ''

### **Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

### **UserData**

Any type of data you wish to associate with system. Set `UserData` to 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
```

...

**Default:** []

## Examples

### Tunable Low-Pass Filter

This example shows how to create the low-pass filter  $F = a/(s + a)$  with one tunable parameter  $a$ .

You cannot use `ltiblock.tf` to represent  $F$ , because the numerator and denominator coefficients of an `ltiblock.tf` block are independent. Instead, construct  $F$  using the tunable real parameter object `realp`.

- 1 Create a tunable real parameter.

```
a = realp('a',10);
```

The `realp` object `a` is a tunable parameter with initial value 10.

- 2 Use `tf` to create the tunable filter `F`:

```
F = tf(a,[1 a]);
```

`F` is a `genss` object which has the tunable parameter `a` in its `Blocks` property. You can connect `F` with other tunable or numeric models to create more complex models of control systems. For an example, see “Control System with Tunable Components”.

### State-Space Model With Both Fixed and Tunable Parameters

This example shows how to create a state-space (`genss`) model having both fixed and tunable parameters.

Create a state-space model having the following state-space matrices:

$$A = \begin{bmatrix} 1 & a+b \\ 0 & ab \end{bmatrix}, \quad B = \begin{bmatrix} -3.0 \\ 1.5 \end{bmatrix}, \quad C = [0.3 \ 0], \quad D = 0,$$

where  $a$  and  $b$  are tunable parameters, whose initial values are  $-1$  and  $3$ , respectively.

- 1 Create the tunable parameters using `realp`.

```
a = realp('a',-1);
```

```
b = realp('b',3);
```

- 2 Define a generalized matrix using algebraic expressions of a and b.

```
A = [1 a+b;0 a*b]
```

A is a generalized matrix whose `Blocks` property contains a and b. The initial value of A is  $M = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix}$ , from the initial values of a and b.

- 3 Create the fixed-value state-space matrices.

```
B = [-3.0;1.5];
```

```
C = [0.3 0];
```

```
D = 0;
```

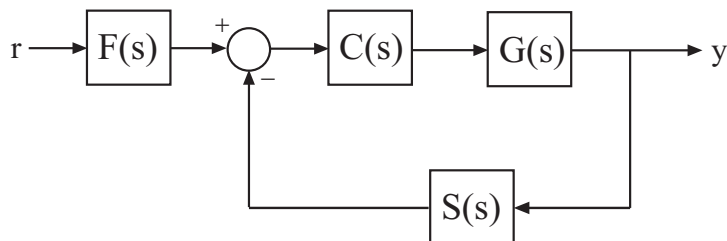
- 4 Use `ss` to create the state-space model.

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

`sys` is a generalized LTI model (`genss`) with tunable parameters a and b.

### Control System With Both Numeric and Tunable Components

This example shows how to create a tunable model of the control system in the following illustration.



The plant response  $G(s) = 1/(s + 1)^2$ . The model of sensor dynamics is  $S(s) = 5/(s + 4)$ . The controller  $C$  is a tunable PID controller, and the prefilter  $F = a/(s + a)$  is a low-pass filter with one tunable parameter,  $a$ .

Create models representing the plant and sensor dynamics.

Because the plant and sensor dynamics are fixed, represent them using numeric LTI models `zpk` and `tf`.

```
G = zpk([], [-1, -1], 1);  
S = tf(5, [1 4]);
```

Create a tunable representation of the controller  $C$ .

```
C = ltiblock.pid('C', 'PID');
```

`C` is a `ltiblock.pid` object, which is a Control Design Block with a predefined proportional-integral-derivative (PID) structure.

Create a model of the filter  $F = a/(s + a)$  with one tunable parameter.

```
a = realp('a', 10);  
F = tf(a, [1 a]);
```

`a` is a `realp` (real tunable parameter) object with initial value 10. Using `a` as a coefficient in `tf` creates the tunable `genss` model object `F`.

Connect the models together to construct a model of the closed-loop response from  $r$  to  $y$ .

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

`T` is a `genss` model object. In contrast to an aggregate model formed by connecting only Numeric LTI models, `T` keeps track of the tunable elements of the control system. The tunable elements are stored in the `Blocks` property of the `genss` model object.

Display the tunable elements of `T`.

```
T.Blocks
```

```
ans =
```

```
    C: [1x1 ltiblock.pid]
```

a: [1x1 realp]

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

## See Also

`realp` | `genmat` | `genfrd` | `tf` | `ss` | `getValue` | `ltiblock.pid` | `feedback` | `connect`

## How To

- “Models with Tunable Coefficients”
- “Dynamic System Models”
- “Control Design Blocks”

**Purpose**

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`. The string `'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)  
    num: {[0 1]}  
    den: {[1 2]}  
  ioDelay: 0  
  Variable: 'z'  
      Ts: 0.1  
  InputDelay: 0  
  OutputDelay: 0  
  InputName: {'voltage'}  
  OutputName: {''}
```

```
InputGroup: [1x1 struct]
OutputGroup: [1x1 struct]
    Name: ''
    Notes: {}
    UserData: 'hello'
```

or query only about the numerator and sample time values by

```
get(h, 'num')

ans =
    [1x2 double]

and

get(h, 'ts')

ans =
    0.1000
```

Because the numerator data (num property) is always stored as a cell array, the first command evaluates to a cell array containing the row vector [0 1].

## 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**

`frdata` | `set` | `ssdata` | `tfdata` | `zpkdata` | `idssdata` | `polydata`

# getBlockValue

---

<b>Purpose</b>	Current value of Control Design Block in Generalized Model
<b>Syntax</b>	<code>val = getBlockValue(M,blockname)</code>
<b>Description</b>	<code>val = getBlockValue(M,blockname)</code> returns the current value of the Control Design Block <code>blockname</code> in the Generalized Model <code>M</code> . (For uncertain blocks, the “current value” is the nominal value of the block.)
<b>Input Arguments</b>	<b>M</b> Generalized LTI Model or Generalized matrix.  <b>blockname</b> Name of the Control Design Block in the model <code>M</code> whose current value is evaluated. To get a list of the Control Design Blocks in <code>M</code> , enter <code>M.Blocks</code> .
<b>Output Arguments</b>	<b>val</b> Numerical LTI model or numerical value, equal to the current value of the Control Design Block <code>blockname</code> .
<b>Examples</b>	Create a tunable genss model, and evaluate the current value of the Control Design Blocks of the model.  <pre>G = zpk([], [-1, -1], 1); C = ltiblock.pid('C', 'PID'); a = realp('a', 10); F = tf(a, [1 a]); T = feedback(G*C, 1)*F;  Cval = getBlockValue(T, 'C')</pre> Continuous-time I-only controller:

```
Ki * ---  
      s
```

With  $K_i = 0.001$

`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.

## See Also

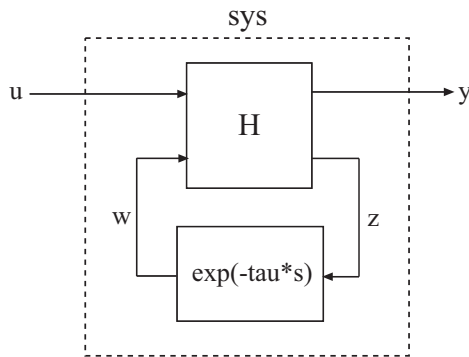
[setBlockValue](#) | [showBlockValue](#) | [getValue](#)

# getDelayModel

**Purpose** 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 ( []).

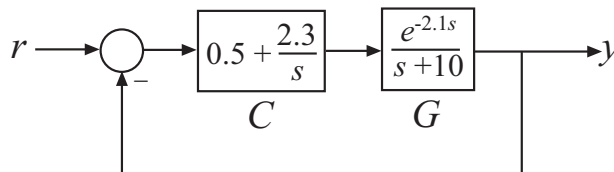
# getDelayModel

---

## 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 arising from the feedback loop.

Decompose `sys` into a delay-free state-space model and the value of the internal delay.

```
[H,tau] = getDelayModel(sys);
```

**See Also** `setDelayModel`

**Concepts**

- “Internal Delays”

<b>Purpose</b>	Crossover frequencies for specified gain
<b>Syntax</b>	<code>wc = getGainCrossover(sys,gain)</code>
<b>Description</b>	<code>wc = getGainCrossover(sys,gain)</code> returns the vector <code>wc</code> of frequencies at which the frequency response of the dynamic system model, <code>sys</code> , has principal gain of <code>gain</code> . For SISO systems, the principal gain is the frequency response. For MIMO models, the principal gain is the largest singular value of <code>sys</code> .
<b>Input Arguments</b>	<p><b>sys - Input dynamic system</b> dynamic system model</p> <p>Input dynamic system, specified as any SISO or MIMO dynamic system model.</p> <p><b>gain - Input gain</b> positive real scalar</p> <p>Input gain in absolute units, specified as a positive real scalar.</p> <ul style="list-style-type: none"><li>• If <code>sys</code> is a SISO model, the gain is the frequency response magnitude of <code>sys</code>.</li><li>• If <code>sys</code> is a MIMO model, gain means the largest singular value of <code>sys</code>.</li></ul>
<b>Output Arguments</b>	<p><b>wc - Crossover frequencies</b> column vector</p> <p>Crossover frequencies, returned as a column vector. This vector lists the frequencies at which the gain or largest singular value of <code>sys</code> is <code>gain</code>.</p>
<b>Examples</b>	<p><b>Unity Gain Crossover</b></p> <p>Find the 0dB crossover of a single-loop control system with plant</p>

# getGainCrossover

---

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

and PI controller

$$C(s) = 1.14 + \frac{0.454}{s}$$

```
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 crossovers are the frequencies at which the open-loop response  $\text{sys} = G*C$  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

$$\text{sys} = \frac{s^2 + 0.05s + 100}{s^2 + 5s + 100}$$

$\text{sys}$  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.

## 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>.

## See Also

`freqresp` | `bode` | `sigma` | `bandwidth` | `getPeakGain`

## Concepts

- “Dynamic System Models”

# getIOTransfer

---

**Purpose** 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.

## Input Arguments

**T - Model of control system**  
generalized state-space model

Model of a control system, specified as a Generalized State-Space (genss) Model.

### in - Input to extracted transfer function

string | cell array of strings

Input to extracted transfer function, specified as a string or cell array of strings. To extract a multiple-input transfer function from the control system, use a cell array of strings. Each string in `in` must match either:

- An input of the control system model `T` (in other words, a string contained in `T.InputName`).
- A loop-opening site in `T`, corresponding to a channel of a loopswitch block in `T`. Use `getLoopID(T)` to get a full list of available loop-opening sites in `T`.

When you specify a loop-opening site as an input `in`, `getIOTransfer` uses the input implicitly associated with the loopswitch channel, arranged as follows.



This input signal models a disturbance entering at the output of the switch.

If a loop-opening site has the same name as an input of  $T$ , then `getIOTransfer` uses the input of  $T$ .

**Example:** `{ 'r', 'X1' }`

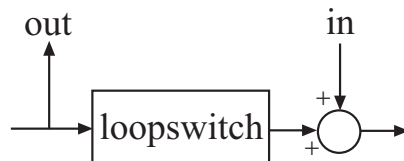
### **out - Output of extracted transfer function**

string | cell array of strings

Output of extracted transfer function, specified as a string or cell array of strings. To extract a multiple-output transfer function from the control system, use a cell array of strings. Each string in `out` must match either:

- An output of the control system model  $T$  (in other words, a string contained in  $T$ .`OutputName`).
- A loop-opening site in  $T$ , corresponding to a channel of a `loopswitch` block in  $T$ . Use `getLoopID(T)` to get a full list of available loop-opening sites in  $T$ .

When you specify a loop-opening site as an output `out`, `getIOTransfer` uses the output implicitly associated with the `loopswitch` channel, arranged as follows.



If a loop-opening site 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**

string | cell array of strings

Locations for opening feedback loops for computation of the response from `in` to `out`, specified as string or cell array of strings that identify loop-opening sites in  $T$ . Loop-opening sites are marked by `loopswitch` blocks in  $T$ . Use `getLoopID(T)` to get a full list of available loop-opening sites in  $T$ .

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 (`genss`) model.

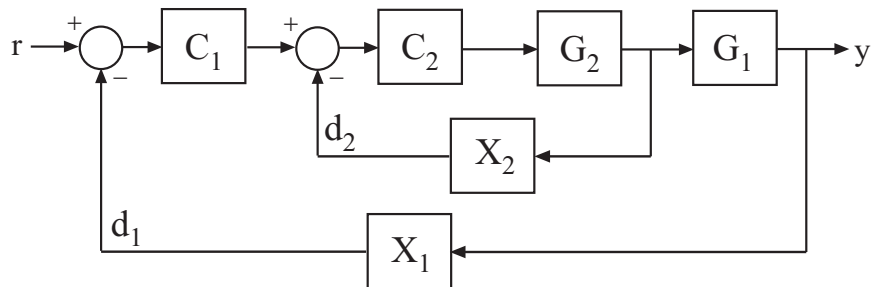
- If both `in` and `out` specify a single signal, then  $T$  is a SISO `genss` model.
- If `in` or `out` identifies multiple signals, then  $T$  is a MIMO `genss` model.

## **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 the loopswitch blocks `X1` and `X2` that mark potential loop-opening or signal injection sites.

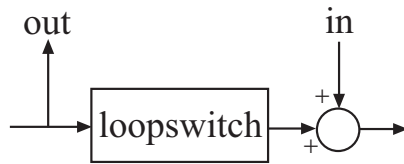
```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = ltiblock.pid('C','pi');
C2 = ltiblock.gain('G',1);
X1 = loopswitch('X1');
X2 = loopswitch('X2');
T = feedback(G1*feedback(G2*C2,X2)*C1,X1);
T.InputName = 'r';
T.OutputName = 'y';
```

If you tuned the free parameters of this model (for example, using the Robust Control Toolbox 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  $d_1$ .

```
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 loopswitch block `X1`, which is the location of  $d_1$ :



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 analyze 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  $d_2$ .

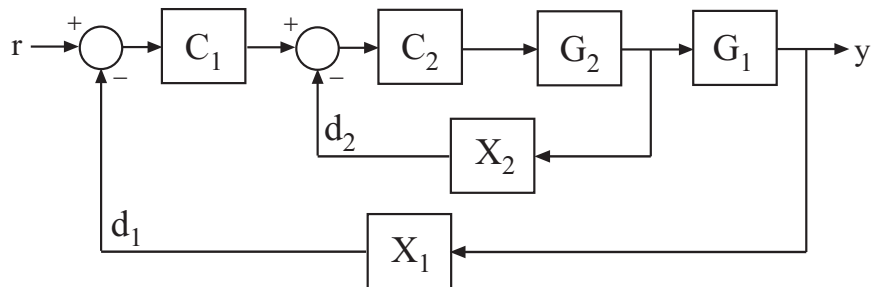
```
H2 = getIOTransfer(T, 'X2', 'y');
```

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  $G_1$  and  $G_2$ , the tunable controllers  $C_1$ , and the loopswitch blocks  $X_1$  and  $X_2$  that mark potential loop-opening or signal injection sites.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
C1 = ltiblock.pid('C','pi');
C2 = ltiblock.gain('G',1);
X1 = loopswitch('X1');
X2 = loopswitch('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 Robust Control Toolbox 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 loop-opening locations in  $T$ ,  $X_1$  and  $X_2$ , are closed. Specifying `'X2'` for the `openings` argument causes `getIOTransfer` to open the loop at  $X_2$  for the purposes of computing the requested transfer from  $r$  to  $y$ . The switch at  $X_1$  remains closed for this computation.

# getIOTransfer

---

## 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 the Robust Control Toolbox tuning command `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™ command `sITunable.getIOTransfer`. Use the latter command when your control system is modeled in Simulink.

## See Also

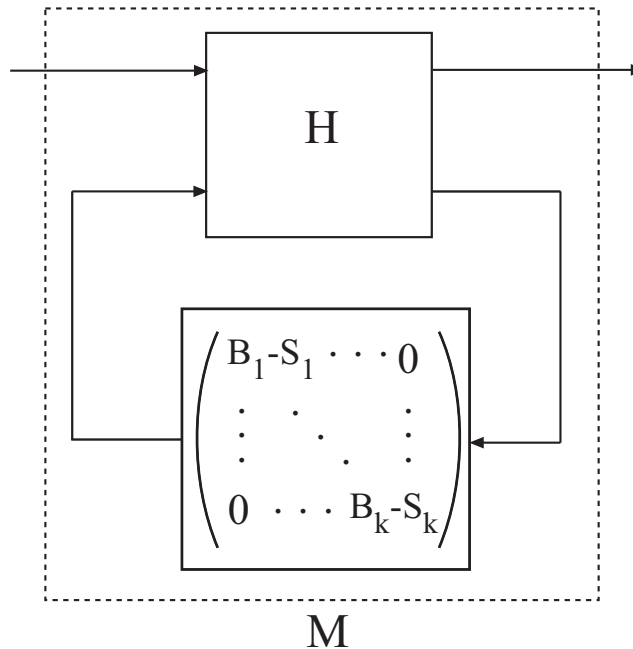
`loopswitch` | `genss` | `getLoopTransfer` |  
`systunesITunable.getIOTransfer` |



**Purpose** 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.

# getLFTModel

---

You can recombine **H**, **B**, and **S** into **M** using `lft`, as follows:

```
M = lft(H,blkdiag(B{:}-S));
```

## 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”.

## 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 `ltiblock.ss`) 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.

## See Also

`genfrd` | `genss` | `genmat` | `lft` | `getValue` | `nblocks`

## How To

- “Generalized Matrices”
- “Generalized and Uncertain LTI Models”
- “Models with Tunable Coefficients”
- “Internal Structure of Generalized Models”

## Purpose

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 loop-opening locations. 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`.

## 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 `loopswitch` blocks in `T`.

### **Locations - Loop-opening locations**

string | cell array of strings

Loop-opening locations in the control system model at which to compute the open-loop point-to-point response, specified as a string or a cell array of strings that identify loop-opening locations in `T`.

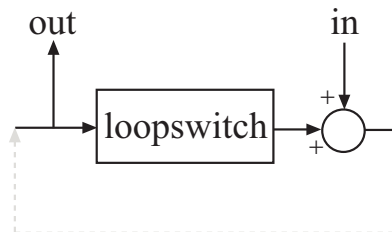
# getLoopTransfer

---

Loop-opening locations are marked by `loopswitch` blocks in `T`. A `loopswitch` block can have single or multiple channels. The `Location` property of a `loopswitch` block gives names to these feedback channels.

The name of any channel in a `loopswitch` block in `T` is a valid entry for the `Locations` argument to `getLoopTransfer`. Use `getSwitches(T)` to get a full list of available loop-opening locations in `T`.

`getLoopTransfer` computes the open-loop response you would obtain by injecting a signal at the implicit input associated with a `loopswitch` 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

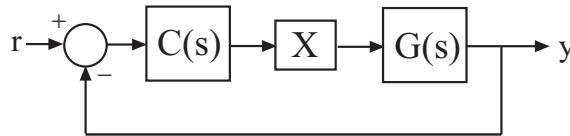
+1 (default) | -1

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

string | cell array of strings

Additional locations for opening feedback loops for computation of the open-loop response, specified as string or cell array of strings that identify loop-opening locations in  $T$ . Loop-opening locations are marked by `loopswitch` blocks in  $T$ . Any channel name contained in the `Location` property of a `loopswitch` block in  $T$  is a valid entry for `openings`.

Use `openings` when you want to compute the open-loop response at one loop-opening 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 `getSwitches(T)` to get a full list of available loop-opening 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 loop-opening location specified by `Locations`, returned as a Generalized State-Space (`genss`) Model.

- If `Locations` is a string specifying a single loop-opening location, then  $L$  is a SISO `genss` model. In this case,  $L$  represents the response

# getLoopTransfer

---

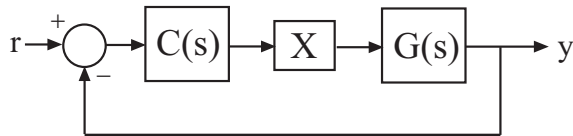
obtained by opening the loop at `Locations`, injecting signals and measuring the return signals at the same location.

- If `Locations` is a string specifying a vector signal, or a cell array identifying multiple loop-opening locations, 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.

## Examples

### Open-Loop Transfer Function at Loop-Opening Location

Compute the open-loop response of the following control system model at a loop-opening location specified by a loopswitch block, `X`.



Create a model of the system by specifying and connecting a numeric LTI plant model `G`, a tunable controller `C`, and the loopswitch block `X`.

```
G = tf([1 2],[1 0.2 10]);  
C = ltiblock.pid('C','pi');  
X = loopswitch('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 loopswitch 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');
```

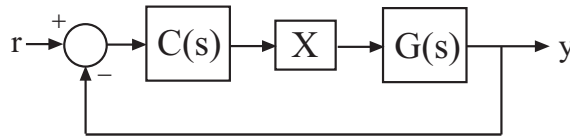
This command computes the positive-feedback transfer function you would obtain by opening the loop at `X`, injecting a signal into `G`, and measuring the resulting response at the output of `C`. By default,

`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 a loop-opening location specified by a `loopswitch` block, `X`.



Create a model of the system by specifying and connecting a numeric LTI plant model `G`, a tunable controller `C`, and the `loopswitch` block `X`.

```
G = tf([1 2],[1 0.2 10]);
C = ltiblock.pid('C','pi');
X = loopswitch('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 `loopswitch` block `X` that identifies the potential loop-opening location.

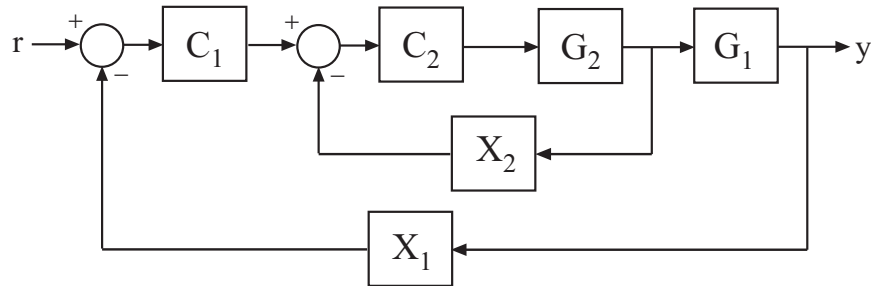
Calculate the open-loop point-to-point loop transfer at the location `X`.

```
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 loopswitch blocks X1 and X2 that mark potential loop-opening locations.

```
G1 = tf(10,[1 10]);  
G2 = tf([1 2],[1 0.2 10]);  
C1 = ltiblock.pid('C','pi');  
C2 = ltiblock.gain('G',1);  
X1 = loopswitch('X1');  
X2 = loopswitch('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-opening location marked the loopswitch block X1 is closed. 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  $L(s) = G_2(s)C_2(s)$ .



## 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 Robust Control Toolbox tuning command `sys tune`.
- `getLoopTransfer` is the `genss` equivalent to the Simulink Control Design command `s1Tunable.getLoopTransfer`. Use the latter command when your control system is modeled in Simulink.

## See Also

`loopswitch` | `genss` | `getIOTransfer` |  
`sys tune s1Tunable.getLoopTransfer` |

# getNominal

---

## **Purpose**

Nominal value of Generalized LTI model or Generalized matrix

---

**Note** `getNominal` has been removed. Use `getValue` instead.

---

**Purpose** 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`

# getPeakGain

---

**Purpose** 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.

## 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**.

### **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.

## Examples

### Peak Gain of Transfer Function

Compute the peak gain of the resonance in the transfer function

$$\text{sys} = \frac{90}{s^2 + 1.5s + 90}.$$

```
sys = tf(90,[1,1.5,90]);  
gpeak = getPeakGain(sys);
```

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

$$\text{sys} = \frac{90}{s^2 + 1.5s + 90}, \text{ with a relative accuracy of } 0.01\%.$$

```
sys = tf(90,[1,1.5,90]);  
gpeak = getPeakGain(sys,0.0001);
```

The second argument specifies a relative accuracy of 0.0001. The `getPeakGain` command returns a value that is within 0.01% of the true peak gain of the transfer function.

### Peak Gain Within Specified Band

Compute the peak gain of the second resonance in the transfer function

$$\text{sys} = \left( \frac{1}{s^2 + 0.2s + 1} \right) \left( \frac{100}{s^2 + s + 100} \right).$$

`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

$$\text{sys} = \left( \frac{1}{s^2 + 0.2s + 1} \right) \left( \frac{100}{s^2 + s + 100} \right).$$

`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`.

## 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>.

## 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 | bode | sigma | getGainCrossover

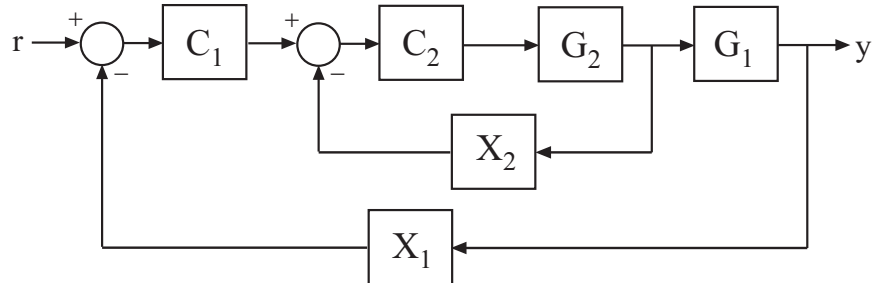
## Concepts

- "Dynamic System Models"



<b>Purpose</b>	Get list of loop opening sites in generalized model of control system
<b>Syntax</b>	<code>Locations = getSwitches(T)</code>
<b>Description</b>	<code>Locations = getSwitches(T)</code> returns the names of all loop-opening sites in a Generalized State-Space Model of a control system. Use these names to calculate open- or closed-loop responses using <code>getLoopTransfer</code> or <code>getIOTransfer</code> .
<b>Input Arguments</b>	<b>T - Model of control system</b> 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 <code>loopswitch</code> blocks in T.
<b>Output Arguments</b>	<b>Locations - Loop-opening sites</b> cell array of strings  Loop-opening sites in the control system model, returned as a cell array of strings. The strings contain the loop channel names. These loop channel names are the contents of the <code>Location</code> property of each <code>loopswitch</code> block in the control system model.
<b>Examples</b>	<b>Loop Opening Sites in Control System Model</b>  Build a closed-loop model of a cascaded feedback loop system, and get a list of loop-opening sites in the model.  Create a model of the following cascaded feedback loop. $C_1$ and $C_2$ are tunable controllers. $X_1$ and $X_2$ are loop-opening sites.

# getSwitches



```
G1 = tf(10,[1 10]);  
G2 = tf([1 2],[1 0.2 10]);  
C1 = ltiblock.pid('C','pi');  
C2 = ltiblock.gain('G',1);  
X1 = loopswitch('X1');  
X2 = loopswitch('X2');  
T = feedback(G1*feedback(G2*C2,X2)*C1,X1);
```

T is a `genss` model whose Control Design Blocks include the tunable controllers and the switches X1 and X2.

Get a list of the loop-opening sites in T.

```
Locations = getSwitches(T)
```

```
Locations =
```

```
'X1'  
'X2'
```

`getSwitches` returns a cell array listing loop-opening sites in the model.

For more complicated closed-loop models, you can use `getSwitches` to keep track of a larger number of loop-opening sites. You can use these loop-opening sites to specify an open-loop response to compute. For instance, the following command computes the open-loop response of the inner loop, with the outer loop open.

```
L = getLoopTransfer(T, 'X2', -1, 'X1');
```

## See Also

[loopswitch](#) | [genss](#) | [getLoopTransfer](#) | [getIOTransfer](#)

## Concepts

- “Generalized Models”

# getValue

---

**Purpose** 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 Robust Control Toolbox tuning commands such as `systeme`, `looptune`, or `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`.

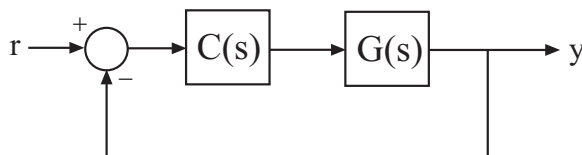
If you do not specify a replacement value for a given Control Design Block of `M`, `getValue` uses the current value of that block.

**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 = ltiblock.pid('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 Robust Control Toolbox tuning commands such as `sys tune`, `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] = systune(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 | systune | looptune | hinfstruct

# gram

---

**Purpose** Controllability and observability gramians

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

**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`.

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  $(C, A)$  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.

## Algorithms

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

$$A W_c + W_c A^T + B B^T = 0$$

or its discrete-time counterpart

$$A W_c A^T - W_c + B B^T = 0$$

Similarly, the observability gramian  $W_o$  solves the Lyapunov equation

$$A^T W_o + W_o A + C^T C = 0$$

in continuous time, and the Lyapunov equation

$$A^T W_o A - W_o + C^T C = 0$$

in discrete time.

## Limitations

The  $A$  matrix must be stable (all eigenvalues have negative real part in continuous time, and magnitude strictly less than one in discrete time).

## References

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

## See Also

balreal | ctrb | lyap | dlyap | obsv

# hasdelay

---

**Purpose** 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** absorbDelay | totaldelay

**Purpose** 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.

**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**

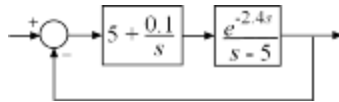
**B - Flag indicating presence of internal delays**  
logical | logical array

Flag indicating presence of internal delays in input model or array, returned as a logical value or logical array.

**Examples**

**Check model for internal delays**

Build a dynamic system model of the following closed-loop system and check the model for internal delays.



```
s = tf('s');
G = exp(-2.4*s)/(s-5);
C = pid(5,0.1);
```

# hasInternalDelay

---

```
sys = feedback(G*C,1);  
B = hasInternalDelay(sys)
```

```
B =
```

```
1
```

The model `sys` has an internal delay because of the transfer delay in the plant `G`. Therefore, `hasInternalDelay` returns 1.

## See Also

`hasdelay` | `getDelayModel`

**Purpose** Hankel singular values of dynamic system

**Syntax**

```

hsv = hsvd(sys)
hsv = hsvd(sys, 'AbsTol', ATOL, 'RelTol', RTOL, 'Offset', ALPHA)
hsv = hsvd(sys, opts)
hsvd(sys)
[hsv, baldata] = hsvd(sys)

```

**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, 'AbsTol', ATOL, 'RelTol', RTOL, 'Offset', ALPHA)` specifies additional options for the stable/unstable decomposition. See the `stabsep` reference page for more information about these options. The default values are `ATOL = 0`, `RTOL = 1e-8`, and `ALPHA = 1e-8`.

*hsv* = `hsvd(sys, opts)` computes the Hankel singular values using the options specified in the `hsvdOptions` object *opts*.

`hsvd(sys)` displays a Hankel singular values plot.

`[hsv, baldata] = hsvd(sys)` returns additional data to speed up model order reduction with `balred`. For example

```

sys = rss(20);           % 20-th order model
[hsv, baldata] = hsvd(sys);
rsys = balred(sys, 8:10, 'Balancing', baldata);
bode(sys, 'b', rsys, 'r--')

```

computes three approximations of *sys* of orders 8, 9, 10.

There is more than one hsvd available. Type

```
help lti/hsvd
```

for more information.

## 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

The `AbsTol`, `RelTol`, and `ALPHA` parameters 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  $G_s$ .

## Examples

### Compute Hankel Singular Values

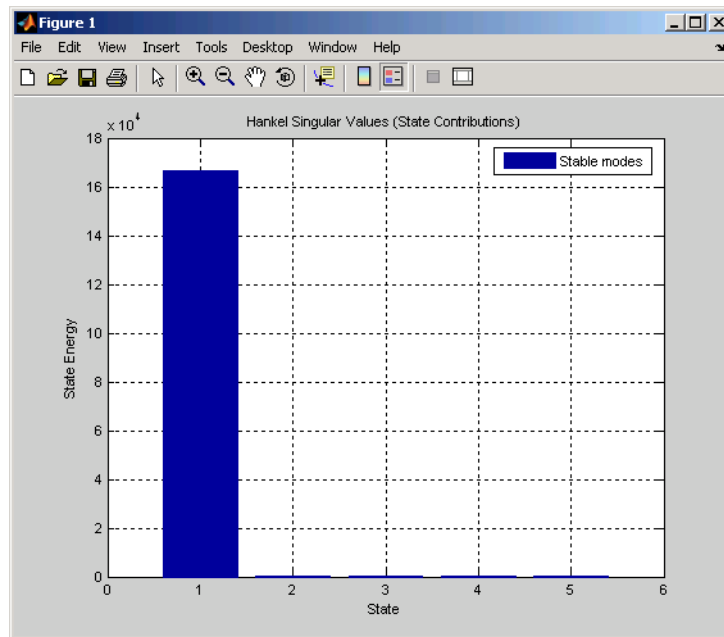
This example illustrates how to compute Hankel singular values.

First, create a system with a stable pole very near to 0, then calculate the Hankel singular values.

```
sys = zpk([1 2],[-1 -2 -3 -10 -1e-7],1)
hsvd(sys)
```

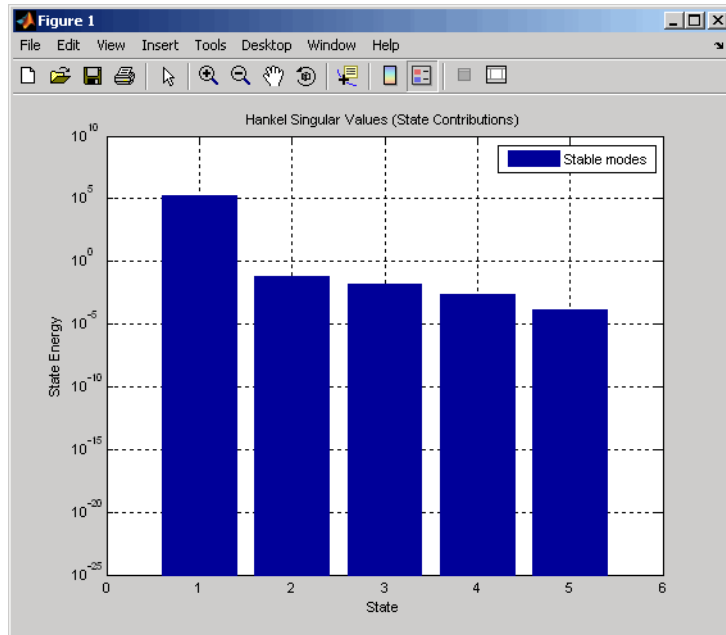
Zero/pole/gain:

$$\frac{(s-1)(s-2)}{(s+1)(s+2)(s+3)(s+10)(s+1e-007)}$$



For a better view of the Hankel singular values, switch the plot to log scale by selecting **Y Scale > Log** from the right-click menu.

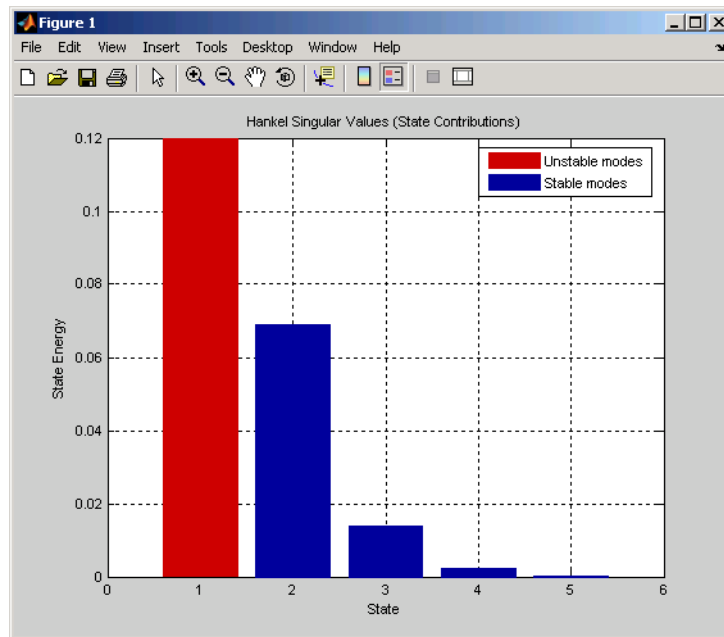




Notice the dominant Hankel singular value with  $1e5$  magnitude, due to the mode  $s = -1e-7$  near the imaginary axis. Set the `offset=1e-6` to treat this mode as unstable

```
hsvd(sys, 'Offset', 1e-7)
```

# hsvd



The dominant Hankel singular value is now shown as unstable.

## See Also

`hsvdOptions` | `balred` | `balreal`

**Purpose** Create option set for computing Hankel singular values and input/output balancing

**Syntax**

```
opts = hsvdOptions
opts = hsvdOptions('OptionName', OptionValue)
```

**Description**

`opts = hsvdOptions` returns the default options for the `hsvd` and `balreal` commands.

`opts = hsvdOptions('OptionName', OptionValue)` accepts one or more comma-separated name/value pairs. Specify *OptionName* inside single quotes.

## Input Arguments

### Name-Value Pair Arguments

#### '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  $\text{AbsTol} + \text{RelTol} * \text{abs}(G)$ . 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)

# hsvdOptions

---

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 `hsvd` and `balreal` reference pages.

## Examples

Compute the Hankel singular values of the system given by:

$$\text{sys} = \frac{(s+0.5)}{(s+10^{-6})(s+2)}$$

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); % create option set  
hsvd(sys,opts) % treats -1e-6 as unstable
```

## See Also

`hsvd` | `balreal`

**Purpose** Create list of Hankel singular value plot options

**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. You can use these options to customize the Hankel singular value plot appearance using the command line.

`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.

This table summarizes the Hankel singular value plot options.

Option	Description
Title, XLabel, YLabel	Label text and style
TickLabel	Tick label style
Grid [off on]	Show or hide the grid
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
YScale [linear log]	Scale for Y-axis
AbsTol, RelTol, Offset	Parameters for the Hankel singular value computation (used only for models with unstable dynamics). See <code>hsvd</code> and <code>stabsep</code> for details.

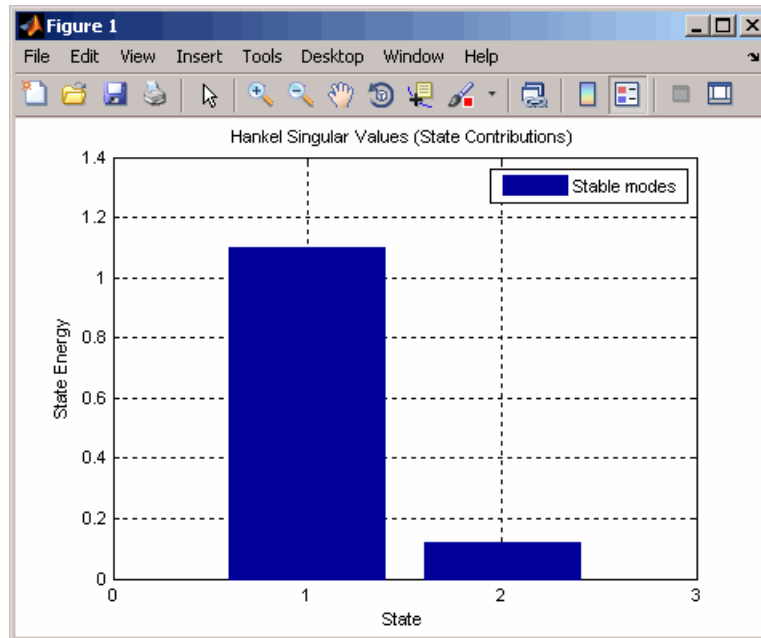
**Examples** In this example, you set the scale for the Y-axis in the HSV plot.

```
P = hsvoptions; % Set the Y-axis scale to linear in options
P.YScale = 'linear'; % Create plot with the options specified by P
```

# hsvoptions

```
h = hsvplot(rss(2,2,3),P);
```

The following HSV plot is created, with a linear scale for the Y-axis.



## See Also

[hsvd](#) | [hsvplot](#) | [getoptions](#) | [setoptions](#) | [stabsep](#)

---

<b>Purpose</b>	Plot Hankel singular values and return plot handle
<b>Syntax</b>	<pre>h = hsvplot(sys) hsvplot(sys) hsvplot(sys, 'AbsTol',ATOL, 'RelTol',RTOL, 'Offset',ALPHA) hsvplot(AX,sys,...)</pre>
<b>Description</b>	<p><code>h = hsvplot(sys)</code> plots the Hankel singular values of an LTI system <code>sys</code> and returns the plot handle <code>h</code>. You can use this handle to customize the plot with the <code>getoptions</code> and <code>setoptions</code> commands. Type</p> <pre>help hsvoptions</pre> <p>for a list of available plot options.</p> <p><code>hsvplot(sys)</code> plots the Hankel singular values of the LTI model <code>sys</code>. See <code>hsvd</code> for details on the meaning and purpose of Hankel singular values. The Hankel singular values for the stable and unstable modes of <code>sys</code> are shown in blue and red, respectively.</p> <p><code>hsvplot(sys, 'AbsTol',ATOL, 'RelTol',RTOL, 'Offset',ALPHA)</code> specifies additional options for computing the Hankel singular values.</p> <p><code>hsvplot(AX,sys,...)</code> attaches the plot to the axes with handle <code>AX</code>.</p>
<b>Tips</b>	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”.
<b>Examples</b>	<p>Use the plot handle to change plot options in the Hankel singular values plot.</p> <pre>sys = rss(20); h = hsvplot(sys,'AbsTol',1e-6); % Switch to log scale and modify Offset parameter setoptions(h,'Yscale','log','Offset',0.3)</pre>
<b>See Also</b>	<code>getoptions</code>   <code>hsvd</code>   <code>hsvoptions</code>   <code>setoptions</code>

# imp2exp

---

**Purpose** 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  $[V;T] = B*[W;I]$ .

```

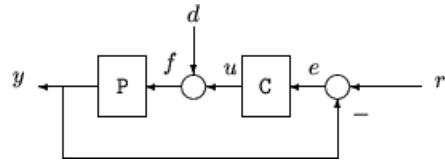
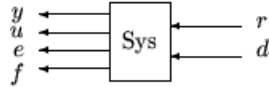
R = ureal('R',1,'Percentage',[-10 40]);
K = ureal('K',2e-3,'Percentage',[-30 30]);
A = [1 -R 0 -K;0 -K 1 0];
Yidx = [1 3];
Uidx = [4 2];
B = imp2exp(A,Yidx,Uidx)
UMAT: 2 Rows, 2 Columns
    K: real, nominal = 0.002, variability = [-30 30]%, 2 occurrences
    R: real, nominal = 1, variability = [-10 40]%, 1 occurrence

```

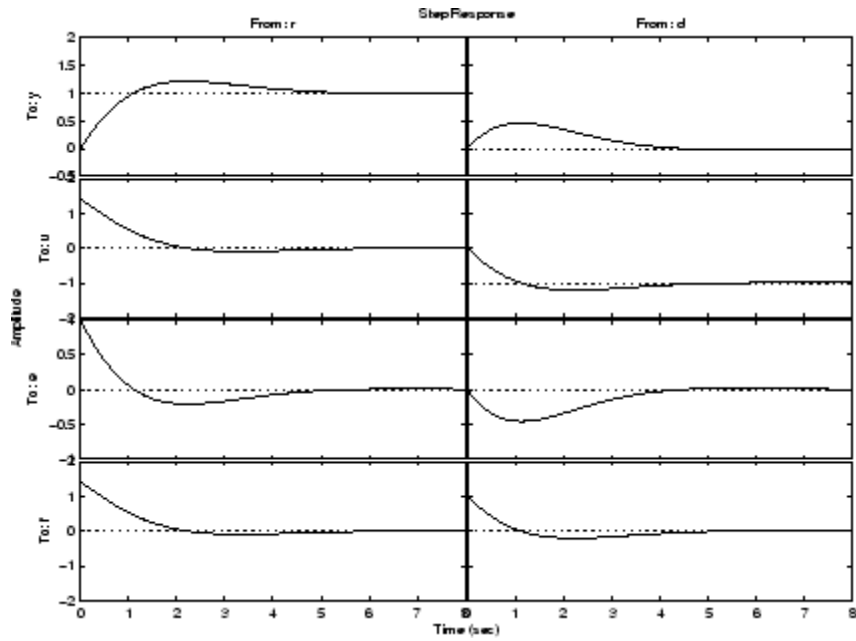
### Scalar Dynamic System Constraint

Consider a standard single-loop feedback connection of controller C and an uncertain plant P, described by the equations  $e=r-y$ ,  $u=Ce$ ;  $f=d+u$ ;  $y=Pf$ .

# imp2exp



```
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
step(Sys)
```



**Algorithms**

The number of rows of A must equal the length of yidx.

**See Also**

iconnect | inv

# impulse

---

**Purpose** 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 sampling 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 sampling 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  $T_s$  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 1-267). The `impulse` command always applies the impulse at  $t=0$ , regardless of  $T_i$ .

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  $t$ )  $\times$  (number of outputs)  $\times$  (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.

## 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”.

## Examples

### Example 1

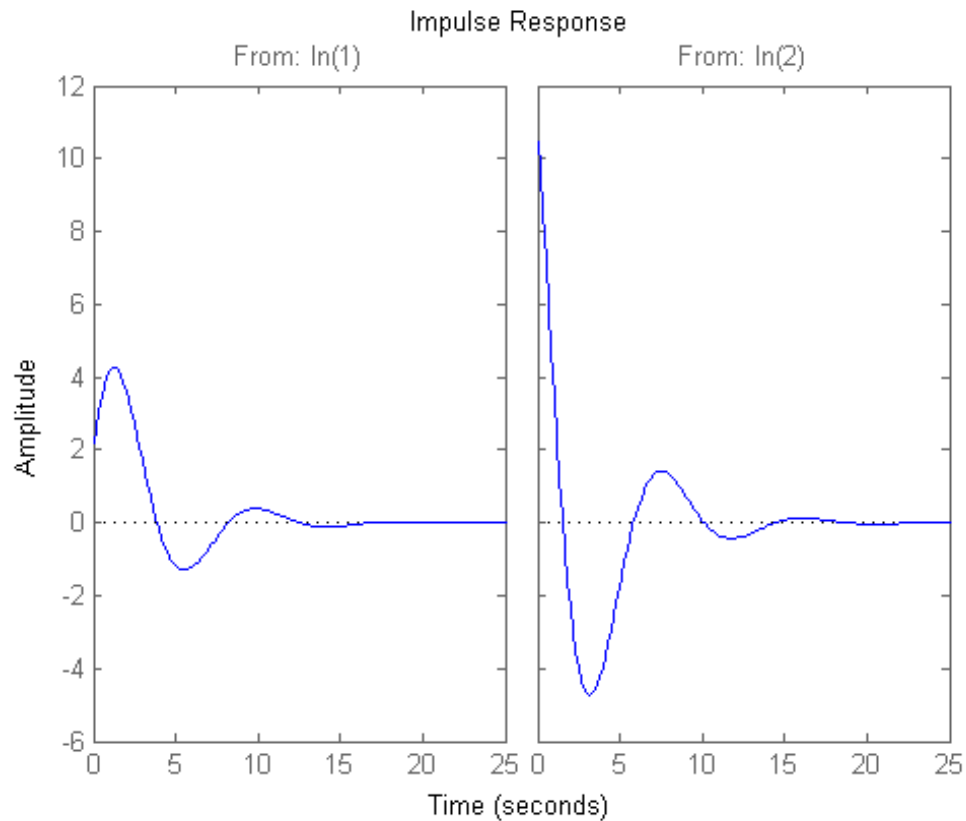
#### Impulse Response Plot of Second-Order State-Space Model

Plot the impulse response of the second-order state-space model

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$
$$y = [1.9691 \quad 6.4493] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

use the following commands.

```
a = [-0.5572 -0.7814;0.7814 0];  
b = [1 -1;0 2];  
c = [1.9691 6.4493];  
sys = ss(a,b,c,0);  
impulse(sys)
```



The left plot shows the impulse response of the first input channel, and the right plot shows the impulse response of the second input channel.

You can store the impulse response data in MATLAB arrays by

```
[y,t] = impulse(sys);
```

Because this system has two inputs, `y` is a 3-D array with dimensions

```
size(y)
```

```
ans =  
  
    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
```

## Example 2

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:')
```

## 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 sampling period is chosen automatically based on the system dynamics, except when a time vector  $t = 0:dt:Tf$  is supplied ( $dt$  is then used as sampling period).

## Limitations

The impulse response of a continuous system with nonzero  $D$  matrix is infinite at  $t = 0$ . `impulse` ignores this discontinuity and returns the lower continuity value  $Cb$  at  $t = 0$ .

## See Also

`ltiview` | `step` | `initial` | `lsim`

# impzplot

---

**Purpose** 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  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling 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  $T_i$ .

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`.

## 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”.

## Examples

### Example 1

Normalize the impulse response of a third-order system.

```
sys = rss(3);  
h = impzplot(sys);  
% Normalize responses  
setoptions(h,'Normalize','on');
```

# impulseplot

---

## 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);
```

## See Also

[getoptions](#) | [impulse](#) | [setoptions](#)

**Purpose**

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  $T_{\text{final}}$  in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling time ( $T_s = -1$ ), `initial` interprets  $T_{\text{final}}$  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  $T_s$  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

---

```
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  $\text{length}(t)$  rows and as many columns as states.

## 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”.

## Examples

Plot the response of the state-space model

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -0.5572 & -0.7814 \\ 0.7814 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
$$y = [1.9691 \quad 6.4493] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

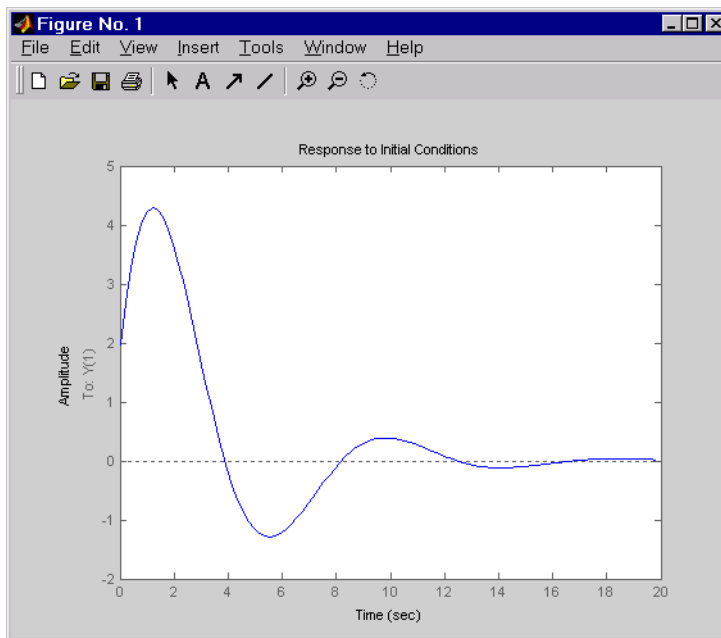
to the initial condition

$$x(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

```
a = [-0.5572 -0.7814; 0.7814 0];
```

```
c = [1.9691 6.4493];  
x0 = [1 ; 0]
```

```
sys = ss(a,[],c,[]);  
initial(sys,x0)
```

**See Also**

[impulse](#) | [lsim](#) | [ltiview](#) | [step](#)

# initialplot

---

**Purpose** 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:  $\dot{x} = A x$ ,  $y = C x$ ,  $x(0) = x_0$

Discrete time:  $x[k+1] = A x[k]$ ,  $y[k] = C x[k]$ ,  $x[0] = x_0$

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  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling 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.
```

## 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”.

## 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.
```

# initialplot

---

## See Also

`getoptions` | `initial` | `setoptions`

**Purpose** 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`

**Purpose** 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

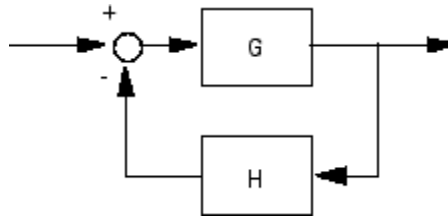
You can verify that

$$H * Hi$$

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

$$\text{inv}(1+g*h) * g$$

this typically leads to nonminimal closed-loop models. For example,

```
g = zpk([],1,1)
h = tf([2 1],[1 0])
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:

$$s (s-1)$$

$$\text{-----}$$
$$(s-1) (s^2 + s + 1)$$

Use feedback to avoid such pitfalls.

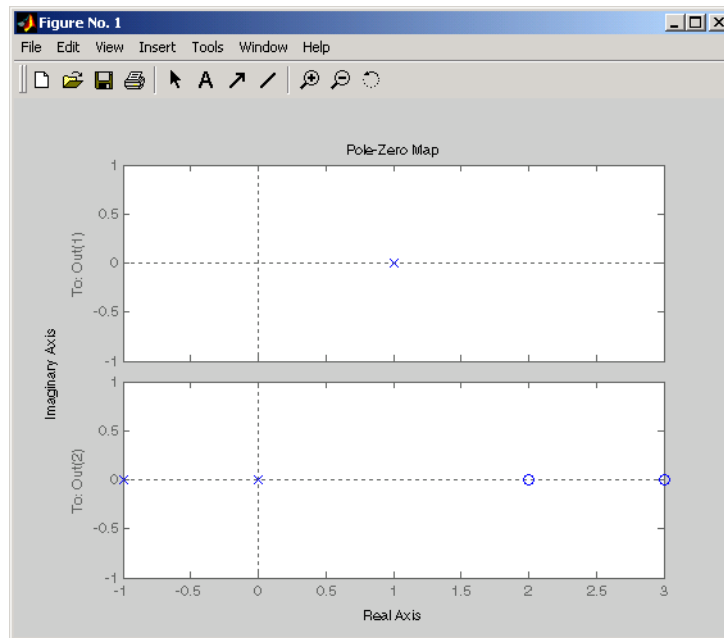
```
cloop = feedback(g,h)
```

Zero/pole/gain:

$$s$$
$$\text{-----}$$
$$(s^2 + s + 1)$$

---

<b>Purpose</b>	Plot pole-zero map for I/O pairs of model
<b>Syntax</b>	<code>iopfzmap(sys)</code> <code>iopfzmap(sys1,sys2,...)</code>
<b>Description</b>	<p><code>iopfzmap(sys)</code> computes and plots the poles and zeros of each input/output pair of the dynamic system model <code>sys</code>. The poles are plotted as x's and the zeros are plotted as o's.</p> <p><code>iopfzmap(sys1,sys2,...)</code> shows the poles and zeros of multiple models <code>sys1,sys2,...</code> on a single plot. You can specify distinctive colors for each model, as in <code>iopfzmap(sys1, 'r', sys2, 'y', sys3, 'g')</code>.</p> <p>The functions <code>sgrid</code> or <code>zgrid</code> can be used to plot lines of constant damping ratio and natural frequency in the <math>s</math> or <math>z</math> plane.</p> <p>For model arrays, <code>iopfzmap</code> plots the poles and zeros of each model in the array on the same diagram.</p>
<b>Tips</b>	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”.
<b>Examples</b>	<b>Example 1</b> Create a one-input, two-output system and plot pole-zero maps for I/O pairs. <pre>H = [tf(-5 , [1 -1]); tf([1 -5 6], [1 1 0])]; iopfzmap(H)</pre>



## Example 2

View the poles and zeros of an over-parameterized state-space model estimated using input-output data.

```
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.

## See Also

[pzmap](#) | [pole](#) | [zero](#) | [sgrid](#) | [zgrid](#) | [iopzplot](#)



**Purpose**

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 LTI 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.

`iopzplot(sys1,sys2,...)` shows the poles and zeros of multiple LTI models `SYS1,SYS2,...` on a single plot. You can specify distinctive colors for each model, as in

```
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.

## 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”.

## Examples

### Example 1

Use the plot handle to change the I/O grouping of a pole/zero map.

```
sys = rss(3,2,2);  
h = iopzplot(sys);  
% View all input-output pairs on a single axis.  
setoptions(h, 'IOGrouping', 'all')
```

### Example 2

View the poles and zeros of an over-parameterized state-space model estimated using input-output data.

```
load iddata1  
sys = ssest(z1,6,ssestOptions('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 1-std 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)  
axis([-20, 10 -30 30])
```

The variability in the pole-zero locations of the second model `sys2` are reduced.

## See Also

`getoptions` | `iopzmap` | `setoptions`

---

<b>Purpose</b>	Determine if dynamic system model is in continuous time
<b>Syntax</b>	<code>bool = isct(sys)</code>
<b>Description</b>	<code>bool = isct(sys)</code> returns a logical value of 1 ( <code>true</code> ) if the dynamic system model <code>sys</code> is a continuous-time model. The function returns a logical value of 0 ( <code>false</code> ) otherwise.
<b>Input Arguments</b>	<b>sys</b> Dynamic system model or array of such models.
<b>Output Arguments</b>	<b>bool</b> Logical value indicating whether <code>sys</code> is a continuous-time model. <code>bool = 1 (true)</code> if <code>sys</code> is a continuous-time model ( <code>sys.Ts = 0</code> ). If <code>sys</code> is a discrete-time model, <code>bool = 0 (false)</code> .  For a static gain, both <code>isct</code> and <code>isdt</code> return <code>true</code> unless you explicitly set the sampling time to a nonzero value. If you do so, <code>isdt</code> returns <code>true</code> and <code>isct</code> returns <code>false</code> .  For arrays of models, <code>bool</code> is <code>true</code> if the models in the array are continuous.
<b>See Also</b>	<code>isdt</code>   <code>isstable</code>

# isdt

---

**Purpose** 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 sampling 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`

**Purpose** Determine whether dynamic system model is empty

**Syntax** `isempty(sys)`

**Description** `isempty(sys)` returns TRUE (logical 1) if the dynamic system model `sys` has no input or no output, and FALSE (logical 0) otherwise. Where `sys` is a FRD model, `isempty(sys)` returns TRUE when the frequency vector is empty. Where `sys` is a model array, `isempty(sys)` returns TRUE 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 TRUE (logical 1) while

```
isempty(ss(1,2,3,4))
```

returns FALSE (logical 0).

**See Also** `issiso` | `size`

# isfinite

---

**Purpose** Determine if model has finite coefficients

**Syntax**  
`B = isfinite(sys)`  
`B = isfinite(sys,'elem')`

**Description** `B = isfinite(sys)` returns 1 (true) if the model `sys` has finite coefficients, and 0 (false) otherwise. If `sys` is a model array, then `B` is true 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.

**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.

**Examples**  
**Check Model for Finite Coefficients**

Create model and check whether its coefficients are all finite.

```
sys = rss(3);  
B = isfinite(sys)
```

```
B =
```

```
1
```

### 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     1     1     1     1
```

When you use the 'elem' input, `isfinite` checks each model individually and returns a logical array indicating which models have all finite coefficients.

**See Also** `isreal`

# isParametric

---

**Purpose** 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 `ltiblock.ss`. If M does not contain parametric Control Design Blocks, `bool = 0` (false).

**See Also** `nblocks`

**How To**

- “Control Design Blocks”
- “Dynamic System Models”
- “Static Models”



<b>Purpose</b>	Determine if dynamic system model is proper
<b>Syntax</b>	<pre>B = isproper(sys) B = isproper(sys,'elem') [B, sysr] = isproper(sys)</pre>
<b>Description</b>	<p><code>B = isproper(sys)</code> returns TRUE (logical 1) if the dynamic system model <code>sys</code> is proper and FALSE (logical 0) otherwise.</p> <p>A proper model has relative degree <math>\leq 0</math> 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.</p> <p>If <code>sys</code> is a model array, then <code>B</code> is TRUE if all models in the array are proper.</p> <p><code>B = isproper(sys,'elem')</code> checks each model in a model array <code>sys</code> and returns a logical array of the same size as <code>sys</code>. The logical array indicates which models in <code>sys</code> are proper.</p> <p>If <code>sys</code> is a proper descriptor state-space model with a non-invertible E matrix, <code>[B, sysr] = isproper(sys)</code> also returns an equivalent model <code>sysr</code> with fewer states (reduced order) and a non-singular E matrix. If <code>sys</code> is not proper, <code>sysr = sys</code>.</p>
<b>Examples</b>	<p><b>Example 1</b></p> <p>The following commands</p> <pre>isproper(tf([1 0],1))           % transfer function s isproper(tf([1 0],[1 1]))      % transfer function s/(s+1)</pre>

return FALSE (logical 0) and TRUE (logical 1), respectively.

## Example 2

Combining state-space models can yield 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
```

```
a =  
      x1    x2    x3    x4  
x1   -2  -2.5  0.5  1.75  
x2    2    0    0    0  
x3    0    0    1    0  
x4    0    0    0    1
```

```
b =  
      u1  
x1    0  
x2    0  
x3    0  
x4   -4
```

```
c =  
      x1    x2    x3    x4  
y1    1  0.5    0    0
```

```
d =  
      u1  
y1    0
```

```
e =  
      x1    x2    x3    x4  
x1    1    0    0    0  
x2    0    1    0    0
```

```

x3    0    0    0    0.5
x4    0    0    0    0

```

H is proper and reducible:

```
[isprop, Hr] = isproper(H)
```

```
isprop =
```

```
1
```

```
a =
```

```

          x1          x2
x1         0    0.1398
x2 -0.06988 -0.0625

```

```
b =
```

```

          u1
x1  -0.125
x2  -0.1398

```

```
c =
```

```

          x1          x2
y1  -0.5   -1.118

```

```
d =
```

```

          u1
y1    1

```

```
e =
```

```

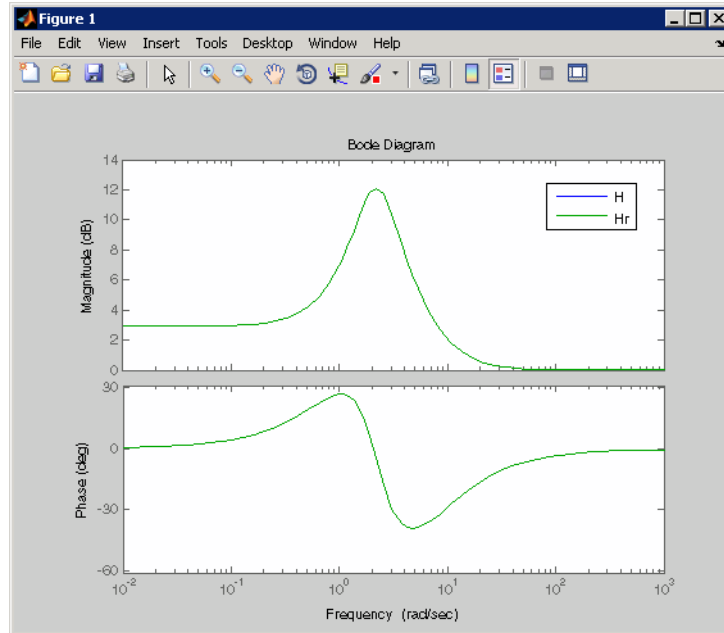
          x1          x2
x1  0.0625    0
x2    0    0.03125

```

Continuous-time model.

H and Hr are equivalent, as a Bode plot demonstrates:

bode(H, Hr)



**See Also**

ss | dss

---

<b>Purpose</b>	Determine if model has real-valued coefficients
<b>Syntax</b>	<pre>B = isreal(sys) B = isreal(sys, 'elem')</pre>
<b>Description</b>	<p><code>B = isreal(sys)</code> returns 1 (true) if the model <code>sys</code> has real-valued coefficients, and 0 (false) otherwise. If <code>sys</code> is a model array, then <code>B</code> is true if all models in <code>sys</code> have real-valued coefficients.</p> <p><code>B = isreal(sys, 'elem')</code> checks each model in the model array <code>sys</code> and returns a logical array of the same size as <code>sys</code>. The logical array indicates which models in <code>sys</code> have real coefficients.</p>
<b>Input Arguments</b>	<p><b>sys - Model or array to check</b> input-output model   model array</p> <p>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.</p>
<b>Output Arguments</b>	<p><b>B - Flag indicating whether model has real-valued coefficients</b> logical   logical array</p> <p>Flag indicating whether model has real-valued coefficients, returned as a logical value or logical array.</p>
<b>Examples</b>	<p><b>Check Model for Real-Valued Coefficients</b></p> <p>Create model and check whether its coefficients are all real-valued.</p> <pre>sys = rss(3); B = isreal(sys)  B =</pre> <p>1</p>

## Check Each Model in Array

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     1     1     1     1
```

When you use the 'elem' input, `isreal` checks each model individually and returns a logical array indicating which models have all real-valued coefficients.

**See Also** `isfinite`

**Purpose** Determine whether system is stable

**Syntax**  
`B = isstable(sys)`  
`B = isstable(sys, 'elem')`

**Description** `B = isstable(sys)` returns 1 (true) if the dynamic system model `sys` has stable dynamics, and 0 (false) otherwise. If `sys` is a model array, then `B` is true if all models in `sys` are stable.

`B = isstable(sys, 'elem')` returns a logical array of the same size 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.

**See Also** pole

# issiso

---

**Purpose** Determine if dynamic system model is single-input/single-output (SISO)

**Syntax** `issiso(sys)`

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

**See Also** `isempty` | `size`



---

<b>Purpose</b>	Determine if model is static or dynamic
<b>Syntax</b>	<pre>B = isstatic(sys) B = isstatic(sys, 'elem')</pre>
<b>Description</b>	<p><code>B = isstatic(sys)</code> returns 1 (true) if the model <code>sys</code> is a static model, and 0 (false) if <code>sys</code> has dynamics such as states or delays. If <code>sys</code> is a model array, then <code>B</code> is true if all models in <code>sys</code> are static.</p> <p><code>B = isstatic(sys, 'elem')</code> checks each model in the model array <code>sys</code> and returns a logical array of the same size as <code>sys</code>. The logical array indicates which models in <code>sys</code> are static.</p>
<b>Input Arguments</b>	<p><b>sys - Model or array to check</b> input-output model   model array</p> <p>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.</p>
<b>Output Arguments</b>	<p><b>B - Flag indicating whether input model is static</b> logical   logical array</p> <p>Flag indicating whether input model is static, returned as a logical value or logical array.</p>
<b>See Also</b>	<code>pole</code>   <code>zero</code>   <code>hasdelay</code>
<b>Concepts</b>	<ul style="list-style-type: none"><li>“Types of Model Objects”</li></ul>

# kalman

---

**Purpose** 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

$$\dot{x} = Ax + Bu + Gw \quad (\text{state equation})$$

$$y = Cx + Du + Hw + v \quad (\text{measurement equation})$$

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

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

$$\begin{bmatrix} \hat{y} \\ \hat{x} \end{bmatrix} = \begin{bmatrix} C \\ I \end{bmatrix} \hat{x} + \begin{bmatrix} D \\ 0 \end{bmatrix} u$$

The filter gain  $L$  is determined by solving an algebraic Riccati equation to be

$$L = (PC^T + \bar{N})\bar{R}^{-1}$$

where

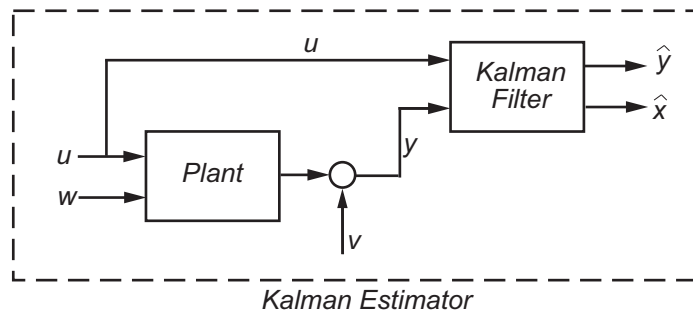
$$\bar{R} = R + HN + N^T H^T + HQH^T$$

$$\bar{N} = G(QH^T + N)$$

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

$$E(w[n]w[n]^T) = Q, \quad E(v[n]v[n]^T) = R, \quad E(w[n]v[n]^T) = N$$

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

$$\bar{R} = R + HN + N^T H^T + HQH^T$$

$$\bar{N} = G(QH^T + N)$$

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

$$\begin{bmatrix} \hat{y}[n | n] \\ \hat{x}[n | n] \end{bmatrix} = \begin{bmatrix} C(I - MC) \\ I - MC \end{bmatrix} \hat{x}[n | n-1] + \begin{bmatrix} (I - CM)D & CM \\ -MD & M \end{bmatrix} \begin{bmatrix} u[n] \\ y[n] \end{bmatrix}$$

where the innovation gain  $M$  is defined as

$$M = PC^T (CPC^T + \bar{R})^{-1}$$

$M$  updates the prediction  $\hat{x}[n | n-1]$  using the new measurement  $y[n]$ .

$$\hat{x}[n | n] = \hat{x}[n | n-1] + M \underbrace{(y[n] - C\hat{x}[n | n-1] - Du[n])}_{\text{innovation}}$$

- 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 1-300). `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 or `sys` are assumed stochastic.

`[kest,L,P,M,Z] = kalman(sys,Qn,Rn,...,type)` specifies the estimator type for discrete-time plants `sys`. The string `type` 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

kalmd | estim | care | dare | lqgreg | lqg | ss

**Purpose** Design discrete Kalman estimator for continuous plant

**Syntax** `kalmd`  
`[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

$$\dot{x} = Ax + Bu + gw \quad (\text{state equation})$$

$$y_v = Cx + Du + v \quad (\text{measurement equation})$$

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

$$Q_d = \int_0^{T_s} e^{A\tau} G Q G^T e^{A^T \tau} d\tau$$

$$R_d = R / T_s$$

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` | `lqgreg` | `lqrd`



**Purpose**

Generalized feedback interconnection of two models (Redheffer star product)

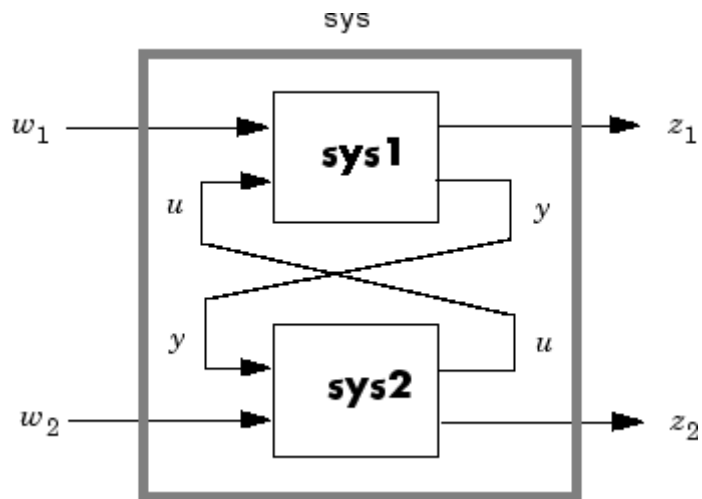
**Syntax**

```
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).



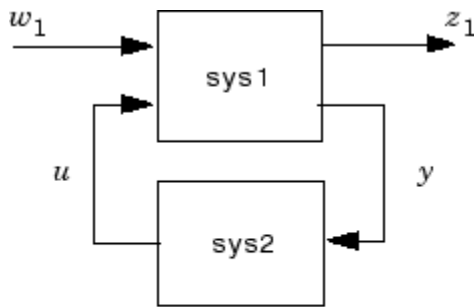
This feedback loop connects the first  $nu$  outputs of `sys2` to the last  $nu$  inputs of `sys1` (signals  $u$ ), and the last  $ny$  outputs of `sys1` to the first  $ny$  inputs of `sys2` (signals  $y$ ). The resulting system `sys` maps the input vector  $[w_1 ; w_2]$  to the output vector  $[z_1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

The abbreviated syntax

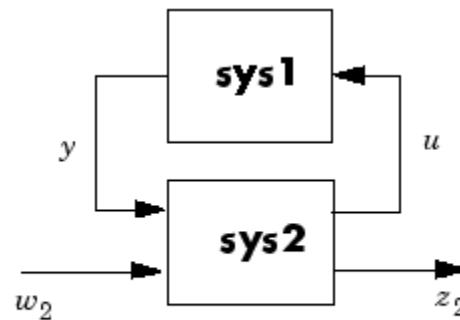
$\text{sys} = \text{lft}(\text{sys1}, \text{sys2})$

produces:

- The lower LFT of  $\text{sys1}$  and  $\text{sys2}$  if  $\text{sys2}$  has fewer inputs and outputs than  $\text{sys1}$ . This amounts to deleting  $w_2$  and  $z_2$  in the above diagram.
- The upper LFT of  $\text{sys1}$  and  $\text{sys2}$  if  $\text{sys1}$  has fewer inputs and outputs than  $\text{sys2}$ . This amounts to deleting  $w_1$  and  $z_1$  in the above diagram.



Lower LFT connection



Upper LFT connection

### Algorithms

The closed-loop model is derived by elementary state-space manipulations.

### Limitations

There should be no algebraic loop in the feedback connection.

### See Also

connect | feedback

**Purpose** Switch for opening and closing feedback loops

**Syntax**  
S = loopswitch(name)  
S = loopswitch(name,N)

**Description** Control Design Block for creating loop-opening locations in a model of a control system. You can combine a `loopswitch` block with numeric LTI models, tunable LTI models, and other Control Design Blocks to build tunable models of control systems. A `loopswitch` block in your control system marks a location where you can optionally open a feedback loop. You can use an optional loop opening to compute open-loop quantities such as point-to-point transfer functions (see `getLoopTransfer`) or stability margins. You can also use a `loopswitch` block to mark an input or output location for computing closed-loop transfer functions with `getIOTransfer`.

`loopswitch` blocks are also useful when tuning a control system using Robust Control Toolbox tuning commands such as `systune`. You can use a `loopswitch` block to mark a loop-opening location for open-loop tuning requirements such as `TuningGoal.LoopShape` or `TuningGoal.Margins`. You can also use a `loopswitch` block to mark the specified input or output for tuning requirements such as `TuningGoal.Gain`.

**Construction** S = `loopswitch(name)` creates a switch block for opening or closing a SISO feedback loop.

S = `loopswitch(name,N)` creates a switch for a MIMO feedback loop with N channels.

### Input Arguments

#### name

Switch block name, specified as a string. This input argument sets the value of the `Name` property of the switch block. (See “Properties” on page 1-310.)

#### N

Number of channels for a multichannel switch, specified as a scalar integer.

## Tips

- By default, the switch `S` is closed. Set `S.Open = true` to create a switch location that is open by default. For a multichannel switch, you can set `S.Open` to a logical vector with `N` entries. Doing so opens only a subset of the feedback loops.

## Properties

### Location

Names of feedback channels in the switch block, specified as a string or a cell array of strings.

By default, the feedback loops are named after the block. For example, if you have a SISO switch block, `X` that has name `'X'`, then `X.Location = 'X'` by default. If you have a multi-channel switch block, then `X.Location = {'X(1)', 'X(2)', ...}` by default. Set `X.Location` to a different value if you want to customize the channel names.

### Open

Switch state, specified as a logical value or vector of logical values. For example, if you have a SISO switch block, `X`, then the value `X.Open = 1` opens the switch. If you have a multi-channel switch block, then `X.Open` is a logical vector with as many entries as `X` has channels.

**Default:** 0 for all channels

### Ts

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

## InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

### **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set `Name` to a string to label the system.

**Default:** ''

## **Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

## **UserData**

Any type of data you wish to associate with system. Set `UserData` to any MATLAB data type.

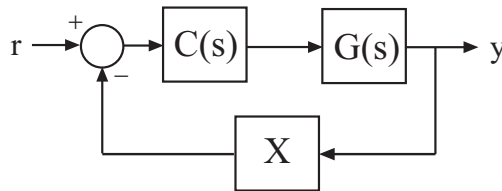
**Default:** []



## Examples

### Feedback Loop with Loop-Opening Switch

Create a model of the following feedback loop with a loop-opening switch in the feedback path.



$G = 1/(s+2)$  is the plant model,  $C$  is a tunable PI controller, and  $X$  is a loop-opening switch.

```

G = tf(1,[1 2]);
C = ltiblock.pid('C','pi');
X = loopswitch('X');
T = feedback(G*C,X);
  
```

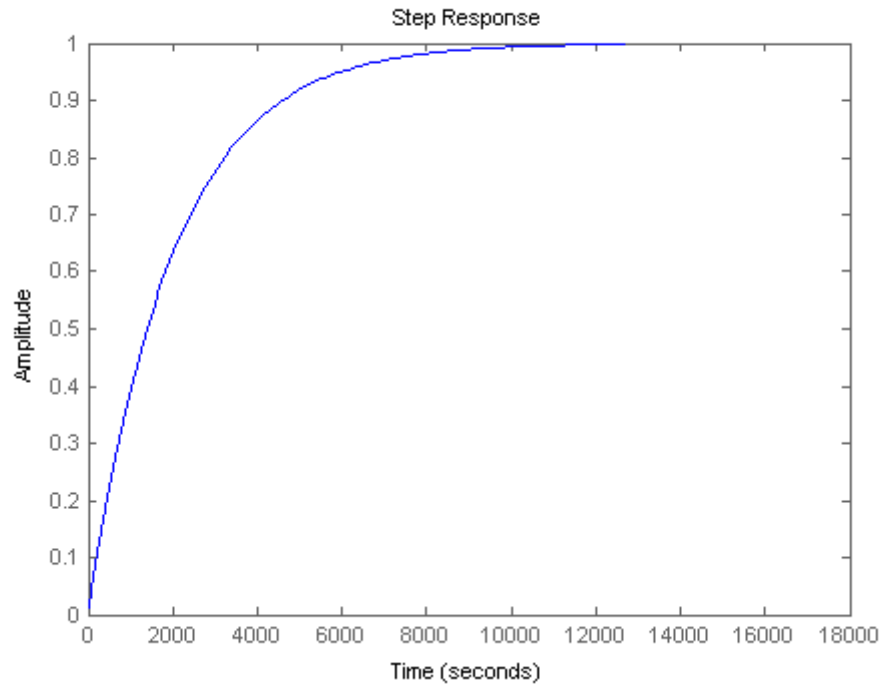
$T$  is a tunable `genss` model. `T.Blocks` contains the Control Design Blocks of the model,  $C$ , and the switch,  $X$ . By default,  $X$  is closed.

Examine the step response of  $T$ .

```
stepplot(T)
```

# loopswitch

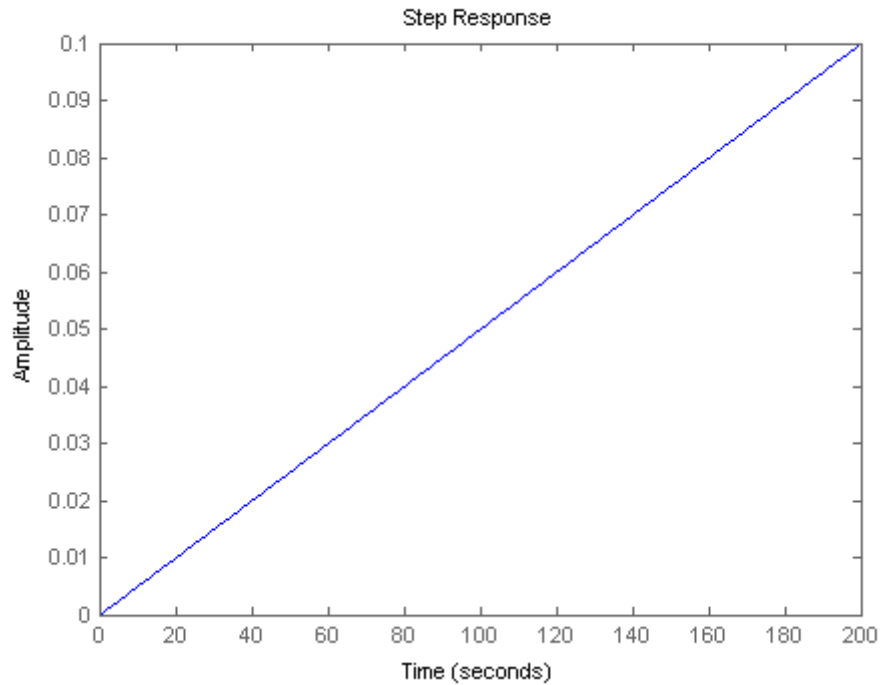
---



With X closed, the response of T is stable.

Open the switch to obtain the open-loop response from  $r$  to  $y$ .

```
T.Blocks.X.Open = true;  
stepplot(T)
```



Close the switch to continue working with the closed-loop model.

```
T.Blocks.X.Open = false;
```

## See Also

`genss` | `getSwitches`

## How To

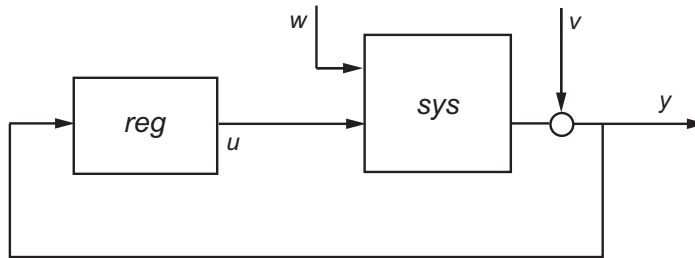
- “Control Design Blocks”
- “Models with Tunable Coefficients”

**Purpose** 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 `sys` 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

$$J = E \left\{ \lim_{\tau \rightarrow \infty} \frac{1}{\tau} \int_0^{\tau} \begin{bmatrix} x^T & u^T \end{bmatrix} Q_{xu} \begin{bmatrix} x \\ u \end{bmatrix} dt \right\}$$

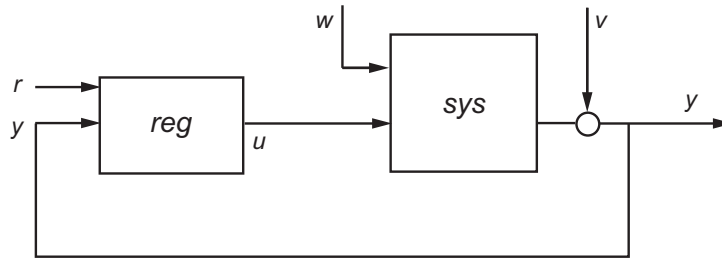
subject to the plant equations

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

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.

## 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).

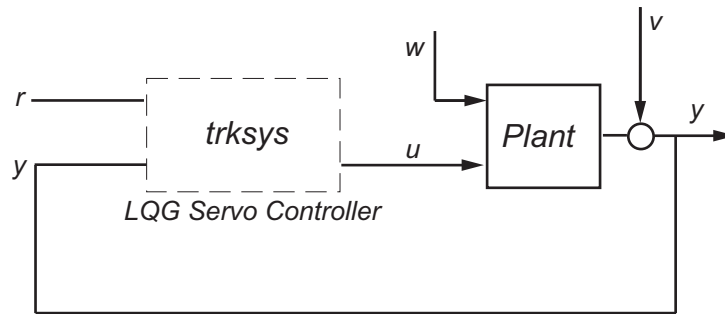
To compute the LQG regulator, `lqg` uses the commands `lqr` and `kalman`. To compute the servo-controller, `lqg` uses the commands `lqi` and `kalman`.

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”.

## 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.

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

$$y = Cx + Du + v$$

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.

## See Also

[lqr](#) | [lqi](#) | [kalman](#) | [lqry](#) | [ss](#) | [care](#) | [dare](#)

# lqgreg

**Purpose** 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.

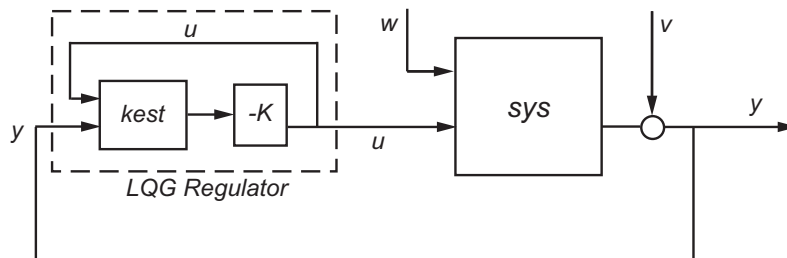
In continuous time, the LQG regulator generates the commands

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

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).

## 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.

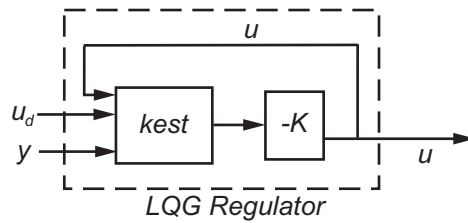
# lqgreg

`r1qg = 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 `r1qg` has  $u_d$  and  $y$  as inputs (see the next figure).

---

**Note** Always use *positive* feedback to connect the LQG regulator to the plant.

---



## Examples

See the example LQG Regulation.

## See Also

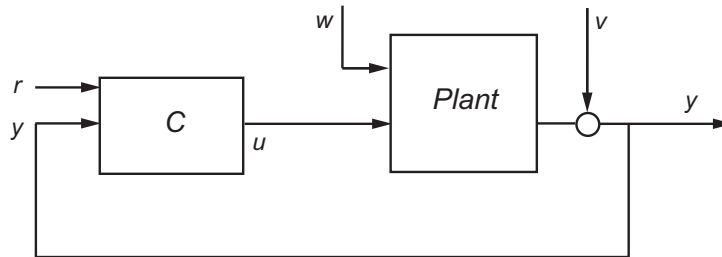
`kalman` | `kalmd` | `lqr` | `dlqr` | `lqrd` | `lqry` | `reg`

**Purpose** 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.

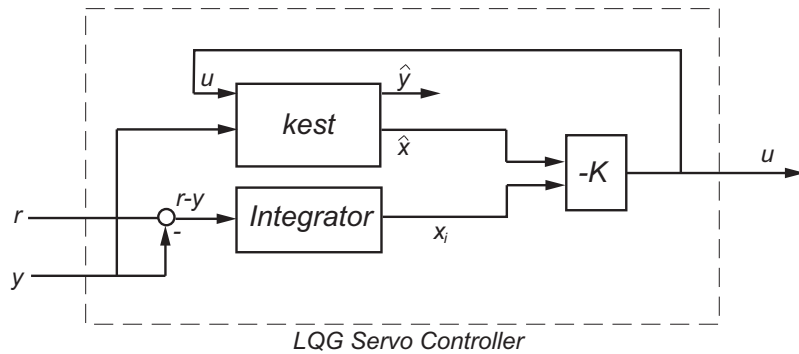



---

**Note** Always use positive feedback to connect the LQG servo controller  $C$  to the plant output  $y$ .

---

$C = \text{lqgtrack}(kest,k)$  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

$$\begin{bmatrix} \dot{\hat{x}} \\ \dot{x}_i \end{bmatrix} = \begin{bmatrix} A - BK_x - LC + LDK_x & -BK_i + LDK_i \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix} + \begin{bmatrix} 0 & L \\ I & -I \end{bmatrix} \begin{bmatrix} r \\ y \end{bmatrix}$$

$$u = [-K_x \quad -K_i] \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix}$$

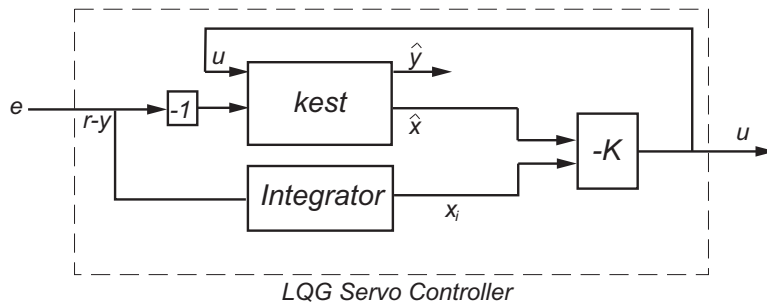
---

**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

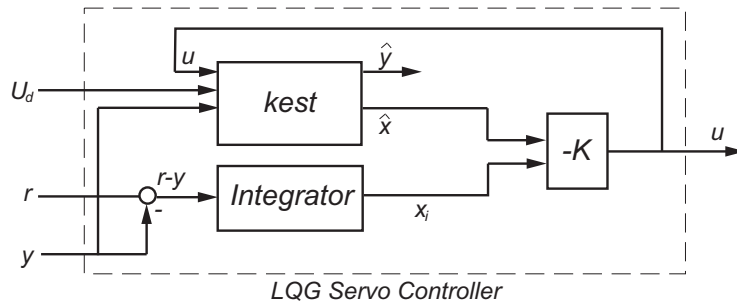
$$\begin{bmatrix} \dot{\hat{x}} \\ \dot{x}_i \end{bmatrix} = \begin{bmatrix} A - BK_x - LC + LDK_x & -BK_i + LDK_i \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix} + \begin{bmatrix} -L \\ I \end{bmatrix} e$$

$$u = [-K_x \quad -K_i] \begin{bmatrix} \hat{x} \\ x_i \end{bmatrix}$$

`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.



## 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).

## Examples

See the example “Design an LQG Servo Controller”.

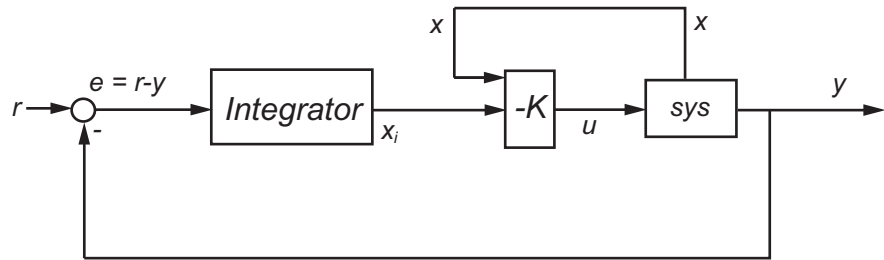
## See Also

`lqg` | `lqi` | `kalman` | `lqgreg` | `lqr`

**Purpose** 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  $r = 0$ )

- $J(u) = \int_0^{\infty} \{z^T Qz + u^T Ru + 2z^T Nu\} dt$  for continuous time

- $J(u) = \sum_{k=0}^{\infty} \{z^T Q z + u^T R u + 2z^T N u\}$  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 sampling 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  $e$ .

## 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

$$\frac{dx}{dt} = E^{-1}Ax + E^{-1}Bu$$

## 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).

---

**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 | lqgtrack | lqg | care | dare

**Purpose** 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  $e = \text{eig}(A - B^*K)$ .  $K$  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  $x[n + 1] = Ax[n] + Bu[n]$ .

$[K, S, e] = \text{LQR}(A, B, Q, R, N)$  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.

**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:

$$\frac{dx}{dt} = E^{-1}Ax + E^{-1}Bu$$

**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).

**See Also**

care | dlqr | lqgreg | lqrd | lqry | lqi

# lqrd

---

**Purpose** 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  $e = \text{eig}(Ad - Bd * Kd)$ .

`[Kd,S,e] = lqrd(A,B,Q,R,N,Ts)` solves the more general problem with a cross-coupling term in the cost function.

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



## 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

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

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

the discretized plant has equations

$$x[n+1] = A_d x[n] + B_d u[n]$$

and the weighting matrices for the equivalent discrete cost function are

$$\begin{bmatrix} Q_d & N_d \\ N_d^T & R_d \end{bmatrix} = \int_0^{T_s} \begin{bmatrix} \Phi^T(\tau) & 0 \\ \Gamma^T(\tau) & I \end{bmatrix} \begin{bmatrix} Q & N \\ N^T & R \end{bmatrix} \begin{bmatrix} \Phi(\tau) & \Gamma(\tau) \\ 0 & I \end{bmatrix} d\tau$$

The integrals are computed using matrix exponential formulas due to Van Loan (see [2]). The plant is discretized using `c2d` and the gain matrix is computed from the discretized data using `dlqr`.

## Limitations

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

## 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`

**Purpose** Form linear-quadratic (LQ) state-feedback regulator with output weighting

**Syntax** `[K,S,e] = lqry(sys,Q,R,N)`

**Description** Given the plant

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

or its discrete-time counterpart, `lqry` designs a state-feedback control

$$u = -Kx$$

that minimizes the quadratic cost function with output weighting

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

(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`

**Purpose**

Simulate time response of dynamic system to arbitrary inputs

**Syntax**

```
lsim
lsim(sys,u,t)
lsim(sys,u,t,x0)
lsim(sys,u,t,x0,'zoh')
lsim(sys,u,t,x0,'foh')
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 time 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 matrix `u` must have as many rows as time samples (`length(t)`) and as many columns as system inputs. Each row `u(i,:)` specifies the input value(s) at the time sample `t(i)`.

The LTI model `sys` can be continuous or discrete, SISO or MIMO. In discrete time, `u` must be sampled at the same rate as the system (`t` is then redundant and can be omitted or set to the empty matrix). In continuous time, the time sampling `dt=t(2)-t(1)` is used to discretize the continuous model. If `dt` is too large (undersampling), `lsim` issues a warning suggesting that you use a more appropriate sample time, but will use the specified sample time. See “Algorithms” on page 1-344 for a discussion of sample times.

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

`lsim(sys,u,t,x0,'zoh')` or `lsim(sys,u,t,x0,'foh')` explicitly specifies how the input values should be interpolated between samples

(zero-order hold or linear interpolation). By default, `lsim` selects the interpolation method automatically based on the smoothness of the signal `U`.

Finally,

```
lsim(sys1,sys2,...,sysN,u,t)
```

simulates the responses of several LTI models to the same input history `t,u` and plots these responses on a single figure. As with `bode` or `plot`, you can specify a particular color, linestyle, and/or marker for each system, for example,

```
lsim(sys1,'y:',sys2,'g--',u,t,x0)
```

The multisystem behavior is similar to that of `bode` or `step`.

When invoked with left-hand arguments,

```
[y,t] = lsim(sys,u,t)
[y,t,x] = lsim(sys,u,t)           % for state-space models only
[y,t,x] = lsim(sys,u,t,x0)       % with initial state
```

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x` (for state-space models only). No plot is drawn on the screen. The matrix `y` has as many rows as time samples (`length(t)`) and as many columns as system outputs. The same holds for `x` with "outputs" replaced by states.

`lsim(sys)` opens the Linear Simulation Tool GUI. For more information about working with this GUI, see [Working with the Linear Simulation Tool](#).

## Examples

### Example 1

Simulate and plot the response of the system

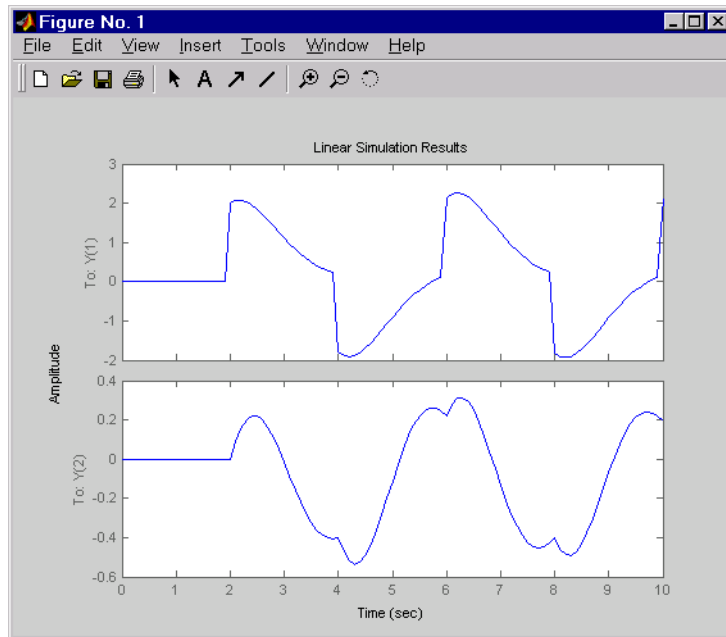
$$H(s) = \left[ \begin{array}{c} \frac{2s^2 + 5s + 1}{s^2 + 2s + 3} \\ \frac{s - 1}{s^2 + s + 5} \end{array} \right]$$

to a square wave with period of four seconds. First generate the square wave with gensig. Sample every 0.1 second during 10 seconds:

```
[u,t] = gensig('square',4,10,0.1);
```

Then simulate with lsim.

```
H = [tf([2 5 1],[1 2 3]) ; tf([1 -1],[1 1 5])]
lsim(H,u,t)
```



## Example 2

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', 'dcmotor'))
z = iddata(y, u, 0.1, 'Name', 'DC-motor');
```

```
[sys, x0] = n4sid(z, 4);
[y,t,x] = lsim(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')
```

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

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

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

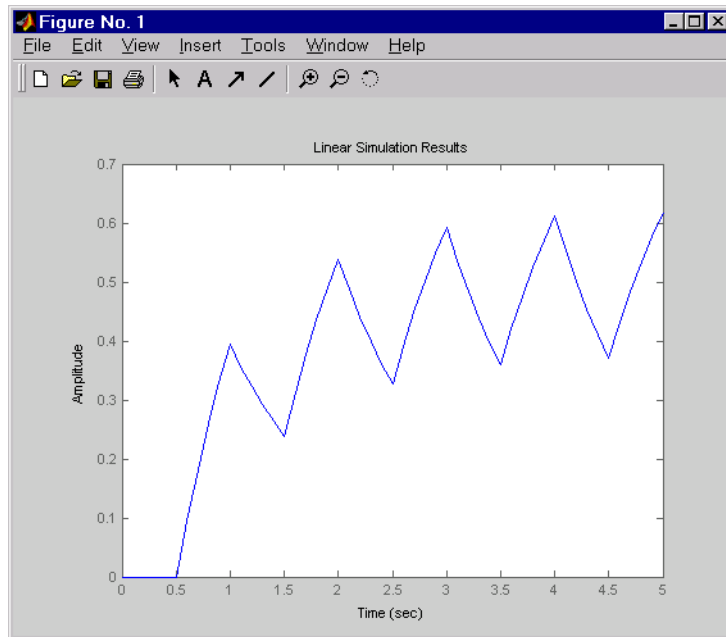
To simulate its response to a square wave with period 1 second, you can proceed as follows:

```
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, gives this warning

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

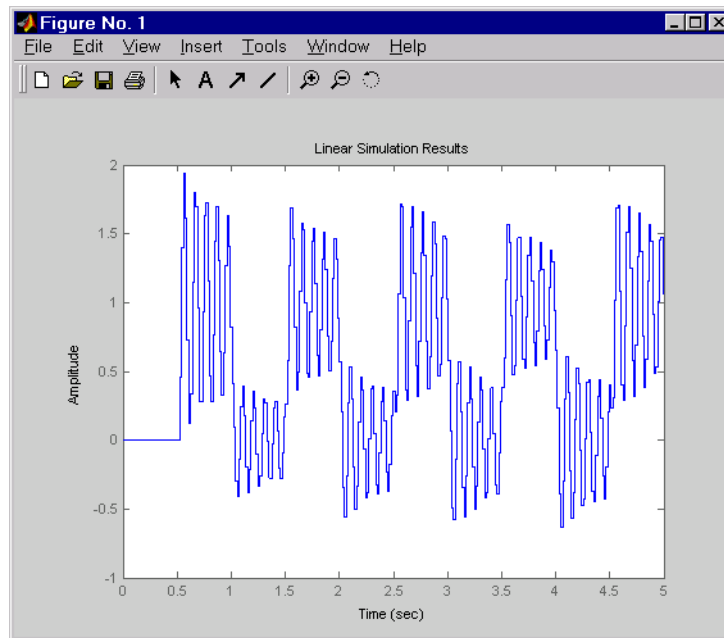


and produces this plot.

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

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

# lsim



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

## See Also

`gensig` | `impulse` | `initial` | `ltiview` | `step` | `sim` | `lsiminfo`



**Purpose** 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  $|y(t) - y_{\text{final}}|$  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 =
```

# lsiminfo

---

```
SettlingTime: 22.8626  
Min: -0.4270  
MinTime: 2.0309  
Max: 0.2845  
MaxTime: 4.0619
```

## See Also

`lsim` | `impulse` | `initial` | `stepinfo`

**Purpose** 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

---

```
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

[getoptions](#) | [lsim](#) | [setoptions](#)

**Purpose** Tunable static gain block

**Syntax**  
`blk = ltiblock.gain(name,Ny,Nu)`  
`blk = ltiblock.gain(name,G)`

**Description** Model object for creating tunable static gains. `ltiblock.gain` lets you parametrize tunable static gains for parameter studies or for automatic tuning with Robust Control Toolbox tuning commands such as `systemtune` or `looptune`.

`ltiblock.gain` is part of the Control Design Block family of parametric models. Other Control Design Blocks include `ltiblock.pid`, `ltiblock.ss`, and `ltiblock.tf`.

**Construction** `blk = ltiblock.gain(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 = ltiblock.gain(name,G)` uses the double array `G` to dimension the block and initialize the tunable parameters.

## Input Arguments

### **name**

String specifying the block Name. (See “Properties” on page 1-352.)

### **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**

# ltiblock.gain

---

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.

## 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 an `ltiblock.gain` parametric model to a numeric (non-tunable) model object, use model commands such as `tf`, `zpk`, or `ss`.

## 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

Field	Description
	<p><code>blk.Gain.Free</code>. Default: Array of zero values.</p>
Free	<p>Array of logical values determining whether the gain entries in <code>blk.Gain.Value</code> are fixed or free parameters.</p> <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 (true) 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. 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. Default: <code>Inf</code>.</p>

## **Ts**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'



Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### InputName

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string '' for all input channels

### **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set `Name` to a string to label the system.

**Default:** ''

### **Notes**

Any text that you want to associate with the system. Set Notes to a string or a cell array of strings.

**Default:** {}

## UserData

Any type of data you wish to associate with system. Set UserData to 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 = ltiblock.gain('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 = ltiblock.gain('gainblock',G);
```

---

Create a 2-by-2 parametric gain block and assign names to the inputs.

```
blk = ltiblock.gain('gainblock',2,2)    % 2 outputs, 2 inputs
blk.InputName = {'Xerror','Yerror'}    % assign input names
```

## See Also

[ltiblock.pid](#) | [ltiblock.pid2](#) | [ltiblock.ss](#) | [ltiblock.tf](#) | [genss](#)  
| [systune](#) | [looptune](#) | [hinfstruct](#)

## How To

- “Control Design Blocks”
- “Models with Tunable Coefficients”

# ltiblock.pid

---

**Purpose** Tunable PID controller

**Syntax**  
`blk = ltiblock.pid(name,type)`  
`blk = ltiblock.pid(name,type,Ts)`  
`blk = ltiblock.pid(name,sys)`

**Description** Model object for creating tunable one-degree-of-freedom PID controllers. `ltiblock.pid` lets you parametrize a tunable SISO PID controller for parameter studies or for automatic tuning with requires Robust Control Toolbox tuning commands such as `systune`, `looptune`, or `hinfstruct`.

`ltiblock.pid2` is part of the family of parametric Control Design Blocks. Other parametric Control Design Blocks include `ltiblock.gain`, `ltiblock.ss`, and `ltiblock.tf`.

**Construction** `blk = ltiblock.pid(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 string `type` sets the controller type by fixing some of these values to zero (see “Input Arguments” on page 1-361).

`blk = ltiblock.pid(name,type,Ts)` creates a discrete-time PID controller with sampling 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 1-362).

`blk = ltiblock.pid(name,sys)` uses the dynamic system model, `sys`, to set the sampling time, `Ts`, and the initial values of the parameters `Kp`, `Ki`, `Kd`, and `Tf`.

### Input Arguments

#### **name**

PID controller Name, specified as a string. (See “Properties” on page 1-362.)

#### **type**

String specifying controller type. Specifying a controller type fixes up to three of the PID controller parameters. `type` can take the following values:

String	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
'PD'	Proportional-derivative with first-order filter on derivative action	$K_i$ is fixed to zero; $K_p$ , $K_d$ , and $T_f$ are free
'PID'	Proportional-integral-derivative with first-order filter on derivative action	$K_p$ , $K_i$ , $K_d$ , and $T_f$ are free

#### **Ts**

Sampling time, specified as a scalar.

#### **sys**

Dynamic system model representing a PID controller.

## Properties

### Kp, Ki, Kd, Tf

Parametrization of the PID gains  $K_p$ ,  $K_i$ ,  $K_d$ , and filter time constant  $T_f$  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 <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>
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 $K_p$ 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.Tf.Maximum = 100</code> 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**

Strings setting the discrete integrator formulas  $IF(z)$  and  $DF(z)$  for the integral and derivative terms, respectively. IFormula and DFormula can have the following values:

String	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**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string `''` for all input channels

### InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string `''` 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

## OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set Name to a string to label the system.

**Default:** ''

### Notes

Any text that you want to associate with the system. Set Notes to a string or a cell array of strings.

**Default:** {}

### UserData

Any type of data you wish to associate with system. Set UserData to 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 = ltiblock.pid('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
```

---

### 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 = ltiblock.pid('piblock',C);
```

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 = ltiblock.pid('pidblock','pid')
blk.InputName = {'error'}      % assign input name
blk.OutputName = {'control'}   % assign output name
```

## 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 an `ltiblock.pid` 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.

## See Also

[ltiblock.pid2](#) | [ltiblock.ss](#) | [ltiblock.tf](#) | [systune](#) | [looptune](#)  
| [hinfstruct](#) | [getValue](#)

## How To

- “Control Design Blocks”
- “Models with Tunable Coefficients”

# ltiblock.pid2

---

**Purpose** Tunable two-degree-of-freedom PID controller

**Syntax**  
`blk = ltiblock.pid2(name,type)`  
`blk = ltiblock.pid2(name,type,Ts)`  
`blk = ltiblock.pid2(name,sys)`

**Description** Model object for creating tunable two-degree-of-freedom PID controllers. `ltiblock.pid2` 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 Robust Control Toolbox tuning commands such as `sysstune`, `looptune`, or `hinfstruct`.

`ltiblock.pid2` is part of the family of parametric Control Design Blocks. Other parametric Control Design Blocks include `ltiblock.gain`, `ltiblock.ss`, and `ltiblock.tf`.

**Construction** `blk = ltiblock.pid2(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 string **type** sets the controller type by fixing some of these values to zero (see “Input Arguments” on page 1-371).

`blk = ltiblock.pid2(name,type,Ts)` creates a discrete-time PID controller with sampling time **Ts**. 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 1-372).

`blk = ltiblock.pid2(name,sys)` uses the dynamic system model, **sys**, to set the sampling time, **Ts**, 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 string. (See “Properties” on page 1-372.)

### **type**

Controller type, specified as a string. Specifying a controller type fixes up to three of the PID controller parameters. **type** can take the following values:

# ltiblock.pid2

String	Controller Type	Effect on PID Parameters
'P'	Proportional only	Ki and Kd are fixed to zero; Tf is fixed to 1; Kp is free
'PI'	Proportional-integral	Kd is fixed to zero; Tf is fixed to 1; Kp and Ki are free
'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**

Sampling 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 `systemtune`:

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>
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

Strings setting the discrete integrator formulas  $IF(z)$  and  $DF(z)$  for the integral and derivative terms, respectively. IFormula and DFormula can have the following values:

String	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**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

## InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string '' for all input channels

### OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set `Name` to a string to label the system.

**Default:** ''

## **Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

## **UserData**

Any type of data you wish to associate with system. Set `UserData` to 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 = ltiblock.pid2('pdblock','PD');  
blk.b.Value = 1;  
blk.c.Value = 0.5;  
blk.Tf.Value = 0.01;  
blk.Tf.Free = false;
```

---

## 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 = ltiblock.pid2('2dofPI',sys);
```

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 = ltiblock.pid2('pidblock','pid');  
blk.InputName = {'reference','measurement'};  
blk.OutputName = {'control'};
```

blk.InputName is a cell array containing two strings, because a two-degree-of-freedom PID controller has two inputs.

# ltiblock.pid2

---

## 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 `ltiblock.pid2` 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.

## See Also

`ltiblock.pid` | `ltiblock.ss` | `ltiblock.tf` | `system` | `looptune` | `hinfstruct` | `getValue`

## How To

- “Control Design Blocks”
- “Models with Tunable Coefficients”

**Purpose** Tunable fixed-order state-space model

**Syntax**

```
blk = ltiblock.ss(name,Nx,Ny,Nu)
blk = ltiblock.ss(name,Nx,Ny,Nu,Ts)
blk = ltiblock.ss(name,sys)
blk = ltiblock.ss(...,Astruct)
```

**Description** Model object for creating tunable fixed-order state-space models. `ltiblock.ss` lets you parametrize a state-space model of a given order for parameter studies or for automatic tuning with Robust Control Toolbox tuning commands such as `systune` or `looptune`.

`ltiblock.ss` is part of the Control Design Block family of parametric models. Other Control Design Blocks include `ltiblock.pid`, `ltiblock.gain`, and `ltiblock.tf`.

**Construction**

`blk = ltiblock.ss(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 = ltiblock.ss(name,Nx,Ny,Nu,Ts)` creates a discrete-time parametric state-space model with sampling time `Ts`.

`blk = ltiblock.ss(name,sys)` uses the dynamic system `sys` to dimension the parametric state-space model, set its sampling time, and initialize the tunable parameters.

`blk = ltiblock.ss(...,Astruct)` creates a parametric state-space model whose  $A$  matrix is restricted to the structure specified in `Astruct`.

### Input Arguments

#### **name**

String specifying the Name of the parametric state-space model `blk`. (See “Properties” on page 1-383.)

#### **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 sampling time.

### **Astruct**

String specifying constraints on the form of the `A` matrix of the parametric state-space model `blk`. `Astruct` can take the following values:

<b>String</b>	<b>Structure of A matrix</b>
'tridiag'	<code>A</code> is tridiagonal. In tridiagonal form, <code>A</code> 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 <code>A</code> are fixed to zero.
'full'	<code>A</code> is full (every entry in <code>A</code> is a free parameter).
'companion'	<code>A</code> is in companion form. In companion form, the characteristic polynomial of the system appears explicitly

String	Structure of A matrix
	in the rightmost column of the A matrix. See canon 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, sampling time, and initial values of the parameters of `blk`. To obtain the dimensions and initial parameter values, `ltiblock.ss` converts `sys` to a state-space model with the structure specified in `Astruct`. If you omit `Astruct`, `ltiblock.ss` converts `sys` into tridiagonal state-space form.

## 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 an `ltiblock.ss` parametric model to a numeric (non-tunable) model object, use model commands such as `ss`, `tf`, or `zpk`.

## 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> .
Free	2-D array of logical values determining whether the corresponding state-space matrix parameters are fixed or free parameters. For example: <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> Defaults: By default, all entries in <code>b</code> , <code>c</code> , and <code>c</code> are tunable. The default free entries in <code>a</code> depend upon the value of <code>Astruct</code> :

Field	Description
	<ul style="list-style-type: none"> <li>• 'tridiag' — entries on the three diagonals of <code>blk.a.Free</code> are 1; the rest are 0.</li> <li>• 'full' — all entries in <code>blk.a.Free</code> are 0.</li> <li>• 'companion' —  <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	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 A matrix remains positive. Default: -Inf.
Maximum	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 A matrix remains negative. Default: Inf.

**StateName**

State names. For first-order models, set `StateName` to a string. For models with two or more states, set `StateName` to a cell array of strings. Use an empty string '' for unnamed states.

**Default:** Empty string '' for all states

## StateUnit

State units. Use `StateUnit` to keep track of the units each state is expressed in. For first-order models, set `StateUnit` to a string. For models with two or more states, set `StateUnit` to a cell array of strings. `StateUnit` has no effect on system behavior.

**Default:** Empty string '' for all states

## Ts

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## TimeUnit

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'



- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

### **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input 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. Set Name to a string to label the system.

**Default:** ''

## Notes

Any text that you want to associate with the system. Set Notes to a string or a cell array of strings.

**Default:** {}

## UserData

Any type of data you wish to associate with system. Set UserData to any MATLAB data type.

**Default:** []

## Examples

Create a parametrized 5th-order SISO model with zero D matrix.

```
blk = ltiblock.ss('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 = ltiblock.ss('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 = ltiblock.ss('ssblock',5,2,2) % 5 states, 2 outputs, 2 inputs  
blk.InputName = {'Xerror','Yerror'} % assign input names
```

**See Also**

ltiblock.pid | ltiblock.pid2 | ltiblock.ss | ltiblock.tf | genss  
| systune | looptune | hinfstruct

**How To**

- “Control Design Blocks”
- “Models with Tunable Coefficients”

# ltiblock.tf

---

**Purpose** Tunable transfer function with fixed number of poles and zeros

**Syntax**

```
blk = ltiblock.tf(name,Nz,Np)
blk = ltiblock.tf(name,Nz,Np,Ts)
blk = ltiblock.tf(name,sys)
```

**Description** Model object for creating tunable SISO transfer function models of fixed order. `ltiblock.tf` lets you parametrize a transfer function of a given order for parameter studies or for automatic tuning with Robust Control Toolbox tuning commands such as `systune` or `looptune`.

`ltiblock.tf` is part of the Control Design Block family of parametric models. Other Control Design Blocks include `deltiblock.pid`, `ltiblock.ss`, and `ltiblock.gain`.

**Construction** `blk = ltiblock.tf(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 = ltiblock.tf(name,Nz,Np,Ts)` creates a discrete-time parametric transfer function with sampling time `Ts`.

`blk = ltiblock.tf(name,sys)` uses the `tf` model `sys` to set the number of poles, number of zeros, sampling time, and initial parameter values.

## Input Arguments

**name**

String specifying the Name of the parametric transfer function `blk`.  
(See “Properties” on page 1-393.)

**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 sampling time.

**sys**

`tf` model providing number of poles, number of zeros, sampling time, and initial values of the parameters of `blk`.

**Tips**

- To convert an `ltiblock.tf` parametric model to a numeric (non-tunable) model object, use model commands such as `tf`, `zpk`, or `ss`.

**Properties****num, den**

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.num` and `blk.den` 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.num` and `blk.den` 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.num.Value</code> has length <math>Nz + 1</math>. <code>blk.den.Value</code> has length <math>Np + 1</math>. The leading coefficient of the denominator (<code>blk.den.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.num.Free(j) = 1</code>, then <code>blk.num.Value(j)</code> is tunable.</li> <li>• If <code>blk.num.Free(j) = 0</code>, then <code>blk.num.Value(j)</code> is fixed.</li> </ul> <p>Default: <code>blk.den.Free(1) = 0</code>; all other entries are 1.</p>



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.num.Minimum(1) = 0</code> ensures that the leading coefficient of the numerator remains positive. Default: <code>-Inf</code> .
Maximum	Maximum value of the parameter. This property places an upper bound on the tuned value of the parameter. For example, setting <code>blk.num.Maximum(1) = 1</code> ensures that the leading coefficient of the numerator does not exceed 1. Default: <code>Inf</code> .

**Ts**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

**TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string `''` for all input channels

### InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string `''` 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

## OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input 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. Set `Name` to a string to label the system.

**Default:** ''

### **Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

### **UserData**

Any type of data you wish to associate with system. Set `UserData` to 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

# ltiblock.tf

---

one integrator regardless of the parameter values, fix the lowest-order coefficient of the denominator to zero.

```
blk = ltiblock.tf('tfblock',2,4); % two zeros, four poles
blk.den.Value(end) = 0; % set last denominator entry to zero
blk.den.Free(end) = 0; % fix it to zero
```

---

Create a parametric transfer function, and assign names to the input and output.

```
blk = ltiblock.tf('tfblock',2,3);
blk.InputName = {'error'}; % assign input name
blk.OutputName = {'control'}; % assign output name
```

## See Also

[ltiblock.pid](#) | [ltiblock.pid2](#) | [ltiblock.ss](#) | [ltiblock.tf](#) | [genss](#)  
| [systune](#) | [looptune](#) | [hinfstruct](#)

## How To

- “Control Design Blocks”
- “Models with Tunable Coefficients”

**Purpose**

LTI Viewer for LTI system response analysis

**Syntax**

```
ltiview
ltiview(sys1,sys2,...,sysn)
ltiview(plottype,sys)
ltiview(plottype,sys,extras)
ltiview('clear',viewers)
ltiview('current',sys1,sys2,...,sysn,viewers)
ltiview(plottype,sys1,sys2,...sysN)
ltiview(plottype,sys1,PlotStyle1,sys2,PlotStyle2,...)
ltiview(plottype,sys1,sys2,...sysN,extras)
```

**Description**

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

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

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

`ltiview(plottype,sys)` initializes an LTI Viewer containing the LTI response type indicated by *plottype* for the LTI model `sys`. The string *plottype* can be any one of the following:

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

or,

*plottype* can be a cell vector containing up to six of these plot types. For example,

```
ltiview({'step';'nyquist'},sys)
```

displays the plots of both of these response types for a given system *sys*.

`ltiview(plottype,sys,extras)` allows the additional input arguments supported by the various LTI model response functions to be passed to the `ltiview` command.

*extras* is one or more input arguments as specified by the function named in *plottype*. These arguments may be required or optional, depending on the type of LTI response. For example, if *plottype* is 'step' then *extras* may be the desired final time, *Tfinal*, as shown below.

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

However, if *plottype* is 'initial', the *extras* arguments must contain the initial conditions *x0* and may contain other arguments, such as *Tfinal*.

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

See the individual references pages of each possible *plottype* commands for a list of appropriate arguments for *extras*.

`ltiview('clear',viewers)` clears the plots and data from the LTI Viewers with handles *viewers*.

`ltiview('current',sys1,sys2,...,sysn,viewers)` adds the responses of the systems *sys1,sys2,...,sysn* to the LTI Viewers with handles *viewers*. If these new systems do not have the same I/O dimensions as those currently in the LTI Viewer, the LTI Viewer is first cleared and only the new responses are shown.



`ltiview(plottype, sys1, sys2, ... sysN)` initializes an LTI Viewer containing the responses of multiple LTI models.

`ltiview(plottype, sys1, PlotStyle1, sys2, PlotStyle2, ...)` initializes the viewer with specified plot styles. See the individual reference pages of the LTI response functions for more information on specifying plot styles.

`ltiview(plottype, sys1, sys2, ... sysN, extras)` initializes the viewer for multiple models using the `extras` input arguments.

## See Also

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

**Purpose** 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.

**Algorithms** `lyap` first transforms the  $A$  and  $B$  matrices to complex Schur form, and then computes the solution of the resulting triangular system. Finally it transforms this solution back[1].

lyap uses SLICOT routines SB03MD and SG03AD for Lyapunov equations and SB04MD (SLICOT) and ZTRSYL (LAPACK) for Sylvester equations.

**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.

$$A = [1 \ 2; -3 \ -4];$$

$$Q = [3 \ 1; 1 \ 1];$$

$$X = \text{lyap}(A,Q)$$

These commands return the following  $X$  matrix:

$X =$

$$\begin{array}{cc} 6.1667 & -3.8333 \\ -3.8333 & 3.0000 \end{array}$$

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

$$A = 5 \quad B = \begin{bmatrix} 4 & 3 \\ 4 & 3 \end{bmatrix} \quad C = [2 \ 1]$$

```
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

## References

[1] Bartels, R.H. and G.W. Stewart, "Solution of the Matrix Equation  $AX + XB = C$ ," *Comm. of the ACM*, Vol. 15, No. 9, 1972.

- 
- [2] Bryson, A.E. and Y.C. Ho, *Applied Optimal Control*, Hemisphere Publishing, 1975. pp. 328–338.
- [3] Barraud, A.Y., “A numerical algorithm to solve  $A X A - X = Q$ ,” *IEEE Trans. Auto. Contr.*, AC-22, pp. 883–885, 1977.
- [4] Hammarling, S.J., “Numerical solution of the stable, non-negative definite Lyapunov equation,” *IMA J. Num. Anal.*, Vol. 2, pp. 303–325, 1982.
- [5] 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.
- [6] Penzl, T., “Numerical solution of generalized Lyapunov equations,” *Advances in Comp. Math.*, Vol. 8, pp. 33–48, 1998.
- [7] 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.

**See Also**

covar | dlyap

# lyapchol

---

**Purpose** Square-root solver for continuous-time Lyapunov equation

**Syntax**  
 $R = \text{lyapchol}(A,B)$   
 $X = \text{lyapchol}(A,B,E)$

**Description**  $R = \text{lyapchol}(A,B)$  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.

$X = \text{lyapchol}(A,B,E)$  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.

**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`

<b>Purpose</b>	Convert magnitude to decibels (dB)
<b>Syntax</b>	<code>ydb = mag2db(y)</code>
<b>Description</b>	<code>ydb = mag2db(y)</code> returns the corresponding decibel (dB) value <i>ydb</i> for a given magnitude <i>y</i> . The relationship between magnitude and decibels is $ydb = 20 \log_{10}(y)$ .
<b>See Also</b>	<code>db2mag</code>

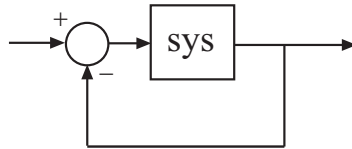
# margin

---

**Purpose** 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.  $W_{pm}$  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  $P_m$  is in degrees. The gain margin  $G_m$  is an absolute magnitude. You can compute the gain margin in dB by

$$G_m_{dB} = 20 \cdot \log_{10}(G_m)$$

`[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.

`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 =
```

$$\frac{0.04798 z + 0.0464}{z^2 - 1.81 z + 0.9048}$$

```
Sample time: 0.1 seconds
Discrete-time transfer function.
```

Compute the gain and phase margins.

```
[Gm,Pm,Wgm,Wpm] = margin(hd)
```

# margin

---

Gm =

2.0517

Pm =

13.5711

Wgm =

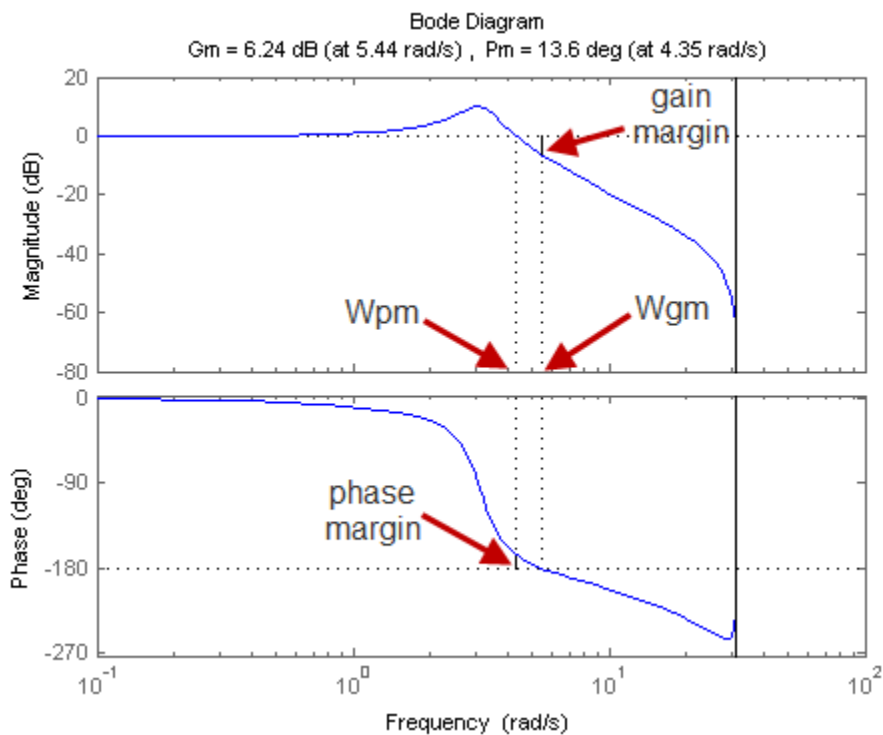
5.4374

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.

## 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 | ltiview

# minreal

---

**Purpose** 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 =
      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.

## 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 | modred | sminreal

**Purpose** Model order reduction

**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 1-419.
- '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**

Consider the continuous fourth-order model

$$h(s) = \frac{s^3 + 11s^2 + 36s + 26}{s^4 + 14.6s^3 + 74.96s^2 + 153.7s + 99.65}$$

To reduce its order, first compute a balanced state-space realization with `balreal`.

```
h = tf([1 11 36 26],[1 14.6 74.96 153.7 99.65]);
[hb,g] = balreal(h);
```

Examine the gramians.

```
g'
```

```
ans =
```

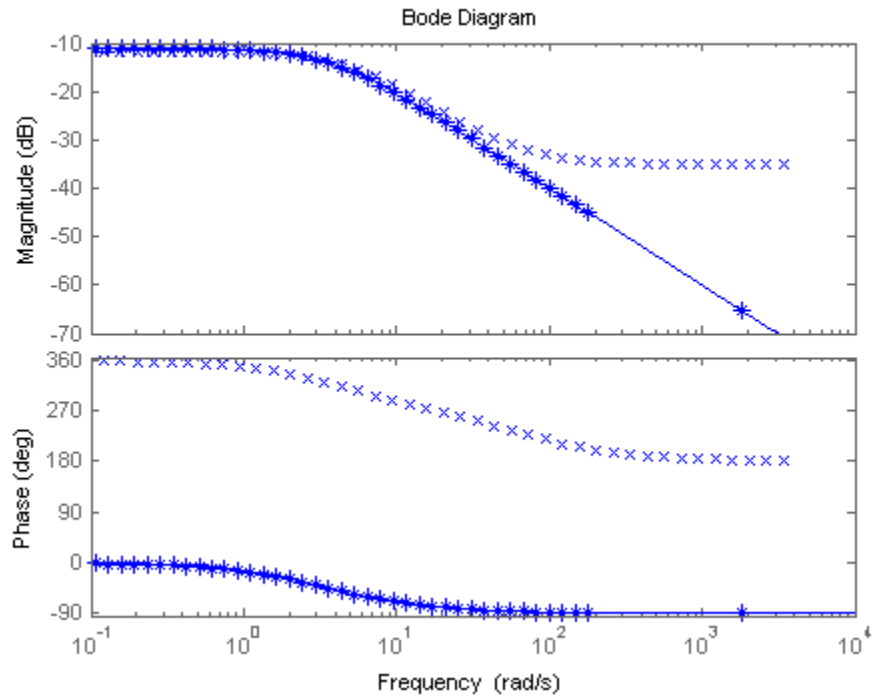
```
0.1394    0.0095    0.0006    0.0000
```

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  $h(s)$ .

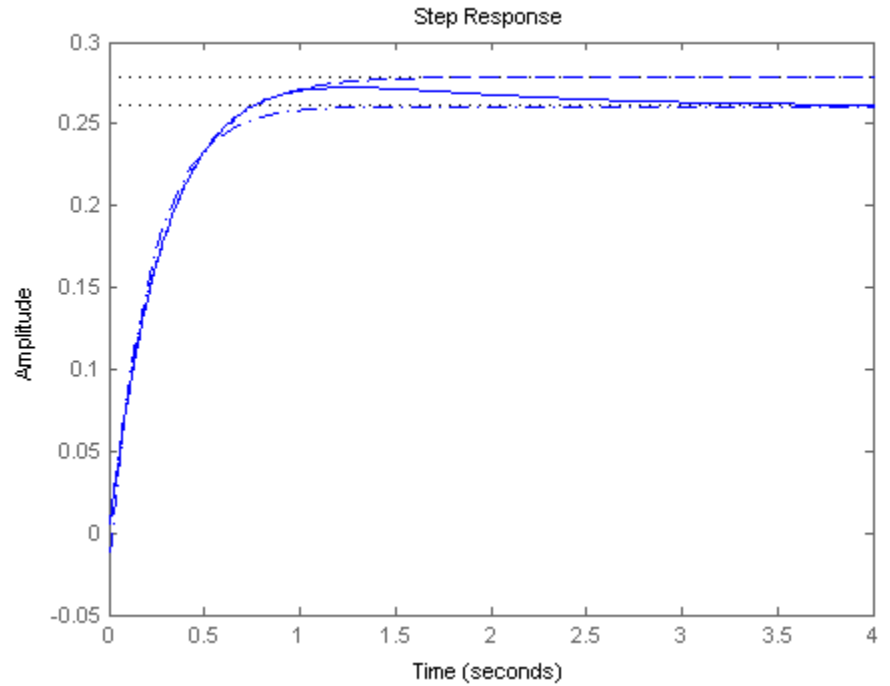
```
bodeplot(h,'-',hmdc,'x',hdel,'*')
```



The reduced-order model `hde1` is clearly a better frequency-domain approximation of  $h(s)$ . Now compare the step responses.

```
stepplot(h, '-', hmdc, '-.', hde1, '--')
```





While `hdel` accurately reflects the transient behavior, only `hmdc` gives the true steady-state response.

## Algorithms

The algorithm for the matched DC gain method is as follows. For continuous-time models

$$\dot{x} = Ax + By$$

$$y = Cx + Du$$

the state vector is partitioned into  $x_1$ , to be kept, and  $x_2$ , to be eliminated.

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} u$$
$$y = [C_1 \quad C_2]x + Du$$

Next, the derivative of  $x_2$  is set to zero and the resulting equation is solved for  $x_2$ . The reduced-order model is given by

$$\dot{x}_1 = [A_{11} - A_{12}A_{22}^{-1}A_{21}]x_1 + [B_1 - A_{12}A_{22}^{-1}B_2]u$$
$$y = [C_1 - C_2A_{22}^{-1}A_{21}]x + [D - C_2A_{22}^{-1}B_2]u$$

The discrete-time case is treated similarly by setting

$$x_2[n+1] = x_2[n]$$

## Limitations

With the matched DC gain method,  $A_{22}$  must be invertible in continuous time, and  $I - A_{22}$  must be invertible in discrete time.

## See Also

balreal | minreal

**Purpose** 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 `Hj` with their poles in disjoint regions `Rj` 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 `Hj` into an LTI array `H` and returns the static gain `H0` separately. Use `H(:, :, j)` to retrieve the submodel `Hj(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 as a string or 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
```

# modsep

---

To extract  $H_1(z)$  and  $H_2(z)$  from the LTI array  $H$ , use

```
H1 = H(:, :, 1); H2 = H(:, :, 2);
```

## See Also

stabsep

<b>Purpose</b>	Number of blocks in Generalized matrix or Generalized LTI model
<b>Syntax</b>	<code>N = nblocks(M)</code>
<b>Description</b>	<code>N = nblocks(M)</code> returns the number of “Control Design Blocks” in the Generalized LTI model or Generalized matrix <code>M</code> .
<b>Input Arguments</b>	<b>M</b> A Generalized LTI model (genss or genfrd model), a Generalized matrix (genmat), or an array of such models.
<b>Output Arguments</b>	<b>N</b> The number of “Control Design Blocks” in <code>M</code> . If a block appears multiple times in <code>M</code> , <code>N</code> reflects the total number of occurrences.  If <code>M</code> is a model array, <code>N</code> is an array with the same dimensions as <code>M</code> . Each entry of <code>N</code> is the number of Control Design Blocks in the corresponding entry of <code>M</code> .

### 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.

**See Also**`genss` | `genfrd` | `genmat` | `getValue`**How To**

- “Control Design Blocks”
- “Generalized Matrices”
- “Generalized and Uncertain LTI Models”

# ndims

---

**Purpose** 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`



<b>Purpose</b>	Superimpose Nichols chart on Nichols plot
<b>Syntax</b>	ngrid
<b>Description</b>	<p>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.</p> <p>The chart relates the complex number <math>H/(1 + H)</math> to <math>H</math>, where <math>H</math> is any complex number. For SISO systems, when <math>H</math> is a point on the open-loop frequency response, then</p> $\frac{H}{1+H}$ <p>is the corresponding value of the closed-loop frequency response assuming unit negative feedback.</p> <p>If the current axis is empty, ngrid generates a new Nichols chart grid in the region <math>-40</math> dB to <math>40</math> dB in magnitude and <math>-360</math> degrees to <math>0</math> degrees in phase. If the current axis does not contain a SISO Nichols frequency response, ngrid returns a warning.</p>

**Examples** Plot the Nichols response with Nichols grid lines for the system.

$$H(s) = \frac{-4s^4 + 48s^3 - 18s^2 + 250s + 600}{s^4 + 30s^3 + 282s^2 + 525s + 60}$$

Type

```
H = tf([-4 48 -18 250 600],[1 30 282 525 60])
```

These commands produce the following result.

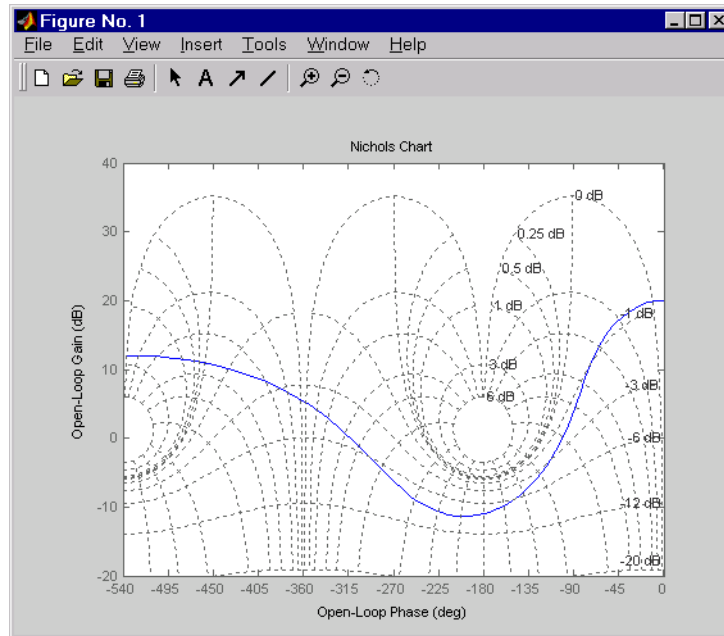
Transfer function:

```
- 4 s^4 + 48 s^3 - 18 s^2 + 250 s + 600
-----
s^4 + 30 s^3 + 282 s^2 + 525 s + 60
```

Type

nichols(H)

ngrid



**See Also**

nichols

**Purpose**

Nichols chart of frequency response

**Syntax**

```
nichols(sys)
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)
[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`)

## 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”.

## Examples

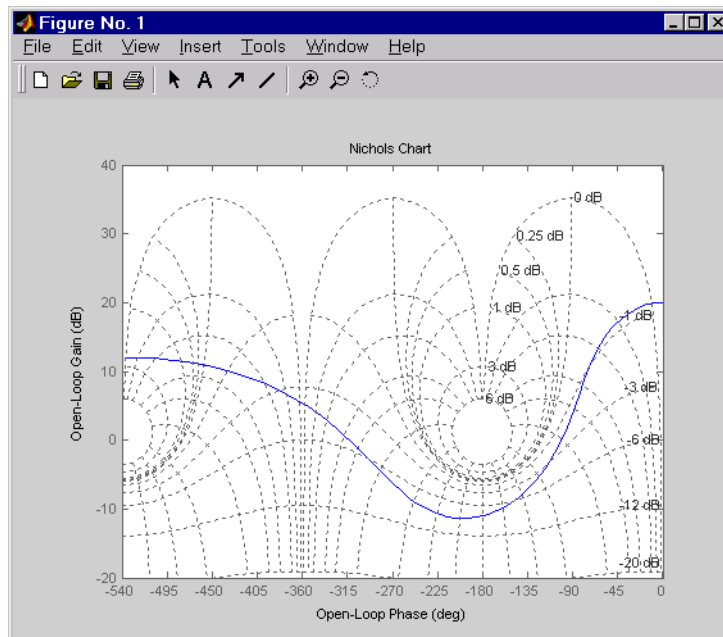
### Nichols Chart of Dynamic System

Display a Nichols chart of the dynamic system:

$$H(s) = \frac{-4s^4 + 48s^3 - 18s^2 + 250s + 600}{s^4 + 30s^3 + 282s^2 + 525s + 60}$$

```
num = [-4 48 -18 250 600];  
den = [1 30 282 525 60];  
H = tf(num,den)
```

```
nichols(H); ngrid
```



The right-click menu for Nichols charts includes the **Tight** option under **Zoom**. You can use this to clip unbounded branches of the Nichols chart.

## Algorithms

See bode.

## See Also

bode | evalfr | freqresp | ltiview | ngrid | nyquist | sigma

# nicholsoptions

---

**Purpose** 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 the following strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles.
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	<p data-bbox="917 319 1326 378">Frequency units, specified as one of the following strings:</p> <ul data-bbox="917 413 1237 1454" style="list-style-type: none"><li data-bbox="917 413 1000 440">• 'Hz'</li><li data-bbox="917 461 1118 489">• 'rad/second'</li><li data-bbox="917 510 1015 538">• 'rpm'</li><li data-bbox="917 558 1015 586">• 'kHz'</li><li data-bbox="917 607 1015 635">• 'MHz'</li><li data-bbox="917 656 1015 683">• 'GHz'</li><li data-bbox="917 704 1178 732">• 'rad/nanosecond'</li><li data-bbox="917 753 1193 781">• 'rad/microsecond'</li><li data-bbox="917 802 1193 829">• 'rad/millisecond'</li><li data-bbox="917 850 1118 878">• 'rad/minute'</li><li data-bbox="917 899 1089 927">• 'rad/hour'</li><li data-bbox="917 947 1074 975">• 'rad/day'</li><li data-bbox="917 996 1089 1024">• 'rad/week'</li><li data-bbox="917 1045 1104 1072">• 'rad/month'</li><li data-bbox="917 1093 1089 1121">• 'rad/year'</li><li data-bbox="917 1142 1222 1170">• 'cycles/nanosecond'</li><li data-bbox="917 1190 1237 1218">• 'cycles/microsecond'</li><li data-bbox="917 1239 1237 1267">• 'cycles/millisecond'</li><li data-bbox="917 1288 1133 1315">• 'cycles/hour'</li><li data-bbox="917 1336 1118 1364">• 'cycles/day'</li><li data-bbox="917 1385 1133 1413">• 'cycles/week'</li><li data-bbox="917 1433 1148 1461">• 'cycles/month'</li></ul>

# nicholsoptions

---

Option	Description
	<ul style="list-style-type: none"><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 multiple systems with different time units, the units of the first system are used.</p>
MagLowerLimMode	Enables a lower magnitude limit Specified as one of the following strings: 'auto'   'manual' <b>Default:</b> 'auto'
MagLowerLim	Specifies the lower magnitude limit
PhaseUnits	Phase units Specified as one of the following strings: 'deg'   'rad' <b>Default:</b> 'deg'
PhaseWrapping	Enables phase wrapping Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'
PhaseMatching	Enables phase matching Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'



Option	Description
PhaseMatchingFreq	Frequency for matching phase
PhaseMatchingValue	The value to make the phase responses close to

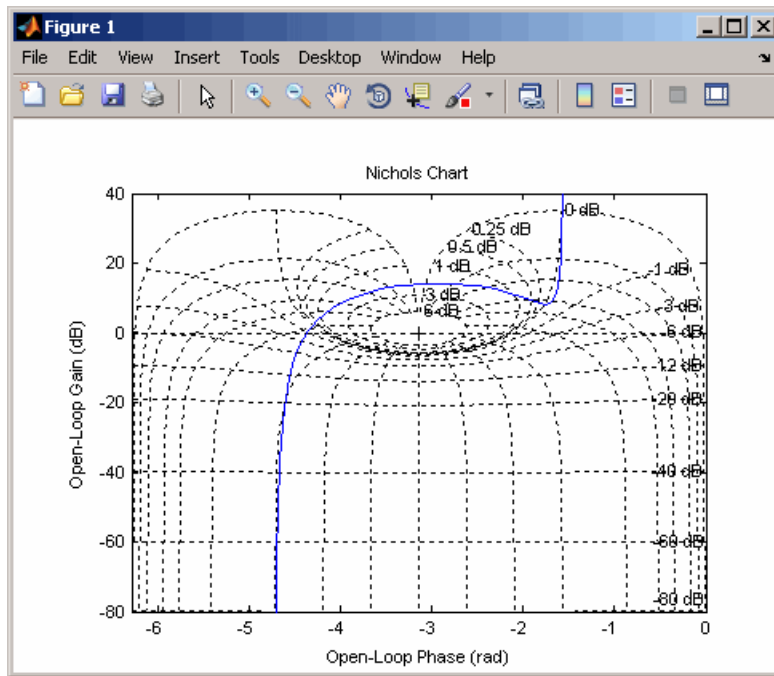
## Examples

In this example, you set the phase units and enable the grid option for the Nichols plot.

```
P = nicholsoptions; % Set phase units to radians and grid to on in options
P.PhaseUnits = 'rad';
P.Grid = 'on'; % Create plot with the options specified by P
h = nicholsplot(tf(1,[1,.2,1,0]),P);
```

The following Nichols plot is created, with the phase units in radians and the grid enabled.

# nicholsoptions



## See Also

[getoptions](#) | [nicholsplot](#) | [setoptions](#)

**Purpose**

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

---

`nicholsplot(..., plotoptions)` plots the Nichols plot with the options specified in `plotoptions`. Type

```
help nicholsoptions
```

for more details.

## 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”.

## 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');
```

## See Also

`getoptions` | `nichols` | `nicholsoptions` | `setoptions`

<b>Purpose</b>	Number of models in model array
<b>Syntax</b>	<code>N = nmodels(sysarray)</code>
<b>Description</b>	<code>N = nmodels(sysarray)</code> returns the number of models in an array of dynamic system models or static models.
<b>Input Arguments</b>	<b>sysarray - Input model array</b> model array  Input model array, specified as an array of input-output models such as numeric LTI models, generalized models, or identified LTI models.
<b>Output Arguments</b>	<b>N - Number of models in array</b> positive integer  Number of models in the input model array, returned as a positive integer.
<b>Examples</b>	<b>Number of Models in Array</b>  Create a 2-by-3-by-5 array of state-space models and confirm the number of models in the array.  <pre>sysarr = rss(2,2,2,2,3,4); N = nmodels(sysarr)</pre> <pre>N =      24</pre>
<b>See Also</b>	<code>ndims</code>   <code>size</code>

# norm

---

## Purpose

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.

**Definitions****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  $w$ :

$$\|H\|_2^2 = \lim_{t \rightarrow \infty} E \{y(t)^T y(t)\}, \quad E(w(t)w(\tau)^T) = \delta(t - \tau)I.$$

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).

## 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)
```

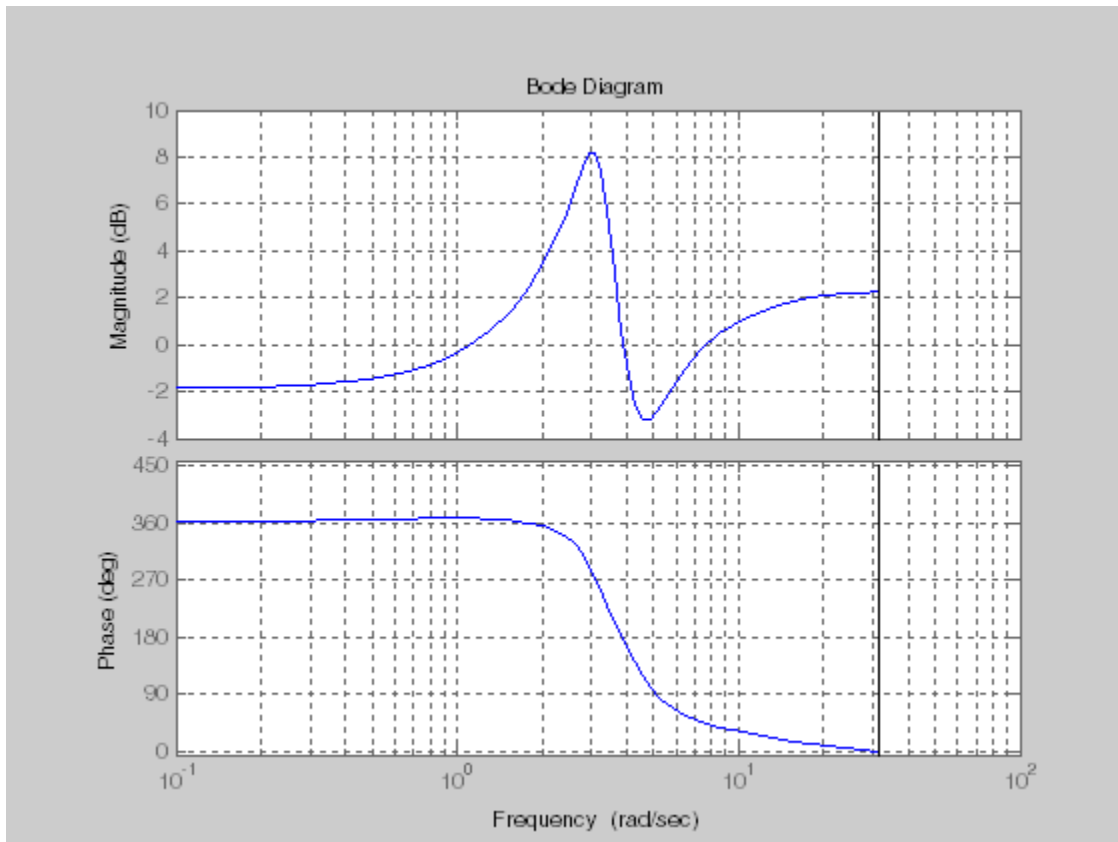
This command returns the result:

```
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
```

**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`

# nyquist

---

**Purpose** 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`.

## 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”.

## 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

### Example 1

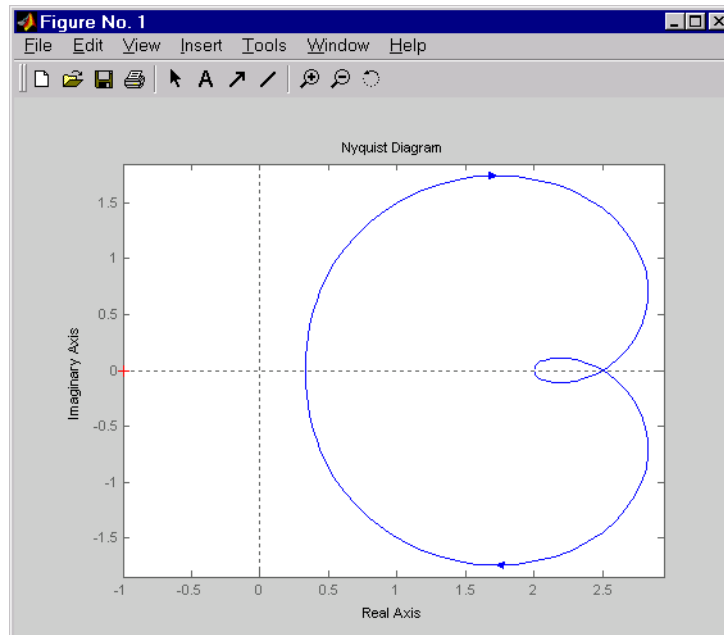
#### Nyquist Plot of Dynamic System

Plot the Nyquist response of the system

# nyquist

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

```
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

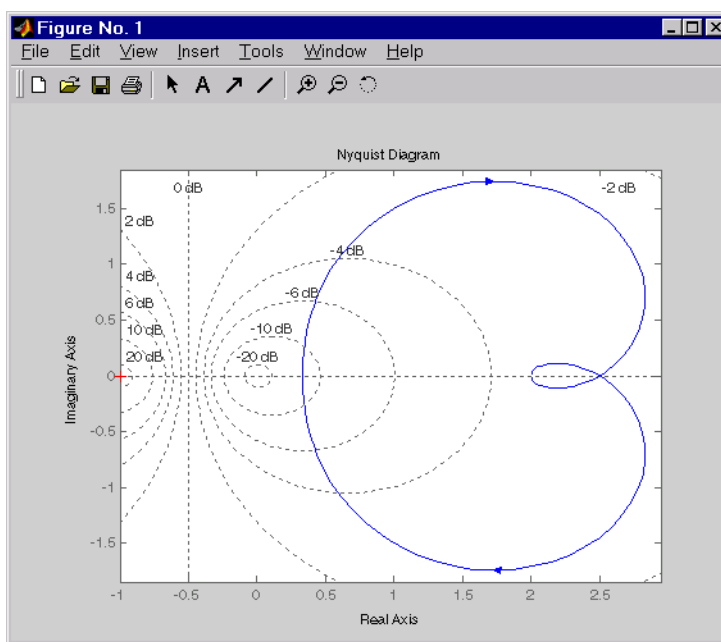
$$T(j\omega) = \left| \frac{G(j\omega)}{1 + G(j\omega)} \right|$$

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

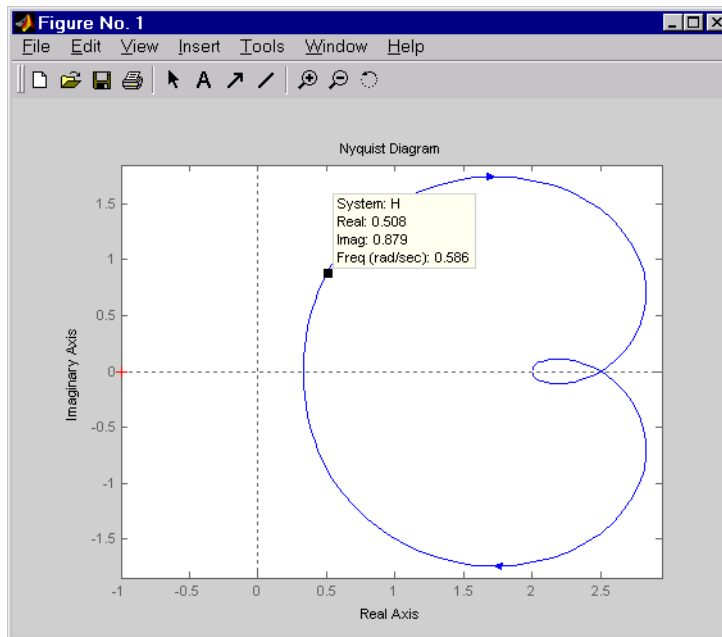
at the MATLAB prompt. This figure shows the M circles for transfer function  $H$ .



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.



## Example 2

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.

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:')
```

## Algorithms

See `bode`.

## See Also

`bode` | `evalfr` | `freqresp` | `ltiview` | `nichols` | `sigma`

# nyquistoptions

---

**Purpose** 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 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 strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabels, OutputLabels	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
FreqUnits	<p data-bbox="516 317 1222 348">Frequency units, specified as one of the following strings:</p> <ul data-bbox="516 383 836 1420" style="list-style-type: none"><li data-bbox="516 383 599 409">• 'Hz'</li><li data-bbox="516 430 718 456">• 'rad/second'</li><li data-bbox="516 477 614 503">• 'rpm'</li><li data-bbox="516 524 614 550">• 'kHz'</li><li data-bbox="516 571 614 597">• 'MHz'</li><li data-bbox="516 618 614 644">• 'GHz'</li><li data-bbox="516 664 777 690">• 'rad/nanosecond'</li><li data-bbox="516 711 792 737">• 'rad/microsecond'</li><li data-bbox="516 758 792 784">• 'rad/millisecond'</li><li data-bbox="516 805 718 831">• 'rad/minute'</li><li data-bbox="516 852 688 878">• 'rad/hour'</li><li data-bbox="516 899 673 925">• 'rad/day'</li><li data-bbox="516 946 688 972">• 'rad/week'</li><li data-bbox="516 992 703 1019">• 'rad/month'</li><li data-bbox="516 1039 688 1065">• 'rad/year'</li><li data-bbox="516 1086 822 1112">• 'cycles/nanosecond'</li><li data-bbox="516 1133 836 1159">• 'cycles/microsecond'</li><li data-bbox="516 1180 836 1206">• 'cycles/millisecond'</li><li data-bbox="516 1227 733 1253">• 'cycles/hour'</li><li data-bbox="516 1274 718 1300">• 'cycles/day'</li><li data-bbox="516 1321 733 1347">• 'cycles/week'</li><li data-bbox="516 1367 747 1394">• 'cycles/month'</li></ul>

# nyquistoptions

Option	Description
	<ul style="list-style-type: none"><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 multiple systems with different time units, the units of the first system are used.</p>
MagUnits	Magnitude units Specified as one of the following strings: 'dB'   'abs' <b>Default:</b> 'dB'
PhaseUnits	Phase units Specified as one of the following strings: 'deg'   'rad' <b>Default:</b> 'deg'
ShowFullContour	Show response for negative frequencies Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'on'
ConfidenceRegionNumberSD	Number of standard deviations to use to plotting the response confidence region (identified models only). <b>Default:</b> 1.
ConfidenceRegionDisplaySpacing	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

**Purpose**

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.

## 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”.

## 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**.

## **See Also**

`getoptions` | `nyquist` | `setoptions`

# obsv

---

**Purpose** 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

$$A = \begin{bmatrix} 1 & 1 \\ 4 & -2 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

is observable. Type



```
Ob = obsv(A,C);  
  
% Number of unobservable states  
unob = length(A) - rank(Ob)
```

These commands produce the following result.

```
unob =  
      0
```

### 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



```
      4   -2
B =
      1   -1
      1   -1

C =
      1    0
      0    1

by typing

[Abar,Bbar,Cbar,T,k] = obsvf(A,B,C)
Abar =
      1    1
      4   -2
Bbar =
      1    1
      1   -1
Cbar =
      1    0
      0    1
T =
      1    0
      0    1
k =
      2    0
```

## 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

# ord2

---

**Purpose** Generate continuous second-order systems

**Syntax** [A,B,C,D] = ord2(wn,z)  
[num,den] = ord2(wn,z)

**Description** [A,B,C,D] = ord2(wn,z) generates the state-space description (A,B,C,D) of the second-order system

$$h(s) = \frac{1}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

given the natural frequency  $\omega_n$  and damping factor  $\zeta$ . 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`

<b>Purpose</b>	Query model order
<b>Syntax</b>	<code>NS = order(sys)</code>
<b>Description</b>	<p><code>NS = order(sys)</code> returns the model order <code>NS</code>. 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).</p> <p><code>order(sys)</code> is an overloaded method that accepts SS, TF, and ZPK models. For LTI arrays, <code>NS</code> is an array of the same size listing the orders of each model in <code>sys</code>.</p>
<b>Caveat</b>	<p><code>order</code> does not attempt to find minimal realizations of MIMO systems. For example, consider this 2-by-2 MIMO system:</p> <pre>s=tf('s'); h = [1, 1/(s*(s+1)); 1/(s+2), 1/(s*(s+1)*(s+2))]; order(h) ans =          6</pre> <p>Although <code>h</code> has a 3rd order realization, <code>order</code> returns 6. Use <code>order(ss(h, 'min'))</code> to find the minimal realization order.</p>
<b>See Also</b>	<code>pole</code>   <code>balred</code>

**Purpose** Padé approximation of model with time delays

**Syntax**

```
[num,den] = pade(T,N)
sysx = pade(sys,N)
sysx = pade(sys,NU,NY,NINT)
```

**Description** `pade` 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] = pade(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,

```
pade(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 = pade(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 “Models with Time Delays” for more information about models with time delays.

`sysx = pade(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)

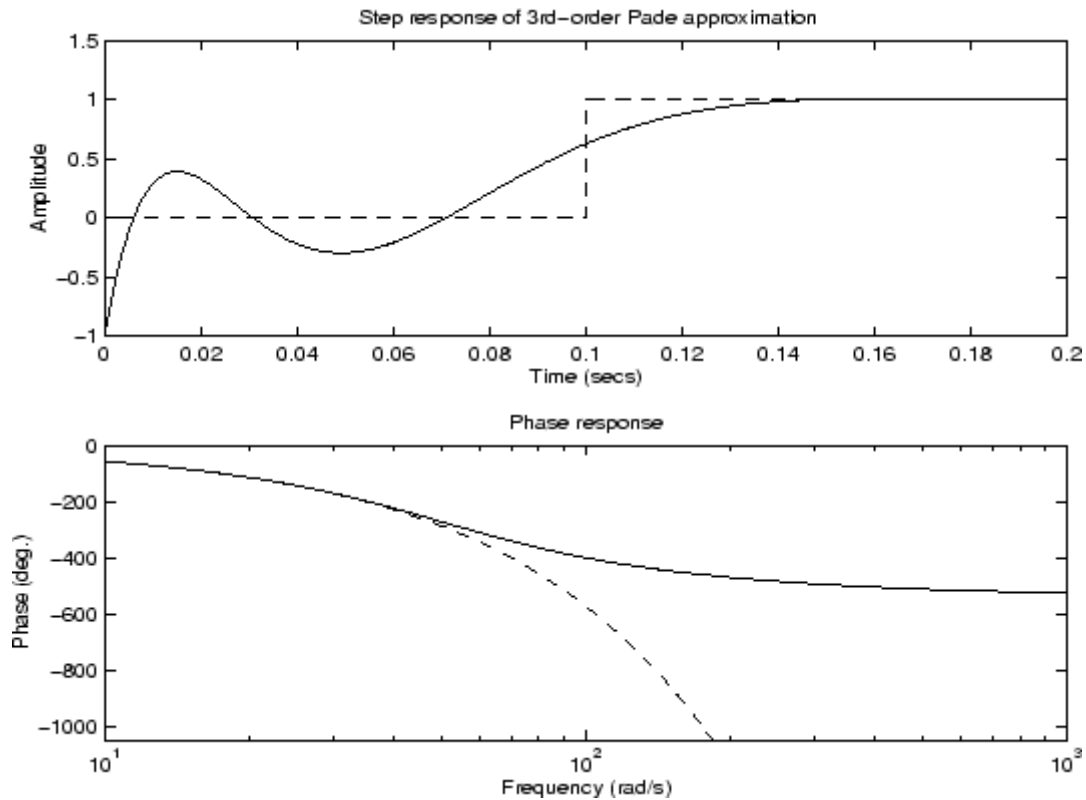
You can use scalar values for NU, NY, or NINT to specify a uniform approximation order. You can also set some entries of NU, NY, or NINT to Inf to prevent approximation of the corresponding delays.

## Examples

### Third-Order Padé Approximation

Compute a third-order Padé approximation of a 0.1 second I/O delay and compare the time and frequency responses of the true delay and its approximation. To do this, type

```
pade(0.1,3)
```



## Limitations

High-order Padé approximations produce transfer functions with clustered poles. Because such pole configurations tend to be very sensitive to perturbations, Padé approximations with order  $N > 10$  should be avoided.

## 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



**How To**

- “Time-Delay Approximation”

# parallel

---

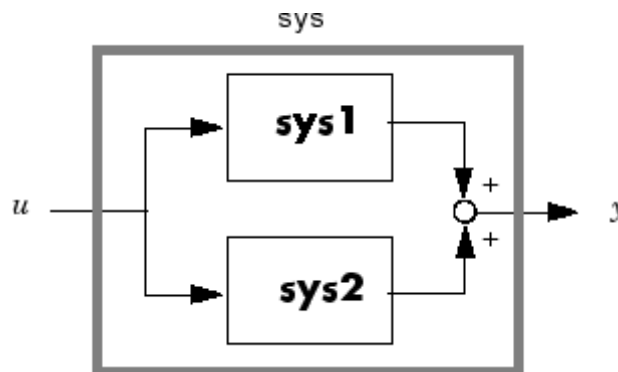
**Purpose** Parallel connection of two models

**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.

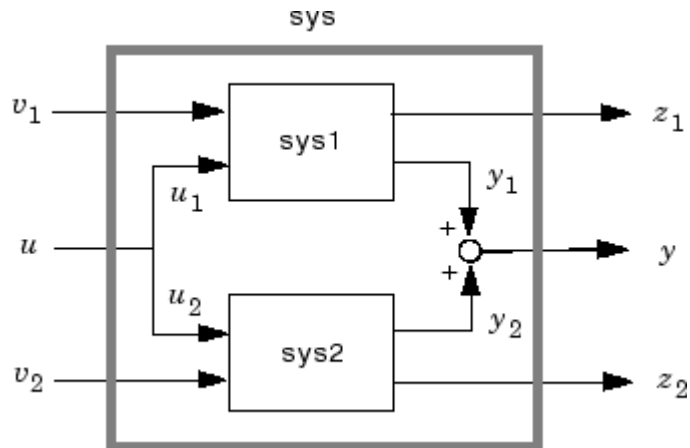
`sys = parallel(sys1,sys2)` forms the basic parallel connection shown in the following figure.



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
```

# parallel

---

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](#)

<b>Purpose</b>	Permute array dimensions in model arrays
<b>Syntax</b>	<code>newarray = permute(sysarray,order)</code>
<b>Description</b>	<code>newarray = permute(sysarray,order)</code> 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.
<b>Input Arguments</b>	<p><b>sysarray - Model array to rearrange</b>  <i>model array</i></p> <p>Model array to rearrange, specified as an array of input-output models such as numeric LTI models, generalized models, or identified LTI models.</p> <p><b>order - Dimensions of rearranged model array</b>  <i>vector</i></p> <p>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].</p> <p><b>Data Types</b>  double</p>
<b>Output Arguments</b>	<p><b>newarray - Rearranged model array</b>  <i>model array</i></p> <p>Rearranged model array, returned as an array of input-output models with the new dimensions as specified in order.</p>
<b>Examples</b>	<p><b>Permute Model Array Dimensions</b></p> <p>Create a 1-by-2-by-3 array of state-space models and rearrange it so that its dimensions are 3-by-2-by-1.</p> <pre>sysarr = rss(2,2,2,1,2,3); newarr = permute(sysarr,[3 2 1]);</pre>

# permute

---

`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.

## **See Also**

`ndims` | `size` | `reshape`

**Purpose**

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  $K_p$ ,  $K_i$ , and  $K_d$  and first-order derivative filter time constant  $T_f$ :

$$C = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}.$$

This representation is in *parallel form*. If all of  $K_p$ ,  $K_i$ ,  $K_d$ , and  $T_f$  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 sampling time  $T_s$ . 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,  $IF(z) = DF(z) = T_s z / (z - 1)$ . To choose different discrete integrator formulas, use the `IFormula` and `DFormula` properties. (See “Properties” on page 1-478 for more information about `IFormula` and `DFormula`). 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.

`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  $K_i = 0$ ,  $K_d = 0$ , and  $T_f = 0$ .

`C = pid(Kp,Ki)` creates a proportional and integral (PI) controller with  $K_d = 0$  and  $T_f = 0$ .

`C = pid(Kp,Ki,Kd)` creates a proportional, integral, and derivative (PID) controller with  $T_f = 0$ .

`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  $K_p = 1$ .

## Tips

- Use `pid` either to create a `pid` controller object from known PID gains and filter time constant, or to convert a dynamic system model to a `pid` object.
- To design a PID controller for a particular plant, use `pidtune` or `pidtool`.
- 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 sampling 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  $N$ :



$$C = K_p \left( 1 + \frac{1}{T_i} \frac{1}{s} + \frac{T_d s}{N s + 1} \right)$$

- 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 1-478 .

- 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 pid controllers than using `c2d`.

## Input Arguments

### **Kp**

Proportional gain.

`Kp` can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (*realp*).
- A tunable generalized matrix (*genmat*), such as a gain surface for gain-scheduled tuning, created using *gainsurf* (requires Robust Control Toolbox software).

When  $K_p = 0$ , the controller has no proportional action.

**Default:** 1

## **K<sub>i</sub>**

Integral gain.

$K_i$  can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (*realp*).
- A tunable generalized matrix (*genmat*), such as a gain surface for gain-scheduled tuning, created using *gainsurf* (requires Robust Control Toolbox software).

When  $K_i = 0$ , the controller has no integral action.

**Default:** 0

## **K<sub>d</sub>**

Derivative gain.

$K_d$  can be:

- A real and finite value.
- An array of real and finite values.
- A tunable parameter (*realp*).

- A tunable generalized matrix (`genmat`), such as a gain surface for gain-scheduled tuning, created using `gainsurf` (requires Robust Control Toolbox software).

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`).
- A tunable generalized matrix (`genmat`), such as a gain surface for gain-scheduled tuning, created using `gainsurf` (requires Robust Control Toolbox software).

When  $T_f = 0$ , the controller has no filter on the derivative action.

**Default:** 0

### **Ts**

Sampling 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  $T_s$ .

### **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 1-478.

## 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`) or generalized matrix (`genmat`), then `C` is a generalized state-space model (`genss`).

## Properties

`pid` controller objects have the following 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` values are real and finite.

**Tf**

Derivative filter time constant.

The Tf property stores the derivative filter time constant of the pid controller object. Tf are real, finite, and greater than or equal to zero.

**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 sampling time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling 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 sampling time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling 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 ( $T_f = 0$ ).

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**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For

discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

## **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string '' for all input channels

## **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set `Name` to a string to label the system.

**Default:** ''

### **Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

### **UserData**

Any type of data you wish to associate with system. Set `UserData` to 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
```

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

$$s^2 + 3.5 s + 25$$

...

**Default:** []

## Examples

### PID Controller with Proportional and Derivative Gains, and Filter Time Constant (PDF Controller)

Create a continuous-time controller with proportional and derivative gains, and filter time constant (PDF controller).

```
Kp=1;
Ki=0;
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.

---

### Discrete-Time PI Controller

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 = pid(5,2.4,'Ts',0.1,'IFormula','Trapezoidal') % Ts = 0.1s
```

This command produces the result:

Discrete-time PI controller in parallel form:

$$K_p + K_i * \frac{T_s(z+1)}{2*(z-1)}$$

with `Kp = 5`, `Ki = 2.4`, `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`, `Ki`, `Kd`, and `Tf`.

```
C = pid(5,2.4,0,0,0.1,'IFormula','Trapezoidal');
```

---

## PID Controller with Custom Input and Output Names

Create a PID controller, and set dynamic system properties `InputName` and `OutputName`.

```
C = pid(1,2,3,'InputName','e','OutputName','u');
```

---

## Array of PID Controllers

Create a 2-by-3 grid of PI controllers with proportional gain ranging from 1–2 and integral gain ranging from 5–9.

Create a grid of PI controllers with proportional gain varying row to row and integral gain varying column to column. To do so, start with arrays representing the gains.

```
Kp = [1 1 1;2 2 2];  
Ki = [5:2:9;5:2:9];  
pi_array = pid(Kp,Ki,'Ts',0.1,'IFormula','BackwardEuler');
```

These commands produce a 2-by-3 array of discrete-time pid objects. All pid 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 stack to build arrays of pid objects.

```
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
```

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 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 times  $T_i$  and  $T_d$ , and filter divisor  $N$ . You can convert any standard form controller to parallel form using pid.

```
stdsys = pidstd(2,3,4,5); % Standard-form controller  
parsys = pid(stdsys)
```

These commands produce a parallel-form controller:

Continuous-time PIDF controller in parallel form:

$$K_p + K_i * \frac{1}{s} + K_d * \frac{s}{T_f*s+1}$$

with  $K_p = 2$ ,  $K_i = 0.66667$ ,  $K_d = 8$ ,  $T_f = 0.8$

---

## Convert Dynamic System to Parallel-Form PID Controller

Convert a continuous-time dynamic system that represents a PID controller to parallel pid form.

The dynamic system

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

represents a PID controller. Use `pid` to obtain  $H(s)$  to in terms of the PID gains  $K_p$ ,  $K_i$ , and  $K_d$ .

```
H = zpk([-1,-2],0,3);  
C = pid(H)
```

These commands produce the result:

Continuous-time PID controller in parallel form:

$$K_p + K_i * \frac{1}{s} + K_d * s$$

with  $K_p = 9$ ,  $K_i = 6$ ,  $K_d = 3$

---

## Convert Discrete-Time Zero-Pole-Gain Model to Parallel-Form PID Controller



Convert a discrete-time dynamic system that represents a PID controller with derivative filter to parallel pid form.

```
% PIDF controller expressed in zpk form
sys = zpk([-0.5,-0.6],[1 -0.2],3,'Ts',0.1)
```

The resulting pid 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 = pid(sys)
```

returns the result

Discrete-time PIDF controller in parallel form:

$$K_p + K_i * \frac{T_s}{z-1} + K_d * \frac{1}{T_f + T_s / (z-1)}$$

with  $K_p = 2.75$ ,  $K_i = 60$ ,  $K_d = 0.020833$ ,  $T_f = 0.083333$ ,  $T_s = 0.1$

Converting using the Trapezoidal formula returns different parameter values:

```
C = pid(sys,'IFormula','Trapezoidal','DFormula','Trapezoidal')
```

This command returns the result:

Discrete-time PIDF controller in parallel form:

$$K_p + K_i * \frac{T_s * (z+1)}{2 * (z-1)} + K_d * \frac{1}{T_f + T_s / 2 * (z+1) / (z-1)}$$

with  $K_p = -0.25$ ,  $K_i = 60$ ,  $K_d = 0.020833$ ,  $T_f = 0.033333$ ,  $T_s = 0.1$

For this particular sys, you cannot write sys in parallel PID form using the BackwardEuler formula for DFormula. 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

First, discretize the controller using the 'zoh' method of c2d.

```
Cc = pid(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 parallel form:

$$K_p + K_i * \frac{T_s}{z-1} + K_d * \frac{1}{T_f + T_s / (z-1)}$$

with  $K_p = 1$ ,  $K_i = 2$ ,  $K_d = 3.0377$ ,  $T_f = 4.0502$ ,  $T_s = 0.1$

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 1-474. 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 parallel form:

$$K_p + K_i * \frac{T_s * z}{z-1} + K_d * \frac{1}{T_f + T_s * z / (z-1)}$$

with  $K_p = 1$ ,  $K_i = 2$ ,  $K_d = 3$ ,  $T_f = 4$ ,  $T_s = 0.1$

## See Also

`pidstd` | `piddata` | `pidtune` | `pidtool` | `ltiblock.pid` | `genss` | `realp`

## Tutorials

- “Proportional-Integral-Derivative (PID) Controller”
- “Discrete-Time Proportional-Integral-Derivative (PID) Controller”

## How To

- “What Are Model Objects?”
- “PID Controllers”

# piddata

---

## Purpose

Access PID data

## 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 the array of `sys` dynamic systems. The indices `J` specify the array entries to extract.

## 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.

## 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.

## 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, the output **Ki** is equal to the Ki value of **sys**.

If **sys** is not a pid object, **Ki** is the integral gain of a parallel PID controller equivalent to **sys**.

If **sys** is an array of dynamic systems, **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, the output **Kd** is equal to the Kd value of **sys**.

If **sys** is not a pid object, **Kd** is the derivative gain of a parallel PID controller equivalent to **sys**.

If **sys** is an array of dynamic systems, **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**

Sampling 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); % sampling 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);
```

## See Also

`pid` | `pidstd` | `get`

**Purpose** 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 `Kp`, integral and derivative times `Ti` and `Td`, 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 sampling time `Ts`. 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,  $IF(z) = DF(z) = T_s z / (z - 1)$ . To choose different discrete integrator formulas, use the `IFormula` and `DFormula` inputs. (See “Properties” on page 1-503 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  $K_p = 1$ .

## 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 `pidtool`.
- 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 sampling 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 1-503 .

- 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`.

**Input Arguments****Kp**

Proportional gain.

Kp must be a real and finite value.

For an array of pidstd controllers, Kp must be an array of real and finite values.

**Default:** 1

**Ti**

Integral time.

Ti must be a real and positive value. When  $Ti = \text{Inf}$ , the controller has no integral action.

For an array of pidstd controllers, Ti must be an array of real and positive values.

**Default:** Inf

**Td**

Derivative time.

Td must be a real, finite, and nonnegative value. When  $Td = 0$ , the controller has no derivative action.

For an array of pidstd controllers, Td must be an array of real, finite, and nonnegative values.

**Default:** 0

**N**

Time constant of the first-order derivative filter.

N must be a real and positive value. When  $N = \text{Inf}$ , the controller has no derivative filter.

For an array of `pidstd` controllers, `N` must be an array of real and positive values.

**Default:** `Inf`

## **Ts**

Sampling 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 ( $T_s = -1$ ).

`Ts` must be a scalar value. In an array of `pidstd` controllers, each controller must have the same `Ts`.

## **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 1-503.

## 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  $K_p$ ,  $T_i$ ,  $T_d$ , and  $N$ . For example, when  $T_d = \text{Inf}$  and  $K_p$  and  $T_i$  are finite and nonzero,  $C$  is a PI controller. Enter `getType(C)` to obtain the controller type.

When the inputs  $K_p, T_i, T_d$ , and  $N$  or the input `sys` are arrays,  $C$  is an array of pidstd objects.

## Properties

pidstd controller objects have the following properties:

### **$K_p$**

Proportional gain.  $K_p$  must be real and finite.

### **$T_i$**

Integral time.  $T_i$  must be real, finite, and greater than or equal to zero.

### **$T_d$**

Derivative time.  $T_d$  must be real, finite, and greater than or equal to zero.

### **$N$**

Derivative time.  $N$  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 sampling time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling 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 sampling time, where the Nyquist limit is large compared to the bandwidth of the controller. For larger sampling 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 ( $N = \text{Inf}$ ).

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**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

## **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'



- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

## InputUnit

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

### OutputUnit

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input 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. Set `Name` to a string to label the system.

**Default:** ''

## **Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

## **UserData**

Any type of data you wish to associate with system. Set `UserData` to 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
```

...

**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  $K_p = 1$ ,  $T_d = 3$ ,  $N = 6$

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  $T_s$  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  $K_p = 1$ ,  $T_i = 0.5$ ,  $T_s = 0.1$

Alternatively, you can create the same discrete-time controller by supplying  $T_s$  as the fifth argument after all four PID parameters  $K_p$ ,  $T_i$ ,  $T_d$ , 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 standard form pid controller to parallel 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 `Kp`, `Ki`, and `Kd` 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 `Kp = 2`, `Ti = 0.66667`, `Td = 2`, `N = 0.4`

---

Convert a continuous-time dynamic system that represents a PID controller to parallel pid 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 `Ti` and `Td`, 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 `Kp = 1`, `Ti = 2`, `Td = 3.2044`, `N = 4`, `Ts = 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 1-499. 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  $K_p = 1$ ,  $T_i = 2$ ,  $T_d = 3$ ,  $N = 4$ ,  $T_s = 0.1$

## See Also

`pid` | `piddata` | `pidtune` | `pidtool`

## Tutorials

- “Proportional-Integral-Derivative (PID) Controller”
- “Discrete-Time Proportional-Integral-Derivative (PID) Controller”

## How To

- “What Are Model Objects?”
- “PID Controllers”

# pidstdata

---

<b>Purpose</b>	Access PIDSTD data
<b>Syntax</b>	<pre>[Kp,Ti,Td,N] = pidstdata(sys) [Kp,Ti,Td,N,Ts] = pidstdata(sys) [Kp,Ti,Td,N,Ts] = pidstdata(sys, J1,...,JN)</pre>
<b>Description</b>	<p><code>[Kp,Ti,Td,N] = pidstdata(sys)</code> returns the proportional gain <code>Kp</code>, integral time <code>Ti</code>, derivative time <code>Td</code>, and filter divisor <code>N</code> of the standard-form controller represented by the dynamic system <code>sys</code>.</p> <p><code>[Kp,Ti,Td,N,Ts] = pidstdata(sys)</code> also returns the sample time <code>Ts</code>.</p> <p><code>[Kp,Ti,Td,N,Ts] = pidstdata(sys, J1,...,JN)</code> extracts the data for a subset of entries in the array of <code>sys</code> dynamic systems. The indices <code>J</code> specify the array entries to extract.</p>
<b>Tips</b>	<p>If <code>sys</code> is not a <code>pidstd</code> controller object, <code>pidstdata</code> returns <code>Kp</code>, <code>Ti</code>, <code>Td</code> and <code>N</code> values of a standard-form controller equivalent to <code>sys</code>.</p> <p>For discrete-time <code>sys</code>, <code>pidstdata</code> returns parameters of an equivalent <code>pidstd</code> controller. This controller has discrete integrator formulas <code>Iformula</code> and <code>Dformula</code> set to <code>ForwardEuler</code>. See the <code>pidstd</code> reference page for more information about discrete integrator formulas.</p>
<b>Input Arguments</b>	<p><b>sys</b></p> <p>SISO dynamic system or array of SISO dynamic systems. If <code>sys</code> is not a <code>pidstd</code> object, it must represent a valid PID controller that can be written in standard PID form.</p> <p><b>J</b></p> <p>Integer indices of <code>N</code> entries in the array <code>sys</code> of dynamic systems.</p>
<b>Output Arguments</b>	<p><b>Kp</b></p> <p>Proportional gain of the standard-form PID controller represented by dynamic system <code>sys</code>.</p>

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**

Sampling 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 = zpke([0.5 0.6],[1,-0.8],1,0.1); % sampling 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);
```

## See Also

`pidstd` | `pid` | `get`

# pidtool

---

**Purpose** Open PID Tuner for PID tuning

**Syntax**

```
pidtool(sys,type)
pidtool(sys,Cbase)
pidtool(sys)
pidtool
```

**Description** `pidtool(sys,type)` launches the PID Tuner GUI and designs a controller of type `type` for plant `sys`.

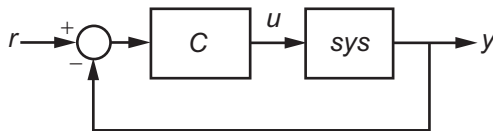
`pidtool(sys,Cbase)` launches the GUI with a baseline controller `Cbase` so that you can compare performance between the designed controller and the baseline controller. If `Cbase` is a `pid` or `pidstd` controller object, the PID Tuner designs a controller of the same form, type, and discrete integrator formulas as `Cbase`.

`pidtool(sys)` designs a parallel-form PI controller.

`pidtool` launches the GUI with default plant of 1 and proportional (P) controller of 1.

**Tips**

- The PID Tuner designs a controller in the feedforward path of a single control loop with unit feedback:



- The 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 the PID Tuner, see “Designing PID Controllers with the PID Tuner GUI”.

## 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 (**idarx**, **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 launching the PID Tuner GUI, click the  button to open the **Import Linear System** dialog box. In that dialog box, you can reimport **sys**, specifying the number of unstable poles where prompted.

### **type**

Controller type (actions) of the controller you are designing, specified as one of the following strings:

String	Type	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
'p'	proportional only	$K_p$	$K_p$
'i'	integral only	$\frac{K_i}{s}$	$K_i \frac{T_s}{z-1}$
'pi'	proportional and integral	$K_p + \frac{K_i}{s}$	$K_p + K_i \frac{T_s}{z-1}$
'pd'	proportional and derivative	$K_p + K_d s$	$K_p + K_d \frac{z-1}{T_s}$
'pdf'	proportional and derivative with first-order filter on derivative term	$K_p + \frac{K_d s}{T_f s + 1}$	$K_p + K_d \frac{1}{T_f + \frac{T_s}{z-1}}$
'pid'	proportional, integral, and derivative	$K_p + \frac{K_i}{s} + K_d s$	$K_p + K_i \frac{T_s}{z-1} + K_d \frac{z-1}{T_s}$
'pidf'	proportional, integral, and derivative with first-order filter on derivative term	$K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}$	$K_p + K_i \frac{T_s}{z-1} + K_d \frac{1}{T_f + \frac{T_s}{z-1}}$

When you use the **type** input, the 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 sampling time  $T_s$ , the PID Tuner designs a discrete-time **pid** controller using the **ForwardEuler** discrete integrator formula. If you want to design a controller having a different discrete integrator formula, use the input **Cbase** instead of **type** or the **Preferences** dialog box. 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, the 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.
- 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, the PID Tuner designs a parallel-form PI controller. You can change the controller form and **type** using the **Form** and **Type** menus after launching the PID Tuner.

## **Examples**

### **Interactive PID Tuning of Parallel-Form Controller**

Launch the 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

pidtool(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')

pidtool(Gd,Cbase)          % Launch PID Tuner
```

The 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 on the PID Tuner GUI.

## Algorithms

Typical PID tuning objectives include:

- Closed-loop stability — The closed-loop system output remains bounded for bounded input.
- Adequate performance — The closed-loop system tracks reference changes and suppresses disturbances as rapidly as possible. The larger the loop bandwidth (the first frequency at which the open-loop gain is unity), the faster the controller responds to changes in the reference or disturbances in the loop.

- Adequate robustness — The loop design has enough phase margin and gain margin to allow for modeling errors or variations in system dynamics.

The MathWorks algorithm for tuning PID controllers helps you meet these objectives by automatically tuning the PID gains to balance performance (response time) and robustness (stability margins).

By default, the algorithm chooses a crossover frequency (loop bandwidth) based upon the plant dynamics, and designs for a target phase margin of 60°. If you change the bandwidth or phase margin using the sliders in the PID Tuner GUI, the algorithm computes PID gains that best meet those targets.

**References**

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

**Alternatives**

For PID tuning at the command line, use `pidtune`. `pidtune` can design controllers for multiple plants at once.

**See Also**

`pid` | `pidstd` | `pidtune`

**Tutorials**

- Designing PID for Disturbance Rejection with PID Tuner

**How To**

- “Designing PID Controllers with the PID Tuner GUI”

# pidtune

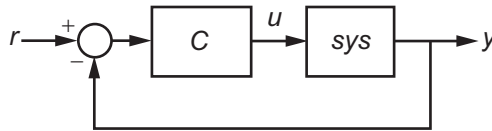
---

**Purpose** 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` in the unit feedback loop



`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  $L = \text{sys} * C$ .

`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.

**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.

## 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 (actions) of the controller to design, specified as one of the following strings.

String	Type	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
'p'	Proportional only	$K_p$	$K_p$
'i'	Integral only	$\frac{K_i}{s}$	$K_i \frac{T_s}{z-1}$
'pi'	Proportional and integral	$K_p + \frac{K_i}{s}$	$K_p + K_i \frac{T_s}{z-1}$
'pd'	Proportional and derivative	$K_p + K_d s$	$K_p + K_d \frac{z-1}{T_s}$
'pdf'	Proportional and derivative with first-order filter on derivative term	$K_p + \frac{K_d s}{T_f s + 1}$	$K_p + K_d \frac{1}{T_f + \frac{T_s}{z-1}}$
'pid'	Proportional, integral, and derivative	$K_p + \frac{K_i}{s} + K_d s$	$K_p + K_i \frac{T_s}{z-1} + K_d \frac{z-1}{T_s}$
'pidf'	Proportional, integral, and derivative with first-order filter on derivative term	$K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}$	$K_p + K_i \frac{T_s}{z-1} + K_d \frac{1}{T_f + \frac{T_s}{z-1}}$



When you use the `type` input, `pidtune` designs a controller in parallel (`pid`) form. Use the input `C0` instead of `type` if you want to design a controller in standard (`pidstd`) form.

If `sys` is a discrete-time model with sampling 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.

### **C0**

`pid` or `pidstd` controller specifying properties of the designed controller. 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 controller with the same `Iformula` and `Dformula` values as `C0`, if `sys` is a discrete-time system. See the `pid` and `pidstd` 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  $L = \text{sys} * C$ . 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. 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 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 `C0` is a pidstd object.

#### **Controller type:**

- If the second argument to `pidtune` is `type`, `C` generally has the specified type.
- If the second argument to `pidtune` is `C0`, `C` generally has the same type as `C0`.

In either case, however, where the algorithm can achieve adequate performance and robustness using a lower-order controller than specified with `type` or `C0`, `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 sampling time as `sys`.
- If you specify `C0`, `C` has the same `Iformula` and `Dformula` as `C0`. If no `C0` is specified, both `Iformula` and `Dformula` are Forward Euler. See the `pid` and `pidstd` 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

### Tune Parallel-Form PID Controller

This example shows how to design a PID controller for the plant

$$\text{sys} = \frac{1}{(s+1)^3}.$$

As a first pass, design a simple PI controller:

```
sys = zpk([],[-1 -1 -1],1); % define the plant
[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.

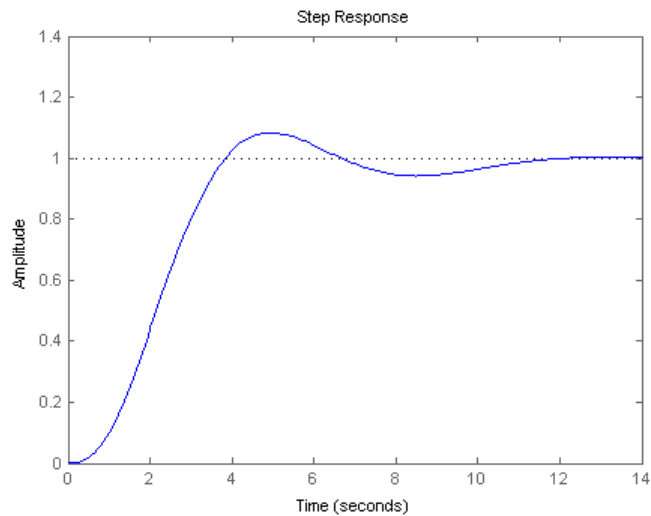
```
info =
```

```
          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 Kp = 2.83, Ki = 0.0495
```

Continuous-time PI controller in parallel form.

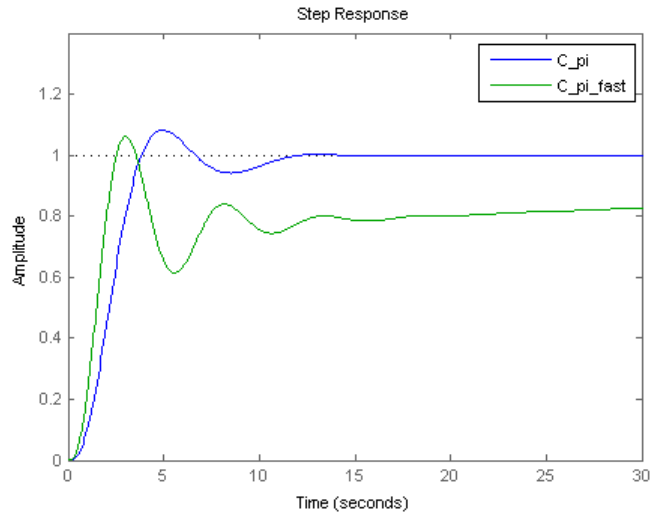
```
info =
```

```
          Stable: 1
CrossoverFrequency: 1
          PhaseMargin: 43.9973
```

The new controller achieves the higher crossover frequency, but at the cost of a reduced phase margin.

Compare the closed-loop step response with the two controllers.

```
T_pi_fast = feedback(C_pi_fast*sys,1);
step(T_pi,T_pi_fast)
axis([0 30 0 1.4])
legend('C_pi','C_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  $G_c$  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  $K_p = 2.72$ ,  $K_i = 1.03$ ,  $K_d = 1.76$ ,  $T_f = 0.00875$

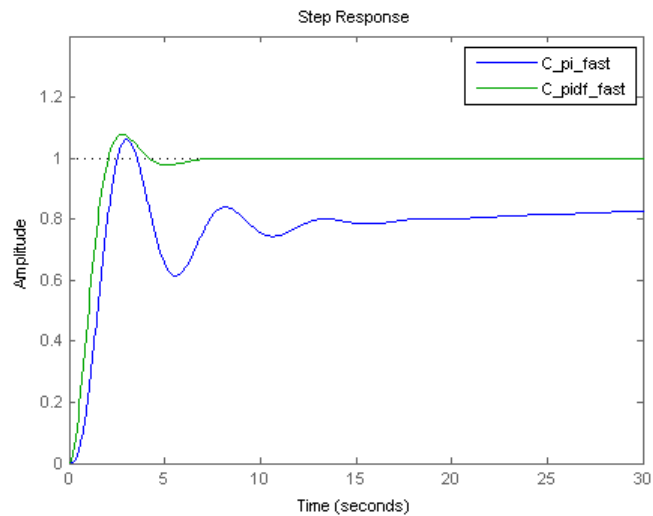
Continuous-time PIDF controller in parallel form.

```
info =  
  
          Stable: 1  
    CrossoverFrequency: 1  
          PhaseMargin: 60.0000
```

The fields of `info` show that the derivative action in the controller allows the tuning algorithm to design a more aggressive controller that achieves the target crossover frequency with a good phase margin.

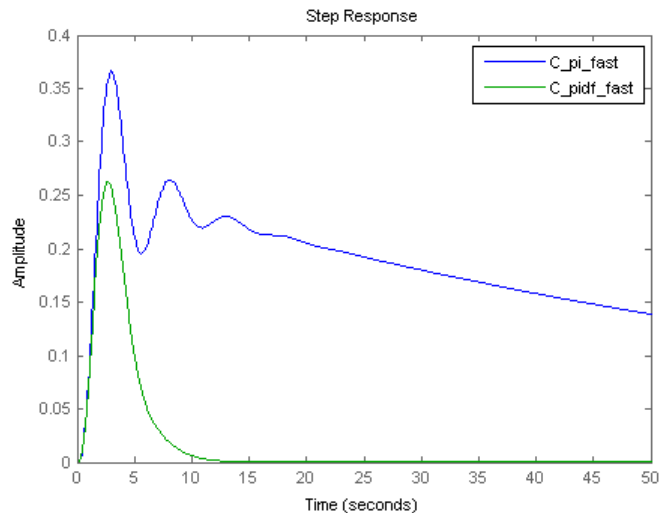
Compare the closed-loop step response and disturbance rejection for the fast PI and PIDF controllers.

```
T_pidf_fast = feedback(C_pidf_fast*sys,1);  
step(T_pi_fast, T_pidf_fast);  
axis([0 30 0 1.4]);  
legend('C_pi_fast', 'C_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('C_pi_fast','C_pidf_fast');
```



This plot shows that the PIDF controller also provides faster disturbance rejection.

## Tune Standard-Form PID Controller

This example shows how to design a PID controller in standard form for the plant defined by



$$\text{sys} = \frac{1}{(s+1)^3}.$$

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

This example shows how to 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 =`

`Ts*z`

$$K_p + K_i * \frac{\text{-----}}{z-1}$$

with  $K_p = -0.518$ ,  $K_i = 10.4$ ,  $T_s = 0.1$

Sample time: 0.1 seconds  
Discrete-time PI controller in parallel form.

Using `CO` as an input causes `pidtune` to design a controller `C` of the same form, type, and discretization method as `CO`. The display shows that the integral term of `C` uses the Backward Euler integration method.

Specify a Trapezoidal integrator and compare the resulting controller.

```
CO_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  $K_i = 10.4$ ,  $T_s = 0.1$

Sample time: 0.1 seconds  
Discrete-time I-only controller.

## Algorithms

Typical PID tuning objectives include:

- Closed-loop stability — The closed-loop system output remains bounded for bounded input.
- Adequate performance — The closed-loop system tracks reference changes and suppresses disturbances as rapidly as possible. The larger the loop bandwidth (the first frequency at which the open-loop gain is unity), the faster the controller responds to changes in the reference or disturbances in the loop.

- Adequate robustness — The loop design has enough phase margin and gain margin to allow for modeling errors or variations in system dynamics.

The MathWorks algorithm for tuning PID controllers helps you meet these objectives by automatically tuning the PID gains to balance performance (response time) and robustness (stability margins).

By default, the algorithm chooses a crossover frequency (loop bandwidth) based upon the plant dynamics, and designs for a target phase margin of 60°. If you specify the crossover frequency using `wc` or the phase margin using `pidtuneOptions`, the algorithm computes PID gains that best meet those targets.

## References

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

## Alternatives

For interactive PID tuning, use the PID Tuner GUI (`pidtool`). See “PID Controller Design for Fast Reference Tracking” for an example of designing a controller using the PID Tuner GUI.

The PID Tuner GUI cannot design controllers for multiple plants at once.

## See Also

`pidtuneOptions` | `pid` | `pidstd` | `pidtool`

## Tutorials

- Designing Cascade Control System with PI Controllers

# pidtuneOptions

---

<b>Purpose</b>	Define options for the pidtune command
<b>Syntax</b>	<code>opt = pidtuneOptions</code> <code>opt = pidtuneOptions(Name,Value)</code>
<b>Description</b>	<code>opt = pidtuneOptions</code> returns the default option set for the pidtune command.  <code>opt = pidtuneOptions(Name,Value)</code> creates an option set with the options specified by one or more <code>Name,Value</code> pair arguments.
<b>Tips</b>	<ul style="list-style-type: none"><li>• Use <code>pidtuneOptions</code> with the <code>pidtune</code> command to design a PID controller for a specified phase margin.</li><li>• When using the <code>pidtune</code> command to design a PID controller for a plant with unstable poles, if your plant model is one of the following:<ul style="list-style-type: none"><li>▪ A <code>frd</code> model</li><li>▪ A <code>ss</code> model with internal delays that cannot be converted to I/O delays</li></ul>then use <code>pidtuneOptions</code> to specify the number of unstable poles in the plant.</li></ul>
<b>Input Arguments</b>	<b>Name-Value Pair Arguments</b> Specify optional comma-separated pairs of <code>Name,Value</code> arguments. <code>Name</code> is the argument name and <code>Value</code> is the corresponding value. <code>Name</code> must appear inside single quotes ( ' '). You can specify several name and value pair arguments in any order as <code>Name1,Value1,...,NameN,ValueN</code> .  <b>'PhaseMargin'</b> Target phase margin in degrees. <code>pidtune</code> attempts to design a controller such that the phase margin is at least the value specified for <code>PhaseMargin</code> . 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

## **'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 PI controller with a target phase margin of 45 degrees. Use pidtuneOptions to specify the phase margin:

```
sys = tf(1,[1 3 3 1]);  
opts = pidtuneOptions('PhaseMargin',45);  
[C,info] = pidtune(sys,'pid',opts);
```

## **See Also**

pidtune

# place

---

<b>Purpose</b>	Pole placement design
<b>Syntax</b>	$K = \text{place}(A,B,p)$ $[K, \text{prec}, \text{message}] = \text{place}(A,B,p)$
<b>Description</b>	Given the single- or multi-input system

$$\dot{x} = Ax + Bu$$

and a vector  $p$  of desired self-conjugate closed-loop pole locations, `place` computes a gain matrix  $K$  such that the state feedback  $u = -Kx$  places the closed-loop poles at the locations  $p$ . In other words, the eigenvalues of  $A - BK$  match the entries of  $p$  (up to the ordering).

$K = \text{place}(A,B,p)$  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  $B$ .

$[K, \text{prec}, \text{message}] = \text{place}(A,B,p)$  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  $B$ .

$l = \text{place}(A',C',p) . '$

## 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)
```

## 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

# pole

---

**Purpose** 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.

**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.

**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`).

**See Also** `damp` | `esort` | `dsort` | `pzmap` | `zero`



**Purpose** 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

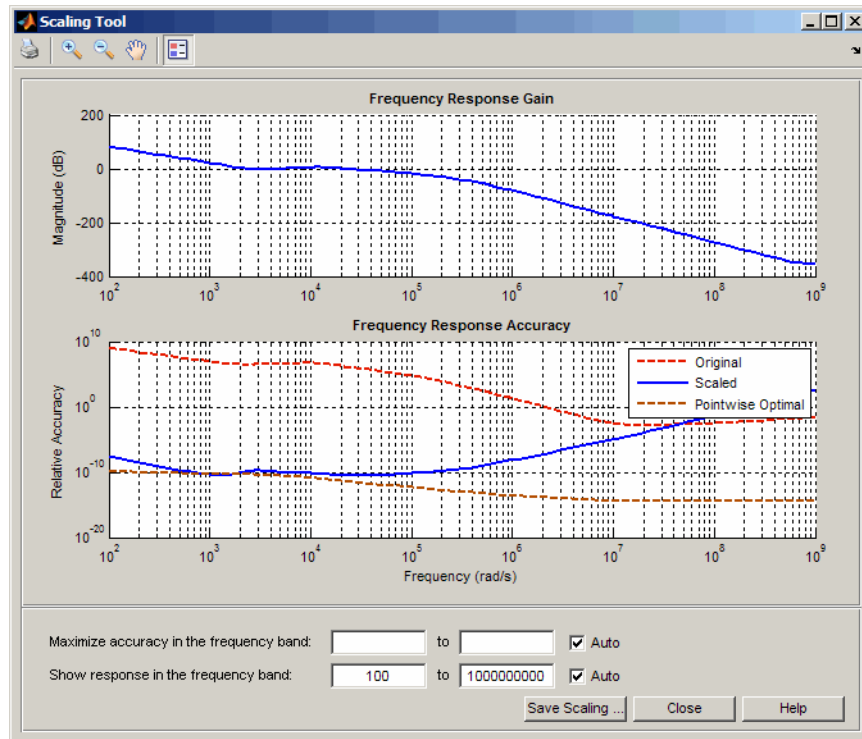
$$\begin{aligned}
 A_s &= T_L A T_R \\
 B_s &= T_L B \\
 C_s &= C T_R \\
 E_s &= T_L E T_R
 \end{aligned}$$

# prescale

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”.

## 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`

**Purpose** 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.

**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”.

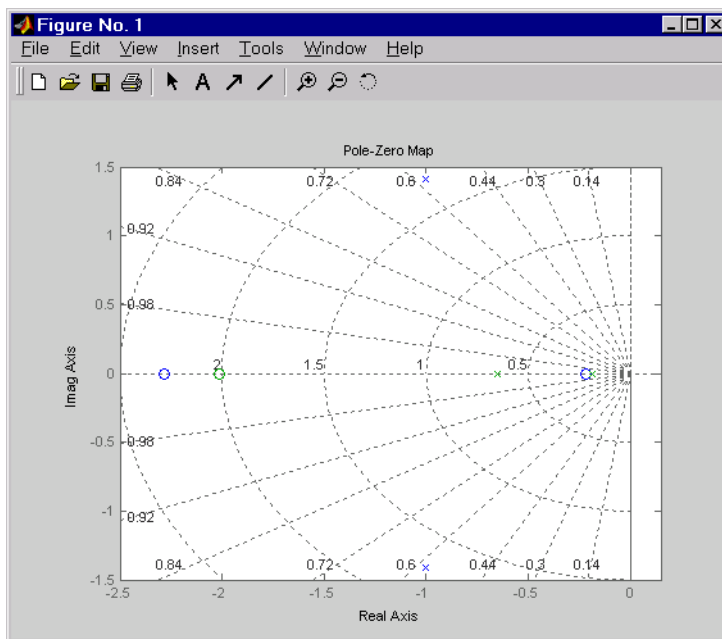
## Examples **Example 1**

### Pole-Zero Plot of Dynamic System

Plot the poles and zeros of the continuous-time system

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

```
H = tf([2 5 1],[1 2 3]); sgrid
pzmap(H)
grid on
```



### Example 2

Plot the pzmap for a 2-input-output discrete-time IDSS model.

```
A = [0.1 0; 0.2 0.9]; B = [.1 .2; 0.1 .02]; C = [10 20; 2 -5]; D = [1 2; 0 1];
sys = idss(A,B,C,D, 'Ts', 0.1);
```

### Algorithms

pzmap uses a combination of pole and zero.

### See Also

damp | esort | dsort | pole | rlocus | sgrid | zgrid | zero | iopzmap

**Purpose** 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.

`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.

**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”.

**Examples**

Use the plot handle to change the color of the plot’s title.

```
sys = rss(3,2,2);  
h = pzplot(sys);  
p = getoptions(h); % Get options for plot.  
p.Title.Color = [1,0,0]; % Change title color in options.  
setoptions(h,p); % Apply options to plot.
```

**See Also**

[getoptions](#) | [pzmap](#) | [setoptions](#) | [iopzplot](#)

# pzoptions

---

**Purpose** 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 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 strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOWGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles



<b>Option</b>	<b>Description</b>
InputVisible, OutputVisible	Visibility of input and output channels
FreqUnits	<p>Frequency units, specified as one of the following strings:</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> </ul>

# pzoptions

---

Option	Description
	<ul style="list-style-type: none"><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 multiple systems with different time units, the units of the first system are used.</p>
TimeUnits	<p>Time units, specified as one of the following strings:</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>

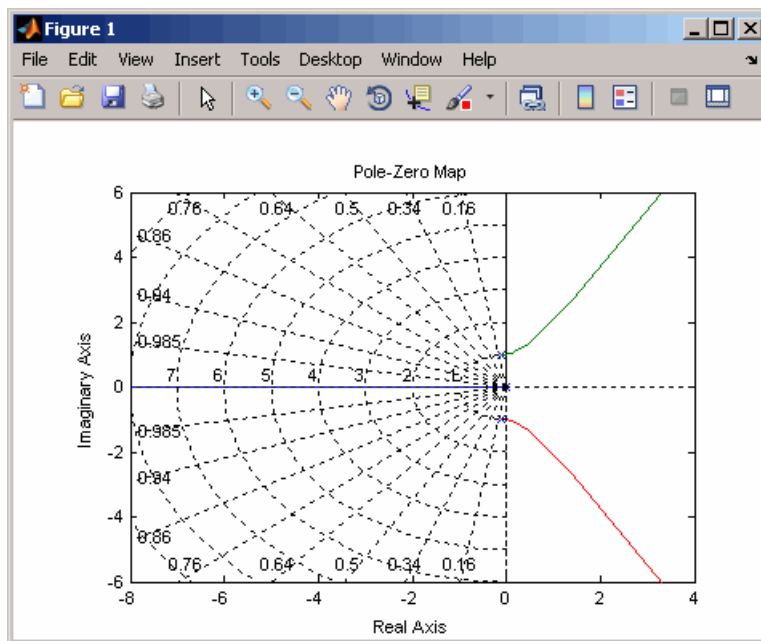
Option	Description
	You can also specify 'auto' which uses time units specified in the TimeUnit property of the input system. For multiple systems with different time units, the units of the first system is used.
ConfidenceRegionNumberSD	Number of standard deviations to use when displaying the confidence region characteristic for identified models (valid only iopzplot).

### 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**      `getoptions` | `iopzplot` | `pzplot` | `setoptions`

---

<b>Purpose</b>	Real tunable parameter
<b>Syntax</b>	<code>p = realp(paramname,initvalue)</code>
<b>Description</b>	<code>p = realp(paramname,initvalue)</code> creates a tunable real-valued parameter with name specified by the string <code>paramname</code> and initial value <code>initvalue</code> . Tunable real parameters can be scalar- or matrix-valued.
<b>Tips</b>	<ul style="list-style-type: none"><li>Use arithmetic operators (+, -, *, /, \, and ^) to combine <code>realp</code> objects into rational expressions or matrix expressions. You can use the resulting expressions in model-creation functions such as <code>tf</code>, <code>zpk</code>, and <code>ss</code> to create tunable models. For more information about tunable models, see “Models with Tunable Coefficients” in the <i>Control System Toolbox User’s Guide</i>.</li></ul>
<b>Input Arguments</b>	<p><b>paramname</b></p> <p>String specifying the name of the <code>realp</code> parameter <code>p</code>. This input argument sets the value of the <code>Name</code> property of <code>p</code>.</p> <p><b>initvalue</b></p> <p>Initial numeric value of the parameter <code>p</code>. <code>initvalue</code> can be a real scalar value or a 2-dimensional matrix.</p>
<b>Output Arguments</b>	<p><b>p</b></p> <p><code>realp</code> parameter object.</p>
<b>Properties</b>	<p><b>Name</b></p> <p>String containing the name of the <code>realp</code> parameter object. The value of <code>Name</code> is set by the <code>paramname</code> input argument to <code>realp</code> and cannot be changed.</p> <p><b>Value</b></p>

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****Tunable Low-Pass Filter**

This example shows how to create the low-pass filter  $F = a/(s + a)$  with one tunable parameter  $a$ .

You cannot use `ltiblock.tf` to represent  $F$ , because the numerator and denominator coefficients of an `ltiblock.tf` block are independent. Instead, construct  $F$  using the tunable real parameter object `realp`.

- 1 Create a tunable real parameter.

```
a = realp('a',10);
```

The `realp` object `a` is a tunable parameter with initial value 10.

- 2 Use `tf` to create the tunable filter `F`:

```
F = tf(a,[1 a]);
```

`F` is a `genss` object which has the tunable parameter `a` in its `Blocks` property. You can connect `F` with other tunable or numeric models to create more complex models of control systems. For an example, see “Control System with Tunable Components”.

---

**Parametric Diagonal Matrix**

This example shows how to create a parametric matrix whose off-diagonal terms are fixed to zero, and whose diagonal terms are tunable parameters.

- 1 Create a parametric matrix whose initial value is the identity matrix.

```
p = realp('P', eye(2));
```

`p` is a 2-by-2 parametric matrix. Because the initial value is the identity matrix, the off-diagonal initial values are zero.

- 2 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;
```

## See Also

[genss](#) | [genmat](#) | [tf](#) | [ss](#)

## How To

- “Models with Tunable Coefficients”



**Purpose**

Form regulator given state-feedback and estimator gains

**Syntax**

```
rsys = reg(sys,K,L)
rsys = reg(sys,K,L,sensors,known,controls)
```

**Description**

`rsys = reg(sys,K,L)` forms a dynamic regulator or compensator `rsys` given a state-space model `sys` of the plant, a state-feedback gain matrix `K`, and an estimator gain matrix `L`. The gains `K` and `L` are typically designed using pole placement or LQG techniques. The function `reg` handles both continuous- and discrete-time cases.

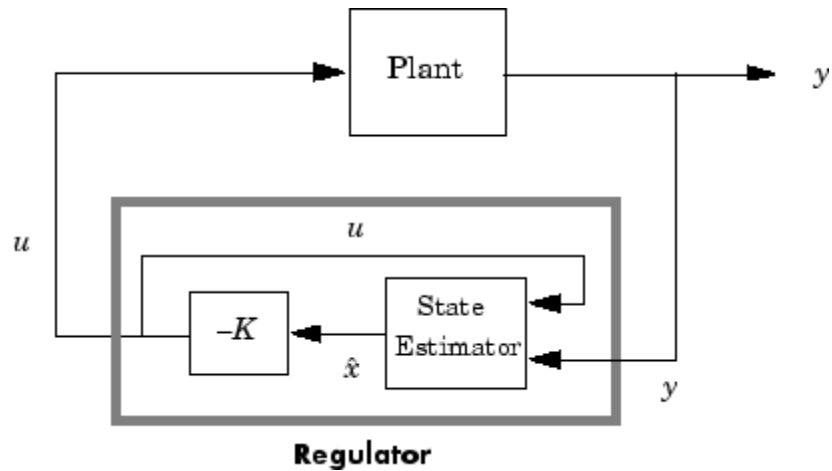
This syntax assumes that all inputs of `sys` are controls, and all outputs are measured. The regulator `rsys` is obtained by connecting the state-feedback law  $u = -Kx$  and the state estimator with gain matrix `L` (see `estim`). For a plant with equations

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

this yields the regulator

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

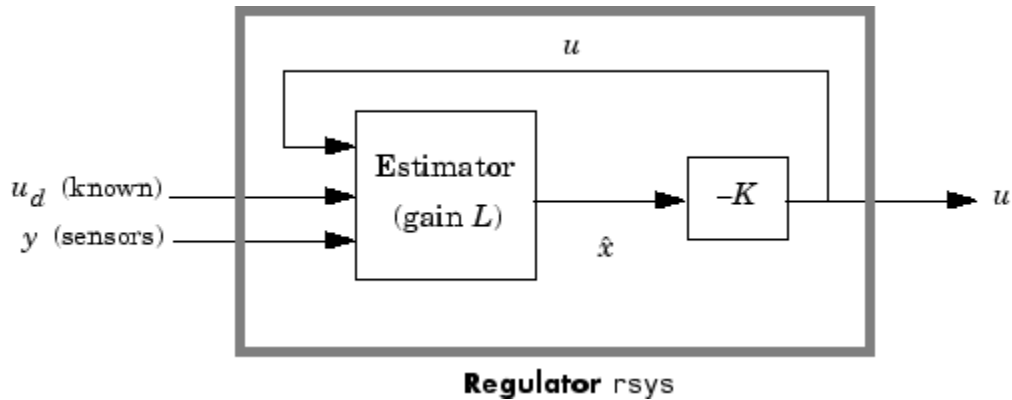
This regulator should be connected to the plant using *positive* feedback.



`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](#) | [lqgreg](#) | [lqr](#) | [dlqr](#) | [place](#)

# replaceBlock

---

<b>Purpose</b>	Replace or update Control Design Blocks in Generalized LTI model
<b>Syntax</b>	<pre>Mnew = replaceBlock(M,Block1,Value1,...,BlockN,ValueN) Mnew = replaceBlock(M,blockvalues) Mnew = replaceBlock(...,mode)</pre>
<b>Description</b>	<p><code>Mnew = replaceBlock(M,Block1,Value1,...,BlockN,ValueN)</code> replaces the Control Design Blocks <code>Block1, ..., BlockN</code> of <code>M</code> with the specified values <code>Value1, ..., ValueN</code>. <code>M</code> is a Generalized LTI model or a Generalized matrix.</p> <p><code>Mnew = replaceBlock(M,blockvalues)</code> specifies the block names and replacement values as field names and values of the structure <code>blockvalues</code>.</p> <p><code>Mnew = replaceBlock(...,mode)</code> performs block replacement on an array of models <code>M</code> using the substitution mode specified by the string <code>mode</code>.</p>
<b>Tips</b>	<ul style="list-style-type: none"><li>• Use <code>replaceBlock</code> 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 1-568.</li></ul>
<b>Input Arguments</b>	<p><b>M</b> Generalized LTI model, Generalized matrix, or array of such models.</p> <p><b>Block1,...,BlockN</b> Names of Control Design Blocks in <code>M</code>. The <code>replaceBlock</code> command replaces each listed block of <code>M</code> with the corresponding values <code>Value1, ..., ValueN</code> that you supply.</p> <p>If a specified Block is not a block of <code>M</code>, <code>replaceBlock</code> that block and the corresponding value.</p> <p><b>Value1,...,ValueN</b></p>

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**

String specifying the block replacement mode for an input array `M` of Generalized matrices or LTI models.

`mode` can take 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.

# replaceBlock

- '-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.

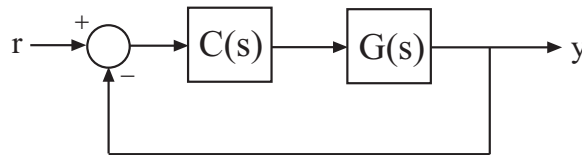
Mnew is a numeric array or numeric LTI model when all the specified replacement values are numeric values or numeric LTI models.

## Examples

### Replace Control Design Block with Numeric Values

This example shows how to replace a tunable PID controller (ltiblock.pid) 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 = ltiblock.pid('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 a Parametric Model over a Matrix of Parameter Values.

This example shows how to sample a parametric model of a second-order filter across a grid of parameter values using `replaceBlock`.

- 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 the parameters.

```
wn = realp('wn', 3);
zeta = realp('zeta', 0.8);
F = tf(wn^2, [1 2*zeta*wn wn^2])
```

```
F =
```

# replaceBlock

---

Generalized continuous-time state-space model with 1 outputs, 1 input  
wn: Scalar parameter, 5 occurrences.  
zeta: Scalar parameter, 1 occurrences.

Type "ss(F)" to see the current value, "get(F)" to see all properties,

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 Sample F over a 2-by-3 grid of (wn,zeta) values.

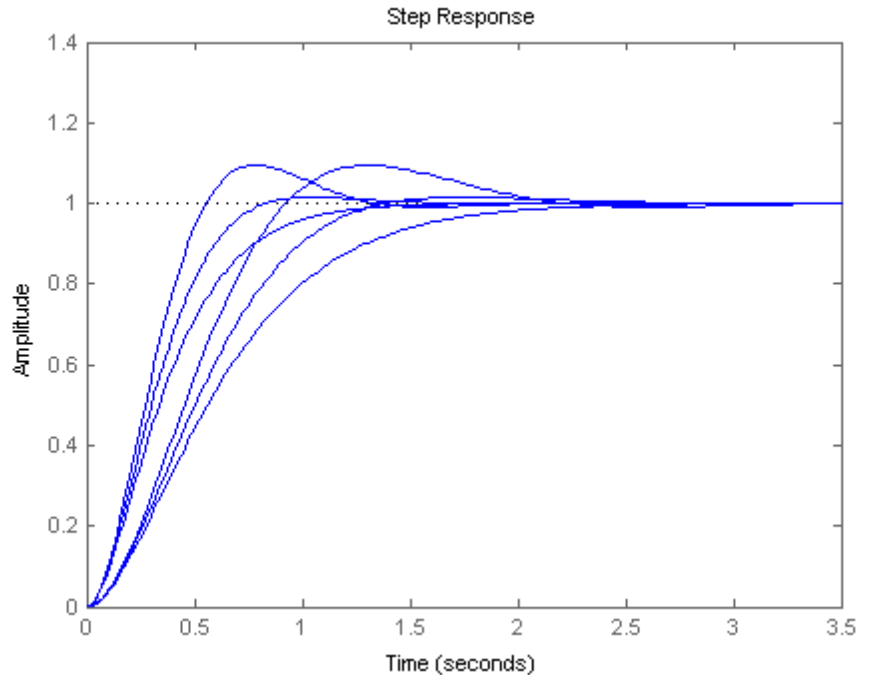
```
wnvals = [3;5];  
zetavals = [0.6 0.8 1.0];  
Fsample = replaceBlock(F, 'wn',wnvals, 'zeta',zetavals);
```

Fsample is 2-by-3 array of state-space models. Each entry in the array is the transfer function for the corresponding (wn,zeta) pair.

- 3 Plot the step response of Fsample.

```
step(Fsample)
```





The step response plots show the variation in the natural frequency and damping constant across the six models in the array `Fsample`.

- 4 You can set the `SamplingGrid` property of the model array to help keep track of which set of parameter values corresponds to which entry in the array. To do so, create a grid of parameter values that matches the dimensions of the array. Then, assign these values to `Fsample.SamplingGrid` with the parameter names.

```
[wngrid,zetagrid] = ndgrid(wnvals,zetavals);  
Fsample.SamplingGrid = struct('wn',wngrid,'zeta',zetagrid);
```

# replaceBlock

---

When you display `H`, the parameter values in `Fsample.SamplingGrid` are displayed along with the each transfer function in the array.

## See Also

`getValue` | `genss` | `genmat` | `nblocks`

## How To

- “Generalized Matrices”
- “Generalized and Uncertain LTI Models”
- “Models with Tunable Coefficients”

<b>Purpose</b>	Replicate and tile models
<b>Syntax</b>	<pre>rsys = repsys(sys,[M N]) rsys = repsys(sys,N) rsys = repsys(sys,[M N S1,...,Sk])</pre>
<b>Description</b>	<p><code>rsys = repsys(sys,[M N])</code> replicates the model <code>sys</code> into an M-by-N tiling pattern. The resulting model <code>rsys</code> has <code>size(sys,1)*M</code> outputs and <code>size(sys,2)*N</code> inputs.</p> <p><code>rsys = repsys(sys,N)</code> creates an N-by-N tiling.</p> <p><code>rsys = repsys(sys,[M N S1,...,Sk])</code> replicates and tiles <code>sys</code> along both I/O and array dimensions to produce a model array. The indices <code>S</code> specify the array dimensions. The size of the array is <code>[size(sys,1)*M, size(sys,2)*N, size(sys,3)*S1, ...]</code>.</p>
<b>Tips</b>	<p><code>rsys = repsys(sys,N)</code> produces the same result as <code>rsys = repsys(sys,[N N])</code>. To produce a diagonal tiling, use <code>rsys = sys*eye(N)</code>.</p>
<b>Input Arguments</b>	<p><b>sys</b> Model to replicate.</p> <p><b>M</b> Number of replications of <code>sys</code> along the output dimension.</p> <p><b>N</b> Number of replications of <code>sys</code> along the input dimension.</p> <p><b>S</b> Numbers of replications of <code>sys</code> along array dimensions.</p>
<b>Output Arguments</b>	<p><b>rsys</b> Model having <code>size(sys,1)*M</code> outputs and <code>size(sys,2)*N</code> inputs.</p>

If you provide array dimensions  $S_1, \dots, S_k$ , `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.
```

## See Also

`append`

**Purpose** 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** `ndims` | `size`

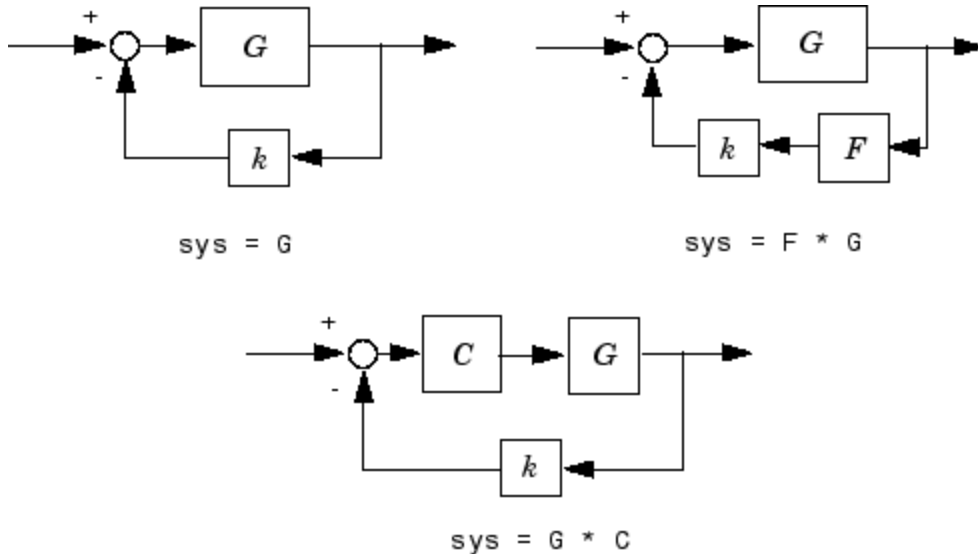
# rlocus

**Purpose** 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  $k(j)$ .

## 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”.

## Examples

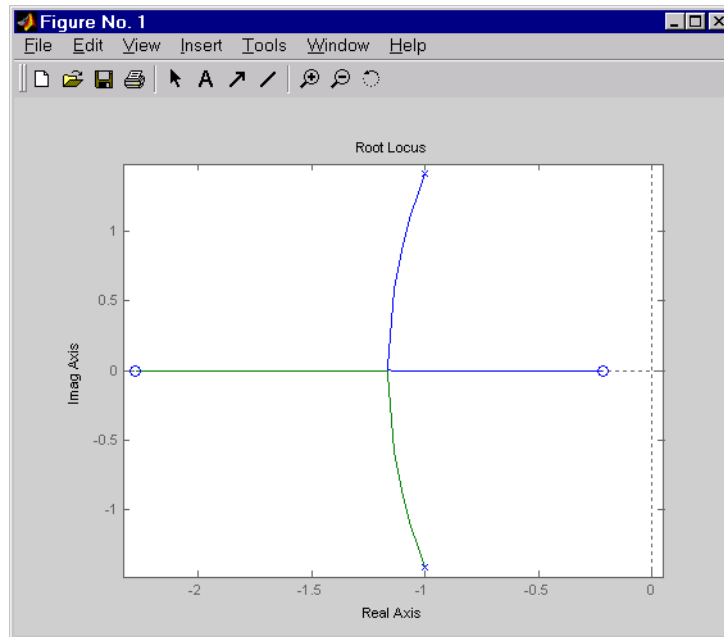
### Root Locus Plot of Dynamic System

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.

## See Also

[pole](#) | [pzmap](#)



**Purpose** 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.

# rlocusplot

---

## 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”.

## 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.
```

## See Also

[getoptions](#) | [rlocus](#) | [pzoptions](#) | [setoptions](#)

**Purpose** 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 nth 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 s1-by-...-by-sn 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** Obtain a random continuous LTI model with three states, two inputs, and two outputs by typing

```
sys = rss(3,2,2)
a =
           x1           x2           x3
x1    -0.54175    0.09729    0.08304
x2     0.09729   -0.89491    0.58707
x3     0.08304    0.58707   -1.95271

b =
           u1           u2
x1    -0.88844   -2.41459
x2         0    -0.69435
x3    -0.07162   -1.39139

c =
```

	x1	x2	x3
y1	0.32965	0.14718	0
y2	0.59854	-0.10144	0.02805

d =

	u1	u2
y1	-0.87631	-0.32758
y2	0	0

Continuous-time system.

**See Also**

[drss](#) | [frd](#) | [tf](#) | [zpk](#)

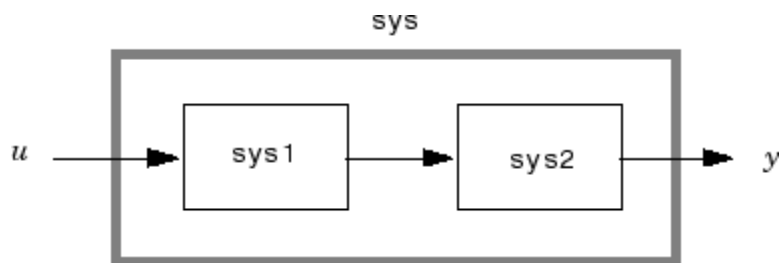
**Purpose** Series connection of two models

**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.

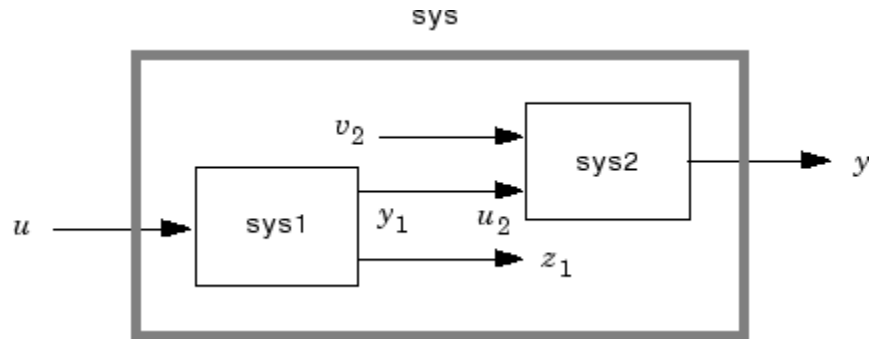


This command is equivalent to the direct multiplication

```
sys = sys2 * sys1
```

`sys = series(sys1,sys2,outputs1,inputs2)` forms the more general series connection.

# series



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`

**Purpose**

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. Like its Handle Graphics<sup>®</sup> counterpart, set uses property name/property value pairs to update property values.

set(sys, 'Property', Value) assigns the value Value to the property of the model sys specified by the string 'Property'. This string can be the full property name (for example, 'UserData') or any unambiguous case-insensitive abbreviation (for example, 'user'). The specified property must be compatible with the model type. For example, if sys is a transfer function, Variable is a valid property but StateName is not. 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
```

## 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.

---

## See Also

get | frd | ss | tf | zpk

## Tutorials

- “Model Properties”

## How To

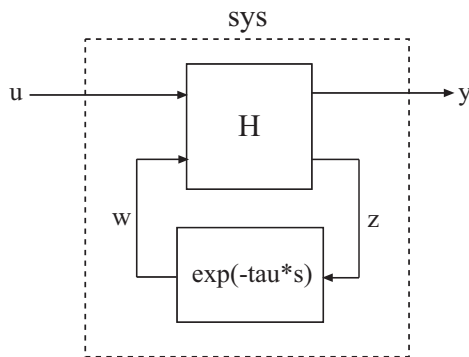
- “What Are Model Objects?”

# setDelayModel

**Purpose** 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`.

## Tips

- `setDelayModel` is an advanced operation and is not the natural way to construct models with internal delays. See “Models with Time Delays” 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 sampling time.

## 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 sampling 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`.

# setDelayModel

---

## See Also

[getDelayModel](#) | [ss](#) | [lft](#)

## Concepts

- “Internal Delays”
- “Models with Time Delays”

**Purpose**

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

# setoptions

properties and values are available for a particular plot, type `help <function>options`. For example, for Bode plots type

```
help bodeoptions
```

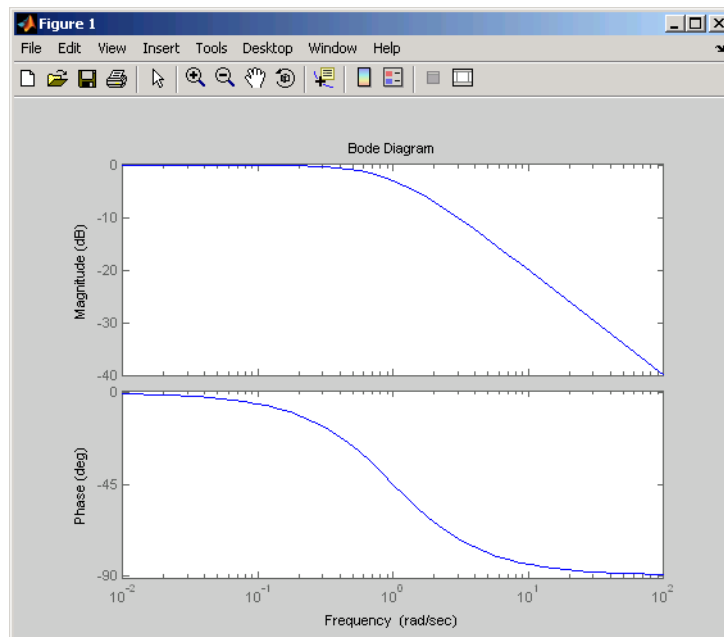
For a list of the properties and values available for each plot type, see “Properties and Values Reference”.

`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

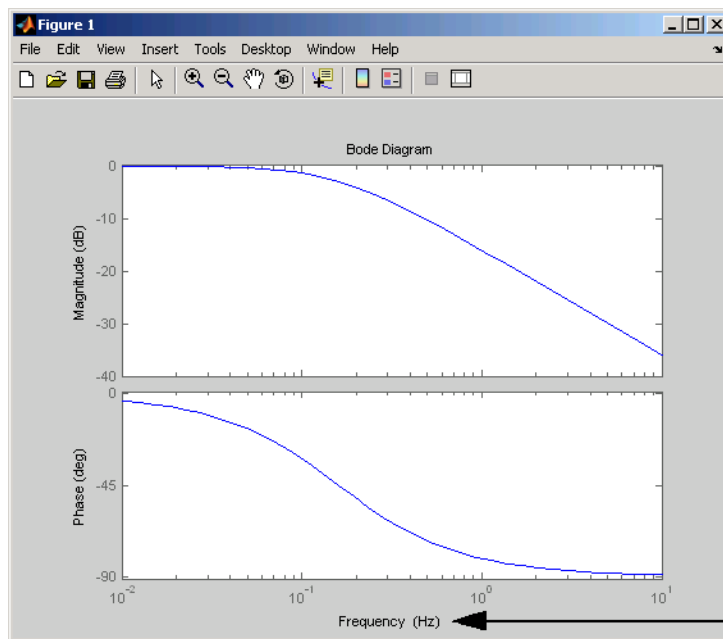
To change frequency units, first create a Bode plot.

```
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

# setBlockValue

---

**Purpose** 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 sampling time of `blockname`.

For example, you can set the value of a tunable PID block (`ltiblock.pid`) 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 the Robust Control Toolbox tuning commands such as `systemtune`, `looptune`, or `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 LTI 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 `hinfstruct`.

# setBlockValue

---

```
G = tf([1,0.0007],[1,0.00034,0.00086]);
Cpi = ltiblock.pid('Cpi','pi');
a = realp('a',10);
FO = tf(a,[1 a]);
CO = Cpi*FO;
TO = feedback(G*CO,1);

T = hinfstruct(TO);
```

The controller model `CO` is a Generalized LTI model with two tunable blocks, `Cpi` and `a`. The closed-loop model `TO` is also a Generalized LTI model with the same blocks. The model `T` contains the tuned values of these blocks.

Propagate the tuned values of the controller in `T` to the controller model `CO`.

```
C = setBlockValue(CO,T)
```

```
C =
```

Generalized continuous-time state-space model with 1 outputs, 1 inputs, 2 states, and the following blocks:

`Cpi`: Parametric PID controller, 1 occurrences.

`a`: Scalar parameter, 2 occurrences.

Type "`ss(C)`" to see the current value, "`get(C)`" to see all properties, and "`C.Blocks`" to interact with the blocks.

`C` is still a Generalized model. The current value of the Control Design Blocks in `C` are set to the values the corresponding blocks of `T`.

Obtain a Numeric LTI model of the controller with the tuned values using `getValue`.

```
CVa1 = getValue(CO,T);
```

This command returns a numerical state-space model of the tuned controller.

## See Also

`getValue` | `getBlockValue` | `showBlockValue` | `genss` | `systune`  
| `looptune` | `hinfstruct`

# setValue

---

**Purpose** 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 sampling time of `blk0`. For example, if `blk0` is a `ltiblock.pid` block, valid types for `val` include `ltiblock.pid`, 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`.

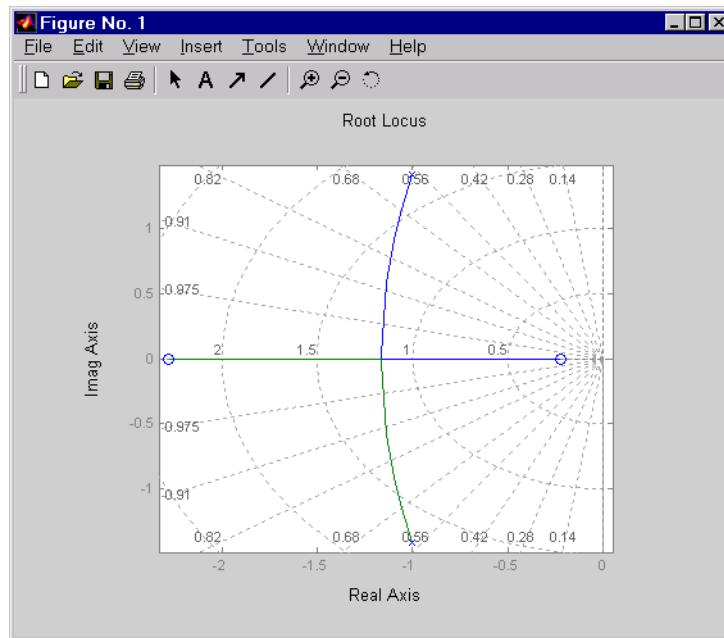
**See Also** `getValue` | `setBlockValue` | `getBlockValue`

- Purpose** 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** [pzmap](#) | [rlocus](#) | [zgrid](#)

<b>Purpose</b>	Display current value of Control Design Blocks in Generalized Model
<b>Syntax</b>	<code>showBlockValue(M)</code>
<b>Description</b>	<code>showBlockValue(M)</code> 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.)
<b>Input Arguments</b>	<b>M</b> Generalized Model.
<b>Examples</b>	Create a tunable genss model, and display the current value of its tunable elements.  <pre>G = zpk([], [-1, -1], 1); C = ltiblock.pid('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 Ki * ---       s  With Ki = 0.001  ----- a = 10</pre>

# showBlockValue

---

## Tips

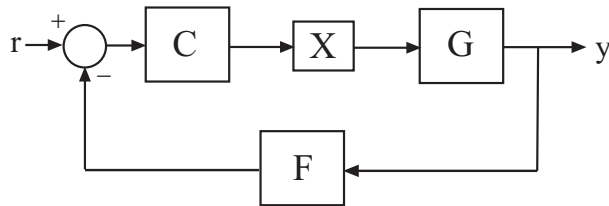
- Displaying the current values of a model is useful, for example, after you have tuned the free parameters of the model using a Robust Control Toolbox tuning command such as `looptune`.
- `showBlockValue` displays the current values of all Control Design Blocks in a model, including tunable, uncertain, and switch blocks. To display the current values of only the tunable blocks, use `showTunable`.

## See Also

`genss` | `getBlockValue` | `setBlockValue` | `showTunable`



<b>Purpose</b>	Display current value of tunable Control Design Blocks in Generalized Model
<b>Syntax</b>	<code>showTunable(M)</code>
<b>Description</b>	<code>showTunable(M)</code> displays the current values of all tunable Control Design Blocks in a generalized LTI model. Tunable control design blocks are parametric blocks such as <code>realp</code> , <code>ltiblock.tf</code> , and <code>ltiblock.pid</code> .
<b>Input Arguments</b>	<p><b>M - Input model</b> generalized LTI model</p> <p>Input model of which to display tunable block values, specified as a generalized LTI model such as a <code>genss</code> model.</p>
<b>Examples</b>	<p><b>Display Block Values of Tuned Control System Model</b></p> <p>Tune the following control system using <code>systemtune</code>, and display the values of the tunable blocks.</p>



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');
```

# showTunable

---

```
num = 33000*(s^2 - 200*s + 90000);
den = (s + 12.5)*(s^2 + 25*s + 63000);
G = num/den;

C0 = ltiblock.pid('C','pi');
a = realp('a',1);
F0 = tf(a,[1 a]);
X = loopswitch('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);
```

`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 = 0.00527
```

```
Name: C
Continuous-time PI controller in parallel form.
-----
a = 68.6
```

`showTunable` displays the values of the tunable blocks only. If you use `showBlockValue` instead, the display also includes the switch block X.

## Tips

- Displaying the current values of tunable blocks is useful, for example, after you have tuned the free parameters of the model using a Robust Control Toolbox 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`.

## See Also

`genss` | `getBlockValue` | `setBlockValue` | `showBlockValue` | `systemtune`

## Concepts

- “Generalized Models”
- “Control Design Blocks”

**Purpose** 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 `[wmin,wmax]`, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the corresponding vector of frequencies. Use `logspace`

to generate logarithmically spaced frequency vectors. 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:

`type = 1` Singular values of the frequency response  $H^{-1}$ , where  $H$  is the frequency response of sys.

`type = 2` Singular values of the frequency response  $I + H$ .

`type = 3` Singular values of the frequency response  $I + H^{-1}$ .

These options are available only for square systems, that is, with the same number of inputs and outputs.

`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 `Nu` input and `Ny` outputs, the array `sv` has `min(Nu, Ny)` rows and as many columns as frequency points (length of `w`). The singular values at the frequency `w(k)` are given by `sv(:, k)`.

## 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”.

## Examples

### Singular Values Plot of Dynamic System

Plot the singular value responses of

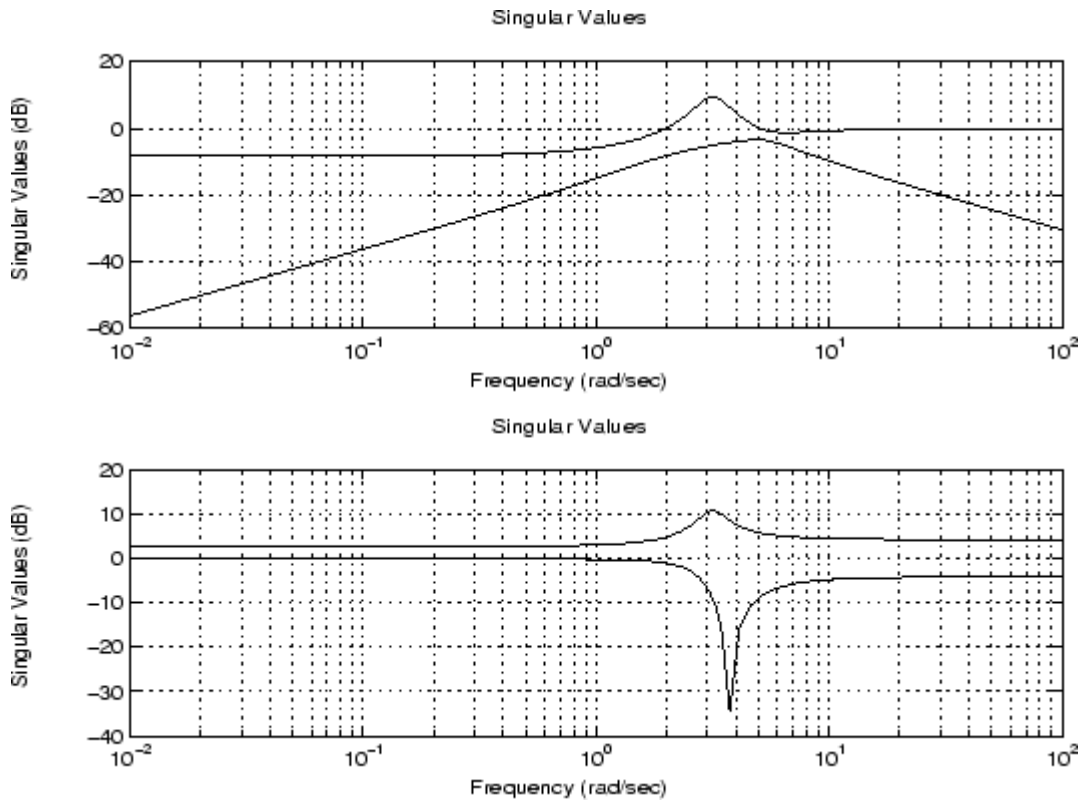
$$H(s) = \begin{bmatrix} 0 & \frac{3s}{s^2 + s + 10} \\ \frac{s+1}{s+5} & \frac{2}{s+6} \end{bmatrix}$$

and  $I + H(s)$ .

You can do this by typing

```
H = [0 tf([3 0],[1 1 10]) ; tf([1 1],[1 5]) tf(2,[1 6])]
```

```
subplot(211)  
sigma(H)  
subplot(212)  
sigma(H,[],2)
```



## 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` | `ltiview` | `nichols` | `nyquist`

# sigmaoptions

---

**Purpose** 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 strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOWGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   '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 strings:</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> </ul>

# sigmaoptions

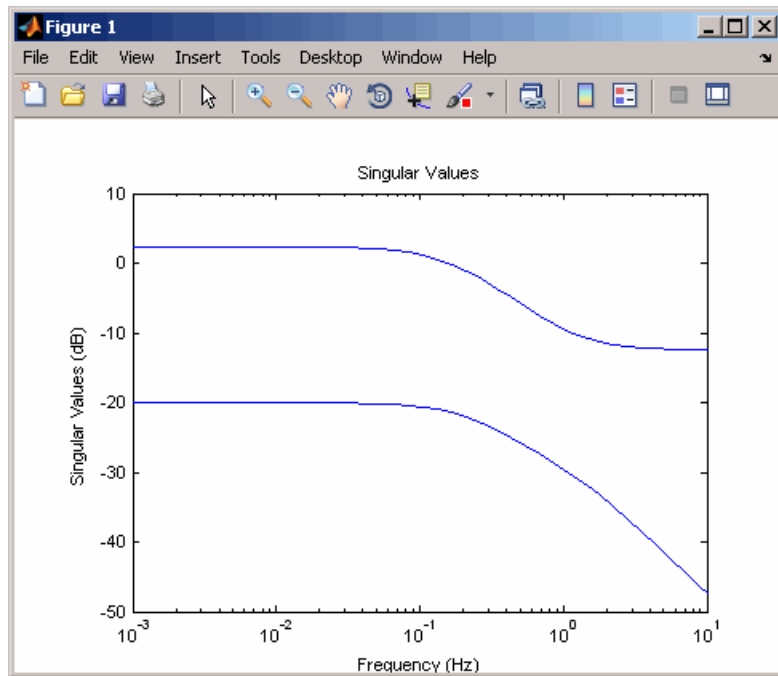
Option	Description
	<ul style="list-style-type: none"><li>'cycles/year'</li></ul> <p><b>Default:</b> 'rad/s'</p> <p>You can also specify 'auto' which uses frequency units rad/TimeUnit relative to system time units specified in the TimeUnit property. For multiple systems with different time units, the units of the first system are used.</p>
FreqScale	Frequency scale Specified as one of the following strings: 'linear'   'log' <b>Default:</b> 'log'
MagUnits	Magnitude units Specified as one of the following strings: 'dB'   'abs' <b>Default:</b> 'dB'
MagScale	Magnitude scale Specified as one of the following strings: '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](#)

# sigmaplot

---

**Purpose** 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.

## 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”.

## Examples

Use the plot handle to change the units to Hz.

```
sys = rss(5);  
h = sigmaplot(sys);  
% Change units to Hz.  
setoptions(h,'FreqUnits','Hz');
```

## See Also

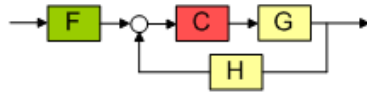
`getoptions` | `setoptions` | `sigma` | `sigmaoptions`

**Purpose** Configure SISO Design Tool at startup

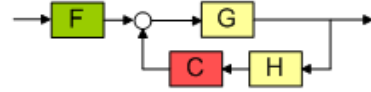
**Syntax** `init_config = sisoinit(config)`

**Description** `init_config = sisoinit(config)` returns a template `init_config` for initializing Graphical Tuning window of the SISO Design Tool with the one of the following control system configurations:

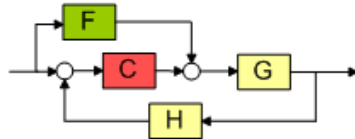
**Configuration 1**



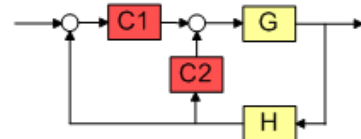
**Configuration 2**



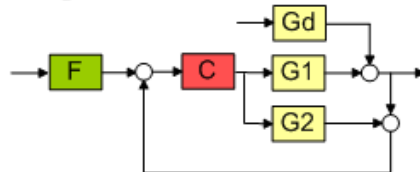
**Configuration 3**



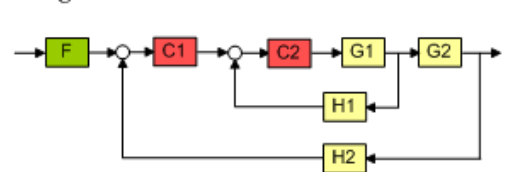
**Configuration 4**



**Configuration 5**



**Configuration 6**



`config` corresponds to the control system configuration. Available configurations include:

- `config = 1` (default) — C in forward path, F in series
- `config = 2` — C in feedback path, F in series
- `config = 3` — C in forward path, feedforward F
- `config = 4` — Nested loop configuration

- `config = 5` — Internal model control (IMC) structure
- `config = 6` — Cascade loop configuration

For each configuration, you can specify the plant models `G` and `H`, initialize the compensator `C` and prefilter `F`, and configure the open- and closed-loop views by specifying the corresponding fields of the structure `init_config`. Then use `sisotool(init_config)` to start the SISO Design Tool in the specified configuration.

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	String
	Description	String
	Value	LTI object
G	Name	String
	Value	<ul style="list-style-type: none"> <li>• LTI object</li> <li>• Row or column array of LTI objects. If the sensor <code>H</code> is also an array of LTI objects, the lengths of <code>G</code> and <code>H</code> must match.</li> </ul>
H	Name	String
	Value	<ul style="list-style-type: none"> <li>• LTI object</li> <li>• Row or column array of LTI objects. If the plant <code>G</code> is also an array of LTI objects, the lengths of <code>H</code> and <code>G</code> must match.</li> </ul>

## Block Properties (Continued)

Block	Properties	Values
C	Name	String
	Description	String
	Value	LTI object

## Loop Properties

Loops	Properties	Values
OL1	Name	String
	Description	String
	View	'rlocus' 'bode'
CL1	Name	String
	Description	String
	View	'bode'

## Examples

Initialize SISO Design Tool with C in feedback path using an LTI model:

```
% Single-loop configuration with C in the feedback path.
T = sisoinit(2);
% Model for plant G.
T.G.Value = tf(1, [1 1]);
% Initial compensator value.
T.C.Value = tf(1,[1 2]);
% Views for tuning Open-Loop OL1.
T.OL1.View = {'rlocus','nichols'};
% Launch SISO Design Tool using configuration T
sisotool(T)
```

---



Initialize SISO Design Tool with C in feedback path using an array of LTI models:

```
% Specify an initial configuration.
initconfig = sisoinit(2);
% Specify model parameters.
m = 3;
b = 0.5;
k = 8:1:10;
T = 0.1:.05:.2;
% Create an LTI array 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;
% Create an LTI array to model variations in sensor H.
for ct = 1:length(T);
    H(:, :, ct) = tf(1, [1/T(ct), 1]);
end
% Assign H to the initial configuration.
initconfig.H.Value = H;
% Specify initial controller.
initconfig.C.Value = tf(1, [1 2]);
% Views for tuning Open-Loop (OL1)
initconfig.OL1.View = {'rlocus', 'bode'};
% Launch SISO Design Tool using initconfig.
sisotool(initconfig)
```

## See Also

sisotool

## How To

- “SISO Design Tool”
- “Control Design Analysis of Multiple Models”

**Purpose** Interactively design and tune SISO feedback loops

**Syntax**

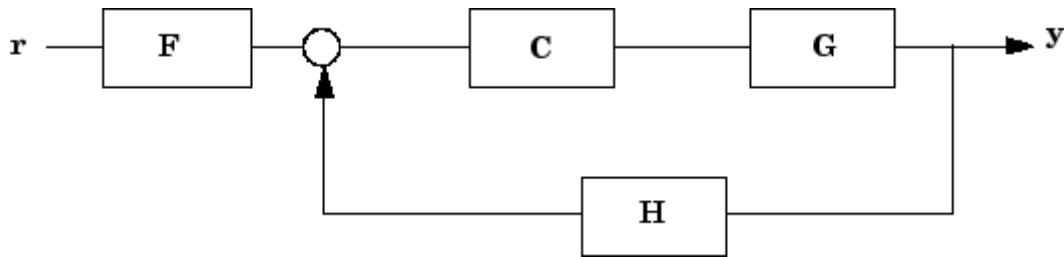
```
sisotool
sisotool(plant)
sisotool(plant,comp)
sisotool(plant,comp,sensor,prefilt)
sisotool(views)
sisotool(views,plant,comp)
sisotool(initdata)
sisotool(sessiondata)
```

**Description** `sisotool` opens a SISO Design GUI for interactive compensator design. This GUI allows you to design a single-input/single-output (SISO) compensator using root locus, Bode diagram, Nichols and Nyquist techniques. You can also automatically design a compensator using this GUI.

By default, the SISO Design Tool:

- Opens the Control and Estimation Tools Manager with a default SISO Design Task node.
- Opens the Graphical Tuning editor with root locus and open-loop Bode diagrams.
- Places the compensator, **C**, in the forward path in series with the plant, **G**.
- Assumes the prefilter, **F**, and the sensor, **H**, are unity gains. Once you specify **G** and **H**, they are *fixed* in the feedback structure.

The default control architecture is shown in this figure.



There are six control architectures available. See `sisoinit` for more information.

This picture shows the SISO Design Graphical editor.

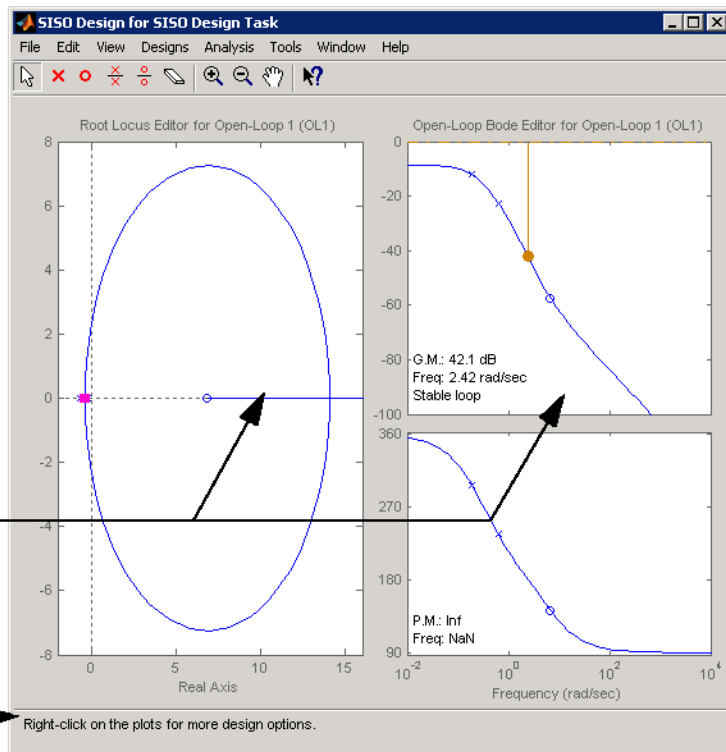


Table 0-1: Use the right-click menu to manipulate the compensator and the plots' appearances. Right-click in any plot

The status bar provides useful information.

`sisotool(plant)` opens the SISO Design Tool, imports `plant`, and initializes the plant model **G** to `plant`. `plant` can be any SISO LTI model created with `ss`, `tf`, `zpk` or `frd`, or a row or column array of LTI models.

`sisotool(plant,comp)` initializes the plant model **G** to `plant`, the compensator **C** to `comp`. `comp` is an LTI object.

`sisotool(plant,comp,sensor,prefilt)` initializes the plant **G** to `plant`, compensator **C** to `comp`, sensor **H** to `sensor`, and the prefilter **F** to `prefilt`. `sensor` is an LTI object or a row or column array of LTI objects. If `plant` is also an array of LTI objects, the lengths of `sensor` and `plant` must match. `prefilt` is an LTI object.

`sisotool(views)` or `sisotool(views,plant,comp)` specifies the initial configuration of the SISO Design Tool. `views` can be any of the following strings (or combination thereof):

- `'rlocus'` — Root Locus plot
- `'bode'` — Bode diagrams of the open-loop response
- `'nichols'` — Nichols plot
- `'filter'` — Bode diagrams of the prefilter **F** and the closed-loop response from the command into **F** to the output of the plant **G**.

For example

```
sisotool('bode')
```

opens a SISO Design Tool with only the Bode Diagrams. If there is more than one view, the views are specified in a cell array.

`sisotool(initdata)` initializes the SISO Design Tool with more general control system configurations. Use `sisoinit` to create the initialization data structure `initdata`.

`sisotool(sessiondata)` opens the SISO Design Tool with a previously saved session where `sessiondata` is the MAT-file for the saved session.

## Examples

Launch SISO Design Tool GUI in default configuration using LTI models:

```
% Create plant G.
G = tf(1, [1 1]);
% Create controller C.
C = tf(1,[1 2]);
% Launch the GUI.
sisotool(G,C)
```

Launch SISO Design Tool GUI in default configuration using an array of LTI models:

```
% Specify model parameters.
m = 3;
b = 0.5;
k = 8:1:10;
T = 0.1:.05:.2;
% Create an LTI array to model variations in plant G.
for ct = 1:length(k);
    G(:,:,ct) = tf(1,[m,b,k(ct)]);
end
% Create an LTI array to model variations in sensor H.
for ct = 1:length(T);
    H(:,:,ct) = tf(1,[1/T(ct), 1]);
end
% Create a controller C.
C = tf(1,[1 2]);
% Launch the GUI.
sisotool(G,C,H)
```

## See Also

[bode](#) | [ltiview](#) | [rlocus](#) | [nichols](#) |

## Tutorials

- “How to Analyze the Controller Design for Multiple Models”
- “Bode Diagram Design”

- “Root Locus Design”
- “Nichols Plot Design”
- “Position Control of a DC Motor”
- “SISO Design Tool”

## **How To**

**Purpose**

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

# size

---

```
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

```
isempty | issiso | ndims
```



**Purpose** 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`.

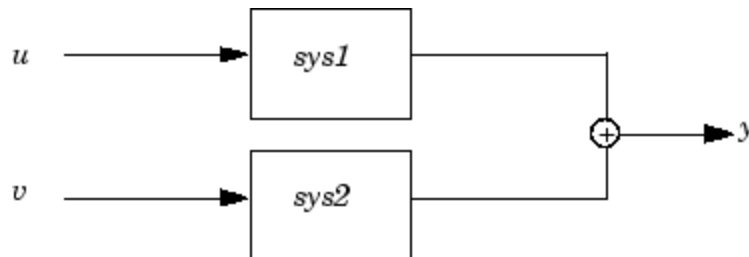
*sminreal* eliminates only structurally non minimal states, i.e., states that can be discarded by looking only at hard zero entries in the *A*, *B*, and *C* matrices. Such structurally nonminimal states arise, for example, when linearizing a Simulink model that includes some unconnected state-space or transfer function blocks.

**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.

**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)
```

# sminreal

---

all of the states of `sys`, including those of `sys2` are retained. To eliminate the unobservable states from `sys2`, while retaining the states of `sys1`, type

```
sminreal(sys(1,1))
```

## See Also

`minreal`

**Purpose** 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,lthisys)
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 1-632 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  $y_{\text{noise}} = HLv(t)$ . 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  $n_y + n_u$  inputs such that the first  $n_u$  inputs represent the channels  $u(t)$

while the remaining by channels represent the noise channels  $v(t)$ . `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  $E dx/dt = Ax + Bu$ , 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. For first-order models, set `StateName` to a string. For models with two or more states, set `StateName` to a cell array of strings. Use an empty string `''` for unnamed states.

**Default:** Empty string `''` for all states

### StateUnit

State units. Use `StateUnit` to keep track of the units each state is expressed in. For first-order models, set `StateUnit` to a string. For models with two or more states, set `StateUnit` to a cell array of strings. `StateUnit` has no effect on system behavior.

**Default:** Empty string `''` 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 sampling period  $T_s$ . 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 delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. 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 sampling period  $T_s$ . For example, `InputDelay = 3` means a delay of three sampling periods.

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 for all input channels

### **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 sampling period  $T_s$ . 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

### **Ts**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'

- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input

---

model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string '' for all input channels

### **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set Name to a string to label the system.

**Default:** ''

**Notes**

Any text that you want to associate with the system. Set Notes to a string or a cell array of strings.

**Default:** {}

**UserData**

Any type of data you wish to associate with system. Set UserData to 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
```

...

**Default:** []

## 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`.

## Examples

### Example 1

#### Discrete-Time State-Space Model

Create a state-space model with a sampling time of 0.25 s and the following state-space matrices:

$$A = \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 3 \end{bmatrix} \quad C = [0 \ 1] \quad D = [0]$$

To do this, enter the following commands:

```
A = [0 1; -5 -2];
B = [0; 3];
C = [0 1];
D = 0;
sys = ss(A,B,C,D,0.25);
```

The last argument sets the sampling time.

### Example 2

#### Discrete-Time State-Space Model with Custom State and Input Names

Create a discrete-time model with matrices A,B,C,D and sample time 0.05 second.

```
sys = ss(A,B,C,D,0.05, 'statename', {'position' 'velocity'}, ...
        'inputname', 'force', ...
        'notes', 'Created 01/16/11');
```

This model has two states labeled `position` and `velocity`, and one input labeled `force` (the dimensions of A,B,C,D should be consistent with these numbers of states and inputs). Finally, a note is attached with the date of creation of the model.

### Example 3

#### Convert Transfer Function Model to State-Space Model

Convert a transfer function model to a state-space model.

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

by typing

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



```
size(sys)
State-space model with 2 outputs, 1 input, and 5 states.
```

The number of states is equal to the cumulative order of the SISO entries of  $H(s)$ .

To obtain a minimal realization of  $H(s)$ , type

```
sys = ss(H, 'min');
size(sys)
State-space model with 2 outputs, 1 input, and 3 states.
```

The resulting state-space model has order of three, which is the minimum number of states needed to represent  $H(s)$ . You can see this number of states by factoring  $H(s)$  as the product of a first-order system with a second-order system.

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

## Example 4

### Descriptor State-Space Model

Create a descriptor state-space model.

```
a = [2 -4; 4 2];
b = [-1; 0.5];
c = [-0.5, -2];
d = [-1];
e = [1 0; -3 0.5];
% Create a descriptor state-space model.
sys1 = dss(a,b,c,d,e);

% Compute an explicit realization.
sys2 = ss(sys1,'explicit')
```

These commands produce the result:

```
a =  
      x1  x2  
x1    2  -4  
x2   20 -20
```

```
b =  
      u1  
x1   -1  
x2   -5
```

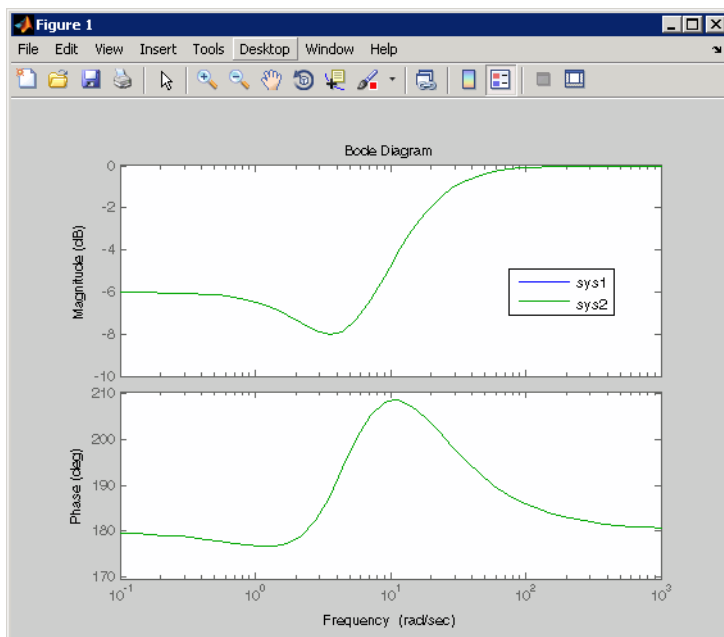
```
c =  
      x1  x2  
y1 -0.5  -2
```

```
d =  
      u1  
y1   -1
```

Continuous-time model.

The result is an explicit state-space model ( $E = I$ ). A Bode plot shows that `sys1` and `sys2` are equivalent.

```
bode(sys1,sys2)
```



## Example 5

### Generalized State-Space Model

This example shows how to create a state-space (genss) model having both fixed and tunable parameters.

Create a state-space model having the following state-space matrices:

$$A = \begin{bmatrix} 1 & a+b \\ 0 & ab \end{bmatrix}, \quad B = \begin{bmatrix} -3.0 \\ 1.5 \end{bmatrix}, \quad C = [0.3 \ 0], \quad D = 0,$$

where  $a$  and  $b$  are tunable parameters, whose initial values are  $-1$  and  $3$ , respectively.

**1** Create the tunable parameters using `realp`.

```
a = realp('a', -1);
```

```
b = realp('b',3);
```

- 2** Define a generalized matrix using algebraic expressions of a and b.

```
A = [1 a+b;0 a*b]
```

A is a generalized matrix whose `Blocks` property contains a and b. The initial value of A is  $M = \begin{bmatrix} 1 & 2 \\ 0 & -3 \end{bmatrix}$ , from the initial values of a and b.

- 3** Create the fixed-value state-space matrices.

```
B = [-3.0;1.5];  
C = [0.3 0];  
D = 0;
```

- 4** Use `ss` to create the state-space model.

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

`sys` is a generalized LTI model (`genss`) with tunable parameters a and b.

### **Example 6**

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 can simply use `ss(sys)` to extract the measured component.

**See Also**

dss | frd | get | set | ssdata | tf | zpk

**Tutorials**

- “State-Space Model”
- “MIMO State-Space Model”

**How To**

- “What Are Model Objects?”
- “State-Space Models”

**Purpose** State coordinate transformation for state-space model

**Syntax** `sysT = ss2ss(sys,T)`

**Description** Given a state-space model `sys` with equations

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

or the innovations form used by the identified state-space (IDSS) models:

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

$$y = Cx + Du + e$$

(or their discrete-time counterpart), `ss2ss` performs the similarity transformation  $\bar{x} = Tx$  on the state vector  $x$  and produces the equivalent state-space model `sysT` with equations.

$$\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu$$

$$y = CT^{-1}\bar{x} + Du$$

or, in the case of an IDSS model:

$$\dot{\bar{x}} = TAT^{-1}\bar{x} + TBu + TKe$$

$$y = CT^{-1}\bar{x} + Du + e$$

(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

# ssdata

---

**Purpose** 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` | `get` | `getdelaymodel` | `set` | `ss` | `tfdata` | `zpkdata`



**Purpose**

Stable-unstable decomposition of LTI model

**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  $ATOL+RTOL*abs(G)$ . 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.

# stabsep

Mode	Continuous Time Region	Discrete Time Region
1	$\text{Re}(s) < -\text{ALPHA} \cdot \max(1,  \text{Im}(s) )$	1 $ z  < 1 - \text{ALPHA}$
2	$\text{Re}(s) > \text{ALPHA} \cdot \max(1,  \text{Im}(s) )$	2 $ z  > 1 + \text{ALPHA}$

The default values are MODE=1 and ALPHA=0.

`[G1,G2] = stabsep(G, 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 -.001 (which is nominally stable).

hns

```
Zero/pole/gain:
0.050075 (s-1)
-----
      (s+0.001) (s-1)
```

## See Also

`stabsepOptions` | `modsep`

**Purpose** Create option set 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  $AbsTol + RelTol * abs(G)$ . 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'

# stabsepOptions

---

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

**Purpose** 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);
```

**Purpose** Step response plot of dynamic system

**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, 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  $T_{\text{final}}$  in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling time ( $T_s = -1$ ), `step` interprets  $T_{\text{final}}$  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 1-665). 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 `idnlmodle1`) `[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`.

## 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”.

## Examples

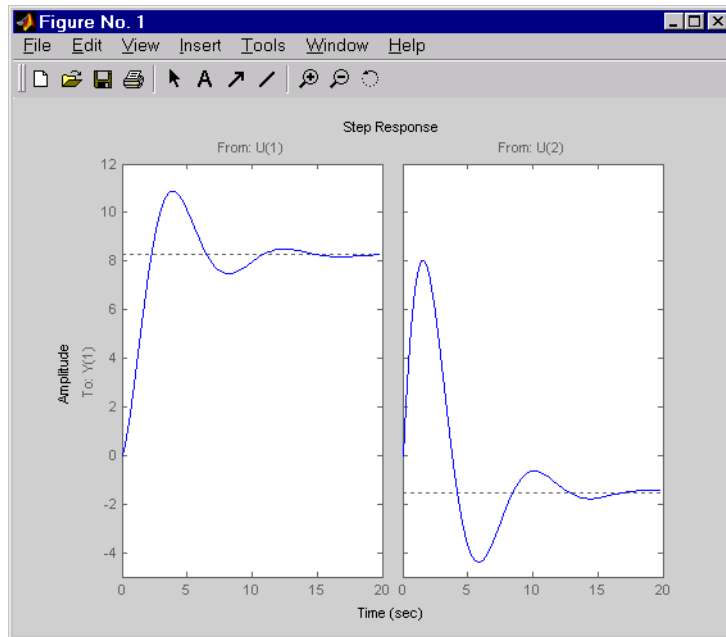
### Example 1

#### 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)
```

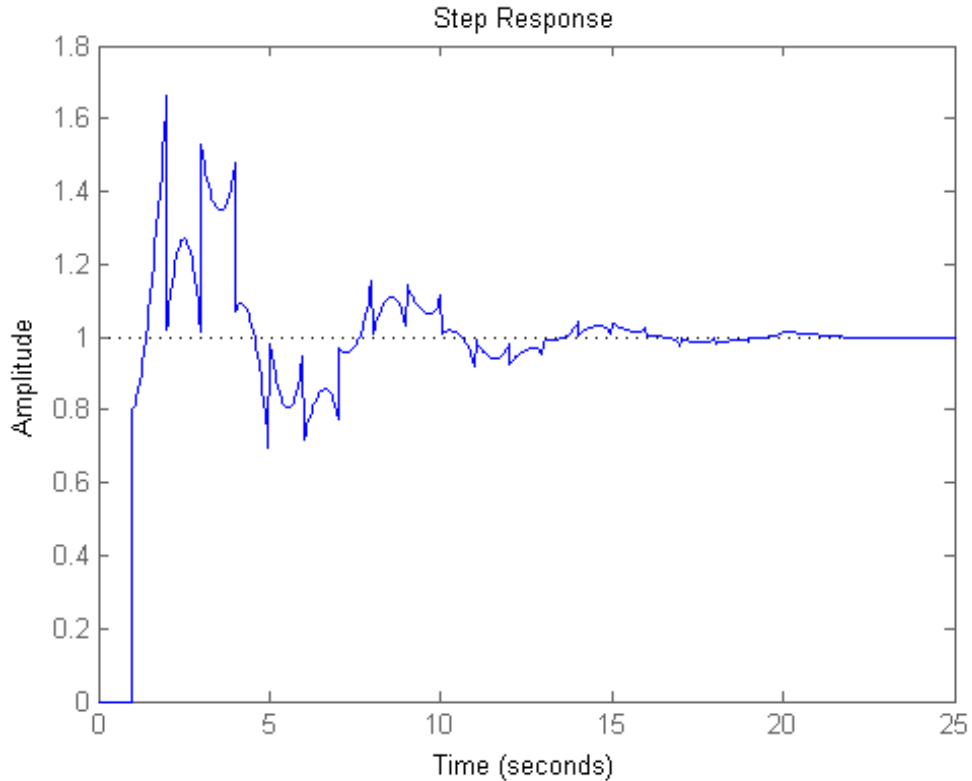


The left plot shows the step response of the first input channel, and the right plot shows the step response of the second input channel.

### 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.

### Example 3

Compare the step response of a parametric identified model to a non-parametric (empirical) model/ Also view their  $3\text{-}\sigma$  confidence regions.

```

load iddata1 z1
sys1 = ssest(z1,4);

parametric model

sys2 = impulseest(z1);

non-parametric model

[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:')

```

#### Example 4

Validation the linearization of a nonlinear ARX model by comparing their small amplitude step responses.

```

nlsys = nlarx(z2,[4 3 10],'tree','custom',...
    {'sin(y1(t-2)*u1(t))+y1(t-2)*u1(t)+u1(t).*u1(t-13)',...
    'y1(t-5)*y1(t-5)*y1(t-1)'},'n1r',[1:5, 7 9]);

```

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)

```

Now validate the usefulness of `sys` by comparing its small-amplitude step response to that of `nlsys`. The nonlinear system `nlsys` is operating an equilibrium level dictated by  $(u_0, y_0)$ . About this steady-state, we

introduce a step perturbation of size 0.1. The corresponding response is computed as follows:

```
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 nonlinear system's equilibrium values. The step response of the linear system is:

```
opt = stepDataOptions;
opt.StepAmplitude = 0.1;
yl = step(sys, t, opt);
```

To compare, add the steady-state offset, `y0`, to the response of the linear system:

```
plot(t, ynl, t, yl+y0)
legend('Nonlinear', 'Linear with offset')
```

## Example 5

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 iddata9
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".

---

`step(sys)`

## Algorithms

Continuous-time models without internal delays are converted to state space and discretized using zero-order hold on the inputs. The sampling period, `dt`, 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 `t` are equisampled with spacing `dt`.

For systems with internal delays, Control System Toolbox software uses variable step solvers. As a result, the time steps `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` | `stepDataOptions` | `initial` | `lsim` | `ltiview`

# stepDataOptions

---

**Purpose** 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.

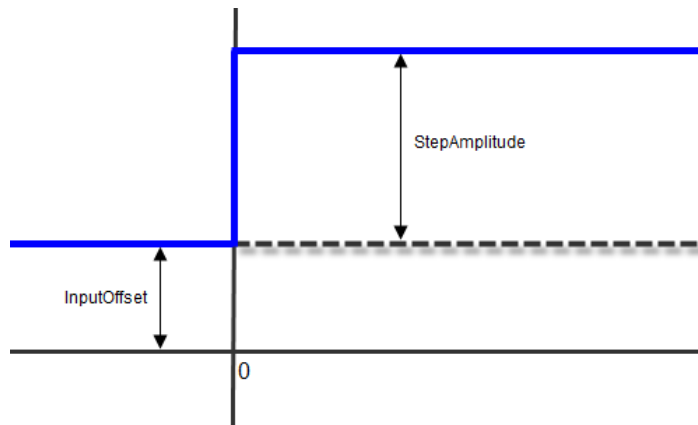
## Input Arguments

### 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**

Specify the input offset and amplitude level for step response.

```
sys = tf(1,[1,1]);  
opt = stepDataOptions('InputOffset',-1,'StepAmplitude',2);  
[y,t] = step(sys,opt)
```

**See Also** [step](#)

# stepinfo

---

**Purpose** 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

step | lsiminfo

# stepplot

---

**Purpose** 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)
```

**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  $t = T_{\text{final}}$ . Express `Tfinal` in the system time units, specified in the `TimeUnit` property of `sys`. For discrete-time systems with unspecified sampling time ( $T_s = -1$ ), `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 `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 `step`). The `stepplot` command always applies the step input at  $t=0$ , regardless of `Ti`.

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)` plots the step response with the options specified in `plotoptions`. Type

```
help timeoptions
```

for more details.

## 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”.

## Examples

### Example 1

Use the plot handle to normalize the responses on a step plot.

```
sys = rss(3);
h = stepplot(sys);
% Normalize responses.
setoptions(h,'Normalize','on');
```

### Example 2

Compare the step response of a parametric identified model to a non-parametric (empirical) model, and view their  $3\text{-}\sigma$  confidence

regions. (Identified models require System Identification Toolbox software.)

```
load iddata1 z1

for parametric model

sys1 = ssest(z1,4);

non-parametric model

sys2 = impulseest(z1);
t = -1:0.1:5;
h = stepplot(sys1,sys2,t);
showConfidence(h, true, 3)
```

The non-parametric model `sys2` shows higher uncertainty.

### Example 3

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);

plotoptions = stepDataOptions('InputOffset', 2, 'StepAmplitude', 0.5);
stepplot(sys,60,plotoptions);
```

### See Also

[getoptions](#) | [setoptions](#) | [step](#)

**Purpose** Create sequence of indexed strings

**Syntax** `strvec = strseq(STR,INDICES)`

**Description** `strvec = strseq(STR,INDICES)` creates a sequence of indexed strings in the string vector `strvec` by appending the integer values `INDICES` to the string `STR`.

---

**Note** You can use `strvec` to aid in system interconnection. For an example, see the `sumblk` reference page.

---

**Examples** Create a string vector by indexing the string 'e' at 1, 2, and 4.

```
strseq('e',[1 2 4])
```

This command returns the following result:

```
ans =
```

```
    'e1'  
    'e2'  
    'e4'
```

**See Also** `strcat` | `connect`

# sumblk

---

**Purpose** 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 the string `formula`. The string `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.

**Tips**

- Use `sumblk` in conjunction with `connect` to interconnect dynamic system models and derive aggregate models for block diagrams.

## Input Arguments

### **formula**

String specifying the equation that relates the input and output signals of the summing junction transfer function `S`. 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 `formula` string. You must provide one `signames` argument for each alias in `formula`.

Specify `signames` as:

- A cell array of name strings.

# sumblk

---

- 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

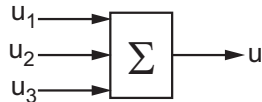
**S**

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 string.

```
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

# sumblk

---

$$e(1) = \text{setpoint}(1) - \alpha + d(1)$$
$$e(2) = \text{setpoint}(2) - q + d(2)$$

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)'
```

## See Also

`connect` | `series` | `parallel` | `strseq`

## How To

- “Multi-Loop Control System”
- “MIMO Control System”

**Purpose** Create transfer function model, convert to transfer function model

**Syntax**

```
sys = tf(num,den)
sys = tf(num,den,Ts)
sys = tf(M)
sys = tf(num,den,ltsys)
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(num,den)` creates a continuous-time transfer function with numerator(s) and denominator(s) specified by `num` and `den`. The output `sys` is:

- A `tf` model object, when `num` and `den` are numeric arrays.
- A generalized state-space model (`genss`) when `num` or `den` include tunable parameters, such as `realp` parameters or generalized matrices (`genmat`).
- An uncertain state-space model (`uss`) when `num` or `den` are uncertain (requires Robust Control Toolbox software).

In the SISO case, `num` and `den` are the real- or complex-valued row vectors of numerator and denominator coefficients ordered in *descending* powers of  $s$ . These two vectors need not have equal length and the transfer function need not be proper. For example, `h = tf([1 0],1)` specifies the pure derivative  $h(s) = s$ .

To create MIMO transfer functions, using one of the following approaches:

- Concatenate SISO `tf` models.

- Use the `tf` command with cell array arguments. In this case, `num` and `den` are cell arrays of row vectors with as many rows as outputs and as many columns as inputs. The row vectors `num{i, j}` and `den{i, j}` specify the numerator and denominator of the transfer function from input `j` to output `i`.

For examples of creating MIMO transfer functions, see “Examples” on page 1-682 and “MIMO Transfer Function Model” in the *Control System Toolbox User Guide*.

If all SISO entries of a MIMO transfer function have the same denominator, you can set `den` to the row vector representation of this common denominator. See “Examples” for more details.

`sys = tf(num,den,Ts)` creates a discrete-time transfer function with sample time `Ts` (in seconds). Set `Ts = -1` to leave the sample time unspecified. The input arguments `num` and `den` are as in the continuous-time case and must list the numerator and denominator coefficients in *descending* powers of  $z$ .

`sys = tf(M)` creates a static gain `M` (scalar or matrix).

`sys = tf(num,den,ltisys)` creates a transfer function with properties inherited from the dynamic system model `ltisys` (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” in the *Control System Toolbox User Guide* for more information.

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 1-688. Note that

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

is a shortcut for

```
sys = tf(num,den)
set(sys,'Property1',Value1,...,'PropertyN',ValueN)
```

### Transfer Functions as Rational Expressions in *s* or *z*

You can also use real- or complex-valued rational expressions to create a TF model. To do so, first type either:

- `s = tf('s')` to specify a TF model using a rational function in the Laplace variable, *s*.
- `z = tf('z',Ts)` to specify a TF model with sample time *Ts* using a rational function in the discrete-time variable, *z*.

Once you specify either of these variables, you can specify TF models directly as rational expressions in the variable *s* or *z* by entering your transfer function as a rational expression in either *s* or *z*.

### Conversion to Transfer Function

`tfsys = tf(sys)` converts 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`, `ltiblock.tf`, or `ltiblock.ss` 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  $\Lambda = LL'$  represents the covariance of noise  $e(t)$ , this equation can also be written as:  $y(t) = Gu(t) + HLv(t)$ , where  $\text{cov}(v(t)) = I$ .

`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(num,den)
```

to create a Generalized state-space (`genss`) model when one or more of the entries `num` and `den` depends on a tunable `realp` or `genmat` model. For more information about Generalized state-space models, see “Models with Tunable Coefficients”.

## Examples

### Example 1

#### Transfer Function Model with One-Input Two-Outputs



Create the one-input, two-output transfer function

$$H(p) = \begin{bmatrix} \frac{p+1}{p^2+2p+2} \\ \frac{1}{p} \end{bmatrix}$$

with input current and outputs torque and ang velocity.

To do this, enter

```
num = {[1 1] ; 1};
den = {[1 2 2] ; [1 0]};
H = tf(num,den,'inputn','current',...
        'outputn',{'torque' 'ang. velocity'},...
        'variable','p')
```

These commands produce the result:

Transfer function from input "current" to output...

```
      p + 1
torque:  -----
      p^2 + 2 p + 2

      1
ang. velocity:  -
                p
```

Setting the 'variable' property to 'p' causes the result to be displayed as a transfer function of the variable  $p$ .

## Example 2

### Transfer Function Model Using Rational Expression

To use a rational expression to create a SISO TF model, type

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

This produces the same transfer function as

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

### Example 3

#### Multiple-Input Multiple-Output Transfer Function Model

Specify the discrete MIMO transfer function

$$H(z) = \begin{bmatrix} \frac{1}{z+0.3} & \frac{z}{z+0.3} \\ \frac{-z+2}{z+0.3} & \frac{3}{z+0.3} \end{bmatrix}$$

with common denominator  $d(z) = z + 0.3$  and sample time of 0.2 seconds.

```
nums = {1 [1 0];[-1 2] 3};
Ts = 0.2;
H = tf(nums,[1 0.3],Ts)    % Note: row vector for common den. d(z)
```

### Example 4

#### Convert State-Space Model to Transfer Function

Compute the transfer function of the state-space model with the following data.

$$A = \begin{bmatrix} -2 & -1 \\ 1 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix}, \quad C = [1 \ 0], \quad D = [0 \ 1].$$

To do this, type

```
sys = ss([-2 -1;1 -2],[1 1;2 -1],[1 0],[0 1]);
tf(sys)
```

These commands produce the result:

Transfer function from input 1 to output:  
 $s - 4.441e-016$   
 -----  
 $s^2 + 4 s + 5$

Transfer function from input 2 to output:  
 $s^2 + 5 s + 8$   
 -----  
 $s^2 + 4 s + 5$

### Example 5

#### Array of Transfer Function Models

You can use a for loop to specify a 10-by-1 array of SISO TF models.

```
H = tf(zeros(1,1,10));
s = tf('s')
for k=1:10,
    H(:, :, k) = k/(s^2+s+k);
end
```

The first statement pre-allocates the TF array and fills it with zero transfer functions.

### Example 6

#### Tunable Low-Pass Filter

This example shows how to create the low-pass filter  $F = a/(s + a)$  with one tunable parameter  $a$ .

You cannot use `ltiblock.tf` to represent  $F$ , because the numerator and denominator coefficients of an `ltiblock.tf` block are independent. Instead, construct  $F$  using the tunable real parameter object `realp`.

- 1 Create a tunable real parameter.

```
a = realp('a',10);
```

The `realp` object `a` is a tunable parameter with initial value 10.

**2** Use `tf` to create the tunable filter `F`:

```
F = tf(a,[1 a]);
```

`F` is a `genss` object which has the tunable parameter `a` in its `Blocks` property. You can connect `F` with other tunable or numeric models to create more complex models of control systems. For an example, see “Control System with Tunable Components”.

### Example 7

Extract the measured and noise components of an identified polynomial model into two separate transfer functions. 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 = 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 $[a_k \ \dots \ a_1 \ a_0]$ to specify the polynomial $a_k z^k + \dots + a_1 z + a_0$ (coefficients ordered in <i>descending</i> powers of $z$ or $q$ ).
'z^-1'	Use the row vector $[b_0 \ b_1 \ \dots \ b_k]$ to specify the polynomial $b_0 + b_1 z^{-1} + \dots + b_k z^{-k}$ (coefficients in <i>ascending</i> powers of $z^{-1}$ ).

For example,

```
g = tf([1 1],[1 2 3],0.1);
```

specifies the discrete transfer function

$$g(z) = \frac{z+1}{z^2+2z+3}$$

because  $z$  is the default variable. In contrast,

```
h = tf([1 1],[1 2 3],0.1,'variable','z^-1');
```

uses the DSP convention and creates

$$h(z^{-1}) = \frac{1 + z^{-1}}{1 + 2z^{-1} + 3z^{-2}} = zg(z).$$

See also `filt` for direct specification of discrete transfer functions using the DSP convention.

Note that `tf` stores data so that the numerator and denominator lengths are made equal. Specifically, `tf` stores the values

```
num = [0 1 1]; den = [1 2 3];
```

for `g` (the numerator is padded with zeros on the left) and the values

```
num = [1 1 0]; den = [1 2 3];
```

for `h` (the numerator is padded with zeros on the right).

## Properties

`tf` objects have the following properties:

### **num**

Transfer function numerator coefficients.

For SISO transfer functions, `num` 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, `num` is a `Ny`-by-`Nu` cell array of the numerator coefficients for each input/output pair.

### **den**

Transfer function denominator coefficients.

For SISO transfer functions, `den` 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  $N_y$  outputs and  $N_u$  inputs, `den` is a  $N_y$ -by- $N_u$  cell array of the denominator coefficients for each input/output pair.

### Variable

String specifying the transfer function display variable. `Variable` can take the following values:

- '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 `num` and `den` 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 sampling period,  $T_s$ .

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 delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. 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 sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

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 for all input channels

### **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 sampling period `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**



Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'
- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

**OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string '' for all input 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. Set `Name` to a string to label the system.

**Default:** ''

**Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

### **UserData**

Any type of data you wish to associate with system. Set UserData to 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
```

...

**Default:** []

## Algorithms

`tf` uses the MATLAB function `poly` to convert zero-pole-gain models, and the functions `zero` and `pole` to convert state-space models.

## See Also

`filt` | `frd` | `get` | `set` | `ss` | `tfdata` | `zpk` | `genss` | `realp` | `genmat` | `ltiblock.tf`

## Tutorials

- “Transfer Function Model Using Numerator and Denominator Coefficients”
- “Discrete-Time Transfer Function Model”
- “MIMO Transfer Function Model”

## How To

- “What Are Model Objects?”
- “Transfer Functions”

**Purpose**

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  $\text{num}\{i,j\}(k)$  and  $\text{sdden}\{i,j\}(k)$  is the 1 standard uncertainty in the value  $\text{den}\{i,j\}(k)$ . If `sys` does not contain uncertainty information, `snum` 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

[get](#) | [ssdata](#) | [tf](#) | [zpkdata](#)

**Purpose** 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 sampling time.

- Tips**
- If `tau` is an integer multiple of `Ts`, then `sys` represents the pure discrete delay  $z^{-N}$ , with  $N = \text{tau}/\text{Ts}$ . 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`.

**Input Arguments**

**tau**  
Time delay to discretize.

**Ts**  
Sampling 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
```

Sampling 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
```

Sampling time: 1

## 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. Valimaki, "Splitting the Unit Delay", *IEEE Signal Processing Magazine*, Vol. 13, No. 1, p.30-60, 1996.

## See Also

c2d | pade | tf

# timeoptions

---

**Purpose** 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 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 strings: 'off'   'on' <b>Default:</b> 'off'
XlimMode, YlimMode	Limit modes
Xlim, Ylim	Axes limits
IOGrouping	Grouping of input-output pairs Specified as one of the following strings: 'none'   'inputs'   'output'   'all' <b>Default:</b> 'none'
InputLabel, OutputLabel	Input and output label styles
InputVisible, OutputVisible	Visibility of input and output channels

Option	Description
Normalize	Normalize responses Specified as one of the following strings: 'on'   'off' <b>Default:</b> 'off'
SettleTimeThreshold	Settling time threshold
RiseTimeLimits	Rise time limits
TimeUnits	Time units, specified as one of the following strings: <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'  You can also specify 'auto' which uses time units specified in the TimeUnit property of the input system. For multiple systems with different time units, the units of the first system is used.

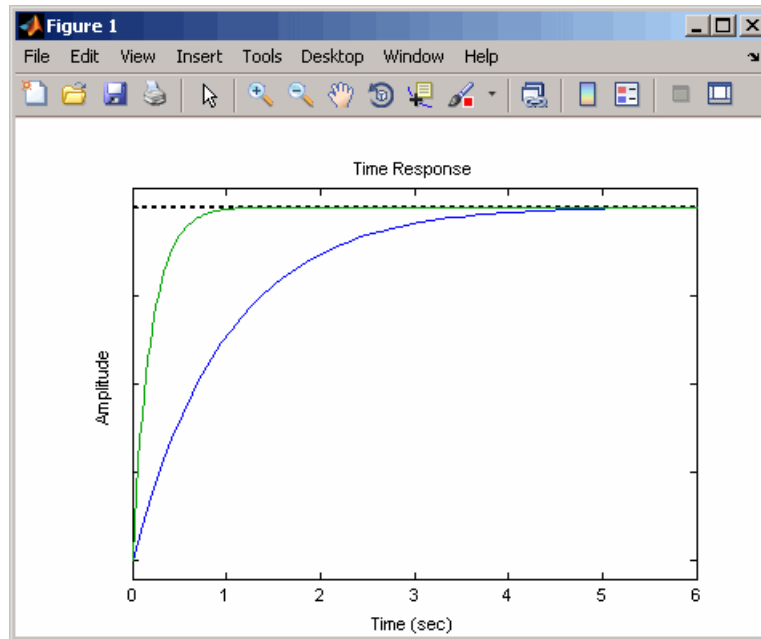
# timeoptions

## 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

[getoptions](#) | [impzplot](#) | [initialplot](#) | [lsimplot](#) | [setoptions](#)  
| [stepplot](#)

**Purpose** 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** `absorbDelay` | `hasdelay`

# tzero

---

**Purpose** 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  $H(s) = D + C(sE - A)^{-1}B$ .

**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 “Unobservable and Uncontrollable Modes of MIMO Model” on page 1-709 for an example.

**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  $sE - A$  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.

## Definitions

### Invariant zeros

For a MIMO state-space model

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

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

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

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.

## Examples

### Transmission Zeros of MIMO Transfer Function

Find the invariant zeros of a MIMO transfer function and confirm that they coincide with the transmission zeros.

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.

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.

### **Unobservable and Uncontrollable Modes of MIMO Model**

Identify the unobservable and uncontrollable modes of a MIMO model using the state-space matrix syntax of tzero.

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.

Scale the state-space matrices of `gasf`.

```
[A,B,C,D] = ssdata(prescale(gasf));
```

To identify the unobservable and uncontrollable modes of `gasf`, you need access to the state-space matrices `A`, `B`, `C`, and `D` of the model. `tzero` does not scale state-space matrices when you use the syntax. Therefore, use `prescale` with `ssdata` to extract scaled values of these matrices.

Use `tzero` to 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 =
```

```
Empty matrix: 0-by-1
```

---

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.

**Algorithms**

tzero is based on SLICOT routines AB08ND, AG08BD, and AB8NXZ. 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.

**Alternatives**

To calculate the zeros and gain of a single-input, single-output (SISO) system, use `zero`.

**See Also**

`pole` | `pzmap` | `zero`

# updateSystem

---

**Purpose** 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 Plot Updates”.

`updateSystem(h,sys,N)` replaces the data used to compute the Nth response in the plot.

## 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**

positive integer | 1 (default)

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);
```

## **Examples**

### **Update System Data in Plot**

Replace plotted step response data with data computed from a different dynamic system model.

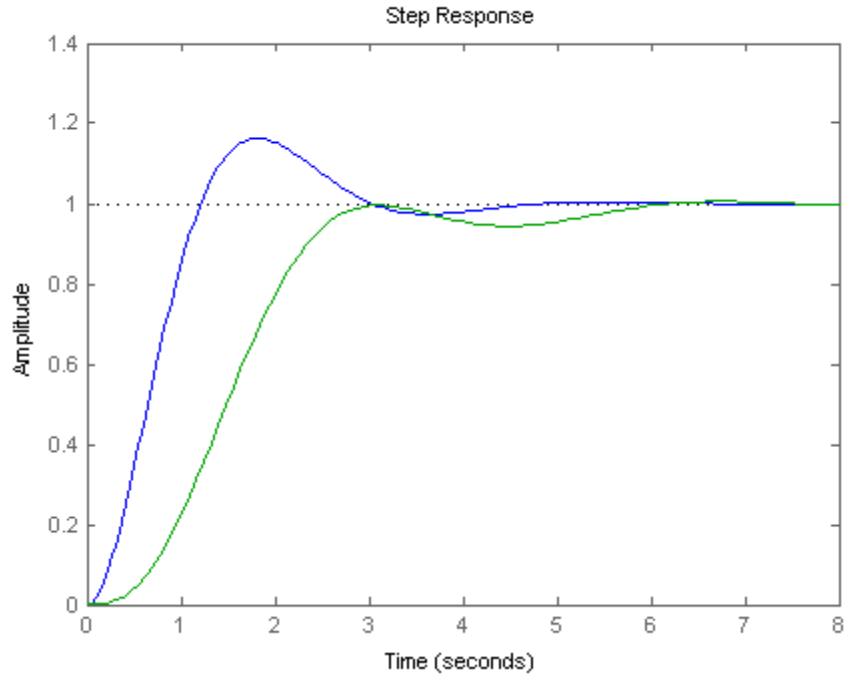
Suppose you have a plant model and pure integrator controller that you designed for that plant. Plot the step responses of the plant and the closed-loop system.

```
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);
```

# updateSystem



`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`.

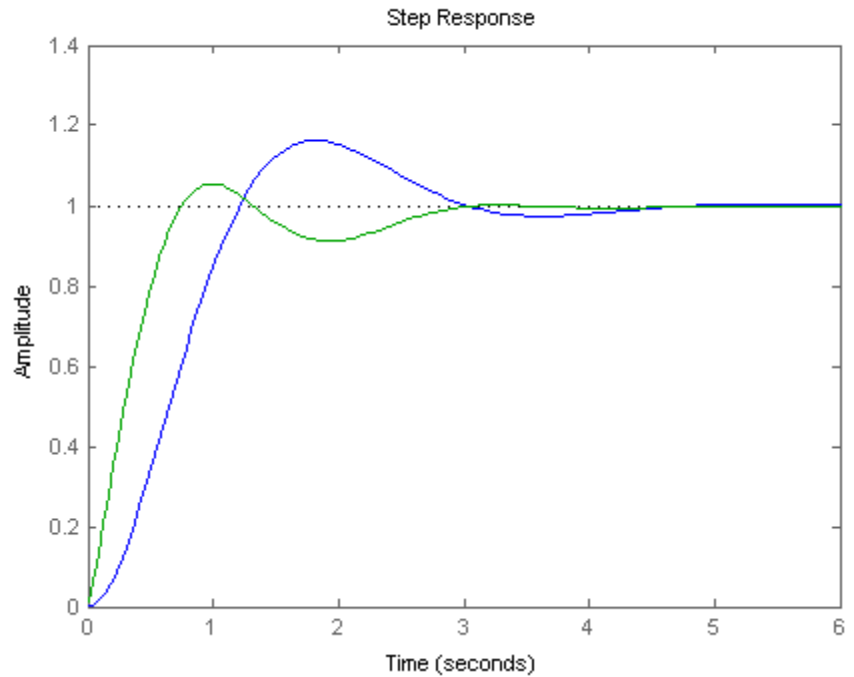
Suppose you also have a PID controller design that you want to analyze. Create a model of the closed-loop system using this alternate controller.

```
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`.

## Related Examples

- “Build GUI With Interactive Plot Updates”

# upsample

---

**Purpose** 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 sampling 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 sampling 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 sampling 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

-----  
 $z^2 + 10z + 2$

Sampling 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<sup>28</sup> + 10 z<sup>14</sup> + 2

Sampling time: 0.16071

The sampling time of sys1 is 0.16071 seconds, which is 14 times faster than the 2.25 second sampling time of sys.

## See Also

d2d | d2c | c2d

# view (genmat)

---

**Purpose** Visualize gain surface as a function of scheduling variables

**Syntax**

```
view(M)
view(M,xvar)
view(M,xvar,yvar)
view(M,xvar,xdata)
view(M,xvar,xdata,yvar,ydata)
```

**Description** `view(M)` plots the values of a 1-D or 2-D array of generalized matrices on a 1-D or 2-D plot. Typically, `M` is a tunable gain surface that you create with `gainsurf`. The plot uses the independent variable values specified in `M.SamplingGrid` if available. Otherwise, The plot uses the indices along each array dimension for the X and Y values.

`view(M,xvar)` plots a 1-D plot of values in the generalized matrix array against a specified independent variable, `xvar`. For a 2-D array, the plot contains multiple traces corresponding to the other dimension in the array. `xvar` must refer to a sampling variable listed in `M.SamplingGrid`.

`view(M,xvar,yvar)` plots the values in the generalized matrix array against the specified independent variables, placing `xvar` on the X axis and `yvar` on the Y axis. `xvar` and `yvar` must refer to sampling variables listed in `M.SamplingGrid`. Use this syntax:

- To specify the order of independent variables plotted along the X and Y axes.
- To select the independent variable values when `M.SamplingGrid` lists more than two independent variables.

`view(M,xvar,xdata)` plots a 1-D plot of values in the generalized matrix array, using `xvar` to name the X axis. This plot also uses the values in `xdata` as the values along the X axis. Use this syntax when `M.SamplingGrid` is empty.

`view(M, xvar, xdata, yvar, ydata)` plots a 2-D plot of values in the generalized matrix array, using `xvar` and `yvar` to name the X and Y axes, respectively. This plot also uses the values in `xdata` and `ydata` as values along the X and Y axes, respectively. Use this syntax when `M.SamplingGrid` is empty.

## Input Arguments

### **M - Array of generalized matrices**

`genmat` array

Array of generalized matrices to plot, specified as a `genmat` array. Typically, `M` represents a variable gain surface that you create using the `gainsurf` command. Optionally, you can set the `SamplingGrid` property of `M` to list the independent variable names and values corresponding to entries in the array. See the `genmat` reference page for more information.

### **xvar - X-axis variable**

string

X-axis variable in the plot, specified as a string.

If the `SamplingGrid` property of `M` specifies independent variable names and values, `xvar` must match one of those variable names. In this case, `view` uses the values stored in `M.SamplingGrid` for the X axis of the plot.

If the `SamplingGrid` property of `M` is empty, `view` uses `xvar` to label the X axis of the plot. In this case, you must also specify values for the X axis using the `xvar` input argument.

### **yvar - Y-axis variable**

string

Y-axis variable in the plot, specified as a string.

If the `SamplingGrid` property of `M` specifies independent variable names and values, `yvar` must match one of those variable names. In this case, `view` uses the values stored in `M.SamplingGrid` for the Y axis of the plot.

## view (genmat)

---

If the `SamplingGrid` property of `M` is empty, `view` uses `yvar` to label the Y axis of the plot. In this case, you must also specify values for the Y axis using the `yvar` input argument.

### **xdata - X-axis values**

numeric vector

X-axis values in the plots, specified as a numeric vector. If the `SamplingGrid` property of `M` is empty, use `xdata` to specify values for `view` to display along the X axis of the plot. The length of `xdata` must match the first array dimension of `M`.

### **ydata - Y-axis values**

numeric vector

Y-axis values in the plots, specified as a numeric vector. If the `SamplingGrid` property of `M` is empty, use `ydata` to specify values for `view` to display along the Y axis of the plot. The length of `ydata` must match the first array dimension of `M`.

## Examples

### **View Gain Surface**

Display a tunable gain surface that depends on two independent variables.

Create a scalar gain,  $K$ , that is a bilinear function of two independent variables,  $\alpha$  and  $\beta$ :

$$K(\alpha, \beta) = K_0 + K_1\alpha + K_2\beta + K_3\alpha\beta.$$

```
[alpha,beta] = ndgrid(0:1:15,300:50:600);  
F1 = alpha;  
F2 = beta;  
F3 = alpha.*beta;  
K = gainsurf('K',1,F1,F2,F3);  
K.SamplingGrid = struct('alpha',alpha,'beta',beta);
```

`gainsurf` initializes all gain surface coefficients to zero. For this example, manually set the coefficients to nonzero values.

```
K.Blocks.K_1.Value = -0.015;  
K.Blocks.K_2.Value = 0.02;  
K.Blocks.K_3.Value = 0.01;
```

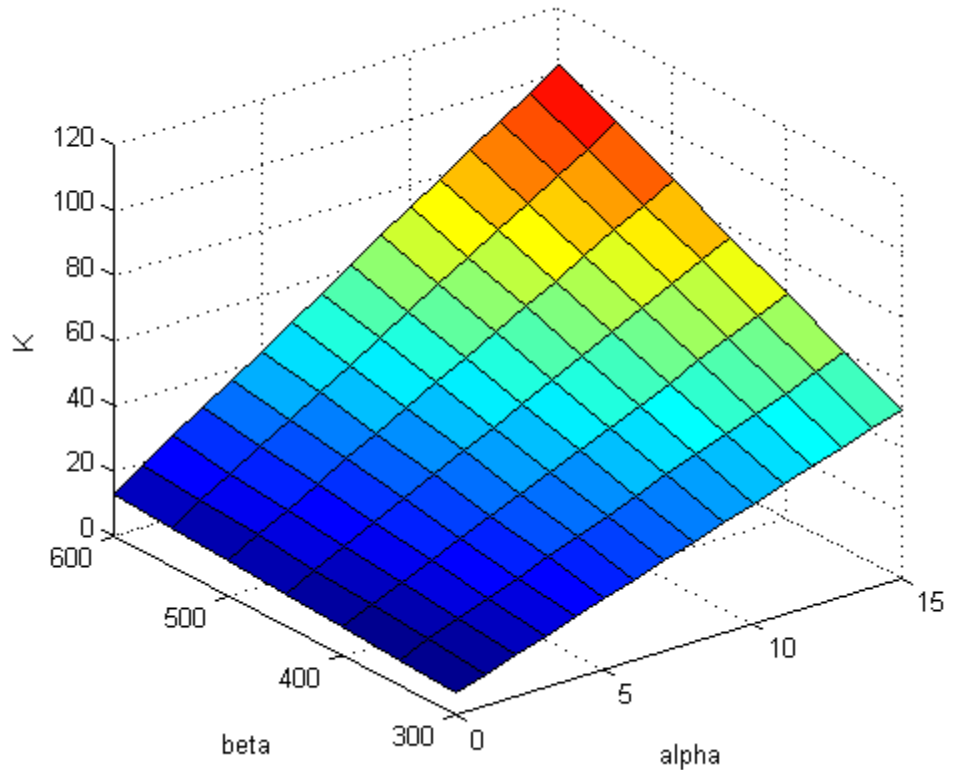
Typically, you would tune the coefficients as part of a control system. You would then use `setBlockValue` to write the tuned coefficients back to `K`, and view the tuned gain surface.

Plot the gain surface.

```
view(K);
```

## view (genmat)

---

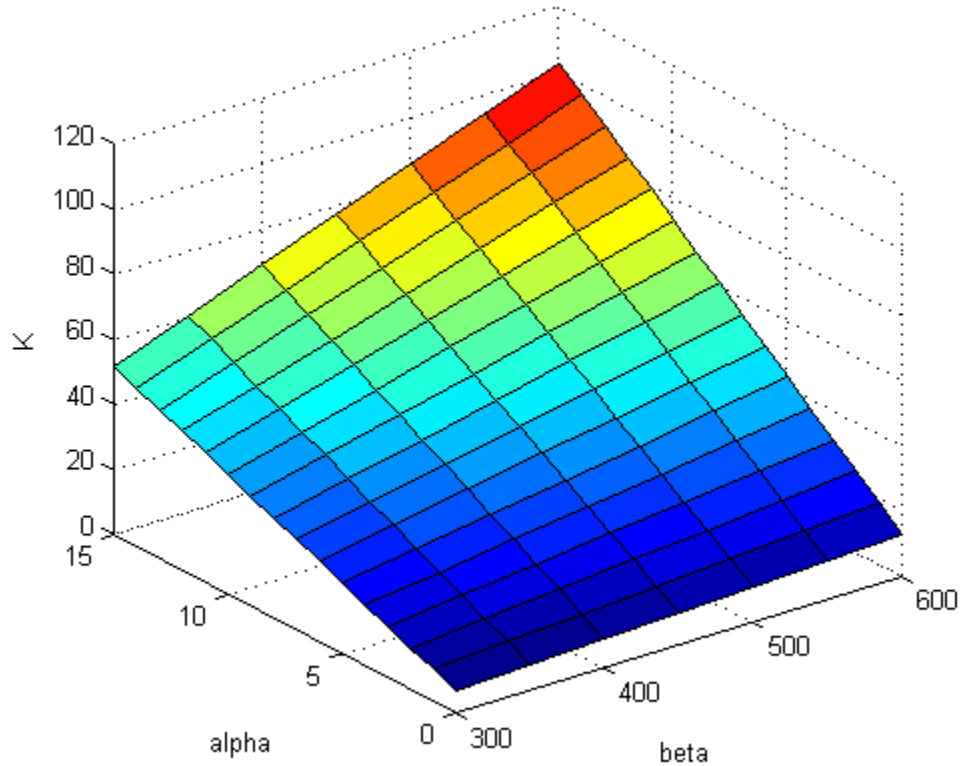


view automatically applies the axis labels and scaling stored in the SamplingGrid property of the gain surface (see the genmat reference page). If the SamplingGrid property is empty, the independent variable axes are unlabeled and the values are index values.

By default, view puts alpha on the X-axis. This ordering arises because SamplingGrid associates alpha with the first dimension of the gain matrix. Reverse the ordering of the independent variables on the X- and Y-axes.



```
view(K, 'beta', 'alpha')
```



## View 1-Dimensional Projections of 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  $\beta$ .

## view (genmat)

---

```
[alpha,beta] = ndgrid(0:1:15,300:50:600);
F1 = alpha;
F2 = beta;
F3 = alpha.*beta;
K = gainsurf('K',1,F1,F2,F3);
SG = struct('alpha',alpha,'beta',beta);
K.SamplingGrid = SG;
```

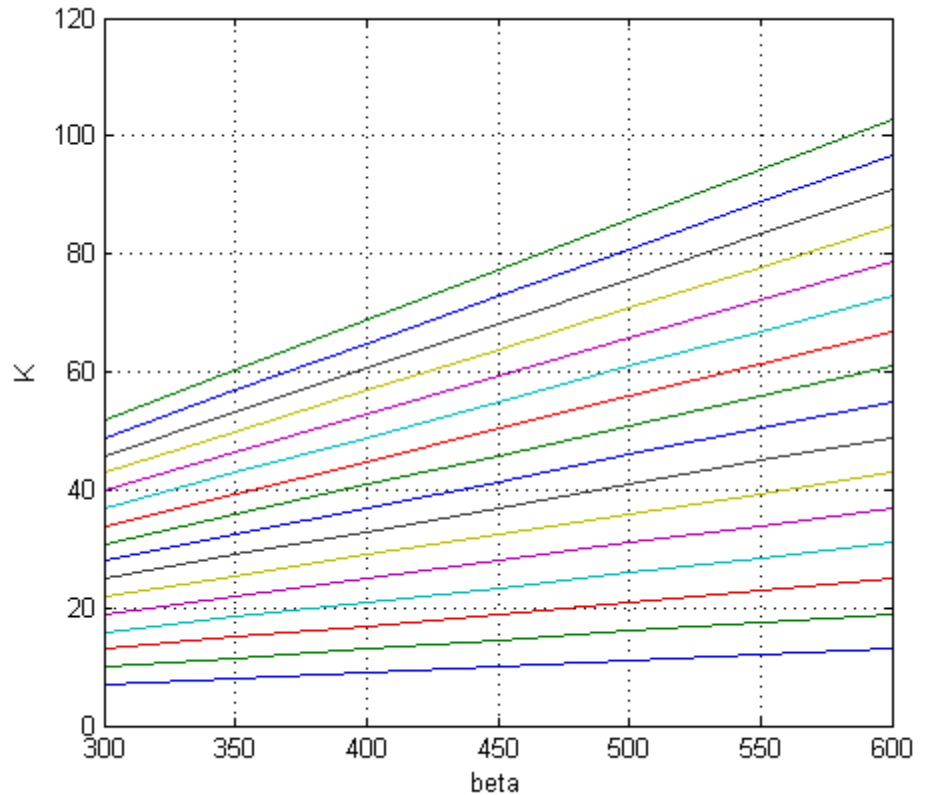
`gainsurf` initializes all gain surface coefficients to zero. For this example, manually set the coefficients to nonzero values.

```
K.Blocks.K_1.Value = -0.015;
K.Blocks.K_2.Value = 0.02;
K.Blocks.K_3.Value = 0.01;
```

Typically, you would tune the coefficients as part of a control system. You would then use `setBlockValue` to write the tuned coefficients back to `K`, and view the tuned gain surface.

Plot the gain as a function of  $\beta$  for all values of  $\alpha$  in the grid of the gain surface.

```
view(K,'beta')
```



view scales the X-axis using the beta values stored in the `SamplingGrid` property of the gain surface. This plot is useful to visualize the full range of gain variation due to one independent variable.

### **View Gain Surface With Specified Independent Variable Names and Values**

Plot a gain surface for which you provide variable names and values.

When plotting a gain surface for which you have not specified a `SamplingGrid` property value, view cannot label the independent

## view (genmat)

---

variable axes. In addition, the independent variable values are just the index values of the gain matrix. (For information about `SamplingGrid`, see the `genmat` reference page). In this case, you can specify variable names and values for the purpose of the plot.

Create a gain surface that is a bilinear function of two independent variables,  $\alpha$  and  $\beta$ .

```
[alpha,beta] = ndgrid(0:1:15,300:50:600);  
F1 = alpha;  
F2 = beta;  
F3 = alpha.*beta;  
K = gainsurf('K',1,F1,F2,F3);
```

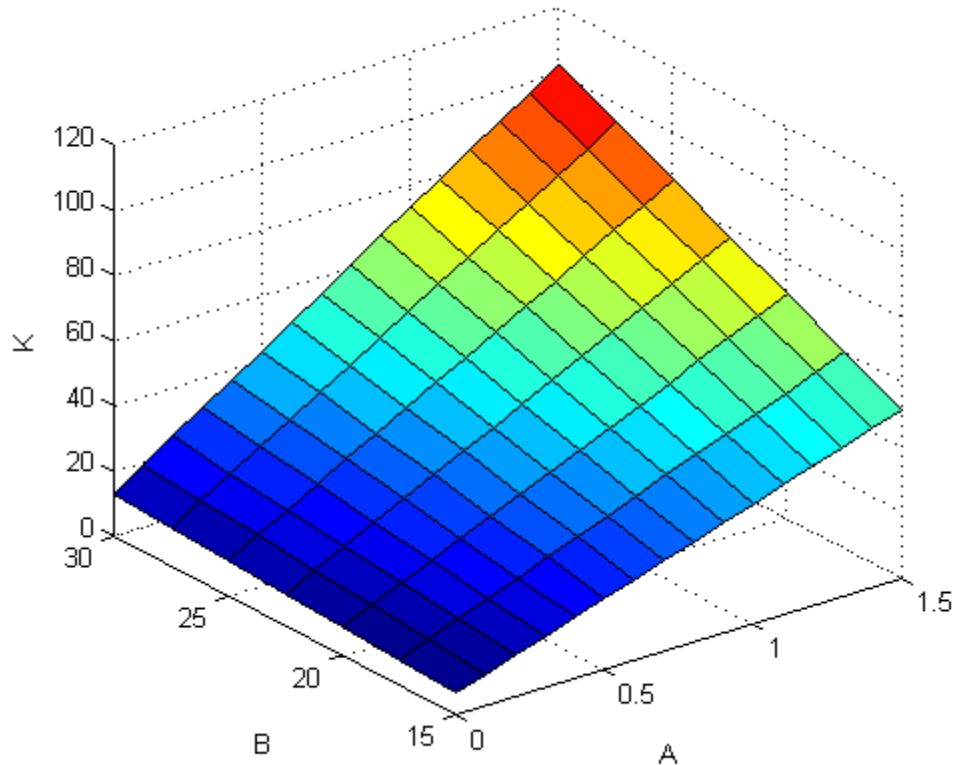
`gainsurf` initializes all gain surface coefficients to zero. For this example, manually set the coefficients to nonzero values.

```
K.Blocks.K_1.Value = -0.015;  
K.Blocks.K_2.Value = 0.02;  
K.Blocks.K_3.Value = 0.01;
```

Typically, you would tune the coefficients as part of a control system. You would then use `setBlockValue` to write the tuned coefficients back to `K`, and view the tuned gain surface.

View the gain surface, specifying variable names and values.

```
view(K, 'A', 0:0.1:1.5, 'B', 15:2.5:30)
```



You can use any variable name and values that you like. The name and value only label the axes and do not affect the gain values themselves, which are stored in `K`. However, the vectors you supply for the variable values must match the sampling dimensions of the gain surface. For example, `K` is created using an 16-element vector for its first dimension. Therefore, the vector you provide of values for that dimension must also have 16 elements.

**See Also** `genmatgainsurfgetValuesetBlockValue`

**Purpose** 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



---

<b>Purpose</b>	Zeros and gain of SISO dynamic system
<b>Syntax</b>	<pre>z = zero(sys) [z,gain] = zero(sys) [z,gain] = zero(sysarr,J1,...,JN)</pre>
<b>Description</b>	<p><code>z = zero(sys)</code> returns the zeros of the single-input, single-output (SISO) dynamic system model, <code>sys</code>.</p> <p><code>[z,gain] = zero(sys)</code> also returns the overall gain of <code>sys</code>.</p> <p><code>[z,gain] = zero(sysarr,J1,...,JN)</code> returns the zeros and gain of the model with subscripts <code>J1, ..., JN</code> in the model array <code>sysarr</code>.</p>
<b>Input Arguments</b>	<p><b>sys</b> SISO dynamic system model.</p> <p>If <code>sys</code> has internal delays, <code>zero</code> sets all internal delays to zero, creating a zero-order Padé approximation. This approximation ensures that the system has a finite number of zeros. <code>zero</code> returns an error if setting internal delays to zero creates singular algebraic loops.</p> <p><b>sysarr</b> Array of dynamic system models.</p> <p><b>J1,...,JN</b> Indices identifying the model <code>sysarr(J1, ..., JN)</code> in the array <code>sysarr</code>.</p>
<b>Output Arguments</b>	<p><b>z</b> Column vector containing the locations of zeros in <code>sys</code>. The zero locations are expressed in the reciprocal of the time units of <code>sys</code>. For example, the zeros are in units of 1/minutes if the <code>TimeUnit</code> property of <code>sys</code> is <code>minutes</code>.</p> <p><b>gain</b></p>

Gain of sys (in the zero-pole-gain sense).

## Examples

### Zero Locations and Gain of Transfer Function

Calculate the zero locations and overall gain of the 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]);  
[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, and zero returns pole locations relative to the new unit.

```
H = chgTimeUnit(H,'minutes');  
[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**            `pole` | `pzmap` | `tzero`

# zgrid

---

**Purpose** 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 for the system

$$H(z) = \frac{2z^2 - 3.4z + 1.5}{z^2 - 1.6z + 0.8}$$

by typing

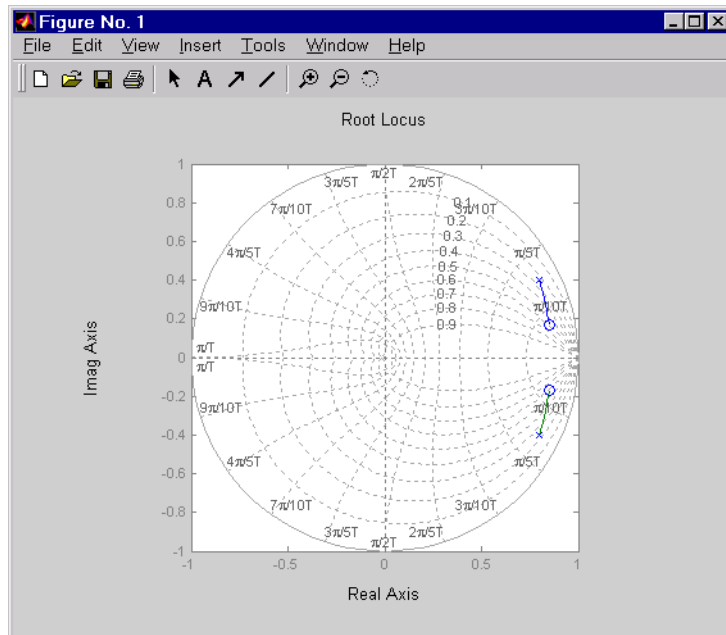
```
H = tf([2 -3.4 1.5],[1 -1.6 0.8],-1)
```

Transfer function:  
 $2z^2 - 3.4z + 1.5$   
 -----  
 $z^2 - 1.6z + 0.8$

Sampling time: unspecified

To see the z-plane grid on the root locus plot, type

```
rlocus(H)
zgrid
axis('square')
```



**See Also** [pzmap](#) | [rlocus](#) | [sgrid](#)

**Purpose** 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  $T_s$  (in seconds). Set  $T_s = -1$  or  $T_s = []$  to leave the sample time unspecified. The input arguments  $z$ ,  $p$ ,  $k$  are as in the continuous-time case.

`sys = zpk(M)` specifies a static gain  $M$ .

`sys = zpk(z,p,k,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 1-739. 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  $T_s$  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  $1:n_u$  while `zsys.InputGroup.Noise` is set to  $n_u+1:n_u+n_y$ . `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  $N_y$  outputs and  $N_u$  inputs, **k** is a  $N_y$ -by- $N_u$  matrix storing the gains for each input/output pair.

**DisplayFormat**

String specifying the way 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<sup>-1</sup>' or 'q<sup>-1</sup>'.

- '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<sup>-1</sup>' or 'q<sup>-1</sup>'.

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 $R = a \pm jb$ )
'roots'	$(s - R)$	$(s^2 - as + \beta)$ , where $a = 2\alpha$ , $\beta = \alpha^2 + b^2$
'frequency'	$(1 - s/\omega_0)$ , where $\omega_0 = R$	$1 - 2\zeta(s/\omega_0) + (s/\omega_0)^2$ , where $\omega_0^2 = \alpha^2 + b^2$ , $\zeta = a/\omega_0$
'time constant'	$(1 - \tau s)$ , 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 sampling time. In discrete time,  $\tau$  and  $\omega_0$  closely match the time constant and natural frequency of the equivalent continuous-time root, provided  $|z-1| T_s (\omega_0 \pi/T_s = \text{Nyquist frequency})$ .

**Default:** 'roots'

### Variable

String specifying the transfer function display variable. Variable can take the following values:

- '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 sampling period, `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 delays. `InputDelay` is a numeric vector specifying a time delay for each input channel. 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 sampling period `Ts`. For example, `InputDelay = 3` means a delay of three sampling periods.

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 for all input channels

### **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 sampling period  $T_s$ . 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

### **Ts**

Sampling 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 sampling 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 sampling time of a discrete-time system.

**Default:** 0 (continuous time)

### **TimeUnit**

String representing the unit of the time variable. For continuous-time models, this property represents any time delays in the model. For discrete-time models, it represents the sampling time  $T_s$ . Use any of the following values:

- 'nanoseconds'
- 'microseconds'

- 'milliseconds'
- 'seconds'
- 'minutes'
- 'hours'
- 'days'
- 'weeks'
- 'months'
- 'years'

Changing this property changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

**Default:** 'seconds'

### **InputName**

Input channel names. Set `InputName` to a string for single-input model. For a multi-input model, set `InputName` to a cell array of strings.

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:** Empty string '' for all input channels

### **InputUnit**

Input channel units. Use `InputUnit` to keep track of input signal units. For a single-input model, set `InputUnit` to a string. For a multi-input model, set `InputUnit` to a cell array of strings. `InputUnit` has no effect on system behavior.

**Default:** Empty string '' 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. Set `OutputName` to a string for single-output model. For a multi-output model, set `OutputName` to a cell array of strings.

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 to 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:** Empty string `''` for all input channels

### **OutputUnit**

Output channel units. Use `OutputUnit` to keep track of output signal units. For a single-output model, set `OutputUnit` to a string. For a multi-output model, set `OutputUnit` to a cell array of strings. `OutputUnit` has no effect on system behavior.

**Default:** Empty string `''` for all input 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. Set `Name` to a string to label the system.

**Default:** ''

### **Notes**

Any text that you want to associate with the system. Set `Notes` to a string or a cell array of strings.

**Default:** {}

### **UserData**

Any type of data you wish to associate with system. Set `UserData` to 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
```

...

**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
```

```
Sampling 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')
```

## 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`

# zpkdata

---

## Purpose

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  $T_s$ .

`[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- $N_z$ , where  $N_z$  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)
```

Sampling time: unspecified

you can extract the zero/pole/gain data embedded in `H` with

```
[z,p,k] = zpkdata(H)
z =
     [      0]
     [-0.5000]
p =
     [    0.3000]
     [2x1 double]
k =
     1
     2
```

To access the zeros and poles of the second output channel of `H`, get the content of the second cell in `z` and `p` by typing

```
z{2,1}
ans =
    -0.5000
p{2,1}
ans =
    0.1000+ 1.0000i
    0.1000- 1.0000i
```

## 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

`get` | `ssdata` | `tfdata` | `zpk`



# Block Reference

---

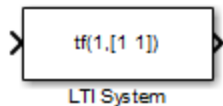
# LTI System

---

## Purpose

Use linear system model object in Simulink

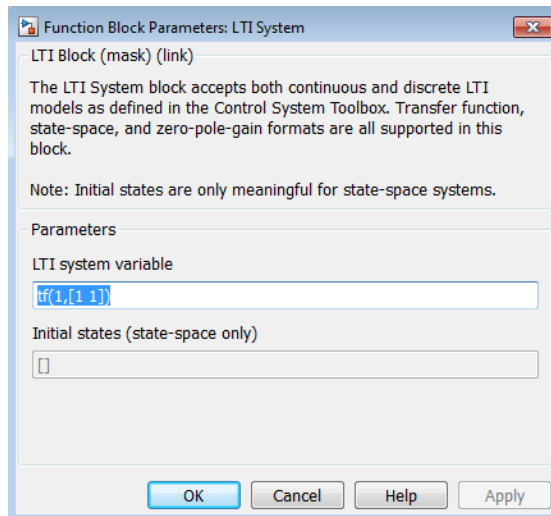
## Description



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.

## Dialog Box



### LTI system variable

Enter your LTI model. This block supports state-space, zero/pole/gain, and transfer function formats. Your model can be discrete- or continuous-time.

### Initial states (state-space only)

If your model is in state-space format, you can specify the initial states in vector format. The default is zero for all states.

# LTI System

---

## A

acker 2-2  
 algebraic loop 1-146  
 append 1-8  
 augstate 1-11

## B

balancing realizations 1-12  
 balreal 1-12  
 bodemag (Bode magnitude plots) 1-38

## C

c2d 1-49  
 cancellation 1-414  
 care 1-67  
 cell array 1-202  
 connection  
   parallel 1-468  
   series 1-583  
 continuous-time  
   conversion to.. *See* conversion, model  
   random model 1-581  
 controllability  
   matrix (ctrb) 1-91  
   staircase form 1-93  
 conversion, model  
   discrete to continuous (d2c) 1-97  
     with negative real poles 1-101  
   resampling  
     discrete models 1-106  
 covar 1-88  
 covariance  
   output 1-88  
   state 1-88  
 crossover frequencies  
   allmargin 1-6  
   margin 1-410  
 ctrb 1-91

ctrbf 1-93

## D

d2c 1-97  
 d2d 1-106  
 dare 1-114  
 dB to magnitude 1-116  
 db2mag 1-116 1-409  
 dcgain 1-117  
 dead time. *See* delays  
 delay2z 1-119  
 delays  
   combining 1-705  
   conversion 1-119 to 1-120  
   delay2z 1-119  
   delayss 1-120  
   existence of, test for 1-244  
   hasdelay 1-244  
 delayss 1-120  
 denominator  
   common denominator 1-680  
   specification 1-147  
 design  
   Kalman estimator 1-300  
   LQG 1-122 1-326  
   pole placement 1-544  
   regulators 1-326 1-563  
   state estimator 1-300  
 digital filter  
   specification 1-147  
 Dirac impulse 1-262  
 discrete-time models  
   Kalman estimator 1-300  
   random 1-127  
 discrete-time random models 1-127  
 discretization  
   available methods 1-59 1-108  
 dlqr 1-122  
 dlyap 1-124

drmodel 1-127  
drss 1-127  
dsort 1-129  
DSP convention 1-147  
dss 1-130

## E

esort 1-133  
estim 1-134  
estimator 1-300  
    current 1-302  
    discrete 1-300  
    discrete for continuous plant 1-305  
evalfr 1-137

## F

feedback 1-143  
    algebraic loop 1-146  
    negative 1-143  
    positive 1-143  
filt 1-147 1-151 1-162  
first-order hold (FOH) 1-59  
frd 1-151  
FRD (frequency response data) objects 1-151  
    data 1-162  
    frdata 1-162  
    singular value plots 1-606  
frdata 1-162  
freqresp 1-164  
frequency  
    crossover 1-410  
frequency response  
    at single frequency (evalfr) 1-137  
    Nichols chart (ngrid) 1-427  
    Nichols plot 1-429

## G

gain

    low frequency (DC) 1-117  
    state-feedback gain 1-122

gensig 1-187  
get 1-201  
gram 1-242  
gramian (gram) 1-13

## H

Hamiltonian matrix and pencil 1-67  
hasdelay 1-244

## I

impulse 1-262  
impulse response 1-262  
inheritance 1-130  
initial 1-271  
initial condition 1-271  
innovation 1-302  
input  
    Dirac impulse 1-262  
    pulse 1-187  
    sine wave 1-187  
    square wave 1-187  
inv 1-278  
inversion  
    limitations 1-279  
isempty 1-287  
isproper 1-291  
issiso 1-298

## K

kalman 1-300  
Kalman estimator  
    current 1-302  
    discrete 1-300  
    innovation 1-302  
    steady-state 1-300  
kalmd 1-305

**L**

LFT (linear-fractional transformation) 1-307  
 LQG (linear quadratic-gaussian) method  
   continuous LQ regulator 1-336  
   cost function 1-122  
   current regulator 1-327  
   discrete LQ regulator 1-122  
   Kalman state estimator 1-300  
   LQ-optimal gain 1-336  
   optimal state-feedback gain 1-336  
   regulator 1-326  
 lqr 1-336  
 lqrd 1-338  
 lqry 1-340  
 lsim 1-341  
 LTI models  
   discrete random 1-127  
   frd 1-151  
   model order reduction 1-416  
   model order reduction (balanced realization) 1-13  
   random 1-581  
   second-order 1-462  
 LTI properties  
   accessing property values (`get`) 1-201  
   displaying properties 1-201  
   inheritance 1-130  
   property names 1-201 1-585  
   property values 1-201 1-585  
   setting 1-585  
 LTI Viewer 1-401  
 ltiview 1-401  
 lyap 1-404  
 Lyapunov equation 1-90 1-243  
   continuous 1-404  
   discrete 1-124

**M**

magnitude to dB 1-409

margin 1-410  
 matched pole-zero 1-59  
 MIMO 1-262  
 minreal 1-414  
 model building  
   appending LTI models 1-8  
   parallel connection 1-468  
   series connection 1-583  
 model order reduction 1-416  
   balanced realization 1-13  
 modred 1-416

**N**

ngrid 1-427  
 nichols 1-429  
 Nichols  
   chart 1-427  
   plot (`nichols`) 1-429  
 noise  
   measurement 1-134  
   process 1-134  
   white 1-88  
 numerator  
   specification 1-147  
   value 1-202  
 nyquist 1-446

**O**

observability  
   matrix (`ctrb`) 1-458  
   staircase form 1-460  
 obsv 1-458  
 obsvf 1-460  
 operations on LTI models  
   append 1-8  
   augmenting state with outputs 1-11  
   diagonal building 1-8  
   sorting the poles 1-129

ord2 1-462  
output  
    covariance 1-88

## P

pade 1-464  
parallel 1-468  
parallel connection 1-468  
place 1-544  
plotting  
    Nichols chart (ngrid) 1-427  
    s-plane grid (sgrid) 1-599  
    z-plane grid (zgrid) 1-734  
pole 1-546 to 1-547  
pole placement 1-544  
pole-zero  
    cancellation 1-414  
    map (pzmap) 1-550  
poles  
    computing 1-546  
    multiple 1-546  
    pole-zero map 1-550  
    s-plane grid (sgrid) 1-599  
    sorting by magnitude (dsort) 1-129  
    z-plane grid (zgrid) 1-734  
pulse 1-187  
pzmap 1-550

## R

random models 1-581  
realization  
    state coordinate transformation 1-648  
realizations 1-630  
    balanced 1-12  
    minimal 1-414  
reduced-order models 1-416  
    balanced realization 1-13  
regulation 1-563

resampling (d2d) 1-106  
Riccati equation  
    continuous (care) 1-67  
    discrete (dare) 1-114  
    for LQG design 1-303  
    H-like 1-69  
rlocus 1-576  
rmodel 1-581  
root locus  
    plot (rlocus) 1-576  
rss 1-581

## S

sample time  
    resampling 1-106  
second-order model 1-462  
series 1-583  
series connection 1-583  
set 1-585  
simulation of linear systems.. *See* time response  
sine wave 1-187  
SISO Design Tool 1-620  
square wave 1-187  
stability margins  
    margin 1-410  
    pole 1-546  
    pzmap 1-550  
stabilizable 1-70  
state  
    augmenting with outputs 1-11  
    covariance 1-88  
    discrete estimator 1-305  
    estimator 1-300  
    feedback 1-122  
    transformation 1-648  
    uncontrollable 1-414  
    unobservable 1-414 1-460  
state-space models  
    balancing 1-12



- descriptor 1-130
- discrete random
  - discrete-time models 1-127
- dss 1-130
- initial condition response 1-271
- random
  - continuous-time 1-581
- realizations 1-630
- scaling 1-547
- state order 1-728
- step response 1-658
- Sylvester equation 1-404
- symplectic pencil 1-115

## T

- time response
  - impulse response (impulse) 1-262
  - initial condition response (initial) 1-271
  - MIMO 1-262
  - response to arbitrary inputs (lsim) 1-341
  - step response (step) 1-658
  - to white noise 1-88
- totaldelay 1-705
- transfer functions
  - common denominator 1-680

- discrete-time 1-147
- discrete-time random 1-127
- DSP convention 1-147
- filt 1-147
- MIMO 1-679
- quick data retrieval (tfdata) 1-697
- random 1-581
- static gain 1-680
- transmission zeros.. *See* zeros
- triangle approximation 1-59
- Tustin approximation 1-59 1-108
  - with frequency prewarping 1-59 1-108
- tzero. . *See* zero

## Z

- zero 1-731
- zero-order hold (ZOH) 1-59 1-108
- zero-pole-gain (ZPK) models
  - MIMO 1-737
  - quick data retrieval (zpkdata) 1-752
  - static gain 1-737
- zeros
  - computing 1-731
  - pole-zero map 1-550
  - transmission 1-731