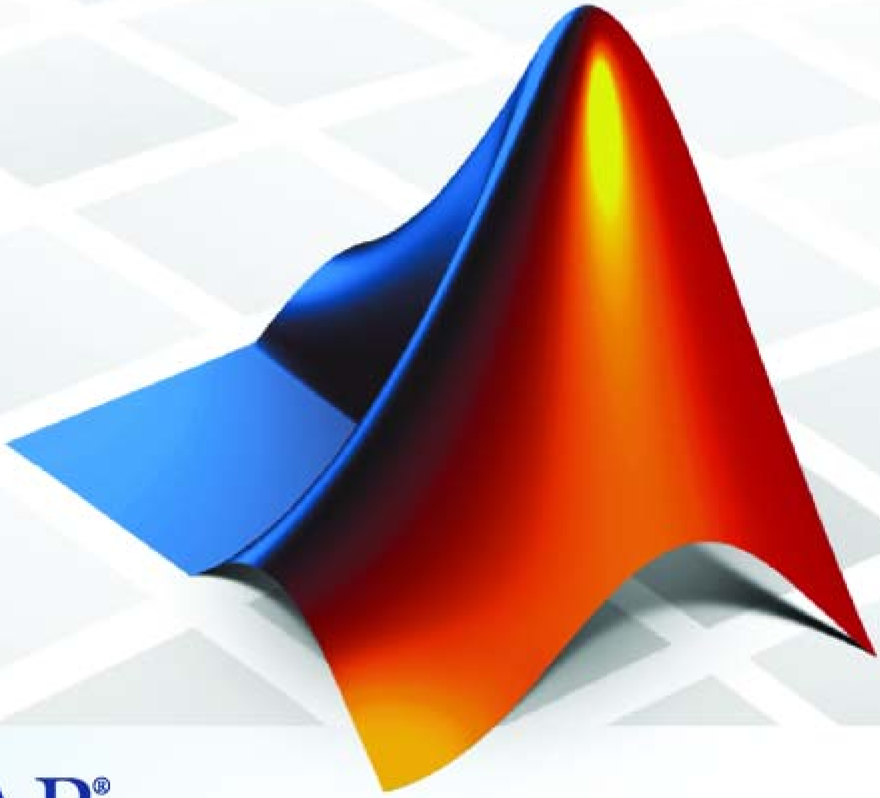
Control System Toolbox 8

Reference



MATLAB®

The MathWorks
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Control System Toolbox Reference

© COPYRIGHT 2001–2007 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

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

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `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)

Contents

1 Functions — By Category

Functions — Alphabetical List

2

Block Reference

3

Index

1

Functions — By Category

General (p. 1-3)	General purpose functions
Creating Linear Models (p. 1-3)	Create LTI SISO and MIMO models
Data Extraction (p. 1-3)	Retrieve data from LTI objects
Conversions (p. 1-4)	Convert between model formats
System Interconnections (p. 1-4)	Connect models
System Gain and Dynamics (p. 1-5)	Retrieve information about system gain and dynamics
Time Domain Analysis (p. 1-6)	Analyze models in the time domain
Frequency Domain Analysis (p. 1-6)	Analyze models in the frequency domain
Classical Design (p. 1-7)	Implement classical control design techniques
Compensator Design (p. 1-7)	Implement basic control design techniques
LQR/LQG Design (p. 1-8)	Implement linear-quadtratic-regulator/linear-quadratic-Gaussian techniques
State-Space Models (p. 1-8)	Create and manipulate SS models
Frequency Response Data (FRD) Models (p. 1-9)	Create and manipulate FRD models
Time Delays (p. 1-10)	Specify and manipulate model time delays
Model Dimensions and Characteristics (p. 1-10)	Extract information about models

Overloaded and Arithmetic Operators (p. 1-11)	Use arithmetic operators to connect and manipulate models
Matrix Equation Solvers (p. 1-12)	Solve Lyapunov and Riccati equations
Command-Line Plot Customization (p. 1-12)	Customize plots from the command line

General

`ctrlpref`	Set Control System Toolbox preferences
`ltimodels`	Help on LTI models
`ltiprops`	Help on LTI model properties

Creating Linear Models

`dss`	Specify descriptor state-space models
`filt`	Specify discrete transfer functions in DSP format
`frd`	Create or convert to frequency-response data models
`set`	Set or modify LTI model properties
`tf`	Create or convert to transfer function model
`zpk`	Create or convert to zero-pole-gain model

Data Extraction

`dssdata`	Extract descriptor state-space data
`frdata`	Access data for frequency response data (FRD) object
`get`	Access LTI property values
`ssdata`	Access state-space model data

`tfdata`	Access transfer function data
`zpkdata`	Access zero-pole-gain data

Conversions

`c2d`	Convert from continuous- to discrete-time models
`chgunits`	Change frequency units of FRD model
`d2c`	Convert from discrete- to continuous-time models
`d2d`	Resample discrete-time LTI model or add input delay
`frd`	Create or convert to frequency-response data models
`ss`	Specify state-space models or convert LTI model to state space
`tf`	Create or convert to transfer function model
`zpk`	Create or convert to zero-pole-gain model

System Interconnections

`append`	Group LTI models by appending their inputs and outputs
`connect`	Arbitrary interconnection of LTI models

`feedback`	Feedback connection of two LTI models
`lft`	Generalized feedback interconnection of two LTI models (Redheffer star product)
`parallel`	Parallel connection of two LTI models
`series`	Series connection of two LTI models

System Gain and Dynamics

`bandwidth`	Frequency response bandwidth
`damp`	Natural frequency and damping of system poles
`dcgain`	Low-frequency (DC) gain of LTI system
`dsort`	Sort discrete-time poles by magnitude
`esort`	Sort continuous-time poles by real part
`iopzmap`	Plot pole-zero map for I/O pairs of LTI model
`modsep`	Region-based modal decomposition
`norm`	Compute LTI model norm
`pole`	Compute poles of LTI system
`pzmap`	Compute pole-zero map of LTI models
`stabsep`	Stable/unstable decomposition of LTI model

Time Domain Analysis

`covar`	Output and state covariance of system driven by white noise
`gensig`	Generate test input signals for `lsim`
`impulse`	Impulse response of LTI model
`initial`	initial condition response of state-space model
`lsim`	Simulate LTI model responses to arbitrary inputs
`lsiminfo`	Compute linear response characteristics
`ltiview`	LTI Viewer for LTI system response analysis
`step`	Step response of LTI systems
`stepinfo`	Compute step response characteristics

Frequency Domain Analysis

`allmargin`	All crossover frequencies and corresponding stability margins
`bode`	Bode diagram of frequency response
`bodemag`	Bode magnitude response of LTI models
`evalfr`	Evaluate frequency response at given frequency
`freqresp`	Frequency response over frequency grid

`ltiview`	LTI Viewer for LTI system response analysis
`margin`	Gain and phase margins and associated crossover frequencies
`nichols`	Nichols plot of LTI models
`nyquist`	Nyquist plot of LTI models
`sigma`	Plot singular values of LTI models

Classical Design

`rlocus`	Evans root locus
`sisotool`	Initialize SISO Design Tool

Compensator Design

`acker`	Pole placement design for single-input systems
`estim`	Form state estimator given estimator gain
`place`	Pole placement design
`reg`	Form regulator given state-feedback and estimator gains

LQR/LQG Design

`augstate`	Append state vector to output vector
`dlqr`	Linear-quadratic (LQ) state-feedback regulator for discrete-time state-space system
`kalman`	Design continuous- or discrete-time Kalman estimator
`kalmd`	Design discrete Kalman estimator for continuous plant
`lqgreg`	Form LQG regulator given state-feedback gain and Kalman estimator
`lqr`	Linear-quadratic (LQ) state-feedback regulator for state-space system
`lqrd`	Design discrete linear-quadratic (LQ) regulator for continuous plant
`lqry`	Form linear-quadratic (LQ) state-feedback regulator with output weighting

State-Space Models

`balreal`	Gramian-based input/output balancing of state-space realizations
`canon`	State-space canonical realization
`ctrb`	Controllability matrix
`gram`	Controllability and observability grammians
`margin`	Gain and phase margins and associated crossover frequencies

`minreal`	Minimal realization or pole-zero cancelation
`modred`	Model order reduction
`ngrid`	Superimpose Nichols chart on Nichols plot
`nichols`	Nichols plot of LTI models
`nyquist`	Nyquist plot of LTI models
`obsv`	Observability matrix
`sminreal`	Perform model reduction based on structure
`ss2ss`	State coordinate transformation for state-space model
`ssbal`	Balance state-space model using diagonal similarity transformation

Frequency Response Data (FRD) Models

`abs`	Entrywise magnitude of frequency response
`chgunits`	Change frequency units of FRD model
`fcat`	Concatenate FRD models along frequency dimension
`fnorm`	Pointwise peak gain of FRD model
`fselect`	Select frequency points or range in FRD model
`imag`	Imaginary part of FRD model
`interp`	Interpolate FRD model
`real`	Real part of frequency response for FRD model

Time Delays

`delay2z`	Replace delays of discrete-time TF, SS, or ZPK models by poles at z=0, or replace delays of FRD models by phase shift
`delayss`	Create state-space models with delayed terms
`getdelaymodel`	State-space representation of internal delays
`hasdelay`	True for LTI model with time delays
`pade`	Padé approximation of model with time delays
`totaldelay`	Total combined I/O delays for LTI model

Model Dimensions and Characteristics

`isct, isdt`	Determine whether LTI model is continuous or discrete
`isempty`	Determine whether LTI model is empty
`isproper`	Determine whether LTI model is proper
`issiso`	Determine whether LTI model is single-input/single-output (SISO)
`ndims`	Provide number of dimensions of LTI model or LTI array

`reshape`	Change shape of LTI array
`size`	Provide output/input/array dimensions of LTI model, model order of TF, SS, and ZPK model, and number of frequencies of FRD model

Overloaded and Arithmetic Operators

+ and —	Add and subtract systems (parallel connection)
*	Multiply systems (series connection)
.*	Element-by-element multiplication
\	Left divide — `sys1\sys2` means `inv(sys1)*sys2`
/	Right divide — `sys1/sys2` means `sys1*inv(sys2)`
^	Powers of given system
’	Pertransposition
.’	Transposition of input/output map
[..]	Concatenate models along inputs or outputs
`conj`	Complex conjugation of model coefficients

`inv`	Invert LTI systems
`stack`	Build LTI array by stacking LTI models or LTI arrays along array dimensions

Matrix Equation Solvers

`care`	Solve continuous-time algebraic Riccati equation
`dare`	Solve discrete-time algebraic Riccati equations (DAREs)
`dlyap`	Solve discrete-time Lyapunov equations
`dlyapchol`	Square-root solver for continuous-time Lyapunov equations
`gcare`	Generalized solver for continuous-time algebraic Riccati equation
`gdare`	Generalized solver for discrete-time algebraic Riccati equation
`lyap`	Solve continuous-time Lyapunov equation
`lyapchol`	Square-root solver for continuous-time Lyapunov equation

Command-Line Plot Customization

`bodeplot`	Plot Bode frequency response and return plot handle
`getoptions`	Return `@PlotOptions` handle or plot options property
`hsvplot`	Plot Hankel singular values and return plot handle
`impulseplot`	Plot impulse response and return plot handle

`initialplot`	Plot initial condition response and return plot handle
`iopzplot`	Plot pole-zero map for I/O pairs and return plot handle
`lsimplot`	Simulate LTI model responses to arbitrary inputs and return plot handle
`nicholsplot`	Plot Nichols frequency responses and return plot handle
`nyquistplot`	Plot Nyquist frequency responses and return plot handle
`pzplot`	Plot pole-zero map of LTI model and return plot handle
`rlocusplot`	Plot root locus and return plot handle
`setoptions`	Set plot options for response plot
`sigmaplot`	Plot singular values of frequency response and return plot handle
`stepplot`	Plot step response of LTI systems and return plot handle

2

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`, `frd/imag`, `frd/real`, `fnorm`

Purpose

Pole placement design for single-input systems

Syntax

```
k = acker(A,b,p)
```

Description

```
k = acker(A,b,p)
```

Given the single-input system

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

and a vector `p` of desired closed-loop pole locations, `acker` (A,b,p) uses Ackermann's formula [1] to calculate a gain vector `k` such that the state feedback $u = -kx$ places the closed-loop poles at the locations `p`. In other words, the eigenvalues of $A - bk$ match the entries of `p` (up to ordering). Here `A` is the state transmitter matrix and `b` is the input to state transmission vector.

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

```
l = acker(a',c',p).'
```

Limitations

`acker` is limited to single-input systems and the pair (A, b) must be controllable.

Note that this method is not numerically reliable and starts to break down rapidly for problems of order greater than 5 or for weakly controllable systems. See `place` for a more general and reliable alternative.

References

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

See Also

`lqr`, `place`, `rlocus`

allmargin

Purpose All crossover frequencies and corresponding stability margins

Syntax

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

Description `S = allmargin(sys)`

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

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

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

In general, stability cannot be assessed for FRD system. In any case when stability cannot be assessed, `S` is set to `NaN`.

`s = allmargin(mag,phase,w,ts)` computes the stability margins from the frequency response data `mag`, `phase`, `w`, and the sampling time, `ts`. `allmargin` expects frequency values `w` in rad/s, magnitude values `mag` in linear scale, and phase values `phase` in degrees. Interpolation is used between frequency points to approximate the true stability margins.

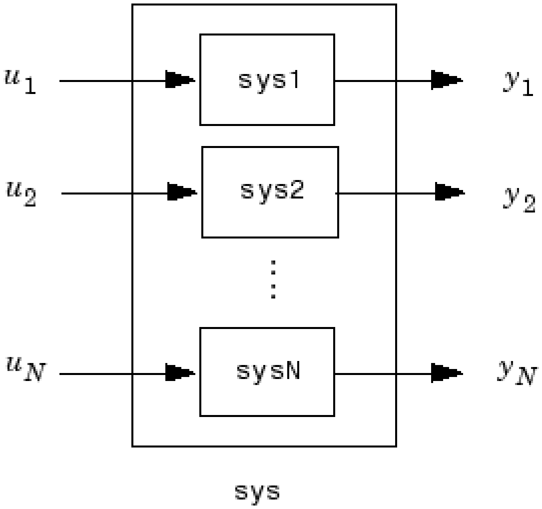
See Also `ltimodels`, `ltiview`, `margin`

Purpose Group LTI 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 LTI models `sys1`,...,`sysN` to form the augmented model sys depicted below.



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

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

For state-space models `sys1` and `sys2` with data (A_1, B_1, C_1, D_1) and (A_2, B_2, C_2, D_2), `append(sys1,sys2)` produces the following state-space model.

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} A_1 & 0 \\ 0 & A_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 & 0 \\ 0 & B_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} C_1 & 0 \\ 0 & C_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} D_1 & 0 \\ 0 & D_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

Arguments

The input arguments `sys1`,..., `sysN` can be LTI models of any type. Regular matrices are also accepted as a representation of static gains, but there should be at least one LTI object in the input list. The LTI models should be either all continuous, or all discrete with the same sample time. When appending models of different types, the resulting type is determined by the precedence rules (see Precedence Rules for details).

There is no limitation on the number of inputs.

Example

The commands

```
sys1 = tf(1,[1 0])
sys2 = ss(1,2,3,4)
sys = append(sys1,10,sys2)
```

produce the state-space model

```
sys

a =
                          x1          x2
             x1            0           0
             x2            0     1.00000
```

```
b =
                    u1           u2           u3
           x1  1.00000            0            0
           x2        0            0      2.00000



c =
                    x1           x2
           y1  1.00000            0
           y2        0            0
           y3        0      3.00000

d =
                    u1           u2           u3
           y1        0            0            0
           y2        0     10.00000            0
           y3        0            0      4.00000

Continuous-time system.
```

See Also `connect`, `feedback`, `parallel`, `series`

augstate

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

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

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

This command prepares the plant so that you can use the `feedback` command to close the loop on a full-state feedback $u = -Kx$.

Limitation

Because `augstate` is only meaningful for state-space models, it cannot be used with TF, ZPK or FRD models.

See Also

`feedback`, `parallel`, `series`

Purpose

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

Syntax

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

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`. You can specify additional options for the stable/unstable decomposition:

```
[sysb,g] = balreal(sys,...
                  'AbsTol',ATOL,'RelTol',RTOL,'Offset',ALPHA)
```

See `stabsep` for more details on these options. The default values are `ATOL=0`, `RTOL=1e-8`, and `ALPHA=1e-8`.

Use `balreal(sys,condmax)` to control 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 $T\mathrm{i} = T^{-1}$.

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.

There are also overloaded methods available. Type

```
help ss/balreal
help lti/balreal
help idmodel/balreal
```

for more information.

Example1

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')
```

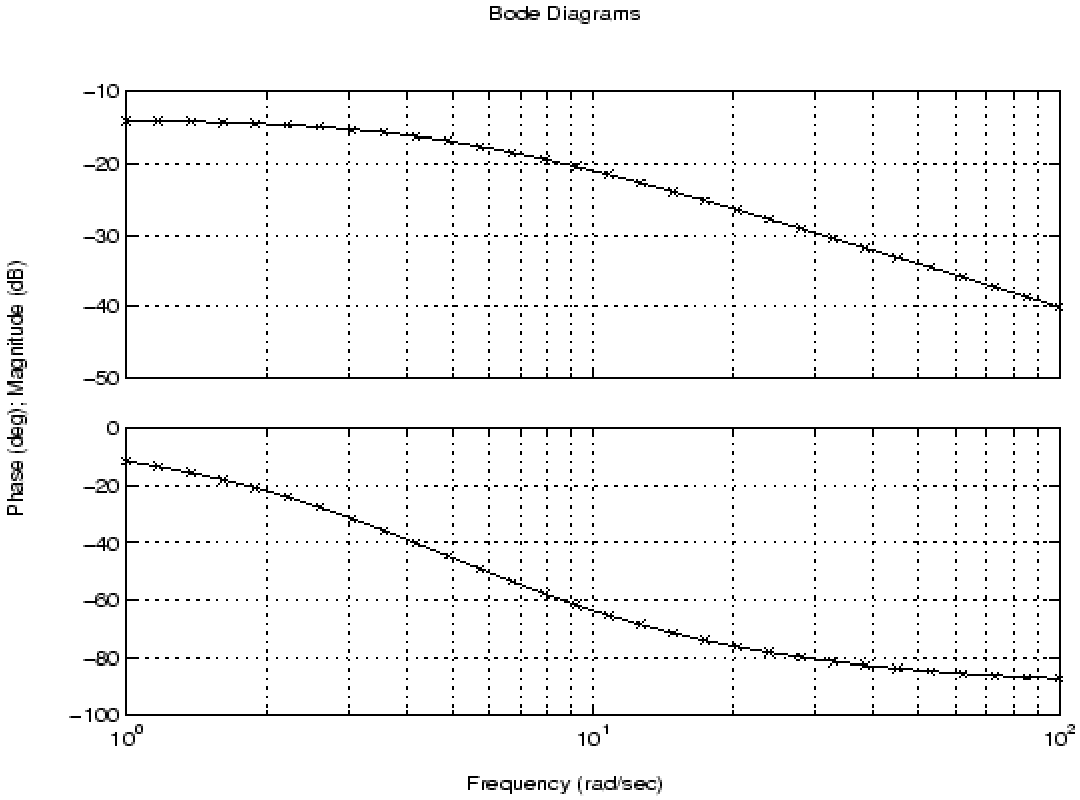
to obtain the following first-order approximation of the original system.

```
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')
```



Example2

Create this unstable system:

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

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

Apply `balreal` to create a balanced gramian realization.

```
[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.

Algorithm

Consider the model

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

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

$$\begin{aligned} \dot{\bar{x}} &= TAT^{-1}\bar{x} + TBu \\ y &= CT^{-1}\bar{x} + Du \end{aligned}$$

and transforms the gramians to

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

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

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

See [1], [2] for details on the algorithm.

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 `gram`, `modred`, `ss`, `ssbal`

Purpose Model order reduction

Syntax

```
rsys = balred(sys,ORDERS)
rsys = balred(sys,ORDERS,...,'Elimination',METHOD)
rsys = balred(sys,ORDERS,...,'Balancing',BALDATA)
```

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. 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

```
sys = balred(sys,ORDERS,'AbsTol',ATOL,...
             'RelTol',RTOL,'Offset',ALPHA)
```

to specify additional options for the stable/unstable decomposition. See `stabsep` for details. The default values are `ATOL=0`, `RTOL=1e-8`, and `ALPHA=1e-8`.

`rsys = balred(sys,ORDERS,...,'Elimination',METHOD)` specifies the state elimination method. Available choices for `METHOD` include:

- `'MatchDC'`: Enforce matching DC gains (default)
- `'Truncate'`: Simply discard the states associated with small Hankel singular values. The `'Truncate'` method tends to produce a better approximation in the frequency domain, but the DC gains are not guaranteed to match.

`rsys = balred(sys,ORDERS,...,'Balancing',BALDATA)` makes use of the balancing data `BALDATA` produced by `hsvd`. Because `hsvd` does most of the work needed to compute `rsys`, this syntax is more efficient when using `hsvd` and `balred` jointly.

`balred` uses implicit balancing techniques to compute the reduced-order approximation `rsys`.

There is more than one `balred` method available. Type

```
help lti/balred
```

for more information.

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

`hsvd`, `lti/order`, `minreal`, `sminreal`

Purpose

Frequency response bandwidth

Syntax

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

Description

`fb = bandwidth(sys)` computes the bandwidth `fb` of the SISO model `sys`, defined as the first frequency where the gain drops below 70.79 percent (-3 dB) of its DC value. The frequency `fb` is expressed in radians per second.

You can create `sys` using `tf`, ss, or `zpk`. See `ltimodels` for details. 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 LTI models, `bandwidth` returns an array of the same size such that

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

See Also

`dcgain`, `issiso`, `ltimodels`

bode

Purpose

Bode diagram of frequency response

Syntax

```
bode
bode(sys)
bode(sys,w)
bode(sys1,sys2,...,sysN)
bode(sys1,sys2,...,sysN,w)
bode(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
[mag,phase,w] = bode(sys)
[mag,phase] = bode(sys,w)
```

Description

`bode` computes the magnitude and phase of the frequency response of LTI models. When invoked without left-side arguments, `bode` produces a Bode plot on the screen. The magnitude is plotted in decibels (dB), and the phase in degrees. The decibel calculation for `mag` is computed as $20\log_{10}(|H(j\omega)|)$, where $|H(j\omega)|$ is the system's frequency response. Bode plots are used to analyze system properties such as the gain margin, phase margin, DC gain, bandwidth, disturbance rejection, and stability.

`bode(sys)` plots the Bode response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. In the MIMO case, `bode` produces an array of Bode plots, each plot showing the Bode response of one particular I/O channel. The frequency range is determined automatically based on the system poles and zeros.

`bode(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in rad/s.

`bode(sys1,sys2,...,sysN)` or `bode(sys1,sys2,...,sysN,w)` plots the Bode responses of several LTI models on a single figure. All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous and discrete systems. This syntax is useful to compare the Bode responses of multiple systems.

`bode(sys1,'PlotStyle1',...,sysN,'PlotStyleN')` specifies which color, linestyle, and/or marker should be used to plot each system. For example,

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

uses red dashed lines for the first system `sys1` and green `'x'` markers for the second system `sys2`.

When invoked with left-side arguments

```
[mag,phase,w] = bode(sys)
[mag,phase] = bode(sys,w)
```

return the magnitude and phase (in degrees) of the frequency response at the frequencies `w` (in rad/s). The outputs `mag` and `phase` are 3-D arrays with the frequency as the last dimension (see "Arguments" below for details). You can convert the magnitude to decibels by

```
magdb = 20*log10(mag)
```

Remark

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

Arguments

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

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

For SISO systems, `mag(1,1,k)` and `phase(1,1,k)` give the magnitude and phase of the response at the frequency ω_k= `w(k)`.

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

MIMO systems are treated as arrays of SISO systems and the magnitudes and phases are computed for each SISO entry h_{ij} independently (h_{ij} is the transfer function from input j to output i). The values `mag(i,j,k)` and `phase(i,j,k)` then characterize the response of h_{ij} at the frequency `w(k)`.

$$\mathrm{mag(i,j,k)} = \left|h_{ij}(j\omega_k)\right|$$

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

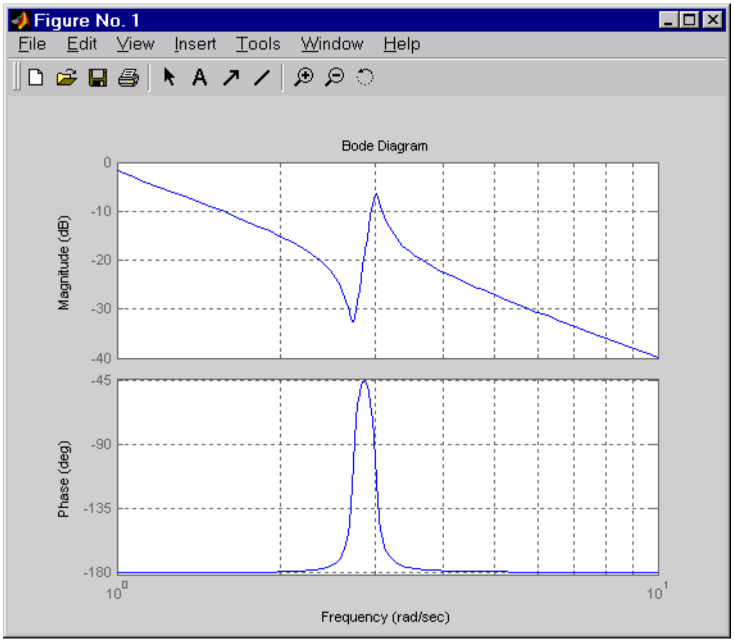
Example

You can plot the Bode response of the continuous SISO system

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

by typing

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

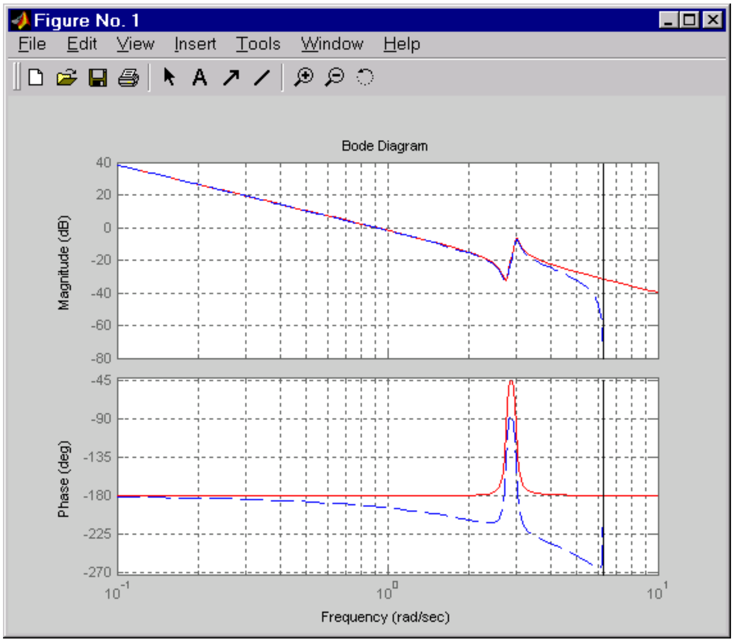


To plot the response on a wider frequency range, for example, from 0.1 to 100 rad/s, type

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

You can also discretize this system using zero-order hold and the sample time $T_s = 0.5$ second, and compare the continuous and discretized responses by typing

```
gd = c2d(g,0.5)
bode(g,'r',gd,'b--')
```



Algorithm

For continuous-time systems, `bode` computes the frequency response by evaluating the transfer function $H(s)$ on the imaginary axis $s = j\omega$. Only positive frequencies ω are considered. For state-space models, the frequency response is $D + C(j\omega - A)^{-1}B\ ,\qquad \omega \geq 0$

When numerically safe, A is diagonalized for maximum speed. Otherwise, A is reduced to upper Hessenberg form and the linear equation $(j\omega - A)X = B$ is solved at each frequency point, taking advantage of the Hessenberg structure. The reduction to Hessenberg form provides a good compromise between efficiency and reliability. See [1] for more details on this technique.

For discrete-time systems, the frequency response is obtained by evaluating the transfer function $H(z)$ on the unit circle. To facilitate interpretation, the upper-half of the unit circle is parametrized as

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

where T_s is the sample time. ω_N is called the *Nyquist frequency*. The equivalent "continuous-time frequency" ω is then used as the x-axis variable. Because $H(e^{j\omega T_s})$

is periodic with period $2\omega_N$, `bode` plots the response only up to the Nyquist frequency ω_N. If the sample time is unspecified, the default value $T_s = 1$ is assumed.

Diagnostics

If the system has a pole on the $j\omega$ axis (or unit circle in the discrete case) and `w` happens to contain this frequency point, the gain is infinite, $j\omega I - A$ is singular, and `bode` produces the warning message

```
Singularity in freq. response due to jw-axis or unit circle pole.
```

References

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

See Also

`evalfr`, `freqresp`, `ltiview`, `nichols`, `nyquist`, `sigma`

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 LTI model SYS (Bode plot without the phase diagram). The frequency range and number of points are chosen automatically.

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

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

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

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

See Also

`bode`, `ltiview`, `ltimodels`

bodeplot

Purpose

Plot Bode frequency response and return plot handle

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 an LTI 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 LTI model `sys` (created with either `tf`, `zpk`, `ss`, or `frd`). The frequency range and number of points are chosen automatically.

`bodeplot(sys1,sys2,...)` graphs the Bode response of multiple LTI 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 2-25 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/s). When `w` is a user-supplied vector `w` of frequencies, in rad/s, the Bode response is drawn for the specified frequencies.

See `logspace` to generate logarithmically spaced frequency vectors.

Examples

Example 1

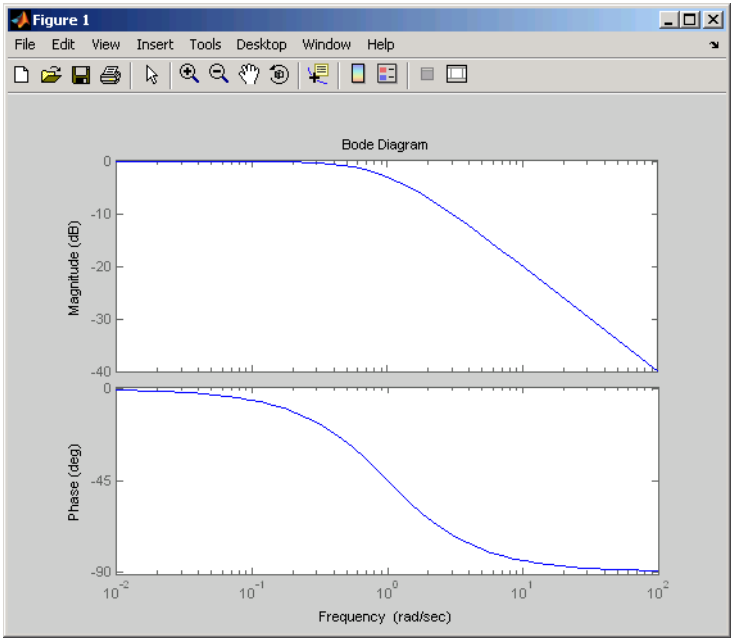
Use the plot handle to change options in a Bode plot.

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

Example 2

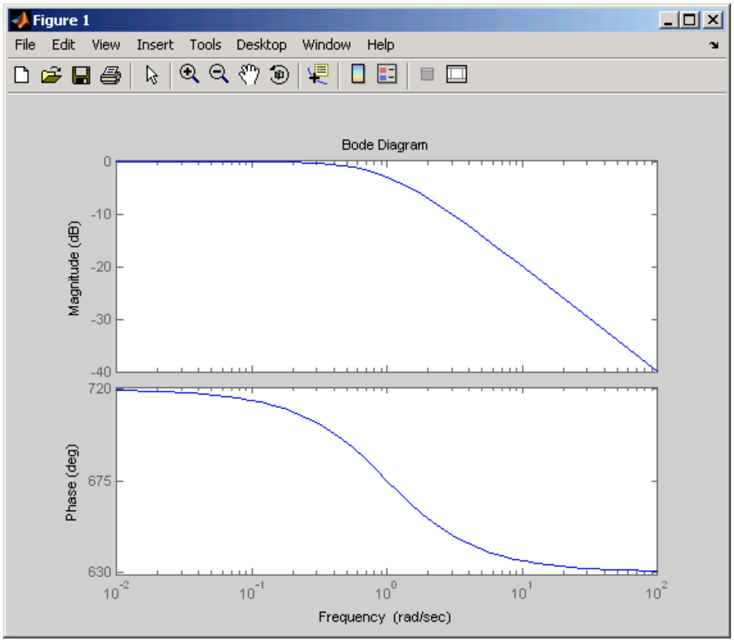
The properties `PhaseMatchingFreq` and `PhaseMatchingValue` are parameters you can use to specify the phase at a specified frequency. For example, enter the following commands.

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



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*360, where N is an integer, and so the plot is set to the nearest allowable phase, namely 675 degrees (or 2*360 - 45 = 675).

See Also

`bode`, `getoptions`, `setoptions`

c2d

Purpose Convert from continuous- to discrete-time models

Syntax

```
sysd = c2d(sys,Ts)
sysd = c2d(sys,Ts,
[sysd,G] = c2d(sys,Ts,method)
```

Description `sysd = c2d(sys,Ts)` discretizes the continuous-time LTI model `sys` using zero-order hold on the inputs and a sample time of `Ts` seconds.

`sysd = c2d(sys,Ts,`*`method`*`)` gives access to alternative discretization schemes. The string *`method`* selects the discretization method among the following:

`'zoh'`	Zero-order hold. The control inputs are assumed piecewise constant over the sampling period `Ts`.
`'foh'`	Triangle approximation (modified first-order hold, see [1], p. 151). The control inputs are assumed piecewise linear over the sampling period `Ts`.
’imp’	Impulse-invariant discretization
`'tustin'`	Bilinear (Tustin) approximation
`'prewarp'`	Tustin approximation with frequency prewarping. You must specify the critical frequency `Wc` (in rad/s) as a fourth input as in `sysd = c2d(sysc,ts,'prewarp',Wc)`
`'matched'`	Matched pole-zero method. See [1], p. 147.

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

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

For state-space systems,

```
[sysd,G] = c2d(sys,Ts,method)
```

returns a matrix `G` that maps the continuous initial conditions x_0 and u_0 to their discrete counterparts $x[0]$ and $u[0]$ according to

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

$$u[0] = u_0$$

Example Consider the system

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

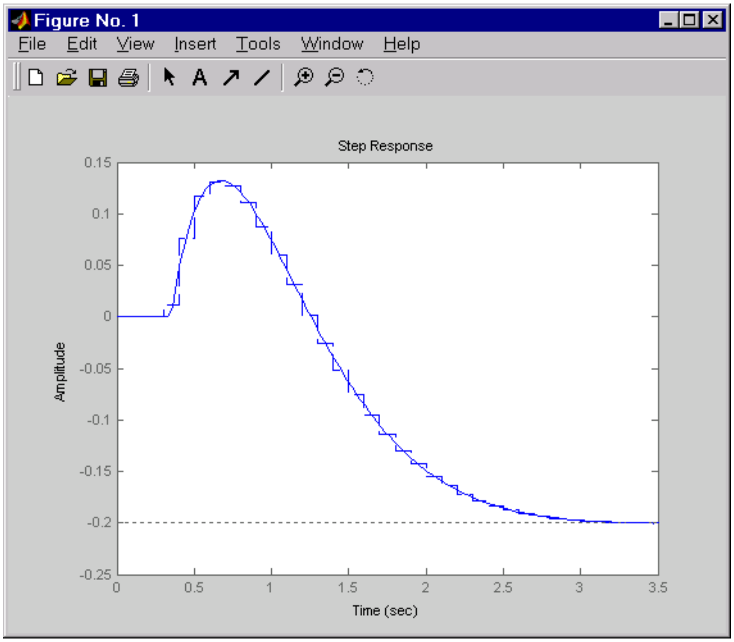
with input delay $T_d = 0.35$ second. To discretize this system using the triangle approximation with sample time $T_s = 0.1$ second, type

```
H = tf([1 -1],[1 4 5],'inputdelay',0.35)
Transfer function:
                        s - 1
exp(-0.35*s) * -------------
                  s^2 + 4 s + 5
Hd = c2d(H,0.1,'foh')
Transfer function:
0.0115 z^3 + 0.0456 z^2 - 0.0562 z - 0.009104
---------------------------------------------
        z^6 - 1.629 z^5 + 0.6703 z^4

Sampling time: 0.1
```

The next command compares the continuous and discretized step responses.

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



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

See Also `d2c`, `d2d`

canon

Purpose

State-space canonical realization

Syntax

```
csys = canon(sys,'modal')
csys = canon(sys,'modal',CONDT)
csys = canon(sys,'companion')
[csys,T] = canon(sys,'type')
```

Description

`canon` computes a canonical state-space model for the continuous or discrete LTI system `sys`. Two types of canonical forms are supported.

Modal Form

`csys = canon(sys,'modal')` returns a realization `csys` in modal form. If A has no repeated eigenvalues, the real eigenvalues appear on the diagonal of the A matrix and the complex conjugate eigenvalues appear in 2-by-2 blocks on the diagonal of A. For a system with eigenvalues $(\lambda_1, \sigma \pm j\omega, \lambda_2)$, the modal A matrix is of the form

$$\begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}$$

`csys = canon(sys,'modal',CONDT)` specifies an upper bound `CONDT` on the condition number of the block-diagonalizing transformation `T`. The default value is `CONDT=1e8`. Increase `CONDT` to reduce the size of the eigenvalue clusters (setting `CONDT=Inf` amounts to diagonalizing A).

Companion Form

`csys = canon(sys,'companion')` produces a companion realization of `sys` where the characteristic polynomial of the system appears explicitly in the rightmost column of the A matrix. For a system with characteristic polynomial

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

the corresponding companion A matrix is

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

For state-space models `sys`,

```
[csys,T] = canon(sys,'type')
```

also returns the state coordinate transformation `T` relating the original state vector x and the canonical state vector x_c, where

$$x_c = Tx$$

This syntax is meaningful only when `sys` is a state-space model.

Algorithm Transfer functions or zero-pole-gain models are first converted to state space using `ss`.

The transformation to modal form uses the matrix P of eigenvectors of the A matrix. The modal form is then obtained as

$$\dot{x}_c = P^{-1}APx_c + P^{-1}Bu$$
$$y = CPx_c + Du$$

The state transformation T returned is the inverse of P.

The reduction to companion form uses a state similarity transformation based on the controllability matrix [1].

Limitations The companion transformation requires that the system be controllable from the first input. The companion form is often poorly conditioned for most state-space computations; avoid using it when possible.

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

See Also `ctrb`, `ctrbf`, `ss2ss`

care

Purpose

Solve continuous-time algebraic Riccati equation

Syntax

```
[X,L,G] = care(A,B,Q)
[X,L,G] = care(A,B,Q,R,S,E)
[X,L,G,report] = care(A,B,Q,...)
[X1,X2,D,L] = care(A,B,Q,...,'factor')
```

Description

`[X,L,G] = care(A,B,Q)` computes the unique solution `X` of the continuous-time algebraic Riccati equation

$$A^T X + XA - XBB^T X + Q = 0$$

The `care` function also returns the gain matrix, $G = R^{-1} B^T XE$.

`[X,L,G] = care(A,B,Q,R,S,E)` solves the more general Riccati equation

$$A^T XE + E^T XA - (E^T XB + S) R^{-1} (B^T XE + S^T) + Q = 0$$

When omitted, `R`, `S`, and `E` are set to the default values `R=I`, `S=0`, and `E=I`. Along with the solution `X`, `care` returns the gain matrix $G = R^{-1}(B^T XE + S^T)$ and a vector `L` of closed-loop eigenvalues, where

```
L=eig(A-B*G,E)
```

`[X,L,G,report] = care(A,B,Q,...)` returns a diagnosis `report` with:

- `-1` when the associated Hamiltonian pencil has eigenvalues on or very near the imaginary axis (failure)
- `-2` when there is no finite stabilizing solution `X`
- The Frobenius norm of the relative residual if `X` exists and is finite.

This syntax does not issue any error message when X fails to exist.

`[X1,X2,D,L] = care(A,B,Q,...,'factor')` returns two matrices `X1`, `X2` and a diagonal scaling matrix `D` such that `X = D*(X2/X1)*D`.

The vector L contains the closed-loop eigenvalues. All outputs are empty when the associated Hamiltonian matrix has eigenvalues on the imaginary axis.

Examples

Example 1

Given

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

you can solve the Riccati equation

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

by

```
a = [-3 2;1 1]
b = [0 ; 1]
c = [1 -1]
r = 3
[x,l,g] = care(a,b,c'*c,r)
```

This yields the solution

```
x

x =
    0.5895    1.8216
    1.8216    8.8188
```

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

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

```
ans =
   -3.4495   -3.5026
    1.4495   -1.4370
```

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

```
l

l =
   -3.5026
   -1.4370
```

Example 2

To solve the H_∞-like Riccati equation

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

rewrite it in the `care` format as

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

You can now compute the stabilizing solution X by

```
B = [B1 , B2]
m1 = size(B1,2)
m2 = size(B2,2)
R = [-g^2*eye(m1) zeros(m1,m2) ; zeros(m2,m1) eye(m2)]

X = care(A,B,C'*C,R)
```

Algorithm

`care` implements the algorithms described in [1]. It works with the Hamiltonian matrix when R is well-conditioned and $E = I$; 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`

Purpose Change frequency units of FRD model

Syntax `sys = chgunits(sys,units)`

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

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

Example

```
w = logspace(1,2,2);
sys = rss(3,1,1);
sys = frd(sys,w)
From input 'input 1' to:
  Frequency(rad/s)        output 1
  ----------------        --------
          10         0.293773+0.001033i
         100         0.294404+0.000109i
Continuous-time frequency response data.
sys = chgunits(sys,'Hz')
sys.freq
ans =
    1.5915
   15.9155
```

See Also `frd`, `get`, `set`

Purpose Form model with complex conjugate coefficients

Syntax

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

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

Example If `sys` is the transfer function

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

then `conj(sys)` produces the transfer function

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

This operation is useful for manipulating partial fraction expansions.

See Also `append`, `ss`, `tf`, `zpk`

connect

Purpose

Arbitrary interconnection of LTI models

Syntax

```
sysc = connect(sysa, sysb, ..., inputs,
   outputs) sysc = connect(sys,Q,inputs,outputs)
```

Description

`sysc = connect(sysa, sysb, ..., inputs, outputs) sysc = connect(sys,Q,inputs,outputs)`

`connect` constructs the aggregate model for a given block diagram interconnection of LTI models. You can specify the block diagram connectivity in two ways, name-based and index-based.

Name-Based Interconnection

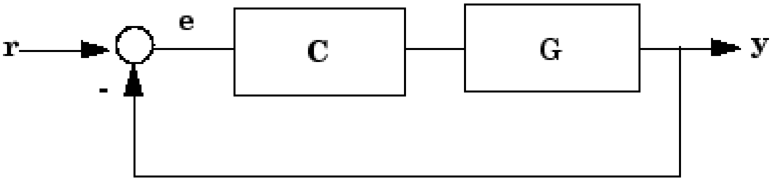
In this approach, you name the input and output signals of all LTI blocks `sys1`, `sys2,...` in the block diagram, including the summation blocks. The aggregate model `sys` is then built by

```
sys = connect
(sys1,sys2,...,inputs,outputs)
```

where `inputs` and `outputs` are the names of the block diagram external I/Os, specified as strings or cell arrays of strings.

Example

Given LTI models C and G, and referring to this block diagram,



you can construct the closed-loop transfer `T` from `r` to `y` as follows

```
C.InputName = e;  C.OutputName = u;
G.InputName = u;  G.OutputName = y;
Sum = ss([1,-1],'InputName',{'r','y'},'OutputName','e');
T = connect(G,G,Sum,'r','y')
```

Index-Based Interconnection

In this approach, first combine all LTI blocks into an aggregate, unconnected model `blksys` using `append`. Then construct a matrix `Q` where each row specifies one of the connections or summing junctions in terms of the input vector `u` and output vector `y` of `blksys`. For example, the row

```
[3 2 0 0]
```

indicates that `y(2)` feeds into u(3), while the row

```
[7 2 -15 6]
```

indicates that `y(2)-y(15)+y(6)` feeds into `u(7)`. The aggregate model `sys` is then obtained by

```
sys = connect(blksys,Q,inputs,outputs)
```

where `inputs` and `outputs` are index vectors into `u` and `y` selecting the block diagram external I/Os.

Example

You can construct the closed-loop model `T` for the block diagram above as follows:

```
blksys = append(C,G);
% u = inputs to C,G.  y = outputs of C,G.
% Here y(1) feeds into u(2) and -y(2) feeds into u(1)
Q = [2 1; 1 -2];
% External I/Os: r drives u(1) and y is y(2)
T = connect(blksys,Q,1,2)
```

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

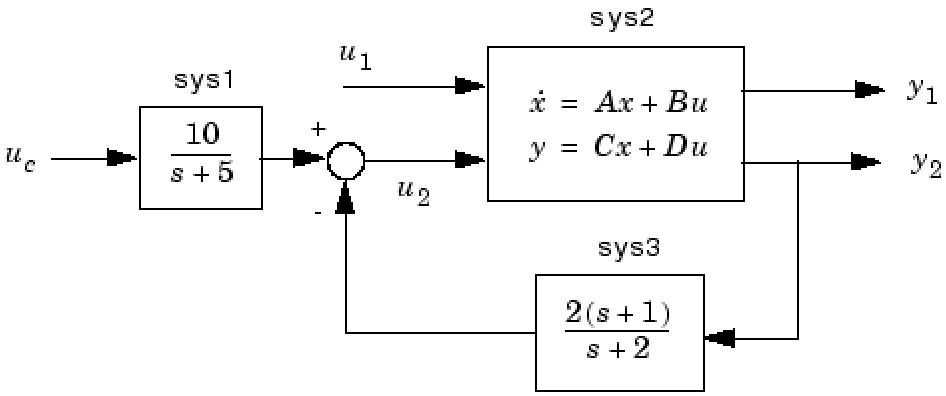
- Make sure the poles of the unconnected model `sys` match the poles of the various blocks in the diagram.
- Check that the final poles and DC gains are reasonable.
- Plot the `step` and `bode` responses of `sysc` and compare them with your expectations.

Delays

The `connect` function supports I/O and internal delays. See Time Delays for more information and examples.

Example

Consider the following block diagram



.

Given the matrices of the state-space model `sys2`

```
A = [ -9.0201  17.7791
      -1.6943  3.2138 ];
B = [ -.5112  .5362
      -.002  -1.8470];
C = [ -3.2897  2.4544
      -13.5009  18.0745];
D = [-.5476  -.1410
    -.6459  .2958 ];
```

Define the three blocks as individual LTI models.

```
sys1 = tf(10,[1 5],'inputname','uc')
sys2 = ss(A,B,C,D,'inputname',{'u1' 'u2'},...
                   'outputname',{'y1' 'y2'})
sys3 = zpk(-1,-2,2)
```

Next append these blocks to form the unconnected model `sys`.

```
sys = append(sys1,sys2,sys3)
```

This produces the block-diagonal model

```
sys

a =
                       x1           x2           x3           x4
           x1          -5            0            0            0
           x2           0      -9.0201       17.779            0
           x3           0      -1.6943       3.2138            0
           x4           0            0            0           -2


b =
                       uc           u1           u2            ?
           x1           4            0            0            0
           x2           0      -0.5112       0.5362            0
           x3           0       -0.002       -1.847            0
           x4           0            0            0       1.4142


c =
                       x1           x2           x3           x4
            ?         2.5            0            0            0
           y1           0      -3.2897       2.4544            0
           y2           0      -13.501       18.075            0
```

```
              ?           0           0           0      -1.4142


d =
                         uc          u1          u2           ?
              ?           0           0           0           0
             y1           0     -0.5476      -0.141           0
             y2           0     -0.6459      0.2958           0
              ?           0           0           0           2

Continuous-time system.
```

Note that the ordering of the inputs and outputs is the same as the block ordering you chose. Unnamed inputs or outputs are denoted b.

To derive the overall block diagram model from `sys`, specify the interconnections and the external inputs and outputs. You need to connect outputs 1 and 4 into input 3 (`u2`), and output 3 (`y2`) into input 4. The interconnection matrix `Q` is therefore

```
Q = [3 1 -4
     4 3  0];
```

Note that the second row of `Q` has been padded with a trailing zero. The block diagram has two external inputs `uc` and `u1` (inputs 1 and 2 of `sys`), and two external outputs `y1` and `y2` (outputs 2 and 3 of `sys`). Accordingly, set `inputs` and `outputs` as follows.

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

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

```
sysc = connect(sys,Q,inputs,outputs)


a =
                         x1          x2          x3          x4
```

```
           x1          -5            0            0            0
           x2     0.84223     0.076636       5.6007      0.47644
           x3     -2.9012      -33.029       45.164      -1.6411
           x4     0.65708      -11.996        16.06      -1.6283


b =
                       uc           u1
           x1           4            0
           x2           0    -0.076001
           x3           0      -1.5011
           x4           0     -0.57391


c =
                       x1           x2           x3           x4
           y1    -0.22148      -5.6818       5.6568     -0.12529
           y2     0.46463      -8.4826       11.356      0.26283


d =
                       uc           u1
           y1           0     -0.66204
           y2           0     -0.40582

Continuous-time system.
```

Note that the inputs and outputs are as desired.

References

[1] Edwards, J.W., "A Fortran Program for the Analysis of Linear Continuous and Sampled-Data Systems," *NASA Report TM X56038*, Dryden Research Center, 1976.

See Also

`append`, `feedback`, `minreal`, `parallel`, `series`, `lft`

Purpose

Output and state covariance of system driven by white noise

Syntax

```
P = covar(sys,W)
P(:,:,i1,...iN)
Q(:,:,i1,...iN)
sys(:,:,i1,...iN)
```

Description

`covar` calculates the stationary covariance of the output y of an LTI model `sys` driven by Gaussian white noise inputs w. This function handles both continuous- and discrete-time cases.

`P = covar(sys,W)` returns the steady-state output response covariance

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

given the noise intensity

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

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

`[P,Q] = covar(sys,W)` also returns the steady-state state covariance

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

when `sys` is a state-space model (otherwise `Q` is set to `[]`).

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

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

Example

Compute the output response covariance of the discrete SISO system

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

due to Gaussian white noise of intensity `W = 5`. Type

```
sys = tf([2 1],[1 0.2 0.5],0.1);
p = covar(sys,5)
```

and MATLAB returns

```
p =
    30.3167
```

You can compare this output of `covar` to simulation results.

```
randn('seed',0)
w = sqrt(5)*randn(1,1000);  % 1000 samples

% Simulate response to w with LSIM:
y = lsim(sys,w);

% Compute covariance of y values
psim = sum(y .* y)/length(w);
```

This yields

```
psim =
    32.6269
```

The two covariance values `p` and `psim` do not agree perfectly due to the finite simulation horizon.

Algorithm

Transfer functions and zero-pole-gain models are first converted to state space with `ss`.

For continuous-time state-space models

$$\begin{aligned} \dot{x} &= Ax + Bw \\ y &= Cx + Dw \end{aligned}$$

Q is obtained by solving the Lyapunov equation

$$AQ + QA^T + BWB^T = 0$$

The output response covariance P is finite only when $D = 0$ and then $P = CQC^T$.

In discrete time, the state covariance solves the discrete Lyapunov equation

$$AQA^T - Q + BWB^T = 0$$

and P is given by $P = CQC^T + DWD^T$

Note that P is well defined for nonzero D in the discrete case.

Limitations

The state and output covariances are defined for *stable* systems only. For continuous systems, the output response covariance P is finite only when the D matrix is zero (strictly proper system).

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

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

where Co has n rows and nm columns.

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

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

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

Example

Check if the system with the following data

```
A =
     1     1
     4    -2

B =
     1    -1
     1    -1
```

is controllable. Type

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

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

and MATLAB returns

```
unco =
     1
```

Limitations

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

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

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

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

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

See Also

`ctrbf`, `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

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

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

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

where (A_c, B_c) is controllable, all eigenvalues of A_{uc} are uncontrollable, and

$$C_c(sI - A_c)^{-1}B_c = C(sI - A)^{-1}B.$$

`[Abar,Bbar,Cbar,T,k] = ctrbf(A,B,C)` decomposes the state-space system represented by `A`, `B`, and `C` into the controllability staircase form, `Abar`, `Bbar`, and `Cbar`, described above. `T` is the similarity transformation matrix and `k` is a vector of length n, where n is the order of the system represented by `A`. Each entry of `k` represents the number of controllable states factored out during each step of the transformation matrix calculation. The number of nonzero elements in `k` indicates how many iterations were necessary to calculate `T`, and `sum(k)` is the number of states in A_c, the controllable portion of `Abar`.

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

Example

Compute the controllability staircase form for

```
A =
     1     1
     4    -2

B =
     1    -1
     1    -1

C =
     1     0
     0     1
```

and locate the uncontrollable mode.

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

Abar =
   -3.0000         0
   -3.0000    2.0000

Bbar =
    0.0000    0.0000
    1.4142   -1.4142

Cbar =
   -0.7071    0.7071
    0.7071    0.7071

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

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

Algorithm

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

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 preferences for the Control System Toolbox. 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.

See Also `sisotool`, `ltiview`

Purpose

Convert from discrete- to continuous-time models

Syntax

Description

`d2c` converts LTI models from discrete to continuous time using one of the following conversion methods:

`'zoh'`	Zero-order hold on the inputs. The control inputs are assumed piecewise constant over the sampling period.
`'tustin'`	Bilinear (Tustin) approximation to the derivative.
`'prewarp'`	Tustin approximation with frequency prewarping.
`'matched'`	Matched pole-zero method of [1] (for SISO systems only).

The string *`method`* specifies the conversion method. If *`method`* is omitted, zero-order hold (`'zoh'`) is assumed. See Continuous/Discrete Conversions of LTI Models for more details on the conversion methods.

Example

Consider the discrete-time model with transfer function

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

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

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

Discretizing the resulting model `Hc` with the zero-order hold method (this is the default method) and sampling period $T_s = 0.1$ gives back the original discrete model $H(z)$. To see this, type

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

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

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

As with zero-order hold, the inverse discretization operation

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

gives back the original $H(z)$.

Algorithm

The `'zoh'` conversion is performed in state space and relies on the matrix logarithm (see `logm` in the MATLAB documentation).

Limitations

The Tustin approximation is not defined for systems with poles at $z = -1$ and is ill-conditioned for systems with poles near $z = -1$.

The zero-order hold method cannot handle systems with poles at $z = 0$. In addition, the `'zoh'` conversion increases the model order for systems with negative real poles, [2]. This is necessary because the matrix logarithm maps real negative poles to complex poles. As a result, a discrete model with a single pole at $z = -0.5$ would be transformed to a continuous model with a single *complex* pole at $\log(-0.5) \approx -0.6931 + j\pi$. Such a model is not meaningful because of its complex time response.

To ensure that all complex poles of the continuous model come in conjugate pairs, `d2c` replaces negative real poles $z = -\alpha$ with a pair of complex conjugate poles near $-\alpha$. The conversion then yields a continuous model with higher order. For example, the discrete model with transfer function

$$H(z) = \frac{z+0.2}{(z+0.5)(z^2+z+0.4)}$$

and sample time 0.1 second is converted by typing

```
Ts = 0.1
H = zpk(-0.2,-0.5,1,Ts) * tf(1,[1 1 0.4],Ts)
```

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

MATLAB responds with

```
Warning: System order was increased to handle real negative poles.

Zero/pole/gain:
  -33.6556 (s-6.273) (s^2 + 28.29s + 1041)
------------------------------------------
(s^2 + 9.163s + 637.3) (s^2 + 13.86s + 1035)
```

Convert `Hc` back to discrete time by typing

```
c2d(Hc,Ts)
```

yielding

```
Zero/pole/gain:
     (z+0.5) (z+0.2)
-------------------------
(z+0.5)^2 (z^2 + z + 0.4)

Sampling time: 0.1
```

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

References

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

[2] Kollár, I., G.F. Franklin, and R. Pintelon, "On the Equivalence of z-domain and s-domain Models in System Identification," *Proceedings of the IEEE Instrumentation and Measurement Technology Conference,* Brussels, Belgium, June, 1996, Vol. 1, pp. 14-19.

See Also

`c2d`, `d2d`, `logm`

Purpose

Resample discrete-time LTI model or add input delay

Syntax

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

Description

`sys1 = d2d(sys,Ts,method)` resamples the discrete-time LTI model `sys` to produce an equivalent discrete-time model `sys1` with the new sample time `Ts` (in seconds). The string `method` specifies the resampling method among the following:

- `'zoh'` — Zero-order hold on the inputs
- `'tustin'` — Bilinear (Tustin) approximation
- `'prewarp'` — Tustin approximation with frequency warping. Specify the critical frequency `Wc` (in rad/s) as a fourth input by

  ```
  sys = d2d(sys,Ts,'prewarp',Wc)
  ```

The default is `'zoh'` when `method` is omitted.

Example

Consider the zero-pole-gain model

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

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

```
H = zpk(0.7,0.5,1,0.1)
H2 = d2d(H,0.05)
Zero/pole/gain:
(z-0.8243)
----------
(z-0.7071)

Sampling time: 0.05
```

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

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

Sampling time: 0.1
```

See Also

`c2d`, `d2c`, ltiexamples

damp

Purpose Natural frequency and damping of system poles

Syntax

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

Description `damp` calculates the damping factor and natural frequencies of the poles of an LTI model `sys`. When invoked without lefthand arguments, a table of the eigenvalues in increasing frequency, along with their damping factors and natural frequencies, is displayed on the screen.

`[Wn,Z] = damp(sys)` returns column vectors `Wn` and `Z` containing the natural frequencies ω_n and damping factors ζ of the poles of `sys`. For discrete-time systems with poles z and sample time T_s, `damp` computes "equivalent" continuous-time poles s by solving

$$z = e^{sT_s}$$

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

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

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

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

Type

```
H = tf([2 5 1],[1 2 3])
Transfer function:
2 s^2 + 5 s + 1
---------------
 s^2 + 2 s + 3
```

Type

```
damp(H)
```

and MATLAB returns

```
Eigenvalue             Damping      Freq. (rad/s)

-1.00e+000 + 1.41e+000i     5.77e-001      1.73e+000
-1.00e+000 - 1.41e+000i     5.77e-001      1.73e+000
```

See Also `eig`, `esort`, `dsort`, `pole`, `pzmap`, `zero`

dare

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

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

The `dare` function also returns the gain matrix, $G = (B^TXB + R)^{-1}B^TXA$, and the vector `L` of closed loop eigenvalues, where

```
L=eig(A-B*G,E)
```

`[X,L,G] = dare(A,B,Q,R,S,E)` solves the more general discrete-time algebraic Riccati equation,

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

or, equivalently, if `R` is nonsingular,

$$E^TXE = F^TXF + -F^TXB(B^TXB + R)^{-1}B^TXF + Q - SR^{-1}S^T$$

where $F = A - BR^{-1}S$. 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$G = (B^TXB + R)^{-1}(B^TXA + S^T)$

and a vector `L` of closed-loop eigenvalues, where

```
L= 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.

Algorithm

`dare` implements the algorithms described in [1]. It uses the QZ algorithm to deflate the extended symplectic pencil and compute its stable invariant subspace.

Limitations

The (A, B) pair must be stabilizable (that is, all eigenvalues of A outside the unit disk must be controllable). In addition, the associated symplectic pencil must have no eigenvalue on the unit circle. Sufficient conditions for this to hold are (Q, A) detectable when $S = 0$ and $R > 0$, or

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

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`

dcgain

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

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

Remark

The DC gain is infinite for systems with integrators.

Example

To compute the DC gain of the MIMO transfer function

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

type

```
H = [1 tf([1 -1],[1 1 3]) ; tf(1,[1 1]) tf([1 2],[1 -3])]
dcgain(H)
ans =
    1.0000   -0.3333
    1.0000   -0.6667
```

See Also `evalfr`, `norm`

delay2z

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

Syntax

`sys = delay2z(sys)`

Description

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

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

Example

```
z=tf('z',-1);
sys=(-.4*z -.1)/(z^2 + 1.05*z + .08)
Transfer function:
-0.4 z - 0.1
-------------------
z^2 + 1.05 z + 0.08
Sampling time: unspecified
sys.InputDelay = 1;
sys = delay2z(sys)
Transfer function:
     -0.4 z - 0.1
-----------------------
z^3 + 1.05 z^2 + 0.08 z
Sampling time: unspecified
```

See Also

`hasdelay`, `pade`, `totaldelay`

Purpose Create state-space models with delayed terms

Syntax

```
sys=delayss(A,B,C,D,delayterms)
sys=delayss(A,B,C,D,ts,delayterms)
```

Description `sys=delayss(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,..,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 `tj`, `Aj`, `Bj`, `Cj`, and `Dj`, respectively. The resulting model `sys` is a state-space (`SS`) model with internal delays.

`sys=delayss(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 Nj, j=1,..,N are time delays expressed as integer multiples of the sampling period `ts`.

Example

To create the model:

$$\frac{dx}{dt} = x(t) - x(t-1.2) + 2u(t-0.5)$$

$$y(t) = x(t-0.5) + u(t)$$

type

```
DelayT(1) = struct('delay',0.5,'a',0,'b',2,'c',1,'d',0);
DelayT(2) = struct('delay',1.2,'a',-1,'b',0,'c',0,'d',0);
sys = delayss(1,0,0,1,DelayT)

a =
       x1
   x1   0

b =
       u1
   x1   2

c =
       x1
   y1   1

d =
       u1
   y1   1

(a,b,c,d values when setting all internal delays to zero)

Internal delays: 0.5  0.5  1.2

Continuous-time model.
```

See Also

getdelaymodel, `ltiprops`, `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

$$J(u) = \sum_{n=1}^{\infty} (x[n]^T Q x[n] + u[n]^T R u[n] + 2x[n]^T N u[n])$$

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 = 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`

Purpose

Solve discrete-time Lyapunov equations

Syntax

```
X = dlyap(A,Q)
X = dlyap(A,B,C)
X = dlyap(A,Q,[],E)
```

Description

`X = dlyap(A,Q)` solves the discrete-time Lyapunov equation

$$AXA^T - X + Q = 0$$

where A and Q are n-by-n matrices.

The solution X is symmetric when Q is symmetric, and positive definite when Q is positive definite and A has all its eigenvalues inside the unit disk.

`X = dlyap(A,B,C)` solves the Sylvester equation $AXB^T - X + C = 0$

where A, B, and C must have compatible dimensions but need not be square.

`X = dlyap(A,Q,[],E)` solves the generalized discrete-time Lyapunov equation $AXA^T - EXE^T + Q = 0$

where Q is a symmetric matrix. The empty square brackets, `[]`, are mandatory. If you place any values inside them, the function will error out.

Algorithm

`dlyap` uses SLICOT routines SB03MD and SG03AD for Lyapunov equations and SB04QD (SLICOT) for Sylvester equations.

Diagnostics

The discrete-time Lyapunov equation has a (unique) solution if the eigenvalues $\alpha_1, \alpha_2, \ldots, \alpha_n$ of A satisfy $\alpha_i\alpha_j \neq 1$ for all (i, j).

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

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

See Also `covar`, `lyap`

Purpose Square-root solver for continuous-time Lyapunov equations

Syntax

```
R = dlyapchol(A,B)
X = dlyapchol(A,B,E)
```

Description `R = dlyapchol(A,B)` computes a Cholesky factorization `X = R'*R` of the solution `X` to the Lyapunov matrix equation:

```
A*X*A'- X + B*B' = 0
```

All eigenvalues of `A` matrix must lie in the open unit disk for `R` to exist.

`X = dlyapchol(A,B,E)` computes a Cholesky factorization `X = R'*R` of `X` solving the Sylvester equation

```
A*X*A' - E*X*E' + B*B' = 0
```

All generalized eigenvalues of (`A`,`E`) must lie in the open unit disk for `R` to exist.

Algorithm `dlyapchol` uses SLICOT routines SB03OD and SG03BD.

See Also `dlyap`, `lyapchol`

drss

Purpose

Generate stable random discrete test model

Syntax

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

Description

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

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

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

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

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

Example

Generate a random discrete LTI system with three states, two inputs, and two outputs.

```
sys = drss(3,2,2)

a =
                        x1           x2           x3
           x1      0.38630     -0.21458     -0.09914
           x2     -0.23390     -0.15220     -0.06572
           x3     -0.03412      0.11394     -0.22618

b =
                        u1           u2
           x1      0.98833      0.51551
           x2            0      0.33395
```

```
          x3      0.42350      0.43291

c =
                       x1           x2           x3
          y1      0.22595      0.76037            0
          y2            0            0            0

d =
                       u1           u2
          y1            0      0.68085
          y2      0.78333      0.46110

Sampling time: unspecified
Discrete-time system.
```

See Also `rss`, `tf`, `zpk`

dsort

Purpose

Sort discrete-time poles by magnitude

Syntax

```
dsort
[s,ndx] = dsort(p)
```

Description

`dsort` sorts the discrete-time poles contained in the vector `p` in descending order by magnitude. Unstable poles appear first.

When called with one lefthand argument, `dsort` returns the sorted poles in `s`.

`[s,ndx] = dsort(p)` also returns the vector `ndx` containing the indices used in the sort.

Example

Sort the following discrete poles.

```
p =
  -0.2410 + 0.5573i
  -0.2410 - 0.5573i
   0.1503
  -0.0972
  -0.2590

s = dsort(p)

s =
  -0.2410 + 0.5573i
  -0.2410 - 0.5573i
  -0.2590
   0.1503
  -0.0972
```

Limitations

The poles in the vector `p` must appear in complex conjugate pairs.

See Also

`eig`, `esort`, `sort`, `pole`, `pzmap`, `zero`

Purpose

Specify descriptor state-space models

Syntax

```
sys = dss(a,b,c,d,e)
sys = dss(a,b,c,d,e,Ts)
sys
```

Description

`sys = dss(a,b,c,d,e)` creates the continuous-time descriptor state-space model

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

The output `sys` is an SS model storing the model data (see LTI Objects). Note that `ss` produces the same type of object. If the matrix $D = 0$, you can simply set `d` to the scalar `0` (zero).

`sys = dss(a,b,c,d,e,Ts)` creates the discrete-time descriptor model

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

with sample time `Ts` (in seconds).

`sys = dss(a,b,c,d,e,ltisys)` creates a descriptor model with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See LTI Properties for an overview of generic LTI properties.

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

```
'Property',Value
```

Each pair specifies a particular LTI property of the model, for example, the input names or some notes on the model history. See `set` and the example below for details.

Example

The command

```
sys = dss(1,2,3,4,5,'td',0.1,'inputname','voltage',...
                                             'notes','Just an example')
```

creates the model

$$\begin{aligned} 5\dot{x} &= x + 2u \\ y &= 3x + 4u \end{aligned}$$

with a 0.1 second input delay. The input is labeled `'voltage'`, and a note is attached to tell you that this is just an example.

See Also

`dssdata`, `get`, `set`, `ss`

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` is equivalent to `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 system with delay-free dynamics).

`[A,B,C,D,E,Ts] = dssdata(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.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`, `ltimodels`, `ltiprops`, `ssdata`

esort

Purpose Sort continuous-time poles by real part

Syntax

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

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

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

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

Example Sort the following continuous eigenvalues.

```
p
p =
  -0.2410+ 0.5573i
  -0.2410- 0.5573i
   0.1503
  -0.0972
  -0.2590

esort(p)

ans =
   0.1503
  -0.0972
  -0.2410+ 0.5573i
  -0.2410- 0.5573i
  -0.2590
```

Limitations The eigenvalues in the vector `p` must appear in complex conjugate pairs.

See Also `dsort`, `sort`, `eig`, `pole`, `pzmap`, `zero`

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

$$\begin{aligned}\dot{x} &= Ax + Bw \\ y &= Cx + Dw\end{aligned}$$

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

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

The discrete-time estimator has similar equations.

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

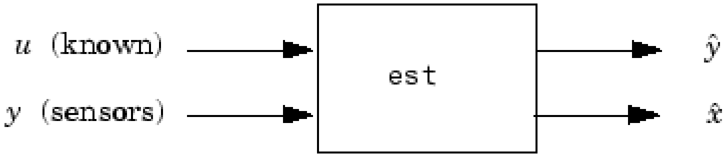
$$\dot{x} = Ax + B_1 w + B_2 u$$

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

The index vectors `sensors` and `known` specify which outputs y are measured and which inputs u are known. The resulting estimator `est` uses both u and y to produce the output and state estimates.

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

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



`estim` handles both continuous- and discrete-time cases. You can use the functions `place` (pole placement) or `kalman` (Kalman filtering) to design an adequate estimator gain L. Note that the estimator poles (eigenvalues of $A - LC$) should be faster than the plant dynamics (eigenvalues of A) to ensure accurate estimation.

Example

Consider a state-space model `sys` with seven outputs and four inputs. Suppose you designed a Kalman gain matrix L using outputs 4, 7, and 1 of the plant as sensor measurements, and inputs 1,4, and 3 of the plant as known (deterministic) inputs. You can then form the Kalman estimator by

```
sensors = [4,7,1];
known = [1,4,3];
est = estim(sys,L,sensors,known)
```

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

See Also

`kalman`, `place`, `reg`

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.

Example To evaluate the discrete-time transfer function

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

at $z = 1 + j$, type

```
H = tf([1 -1],[1 1 1],-1)
z = 1+j
evalfr(H,z)
ans =
  2.3077e-01 +  1.5385e-01i
```

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

See Also `bode`, `freqresp`, `sigma`

lti/exp

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`

Purpose

Concatenate FRD models along frequency dimension

Syntax

```
sys = fcat(sys1,sys2,...)
```

Description

`sys = fcat(sys1,sys2,...)` takes two or more FRD models and merges their frequency responses into a single FRD model `sys`. The frequency vectors of `sys1, sys2,...` should not intersect and are merged together. The resulting frequency vector is sorted by increasing frequency.

See Also

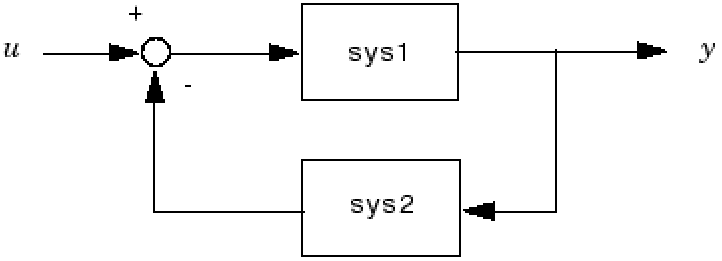
`fselect`, `interp`, `frd`

Purpose Feedback connection of two LTI models

Syntax

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

Description `sys = feedback(sys1,sys2)` returns an LTI model `sys` for the negative feedback interconnection.



The closed-loop model `sys` has u as input vector and y as output vector. The LTI models `sys1` and `sys2` must be both continuous or both discrete with identical sample times. Precedence rules are used to determine the resulting model type (see Precedence Rules).

To apply positive feedback, use the syntax

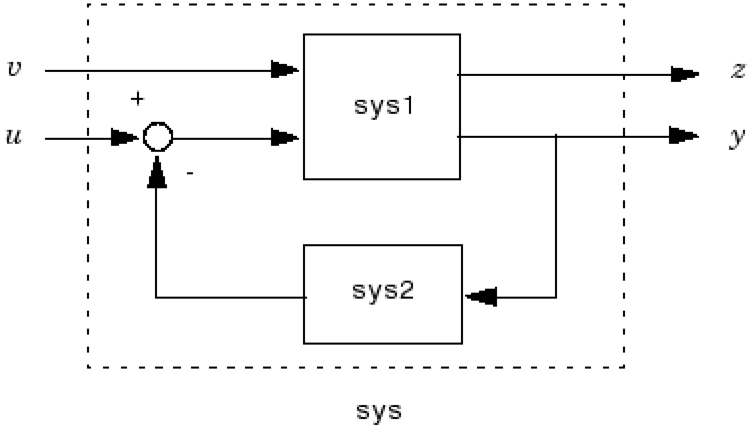
```
sys = feedback(sys1,sys2,+1)
```

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

Finally,

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

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



The vector `feedin` contains indices into the input vector of `sys1` and specifies which inputs u are involved in the feedback loop. Similarly, `feedout` specifies which outputs y of `sys1` are used for feedback. The resulting LTI model `sys` has the same inputs and outputs as `sys1` (with their order preserved). As before, negative feedback is applied by default and you must use

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

to apply positive feedback.

For more complicated feedback structures, use `append` and `connect`.

Remark

You can specify static gains as regular matrices, for example,

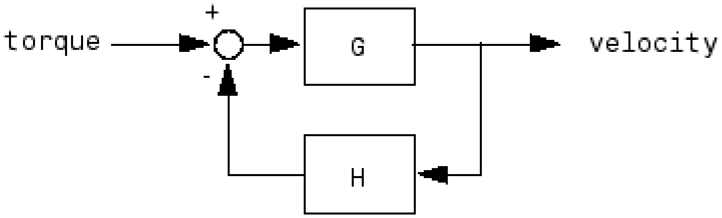
```
sys = feedback(sys1,2)
```

However, at least one of the two arguments `sys1` and `sys2` should be an LTI object. For feedback loops involving two static gains `k1` and `k2`, use the syntax

```
sys = feedback(tf(k1),k2)
```

Examples

Example 1



To connect the plant

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

with the controller

$$H(s) = \frac{5(s+2)}{s+10}$$

using negative feedback, type

```
G = tf([2 5 1],[1 2 3],'inputname','torque',...
                        'outputname','velocity');
H = zpk(-2,-10,5)
Cloop = feedback(G,H)
```

and MATLAB returns

```
Zero/pole/gain from input "torque" to output "velocity":
0.18182 (s+10) (s+2.281) (s+0.2192)
-----------------------------------
 (s+3.419) (s^2 + 1.763s + 1.064)
```

The result is a zero-pole-gain model as expected from the precedence rules. Note that `Cloop` inherited the input and output names from `G`.

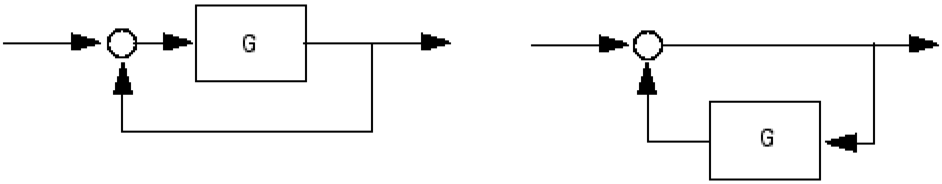
Example 2

Consider a state-space plant `P` with five inputs and four outputs and a state-space feedback controller `K` with three inputs and two outputs. To connect outputs 1, 3, and 4 of the plant to the controller inputs, and the controller outputs to inputs 4 and 2 of the plant, use

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

Example 3

You can form the following negative-feedback loops



by

```
Cloop = feedback(G,1)        % left diagram
Cloop = feedback(1,G)b       % 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 LTI property of the model, for example, the input names or the transfer function variable. See LTI Properties and the `set` entry for additional information on LTI properties and admissible property values.

Arguments

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

MIMO transfer functions are regarded as arrays of SISO transfer functions (one per I/O channel), each of which is characterized by its numerator and denominator. The input arguments `num` and `den` are then cell arrays of row vectors such that:

- `num` and `den` have as many rows as outputs and as many columns as inputs.
- Their (i, j) entries `num{i,j}` and `den{i,j}` specify the numerator and denominator of the transfer function from input `j` to output `i`.

If all SISO entries have the same denominator, you can also set `den` to the row vector representation of this common denominator. See also MIMO Transfer Function Models for alternative ways to specify MIMO transfer functions.

Remark

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

Example

Typing the commands

```
num = {1 , [1 0.3]}
den = {[1 1 2] ,[5 2]}
H = filt(num,den,'inputname',{'channel1' 'channel2'})
```

creates the two-input digital filter

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

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

See Also `tf`, `zpk`, `ss`

Purpose Pointwise peak gain of FRD model

Syntax

```
fnrm = fnorm(sys)
fnrm = fnorm(sys,ntype)
```

Description `fnrm = fnorm(sys)` computes the pointwise 2-norm of the frequency response contained in the FRD model `sys`, that is, the peak gain at each frequency point. The output `fnrm` is an FRD object containing the peak gain across frequencies.

`fnrm = fnorm(sys,ntype)` computes the frequency response gains using the matrix norm specified by `ntype`. See `norm` for valid matrix norms and corresponding NTYPE values.

See Also `lti/norm`, `frd/abs`

Purpose

Create or convert to frequency-response data models

Syntax

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

Description

`sys = frd(response,frequency)` creates an FRD model `sys` from the frequency response data stored in the multidimensional array `response`. The vector `frequency` represents the underlying frequencies for the frequency response data. See Data Format for the Argument Response in FRD Models on page 2-95 for a list of response data formats.

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

`sys = frd` creates an empty FRD model.

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

```
'PropertyName',PropertyValue
```

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

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

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

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

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

Arguments

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

Data Format for the Argument Response in FRD Models

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

Remarks

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

Example

Type the commands

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

to create a SISO FRD model.

See Also `chgunits`, `frdata`, `set`, `ss`, `tf`, `zpk`

Purpose

Access data for frequency response data (FRD) object

Syntax

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

Description

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

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

See Data Format for the Argument Response in FRD Models on page 2-95 for more information on the data format for FRD response data.

For SISO FRD models, the syntax

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

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

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

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

Arguments

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

Example

Typing the commands

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

returns the FRD model data

```
resp =
 0.2040 + 0.4565i
   2.4359 - 4.3665i
freq =
    10
   100
```

See Also `frd`, `get`, `set`

Purpose

Frequency response over frequency grid

Syntax

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

Description

`H = freqresp(sys,w)` computes the frequency response of the LTI model `sys` at the real frequency points specified by the vector `w`. `sys` can be a TF, SS, ZPK, or FRD object. The frequencies must be in rad/s. For single LTI Models, `freqresp(sys,w)` returns a 3-D array `H` with the frequency as the last dimension (see "Arguments" below). For LTI arrays of size `[Ny Nu S1 ... Sn]`, `freqresp(sys,w)` returns a [`Ny`-by-`Nu`-by-`S1`-by-...-by-`Sn`] length (w) array.

In continuous time, the response at a frequency ω is the transfer function value at $s = j\omega$. For state-space models, this value is given by

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

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

where T_s is the sample time. The transfer function is then evaluated at the resulting z values. The default $T_s = 1$ is used for models with unspecified sample time.

Remark

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

Arguments

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

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

For SISO systems, `H(1,1,k)` gives the scalar response at the frequency `w(k)`. For MIMO systems, the frequency response at `w(k)` is `H(:,:,k)`, a matrix with as many rows as outputs and as many columns as inputs.

Example

Compute the frequency response of

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

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

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

        0                0.5000- 0.5000i
  -0.2000+ 0.6000i   1.0000


H(:,:,2) =

        0                0.0099- 0.0990i
   0.9423+ 0.2885i   1.0000


H(:,:,3) =

        0                0.0001- 0.0100i
   0.9994+ 0.0300i   1.0000
```

The three displayed matrices are the values of $P(j\omega)$ for
$\omega = 1,\qquad \omega = 10,\qquad \omega = 100$

The third index in the 3-D array `H` is relative to the frequency vector `w`, so you can extract the frequency response at $\omega = 10$ rad/sec by

```
H(:,:,w==10)

ans =
```

```
        0              0.0099- 0.0990i
   0.9423+ 0.2885i   1.0000
```

Algorithm

For transfer functions or zero-pole-gain models, `freqresp` evaluates the numerator(s) and denominator(s) at the specified frequency points. For continuous-time state-space models (A, B, C, D), the frequency response is

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

For efficiency, A is reduced to upper Hessenberg form and the linear equation $(j\omega - A)X = B$ is solved at each frequency point, taking advantage of the Hessenberg structure. The reduction to Hessenberg form provides a good compromise between efficiency and reliability. See [1] for more details on this technique.

Diagnostics

If the system has a pole on the $j\omega$ axis (or unit circle in the discrete-time case) and `w` happens to contain this frequency point, the gain is infinite, $j\omega I - A$ is singular, and `freqresp` produces the following warning message.

```
Singularity in freq. response due to jw-axis or unit circle pole.
```

References

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

See Also

`evalfr`, `bode`, `nyquist`, `nichols`, `sigma`, `ltiview`, `interp`

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.

`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`, `fcat`, `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 jw-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`

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`

Purpose

Generate test input signals for `lsim`

Syntax

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

Description

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

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

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

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

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

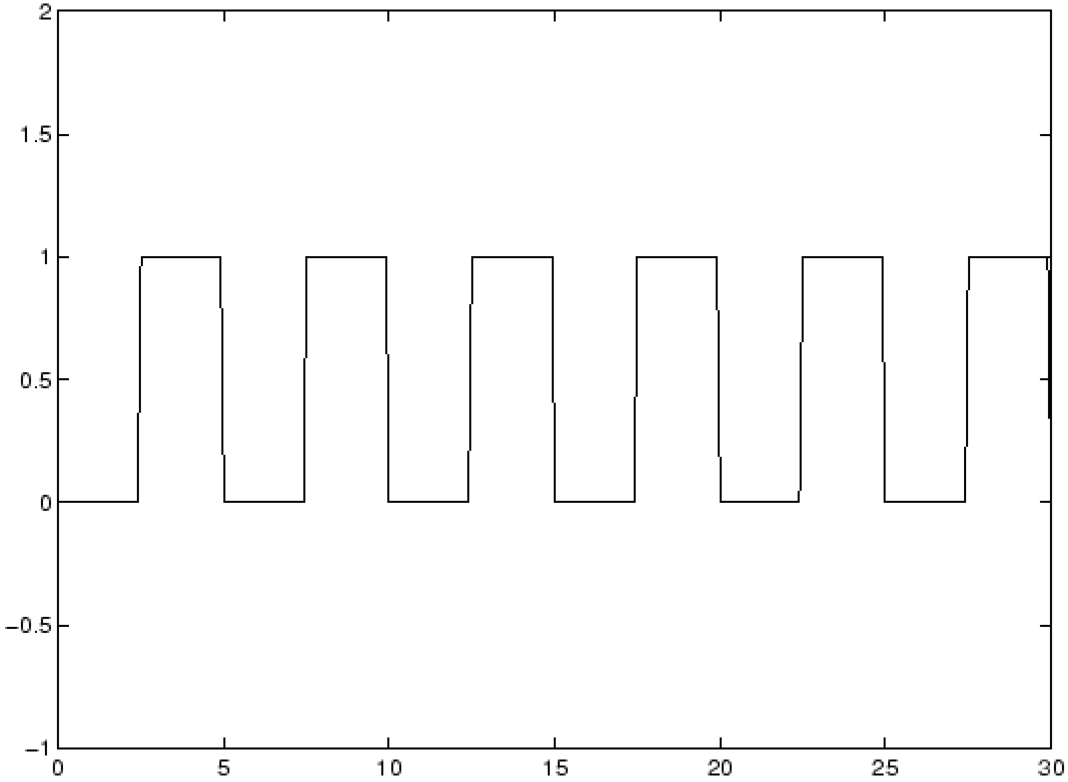
Example

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

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

Plot the resulting signal.

```
plot(t,u)
axis([0 30 -1 2])
```



See Also `lsim`

Purpose

Access LTI 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 LTI model `sys`. The string `'PropertyName'` can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`). You can specify any generic LTI property, or any property specific to the model `sys` (see LTI Properties for details on generic and model-specific LTI properties).

`Struct = get(sys)` converts the TF, SS, or ZPK object `sys` into a standard MATLAB structure with the property names as field names and the property values as field values.

Without left-side argument,

```
get(sys)
```

displays all properties of `sys` and their values.

Example

Consider the discrete-time SISO transfer function defined by

```
h = tf(1,[1 2],0.1,'inputname','voltage','user','hello')
```

You can display all LTI properties of `h` with

```
get(h)
            num: {[0 1]}
            den: {[1 2]}
        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]`.

Remark

An alternative to the syntax

```
Value = get(sys,'PropertyName')
```

is the structure-like referencing

```
Value = sys.PropertyName
```

For example,

```
sys.Ts
sys.a
sys.user
```

return the values of the sample time, A matrix, and `UserData` property of the (state-space) model `sys`.

See Also `frdata`, `set`, `ssdata`, `tfdata`, `zpkdata`

getdelaymodel

Purpose State-space representation of internal delays

Syntax

```
[[A,B1,B2,C1,C2,D11,D12,D21,D22,E,tau] = getdelaymodel(sys,
   'mat')
[H,tau] = getdelaymodel(sys,'lft')
```

Description

`[[A,B1,B2,C1,C2,D11,D12,D21,D22,E,tau] = getdelaymodel(sys,'mat')` returns the matrices `A`,`B1`,`B2`, etc. and vector `tau` of internal delays for the state-space model `sys` . The `E` matrix is set to `[]` for explicit models with no `E` matrix.

State-space models with internal delays are represented by differential-algebraic equations of the form:

$E\ dx/dt = A\ x + B1\ u + B2\ w$

$y = C1\ x + D11\ u + D12\ w$

$z = C2\ x + D21\ u + D22\ w$

$w(t) = z(t - tau)$

or their discrete-time counterparts:

$E\ x[k+1] = A\ x[k] + B1\ u[k] + B2\ w[k]$

$y[k] = C1\ x[k] + D11\ u[k] + D12\ w[k]$

$z[k] = C2\ x[k] + D21\ u[k] + D22\ w[k]$

$w[k] = z[k - tau]$

where u,y are the external inputs and outputs, and tau is the vector of internal delays. These equations correspond to this block diagram:

where H(s) is the delay-free state-space model mapping [u;w] to [y;z].

`[H,tau] = getdelaymodel(sys,'lft')` returns the state-space model `H` and vector `tau` of internal delays making up the block diagram above.

Note that for models without internal delays:

- Only `A`,`B1`,`C1`,`D11` (and possibly `E`) are non-empty
- `tau` is empty and `H` is equal to `sys`.

See Also

`delayss`, `dss`, `ss`, `setdelaymodel`

getoptions

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.

See Also `setoptions`

Purpose

Controllability and observability grammians

Syntax

`gram`

Description

`gram` calculates controllability and observability grammians. You can use grammians to study the controllability and observability properties of state-space models and for model reduction [1] . They have better numerical properties than the controllability and observability matrices formed by `ctrb` and `obsv`.

Given the continuous-time state-space model

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

the controllability grammian is defined by

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

and the observability grammian by

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

The discrete-time counterparts are

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

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

Use the commands

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

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

Algorithm

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

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

or its discrete-time counterpart

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

Similarly, the observability grammian W_o solves the Lyapunov equation

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

in continuous time, and the Lyapunov equation

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

in discrete time.

Limitations

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

References

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

See Also

`balreal`, `ctrb`, `lyap`, `dlyap`, `obsv`

Purpose True for LTI model with time delays

Syntax `hasdelay(sys)`

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

See Also `delay2z`, `totaldelay`

hsvd

Purpose

Compute Hankel singular values of LTI model

Syntax

```
hsv = hsvd(sys)
hsvd(sys)
[hsv,baldata] = hsvd(sys)
```

Description

`hsv = hsvd(sys)` computes the Hankel singular values `hsv` of the LTI model `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`. Use

```
hsv = hsvd(sys,'AbsTol',ATOL,...
           'RelTol',RTOL,'Offset',ALPHA)
```

to specify additional options for the stable/unstable decomposition, see `STABSEP` for details. The default values are `ATOL=0`, `RTOL=1e-8`, and `ALPHA=1e-8`.

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

`[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.

Algorithm

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`.

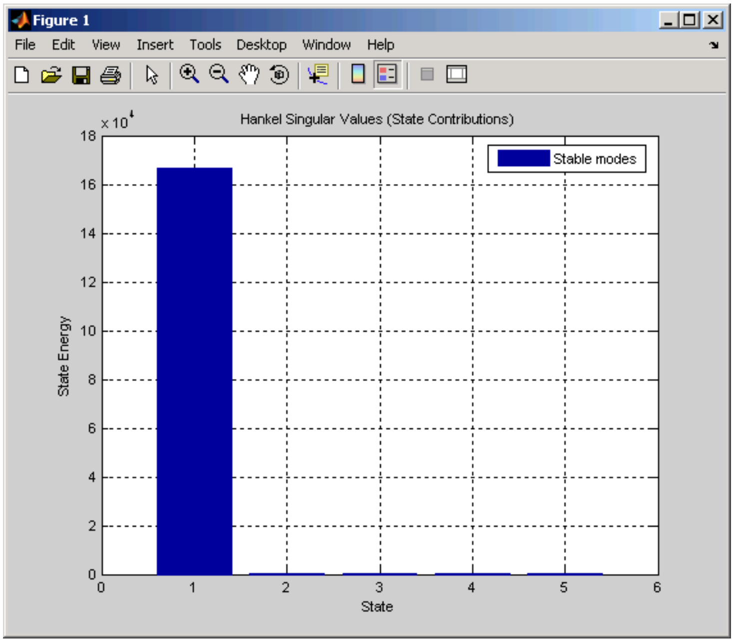
Example

This example illustrates the use of `offset`.

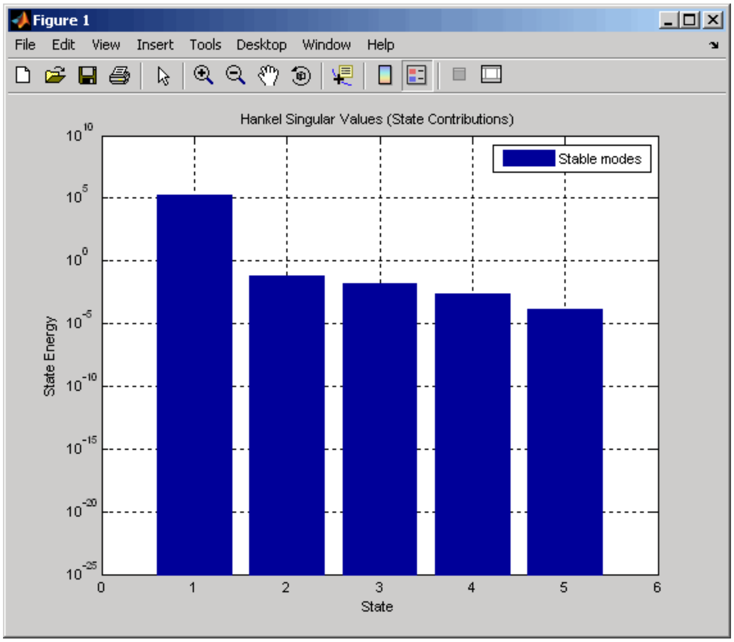
First, create a system with a stable pole very near to 0, then calculate the Hankel singular values.

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

Zero/pole/gain:
            (s-1) (s-2)
-----------------------------------
(s+1) (s+2) (s+3) (s+10) (s+1e-007)
```

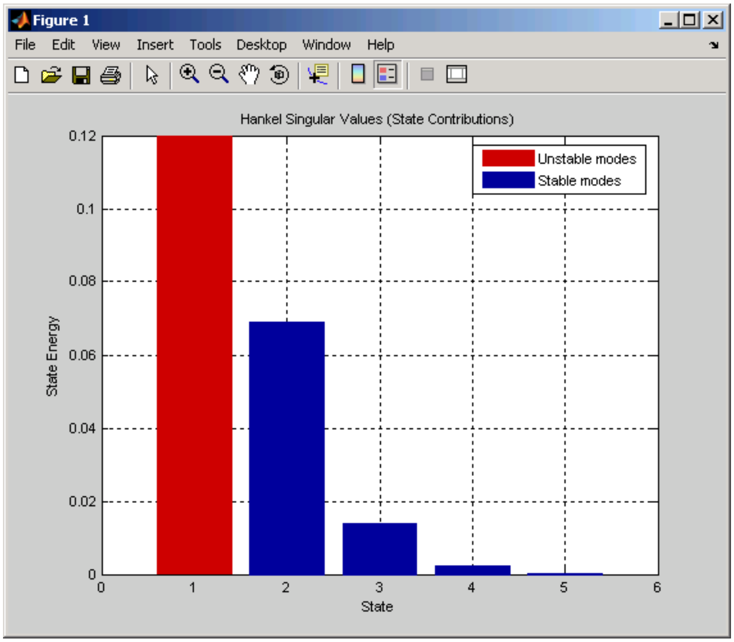


For a better view of the Hankel singular values, switch the plot to log scale by selecting **Y Scale > Log** from the right-click menu.



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)
```



The dominant Hankel singular value is now shown as unstable.

See Also

`balred`, `balreal`

hsvplot

Purpose

Plot Hankel singular values and return plot handle

Syntax

```
h = hsvplot(sys)
hsvplot(sys)
hsvplot(sys, AbsTol',ATOL,'RelTol',RTOL,'Offset',ALPHA)
hsvplot(AX,sys,...)
```

Description

`h = hsvplot(sys)` plots the Hankel singular values of an LTI system `sys` and returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help hsvoptions
```

for a list of available plot options.

`hsvplot(sys)` plots the Hankel singular values of the LTI model `sys`. See `hsvd` for details on the meaning and purpose of Hankel singular values. The Hankel singular values for the stable and unstable modes of `sys` are shown in blue and red, respectively.

`hsvplot(sys, AbsTol',ATOL,'RelTol',RTOL,'Offset',ALPHA)` specifies additional options for computing the Hankel singular values.

`hsvplot(AX,sys,...)` attaches the plot to the axes with handle `AX`.

Example

Use the plot handle to change plot options in the Hankel singular values plot.

```
sys = rss(20);
h = hsvplot(sys,'AbsTol',1e-6);
% Switch to log scale and modify Offset parameter
setoptions(h,'Yscale','log','Offset',0.3)
```

See Also

`getoptions`, `hsvd`, `setoptions`

Purpose

Imaginary part of FRD model

Syntax

```
imagfrd = imag(sys)
```

Description

`imagfrd = imag(sys)` computes the imaginary part of the frequency response contained in the FRD model `sys`, including the contribution of input, output, and I/O delays. The output `imagfrd` is an FRD object containing the values of the imaginary part across frequencies.

See Also

`frd/real`, `frd/abs`

impulse

Purpose Impulse response of LTI model

Syntax

```
impulse
impulse(sys)
impulse(sys,t)
```

Description

`impulse` calculates the unit impulse response of a linear system. The impulse response is the response to a Dirac input $\delta(t)$ for continuous-time systems and to a unit pulse at $t = 0$ for discrete-time systems. Zero initial state is assumed in the state-space case. When invoked without left-hand arguments, this function plots the impulse response on the screen.

`impulse(sys)` plots the impulse response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. The impulse response of multi-input systems is the collection of impulse responses for each input channel. The duration of simulation is determined automatically to display the transient behavior of the response.

`impulse(sys,t)` sets the simulation horizon explicitly. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form

```
t = 0:dt:Tfinal
```

For discrete systems, the spacing `dt` should match the sample period. For continuous systems, `dt` becomes the sample time of the discretized simulation model (see "Algorithm"), so make sure to choose `dt` small enough to capture transient phenomena.

To plot the impulse responses of several LTI models `sys1`,..., `sysN` on a single figure, use

```
impulse(sys1,sys2,...,sysN)
impulse(sys1,sys2,...,sysN,t)
```

As with `bode` or `plot`, you can specify a particular color, linestyle, and/or marker for each system, for example,

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

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

When invoked with left-side arguments,

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

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x` (for state-space models only). No plot is drawn on the screen. For single-input systems, `y` has as many rows as time samples (length of `t`), and as many columns as outputs. In the multi-input case, the impulse responses of each input channel are stacked up along the third dimension of `y`. The dimensions of `y` are then

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

and `y(:,:,j)` gives the response to an impulse disturbance entering the `j`th input channel. Similarly, the dimensions of `x` are

$$(\text{length of } \mathtt{t}) \times (\text{number of states}) \times (\text{number of inputs})$$

Example

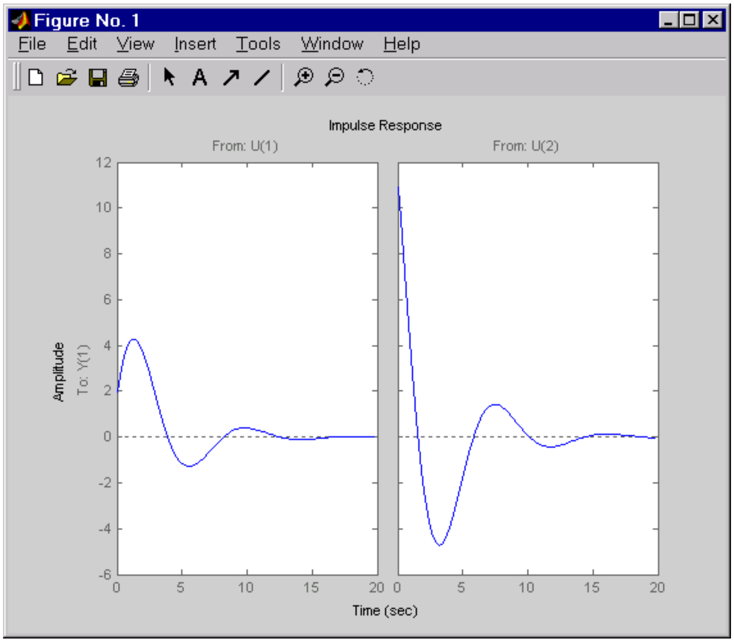
To plot the impulse response of the second-order state-space model

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

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

use the following commands.

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



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

You can store the impulse response data in MATLAB arrays by

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

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

```
size(y)
```

```
ans =
   101       1       2
```

(the first dimension is the length of `t`). The impulse response of the first input channel is then accessed by

```
y(:,:,1)
```

Algorithm

Continuous-time models are first converted to state space. The impulse response of a single-input state-space model

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

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

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

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`

impulseplot

Purpose

Plot impulse response and return plot handle

Syntax

```
h = impulseplot(sys)
impulseplot(sys)
impulseplot(sys,Tfinal)
impulseplot(sys,t)
impulseplot(sys1,sys2,...,t)
impulseplot(AX,...)
impulseplot(..., plotoptions)
```

Description

`h = impulseplot(sys)` plots the impulse response of the LTI model `sys` (created with either `tf`, `zpk`, or `ss`). For multiinput 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. `impulseplot` 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.

`impulseplot(sys)` plots the impulse response of the LTI model without returning the plot handle.

`impulseplot(sys,Tfinal)` simulates the impulse response from t=0 to the final time `t=Tfinal`. For discrete-time systems with unspecified sampling time, `Tfinal` is interpreted as the number of samples.

`impulseplot(sys,t)` uses the user-supplied time vector `t` for simulation. 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. The impulse is always assumed to arise at `t=0` (regardless of `Ti`).

`impulseplot(sys1,sys2,...,t)` plots the impulse response of multiple LTI models `sys1,sys2,...` on a single plot. The time vector `t` is optional. You can also specify a color, line style, and marker for each system, as in

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

`impulseplot(AX,...)` plots into the axes with handle `AX`.

`impulseplot(..., plotoptions)` plots the impulse response with the options specified in `plotoptions`. Type

```
help timeoptions
```

for more detail.

Example

Normalize the impulse response of a third-order system.

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

See Also

`getoptions`, `impulse`, `setoptions`

initial

Purpose initial condition response of state-space model

Syntax

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

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

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

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

`initial(sys,x0)` plots the response of `sys` to an initial condition `x0` on the states. `sys` can be any *state-space* model (continuous or discrete, SISO or MIMO, with or without inputs). The duration of simulation is determined automatically to reflect adequately the response transients.

`initial(sys,x0,t)` explicitly sets the simulation horizon. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form

```
t = 0:dt:Tfinal
```

For discrete systems, the spacing `dt` should match the sample period. For continuous systems, `dt` becomes the sample time of the discretized simulation model (see `impulse`), so make sure to choose `dt` small enough to capture transient phenomena.

To plot the initial condition responses of several LTI models on a single figure, use

```
initial(sys1,sys2,...,sysN,x0)
initial(sys1,sys2,...,sysN,x0,t)
```

(see `impulse` for details).

When invoked with left-side arguments,

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

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x`. No plot is drawn on the screen. The array `y` has as many rows as time samples (length of `t`) and as many columns as outputs. Similarly, `x` has `length(t)` rows and as many columns as states.

Example

Plot the response of the state-space model

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

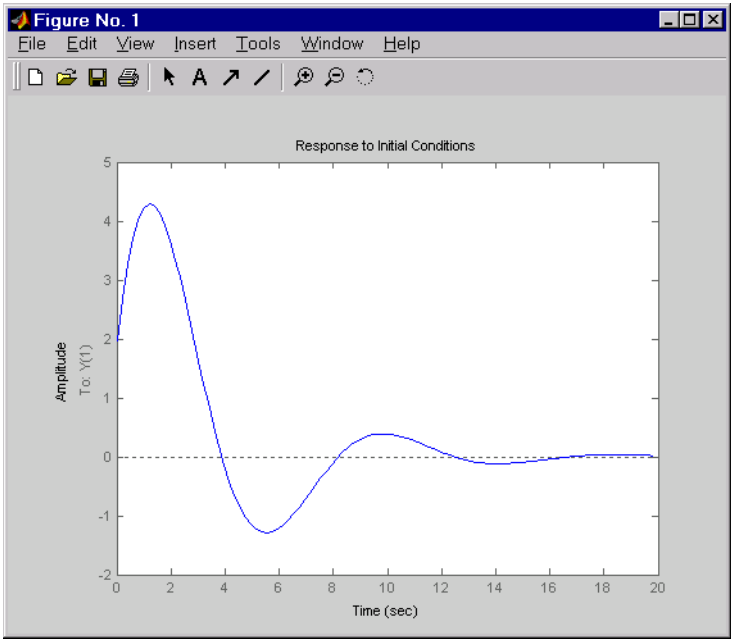
$$y = \begin{bmatrix} 1.9691 & 6.4493 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

to the initial condition

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

```
a = [-0.5572   -0.7814;0.7814  0];
c = [1.9691  6.4493];
x0 = [1 ; 0]

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



See Also `impulse`, `lsim`, `ltiview`, `step`

Purpose

Plot initial condition response and return plot handle

Syntax

```
initialplot(sys,x0)
initialplot(sys,x0,Tfinal)
initialplot(sys,x0,t)
initialplot(sys1,sys2,...,x0,t)
initialplot(AX,...)
initialplot(..., plotoptions)
```

Description

`initialplot(sys,x0)` plots the undriven response of the state-space model `sys` (created with `ss`) with initial condition `x0` on the states. This response is characterized by these equations:

Continuous time: x = A x, y = C x, x(0) = x0

Discrete time: x[k+1] = A x[k], y[k] = C x[k], x[0] = x0

The time range and number of points are chosen automatically. `initialplot` also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

`initialplot(sys,x0,Tfinal)` simulates the time response from `t=0` to the final time `t=Tfinal`. For discrete-time models with unspecified sample time, `Tfinal` should be the number of samples.

`initialplot(sys,x0,t)` specifies a time vector `t` to be used for simulation. For discrete systems, `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 of the continuous model.

`initialplot(sys1,sys2,...,x0,t)` plots the response of multiple LTI models `sys1,sys2,...` on a single plot. The time vector `t` is optional. 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.

Example

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.
```

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`.

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`, `ltimodels`

Purpose

Invert LTI systems

Syntax

`inv`

Description

`inv` inverts the input/output relation

$$y = G(s)u$$

to produce the LTI system with the transfer matrix $H(s) = G(s)^{-1}$.

$$u = H(s)y$$

This operation is defined only for square systems (same number of inputs and outputs) with an invertible feedthrough matrix D. `inv` handles both continuous- and discrete-time systems.

Example

Consider

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

At the MATLAB prompt, type

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

to invert it. MATLAB returns

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

 #2:  0

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

```
          s + 1

     #2:  1
```

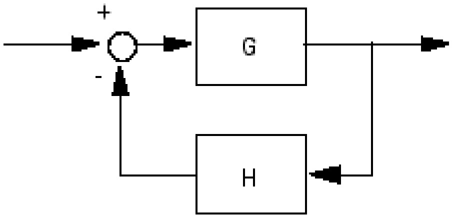
You can verify that

```
H * Hi
```

is the identity transfer function (static gain `I`).

Limitations

Do not use `inv` to model feedback connections such as



While it seems reasonable to evaluate the corresponding closed-loop transfer function $(I + GH)^{-1}G$ as

```
inv(1+g*h) * g
```

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

```
g = zpk([],1,1)
h = tf([2 1],[1 0])
cloop = inv(1+g*h) * g
```

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

```
cloop

Zero/pole/gain:
      s (s-1)
```

```
-------------------
(s-1) (s^2 + s + 1)
```

Use `feedback` to avoid such pitfalls.

```
cloop = feedback(g,h)

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

Purpose Plot pole-zero map for I/O pairs of LTI model

Syntax

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

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

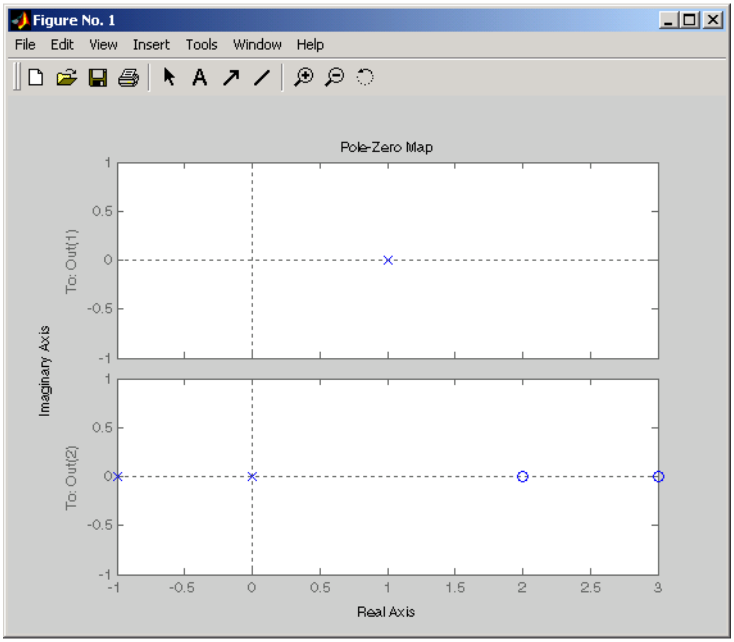
`iopzmap(sys1,sys2,...)` shows the poles and zeros of multiple LTI models `sys1`,`sys2`,... on a single plot. You can specify distinctive colors for each model, as in `iopzmap(sys1,'r',sys2,'y',sys3,'g')`.

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

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

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

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



See Also `pzmap`, `pole`, `zero`, `sgrid`, `zgrid`, `ltimodels`

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.

Example

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')
```

See Also

`getoptions`, `iopzmap`, `setoptions`

Purpose

Determine whether LTI model is continuous or discrete

Syntax

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

Description

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

`boo = isdt(sys)` returns `1` (true) if `sys` is discrete and `0` (false) otherwise. Discrete-time LTI models have a nonzero sample time, except for empty models and static gains, which are regarded as either continuous or discrete as long as their sample time is not explicitly set to a nonzero value. Thus both

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

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

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

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

See Also

`isa`, `isempty`, `isproper`

Purpose Determine whether LTI model is empty

Syntax `isempty(sys)`

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

Example Both commands

```
isempty(tf)   % tf by itself returns an empty transfer function
isempty(ss(1,2,[],[]))
```

return `1` (true) while

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

returns `0` (false).

See Also `issiso`, `size`

Purpose

Determine whether LTI model is proper

Syntax

```
isproper(sys)
```

Description

`isproper(sys)` returns `1` (true) if the LTI model `sys` is proper and `0` (false) otherwise.

State-space models are always proper. SISO transfer functions or zero-pole-gain models are proper if the degree of their numerator is less than or equal to the degree of their denominator. MIMO transfer functions are proper if all their SISO entries are proper.

Example

The following commands

```
isproper(tf([1 0],1))         % transfer function s
isproper(tf([1 0],[1 1]))    % transfer function s/(s+1)
```

return false and true, respectively.

lti/isstable

Purpose Determine whether system is stable

Syntax `isstable(sys)`

Description `isstable(sys)` returns `TRUE` if the LTI model `sys` has stable dynamics, and `FALSE` otherwise. For LTI arrays, `isstable` returns a logical array where the k-th entry indicates the stability of the k-th model.

`isstable` is only supported for analytical models with a finite number of poles.

See Also `ltimodels`

Purpose Determine whether LTI model is single-input/single-output (SISO)

Syntax `issiso(sys)`

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

See Also `isempty`, `size`

kalman

Purpose Design continuous- or discrete-time Kalman estimator

Syntax `kalman`

Description `kalman` designs a Kalman state estimator given a state-space model of the plant and the process and measurement noise covariance data. The Kalman estimator is the optimal solution to the following continuous or discrete estimation problems.

Continuous-Time Estimation

Given the continuous plant

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

with known inputs u and process and measurement white noise w, v satisfying

$$E(w) = E(v) = 0\ , \quad E(ww^T) = Q\ , \quad E(vv^T) = R\ , \quad E(wv^T) = N$$

construct a state estimate $\hat{x}(t)$ that minimizes the steady-state error covariance

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

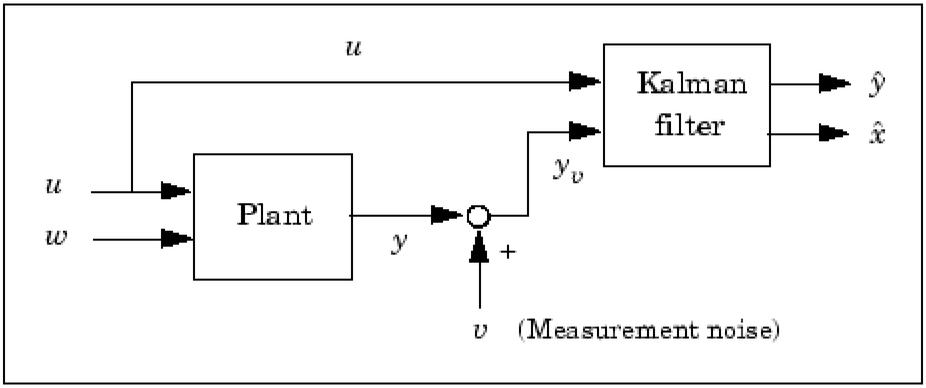
The optimal solution is the Kalman filter with equations

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

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

where the filter gain L is determined by solving an algebraic Riccati equation. This estimator uses the known inputs u and the measurements y_v to generate the output and state estimates $\hat{y}$ and $\hat{x}$. Note that $\hat{y}$ estimates the true plant output

$$y = Cx + Du + Hw$$



Kalman estimator

Discrete-Time Estimation

Given the discrete plant

$$\begin{aligned} x[n+1] &= Ax[n] + Bu[n] + Gw[n] \\ y_v[n] &= Cx[n] + Du[n] + Hw[n] + v[n] \end{aligned}$$

and the noise covariance data

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

the Kalman estimator has equations

$$\hat{x}[n+1|n] = A\hat{x}[n|n-1] + Bu[n] + L(y_v[n] - C\hat{x}[n|n-1] - Du[n])$$

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

and generates optimal "current" output and state estimates $\hat{y}[n|n]$ and $\hat{x}[n|n]$ using all available measurements including $y_v[n]$. The gain matrices L and M are derived by solving a discrete Riccati equation.

The innovation gain M is used to update the prediction $\hat{x}[n|n-1]$ using the new measurement $y_v[n]$.

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

Usage

`[kest,L,P] = kalman(sys,Qn,Rn,Nn)` returns a state-space model `kest` of the Kalman estimator given the plant model `sys` and the noise covariance data `Qn`, `Rn`, `Nn` (matrices Q, R, N above). `sys` must be a state-space model with matrices

$$A,\ [B\ G],\ C,\ [D\ H]$$

The resulting estimator `kest` has $[u\ ;\ y_v]$ as inputs and $[\hat{y}\ ;\ \hat{x}]$ (or their discrete-time counterparts) as outputs. You can omit the last input argument `Nn` when $N = 0$.

The function `kalman` handles both continuous and discrete problems and produces a continuous estimator when `sys` is continuous, and a discrete estimator otherwise. In continuous time, `kalman` also returns the Kalman gain `L` and the steady-state error covariance matrix `P`. Note that `P` is the solution of the associated Riccati equation. In discrete time, the syntax

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

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

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

Finally, use the syntaxes

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

for more general plants `sys` where the known inputs u and stochastic inputs w are mixed together, and not all outputs are measured. The index vectors `sensors` and `known` then specify which outputs y of `sys` are measured and which inputs $u$are known. All other inputs are assumed stochastic.

Example

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

Limitations

The plant and noise data must satisfy:

- (C, A) detectable
- $\overline{R} > 0$ and $\overline{Q} - \overline{N}\overline{R}^{-1}\overline{N}^T \geq 0$
- $(A - \overline{N}\overline{R}^{-1}C, \overline{Q} - \overline{N}\overline{R}^{-1}\overline{N}^T)$ has no uncontrollable mode on the imaginary axis (or unit circle in discrete time)

with the notation

$$\overline{Q} = GQG^T$$

$$\overline{R} = R + HN + N^TH^T + HQH^T$$

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

References

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

See Also

`care`, `dare`, `estim`, `kalmd`, `lqgreg`, `lqr`

kalmd

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

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

with process noise w and measurement noise v satisfying

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

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

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

The integral is computed using the matrix exponential formulas in [2]. A discrete-time estimator is then designed for the discretized plant and noise. See `kalman` for details on discrete-time Kalman estimation.

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

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

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`

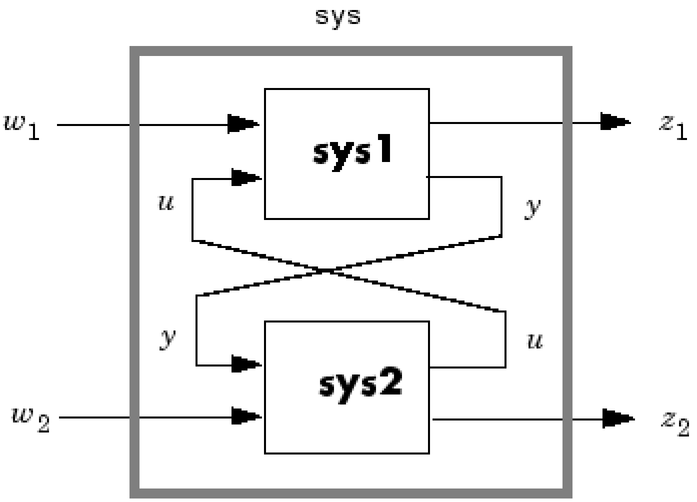
Purpose Generalized feedback interconnection of two LTI models (Redheffer star product)

Syntax

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

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

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



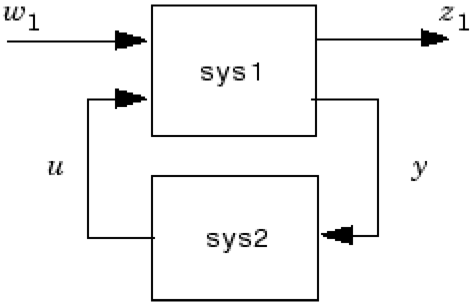
This feedback loop connects the first `nu` outputs of `sys2` to the last `nu` inputs of `sys1` (signals u), and the last `ny` outputs of `sys1` to the first `ny` inputs of `sys2` (signals y). The resulting system `sys` maps the input vector $[w_1 ; w_2]$ to the output vector $[z_1 ; z_2]$.

The abbreviated syntax

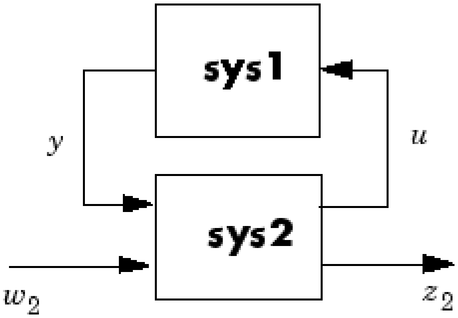
```
sys = lft(sys1,sys2)
```

produces:

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



Lower LFT connection



Upper LFT connection

Algorithm

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

Limitations

There should be no algebraic loop in the feedback connection.

See Also

`connect`, `feedback`

Purpose Continuous linear-quadratic-Gaussian (LQG) control synthesis

Syntax

```
reg = lqg(sys,QXU,QWV)
```

Description `reg = lqg(sys,QXU,QWV)` computes an optimal LQG regulator `reg` given a state-space model SYS of the plant and some weighting matrices `QXU` and `QWV`. The dynamic regulator `u = REG * y` generates the control signal `u` from the noisy measurements `y`. Use positive feedback to connect this regulator to the plant.

The LQG regulator minimizes the cost function

$$J = \lim_{T \to \infty} \int_0^T [x', u'] QXU \begin{bmatrix} x \\ u \end{bmatrix} dt$$

subject to the plant equations

```
dx/dt = Ax + Bu + w
y     = Cx + Du + v
```

where the process noise w and measurement noise v are Gaussian white noises with covariance:

```
E([w;v] * [w',v']) = QWV
```

`lqg` can be used for both continuous- and discrete-time plants and uses the commands `lqr` and `kalman` to compute the LQG regulator.

See Also `lqr`, `kalman`, `lqry`, `ss`, `care`, `dare`

Purpose

Form LQG regulator given state-feedback gain and Kalman estimator

Syntax

```
lqgreg
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

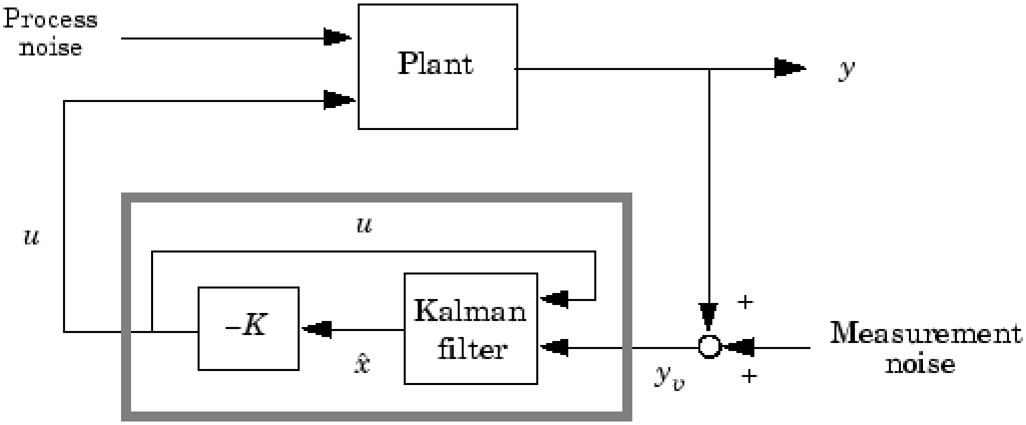
$$u = -K\hat{x}$$

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

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

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

where y_v is the vector of plant output measurements (see `kalman` for background and notation). The diagram below shows this dynamic regulator in relation to the plant.



LQG regulator

In discrete time, you can form the LQG regulator using either the prediction $\hat{x}[n|n-1]$ of $x[n]$ based on measurements up to $y_v[n-1]$, or the current state estimate $\hat{x}[n|n]$ based on all available measurements including $y_v[n]$. While the regulator

$$u[n] = -K\hat{x}[n|n-1]$$

is always well-defined, the *current regulator*

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

is causal only when $I - KMD$ is invertible (see `kalman` for the notation). In addition, practical implementations of the current regulator should allow for the processing time required to compute $u[n]$ once the measurements $y_v[n]$ become available (this amounts to a time delay in the feedback loop).

Usage

`rlqg = lqgreg(kest,k)` returns the LQG regulator `rlqg` (a state-space model) given the Kalman estimator `kest` and the state-feedback gain matrix `k`. The same function handles both continuous- and discrete-time cases. Use consistent tools to design `kest` and `k`:

- Continuous regulator for continuous plant: use `lqr` or `lqry` and `kalman`.
- Discrete regulator for discrete plant: use `dlqr` or `lqry` and `kalman`.
- Discrete regulator for continuous plant: use `lqrd` and `kalmd`.

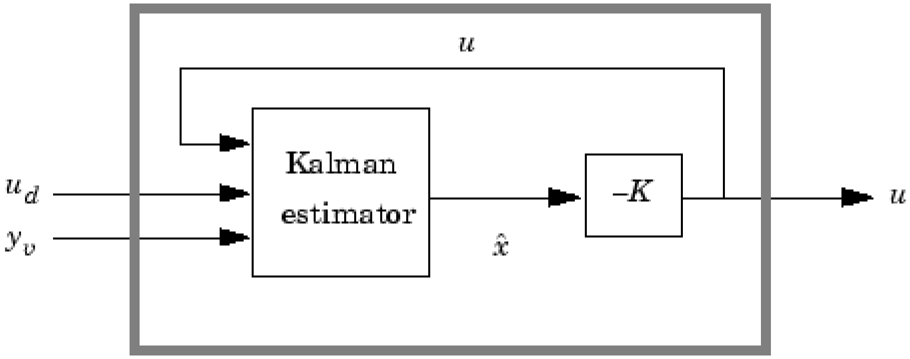
In discrete time, `lqgreg` produces the regulator $u[n] = -K\hat{x}[n|n-1]$ by default (see "Description"). To form the "current" LQG regulator $u[n] = -K\hat{x}[n|n]$ instead, use the syntax

```
rlqg = lqgreg(kest,k,'current')
```

This syntax is meaningful only for discrete-time problems.

`rlqg = lqgreg(kest,k,controls)` handles estimators that have access to additional known plant inputs u_d. The index vector `controls` then specifies which estimator inputs are the controls u, and the resulting LQG regulator `rlqg` has u_d and y_v as inputs (see figure below).

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



LQG regulator

Example See the example LQG Regulation.

See Also `kalman`, `kalmd`, `lqr`, `dlqr`, `lqrd`, `lqry`, `reg`

Purpose

Linear-quadratic (LQ) state-feedback regulator for state-space system

Syntax

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

Description

`[K,S,e] = lqr(SYS,Q,R,N)` calculates the optimal gain matrix `K` such that:

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 = eig(A-B*K)`. Note that K is derived from S by

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

For a discrete-time state-space model, `u[n]=-Kx[n]` minimizes

$$J = \sum \{x'Qx + u'Ru + 2x'Nu\}$$

subject to `x[n+1]=Ax[n]+Bu[n]`.

`[K,S,e] = LQR(A,B,Q,R,N)` is an equivalent syntax for continuous-time models with dynamics `dx/dt=Ax+Bu`.

In all cases, the default value `N=0` is assumed when `N` is omitted.

Limitations

The problem data must satisfy:

- The pair (A, B) is stabilizable.
- $R > 0$ and $Q - NR^{-1}N^T \geq 0$.
- $(Q - NR^{-1}N^T, A - BR^{-1}N^T)$ has no unobservable mode on the imaginary axis.

See Also `care`, `dlqr`, `lqgreg`, `lqrd`, `lqry`

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 = eig(Ad-Bd*Kd)`.

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

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

Algorithm

The equivalent discrete gain matrix `Kd` is determined by discretizing the continuous plant and weighting matrices using the sample time `Ts` and the zero-order hold approximation.

With the notation

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

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

the discretized plant has equations

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

and the weighting matrices for the equivalent discrete cost function are

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

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

Limitations

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

References

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

[2] Van Loan, C.F., "Computing Integrals Involving the Matrix Exponential," *IEEE Trans. Automatic Control*, AC-15, October 1970.

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

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

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

$$u = -Kx$$

that minimizes the quadratic cost function with output weighting

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

(or its discrete-time counterpart). The function `lqry` is equivalent to `lqr` or `dlqr` with weighting matrices:

$$\begin{bmatrix}\bar{Q} & \bar{N}\\ \bar{N}^T & \bar{R}\end{bmatrix} = \begin{bmatrix}C^T & 0\\ D^T & I\end{bmatrix}\begin{bmatrix}Q & N\\ N^T & R\end{bmatrix}\begin{bmatrix}C & D\\ 0 & I\end{bmatrix}$$

`[K,S,e] = lqry(sys,Q,R,N)` returns the optimal gain matrix `K`, the Riccati solution `S`, and the closed-loop eigenvalues `e = eig(A-B*K)`. The state-space model `sys` specifies the continuous- or discrete-time plant data (A, B, C, D). The default value `N=0` is assumed when `N` is omitted.

Example

See LQG Design for the x-Axis for an example.

Limitations

The data $A, B, \bar{Q}, \bar{R}, \bar{N}$ must satisfy the requirements for `lqr` or `dlqr`.

See Also

`lqr`, `dlqr`, `kalman`, `lqgreg`

lsim

Purpose

Simulate LTI model responses 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 LTI model `sys` to the input time history `t`,`u`. The vector `t` specifies the time samples for the simulation and consists of regularly spaced time samples.

```
t = 0:dt:Tfinal
```

The matrix `u` must have as many rows as time samples (`length(t)`) and as many columns as system inputs. Each row `u(i,:)` specifies the input value(s) at the time sample `t(i)`.

The LTI model `sys` can be continuous or discrete, SISO or MIMO. In discrete time, `u` must be sampled at the same rate as the system (`t` is then redundant and can be omitted or set to the empty matrix). In continuous time, the time sampling `dt=t(2)-t(1)` is used to discretize the continuous model. If `dt` is too large (undersampling), `lsim` issues a warning suggesting that you use a more appropriate sample time, but will use the specified sample time. See “Algorithm” on page 2-169 for a discussion of sample times.

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

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

Finally,

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

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

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

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

When invoked with left-hand arguments,

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

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x` (for state-space models only). No plot is drawn on the screen. The matrix `y` has as many rows as time samples (`length(t)`) and as many columns as system outputs. The same holds for `x` with "outputs" replaced by states. Note that the output `t` may differ from the specified time vector when the input data is undersampled (see "Algorithm" on page 2-169).

`lsim(sys)` opens the Linear Simulation Tool GUI. For more information about working with this GUI, see Working with the Linear Simulation Tool in the Control System Toolbox Getting Started guide.

Example

Simulate and plot the response of the system

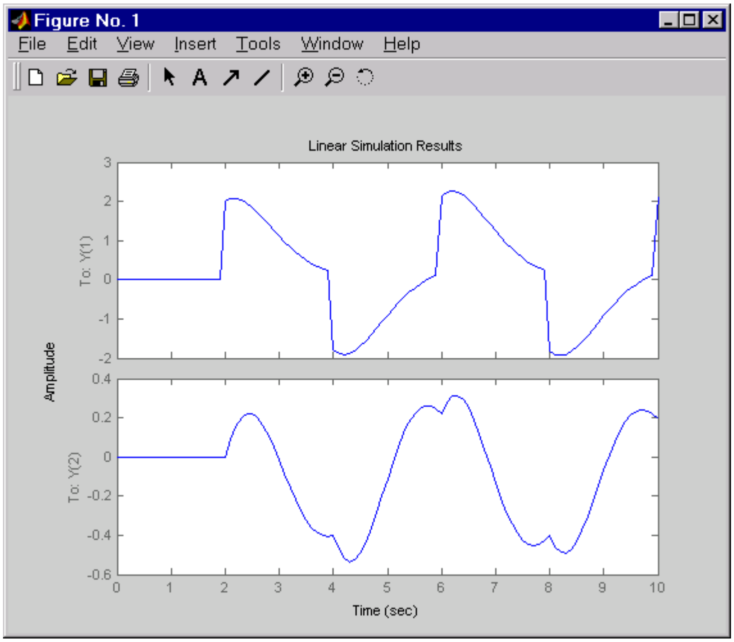
$$H(s) = \begin{bmatrix} \dfrac{2s^2+5s+1}{s^2+2s+3} \\ \dfrac{s-1}{s^2+s+5} \end{bmatrix}$$

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

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

Then simulate with `lsim`.

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



Algorithm

Discrete-time systems are simulated with `ltitr` (state space) or `filter` (transfer function and zero-pole-gain).

Continuous-time systems are discretized with `c2d` using either the `'zoh'` or `'foh'` method (`'foh'` is used for smooth input signals and `'zoh'` for discontinuous signals such as pulses or square waves). The sampling period is set to the spacing `dt` between the user-supplied time samples `t`.

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

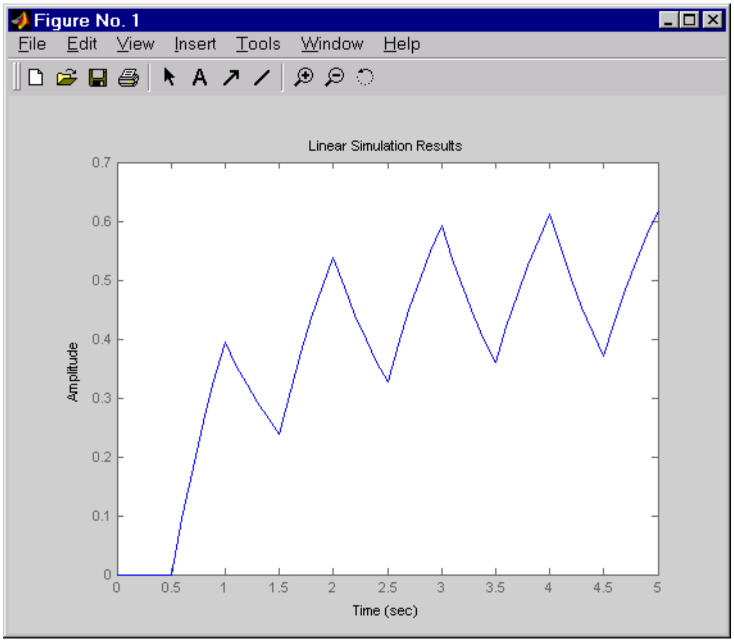
$$H(s) = \frac{\omega^2}{s^2 + 2s + \omega^2}, \qquad \omega = 62.83$$

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

```
w2 = 62.83^2
h = tf(w2,[1 2 w2])
t = 0:0.1:5;                % vector of time samples
u = (rem(t,1)>=0.5);    % square wave values
lsim(h,u,t)
```

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

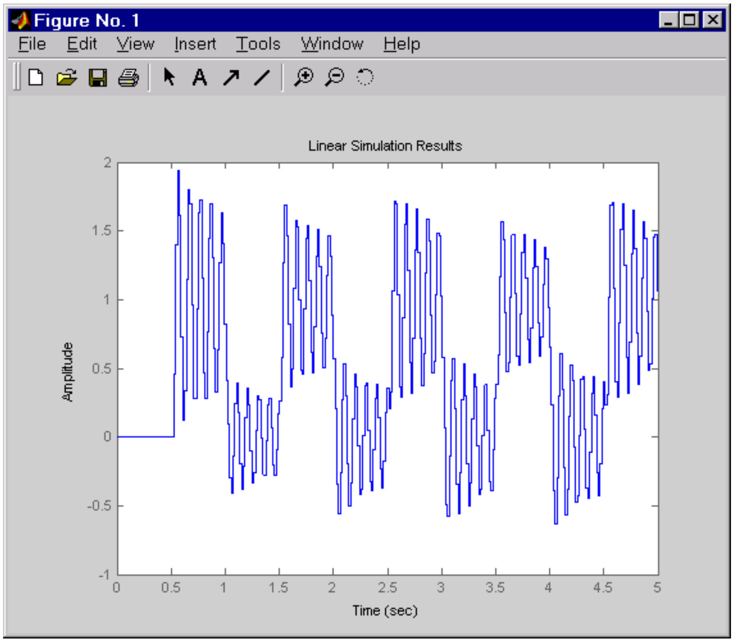
```
Warning: Input signal is undersampled. Sample every 0.016 sec or
faster.
```



and produces this plot.

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

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



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

See Also

`gensig`, `impulse`, `initial`, `ltiview`, `step`

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 responsehas settled when the error `|y(t) - yfinal|` becomes smaller than a fraction `ST` of its peak value. The default value is `ST=0.02` (2%).

Example Create a fourth order transfer function and ascertain the response characteristics.

```
sys = tf([1 -1],[1 2 3 4]);
[y,t] = impulse(sys);
s = lsiminfo(y,t,0)  % final value is 0
s =
```

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

See Also `lsim`, `impulse`, `initial`, `stepinfo`, `ltimodels`

lsimplot

Purpose

Simulate LTI model responses 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 LTI model `sys` (created with `tf`, `zpk`, or `ss`), 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 LTI 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 LTI model `sys` to the input signal described by `u` and `t`. The time vector `t` consists of regularly spaced time samples. For MIMO systems, `u` is a matrix with as many columns as inputs and whose `i`th row specifies the input value at time `t(i)`. For SISO systems `u` can be specified either as a row or column vector. For example,

```
t = 0:0.01:5;
u = sin(t);
lsimplot(sys,u,t)
```

simulates the response of a single-input model `sys` to the input `u(t)=sin(t)` during 5 seconds.

For discrete-time models, `u` should be sampled at the same rate as `sys` (`t` is then redundant and can be omitted or set to the empty matrix).

For continuous-time models, choose the sampling period `t(2)-t(1)` small enough to accurately describe the input `u`. `lsim` issues a warning when `u` is undersampled, and hidden oscillations can occur.

`lsimplot(sys,u,t,x0)` specifies the initial state vector `x0` at time `t(1)` (for state-space models only). `x0` is set to zero when omitted.

`lsimplot(sys1,sys2,...,u,t,x0)` simulates the responses of multiple LTI models `sys1,sys2,...` on a single plot. The initial condition `x0` is optional. You can also specify a color, line style, and marker for each system, as in

```
lsimplot(sys1,'r',sys2,'y--',sys3,'gx',u,t)
```

`lsimplot(AX,...)` plots into the axes with handle `AX`.

`lsimplot(..., plotoptions)` plots the initial condition response with the options specified in `plotoptions`. Type

```
help timeoptions
```

for more detail.

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

See Also

`getoptions`, `lsim`, `setoptions`

ltimodels

Purpose

Help on LTI models

Syntax

```
ltimodels
timodels(modeltype)
```

Description

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

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

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

Note that you can type

```
ltimodels zpk
```

as a shorthand for

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

See Also

`frd`, `ltiprops`, `ss`, `tf`, `zpk`

Purpose

Help on LTI model properties

Syntax

```
ltiprops
ltiprops(
```

Description

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

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

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

Note that you can type

```
ltiprops tf
```

as a shorthand for

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

See also

`get`, `ltimodels`, `set`

ltiview

Purpose

LTI Viewer for LTI system response analysis

Syntax

```
ltiview
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.

Finally,

```
ltiview(plottype,sys1,sys2,...sysN)
ltiview(plottype,sys1,PlotStyle1,sys2,PlotStyle2,...)
ltiview(plottype,sys1,sys2,...sysN,extras)
```

initializes an LTI Viewer containing the responses of multiple LTI models, using the plot styles in `PlotStyle`, when applicable. See the individual reference pages of the LTI response functions for more information on specifying plot styles.

See Also `bode`, `impulse`, `initial`, `lsim`, `nichols`, `nyquist`, `pzmap`, `sigma`, `step`

lyap

Purpose

Solve continuous-time Lyapunov equation

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 matrix equation. Lyapunov equations arise in several areas of control, including stability theory and the study of the RMS behavior of systems.

`X = lyap(A,Q)` solves the Lyapunov equation

$$AX + XA^T + Q = 0$$

where A and Q are square matrices of identical sizes. The solution `X` is a symmetric matrix if Q is.

`X = lyap(A,B,C)` solves the 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. The empty square brackets, `[]`, are mandatory. If you place any values inside them, the function will error out.

Algorithm

`lyap` transforms the A and B matrices to complex Schur form, computes the solution of the resulting triangular system, and transforms this solution back[1].

`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 $\alpha_1, \alpha_2, \ldots, \alpha_n$ of A and $\beta_1, \beta_2, \ldots, \beta_n$ of B satisfy

$$\alpha_i + \beta_j \neq 0 \qquad \text{for all pairs } (i, j)$$

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

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

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.

See Also

`covar`, `dlyap`

Purpose

Square-root solver for continuous-time Lyapunov equation

Syntax

```
R = lyapchol(A,B)
X = lyapchol(A,B,E)
```

Description

`R = lyapchol(A,B)` computes a Cholesky factorization `X = R'*R` of the solution `X` to the Lyapunov matrix equation:

```
A*X + X*A' + B*B' = 0
```

All eigenvalues of matrix `A` must lie in the open left half-plane for `R` to exist.

`X = lyapchol(A,B,E)` computes a Cholesky factorization `X = R'*R` of `X` solving the generalized Lyapunov equation:

```
A*X*E' + E*X*A' + B*B' = 0
```

All generalized eigenvalues of (`A`,`E`) must lie in the open left half-plane for `R` to exist.

Algorithm

`lyapchol` uses SLICOT routines SB03OD and SG03BD.

See Also

`lyap`, `dlyapchol`

margin

Purpose

Gain and phase margins and associated crossover frequencies

Syntax

```
margin
[Gm,Pm,Wg,Wp] = margin(sys)
[Gm,Pm,Wg,Wp] = margin(mag,phase,w)
```

Description

`margin` calculates the minimum gain margin, `Gm`, phase margin, `Pm`, and associated crossover frequencies of SISO open-loop models, `Wg` and `Wp`. The gain and phase margins indicate the relative stability of the control system when the loop is closed. When invoked without left-hand arguments, `margin` produces a Bode plot and displays the margins on this plot.

The gain margin is the amount of gain increase required to make the loop gain unity at the frequency where the phase angle is -180°. In other words, the gain margin is $1/g$ if g is the gain at the -180° phase frequency. Similarly, the phase margin is the difference between the phase of the response and -180° when the loop gain is 1.0. The frequency at which the magnitude is 1.0 is called the *unity-gain frequency* or *crossover frequency*. It is generally found that gain margins of three or more combined with phase margins between 30 and 60 degrees result in reasonable trade-offs between bandwidth and stability.

`[Gm,Pm,Wg,Wp] = margin(sys)` computes the gain margin `Gm`, the phase margin `Pm`, and the corresponding crossover frequencies `Wg` and `Wp`, given the SISO open-loop model `sys`. `Wg` is the frequency where the gain is 0dB, and `Wp` is the frequency where the phase is -180°. This function handles both continuous- and discrete-time cases. When faced with several crossover frequencies, `margin` returns the smallest gain and phase margins.

The phase margin `Pm` is in degrees. The gain margin `Gm` is an absolute magnitude. You can compute the gain margin in dB by

```
Gm_dB = 20*log10(Gm)
```

`[Gm,Pm,Wg,Wp] = margin(mag,phase,w)` derives the gain and phase margins from the Bode frequency response data (magnitude, phase, and frequency vector). Interpolation is performed between the frequency points to estimate the margin values. This approach is generally less accurate.

When invoked without left-hand argument,

```
margin(sys)
```

plots the open-loop Bode response with the gain and phase margins marked by vertical lines. By default, gain margins are expressed in dB when plotting.

Example

You can compute the gain and phase margins of the open-loop discrete-time transfer function. Type

```
hd = tf([0.04798 0.0464],[1 -1.81 0.9048],0.1)
```

MATLAB responds with

```
Transfer function:
 0.04798 z + 0.0464
---------------------
z^2 - 1.81 z + 0.9048

Sampling time: 0.1
```

Type

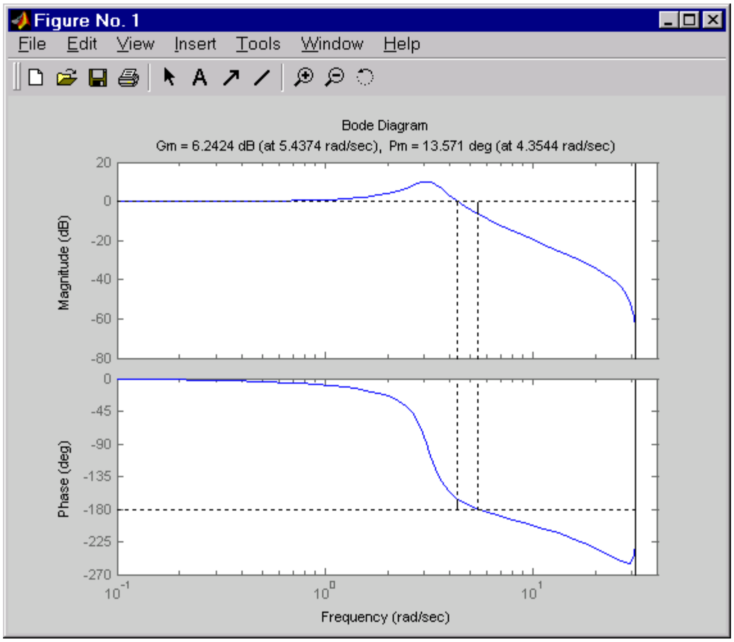
```
[Gm,Pm,Wg,Wp] = margin(hd);
```

and MATLAB returns

```
ans =
    2.0517   13.5711    5.4374    4.3544
```

You can also display these margins graphically.

```
margin(hd)
```



Algorithm

The phase margin is computed using H_∞ theory, and the gain margin by solving $H(j\omega) = \overline{H(j\omega)}$ for the frequency ω.

See Also

`bode`, `ltiview`

Purpose Minimal realization or pole-zero cancelation

Syntax

```
sysr = minreal(sys)
sysr = minreal(sys,tol)
[sysr,u] = minreal(sys,tol)
```

Description

`sysr = minreal(sys)` eliminates uncontrollable or unobservable state in state-space models, or cancels pole-zero pairs in transfer functions or zero-pole-gain models. The output `sysr` has minimal order and the same response characteristics as the original model `sys`.

`sysr = minreal(sys,tol)` specifies the tolerance used for state elimination or pole-zero cancellation. The default value is `tol = sqrt(eps)` and increasing this tolerance forces additional cancellations.

`[sysr,u] = minreal(sys,tol)` returns, for state-space model `sys`, an orthogonal matrix U such that `(U*A*U',U*B,C*U')` is a Kalman decomposition of (`A`,`B`,`C`)

Example

The commands

```
g = zpk([],1,1)
h = tf([2 1],[1 0])
cloop = inv(1+g*h) * g
```

produce the nonminimal zero-pole-gain model by typing `cloop`.

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

To cancel the pole-zero pair at $s = 1$, type

```
cloop = minreal(cloop)
```

and MATLAB returns

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

Algorithm

Pole-zero cancellation is a straightforward search through the poles and zeros looking for matches that are within tolerance. Transfer functions are first converted to zero-pole-gain form.

See Also

`balreal`, `modred`, `sminreal`

Purpose

Model order reduction

Syntax

```
modred
rsys = modred(sys,elim)
rsys = modred(sys,elim,'method')
```

Description

`modred` 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 X2 is to be discarded, and the reduced state is set to Xr = X1+T*X2 where T is chosen to enforce matching DC gains (steady-state response) between `sys` and `rsys`.

`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'`: Enforce matching DC gains (default)
- `'Truncate'`: Simply delete X2 and sets Xr = X1.

The `'Truncate'` option tends to produces 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`.

Example

Consider the continuous fourth-order model

$$h(s) = \frac{s^3 + 11s^2 + 36s + 26}{s^4 + 14.6s^3 + 74.96s^2 + 153.7s + 99.65}$$

To reduce its order, first compute a balanced state-space realization with `balreal` by typing

```
h = tf([1 11 36 26],[1 14.6 74.96 153.7 99.65])
[hb,g] = balreal(h)
g'
```

MATLAB returns

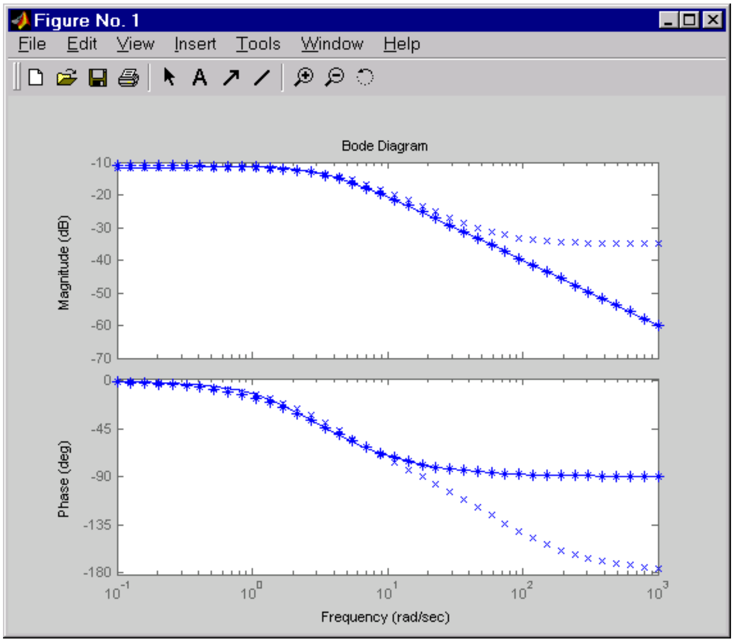
```
ans =
   1.3938e-01   9.5482e-03   6.2712e-04   7.3245e-06
```

The last three diagonal entries of the balanced grammians are small, so eliminate the last three states with `modred` using both matched DC gain and direct deletion methods.

```
hmdc = modred(hb,2:4,'MatchDC')
hdel = modred(hb,2:4,'Truncate')
```

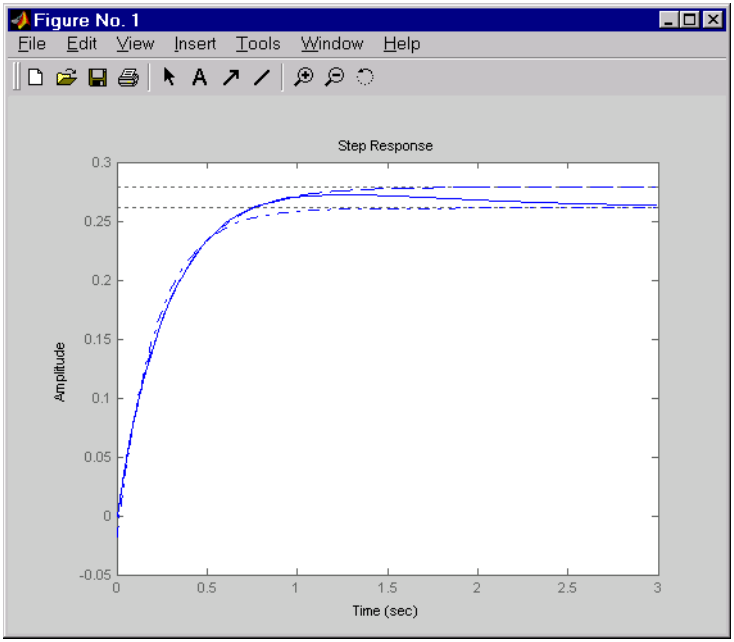
Both `hmdc` and `hdel` are first-order models. Compare their Bode responses against that of the original model $h(s)$.

```
bode(h,'-',hmdc,'x',hdel,'*')
```



The reduced-order model `hdel` is clearly a better frequency-domain approximation of $h(s)$. Now compare the step responses.

```
step(h,'-',hmdc,'-.',hdel,'--')
```



While `hdel` accurately reflects the transient behavior, only `hmdc` gives the true steady-state response.

Algorithm

The algorithm for the matched DC gain method is as follows. For continuous-time models

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

the state vector is partitioned into x_1, to be kept, and x_2, to be eliminated.

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} u$$

$$y = \begin{bmatrix} C_1 & C_2 \end{bmatrix} x + Du$$

Next, the derivative of x_2 is set to zero and the resulting equation is solved for x_1. The reduced-order model is given by

$$\dot{x}_1 = [A_{11} - A_{12}A_{22}^{-1}A_{21}]x_1 + [B_1 - A_{12}A_{22}^{-1}B_2]u$$

$$y = [C_1 - C_2A_{22}^{-1}A_{21}]x + [D - C_2A_{22}^{-1}B_2]u$$

The discrete-time case is treated similarly by setting

$$x_2[n+1] = x_2[n]$$

Limitations

With the matched DC gain method, A_{22} must be invertible in continuous time, and $I - A_{22}$ must be invertible in discrete time.

See Also

`balreal`, `minreal`

modsep

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:

$$G(s) = H0 + \sum_{j=1}^{N} Hj(s)$$

`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,...)
```

Example To decompose `G` into `G(z) = H0 + H1(z) + H2(z)` where `H1` and `H2` have their poles inside and outside the unit disk respectively, use

```
[H,H0]  = modsep(G,2,@udsep)
```

where the function `udsep` is defined by

```
function r = udsep(p)
if abs(p)<1, r = 1;  % assign r=1 to poles inside unit disk
else         r = 2;  % assign r=2 to poles outside unit disk
end
```

To extract `H1(z)` and `H2(z)` from the LTI array `H`, use

```
H1 = H(:,:,1);  H2 = H(:,:,2);
```

See Also

`stabsep`

ndims

Purpose

Provide number of dimensions of LTI model or LTI array

Syntax

```
n = ndims(sys)
```

Description

`n = ndims(sys)` is the number of dimensions of an LTI model or an array of LTI models `sys`. A single LTI model has two dimensions (one for outputs, and one for inputs). An LTI array has $2+p$ dimensions, where $p \geq 2$ is the number of array dimensions. For example, a 2-by-3-by-4 array of models has 2+3=5 dimensions.

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

Example

```
sys = rss(3,1,1,3);
ndims(sys)
ans =
     4
```

`ndims` returns 4 for this 3-by-1 array of SISO models.

See Also

`size`

Purpose

Superimpose Nichols chart on Nichols plot

Syntax

`ngrid`

Description

`ngrid` superimposes Nichols chart grid lines over the Nichols frequency response of a SISO LTI system. The range of the Nichols grid lines is set to encompass the entire Nichols frequency response.

The chart relates the complex number $H/(1+H)$ to H, where H is any complex number. For SISO systems, when H is a point on the open-loop frequency response, then

$$\frac{H}{1+H}$$

is the corresponding value of the closed-loop frequency response assuming unit negative feedback.

If the current axis is empty, `ngrid` generates a new Nichols chart grid in the region -40 dB to 40 dB in magnitude and -360 degrees to 0 degrees in phase. If the current axis does not contain a SISO Nichols frequency response, `ngrid` returns a warning.

Example

Plot the Nichols response with Nichols grid lines for the system.

$$H(s) = \frac{-4s^4+48s^3-18s^2+250s+600}{s^4+30s^3+282s^2+525s+60}$$

Type

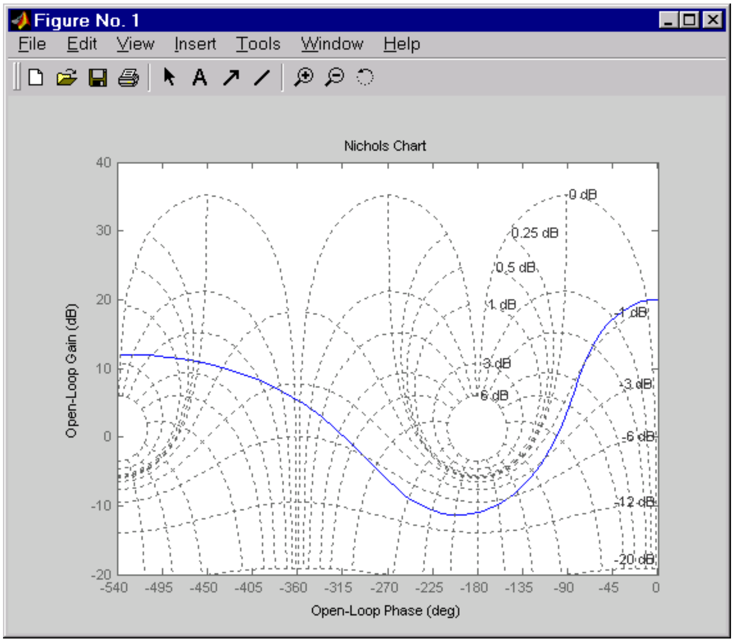
```
H = tf([-4 48 -18 250 600],[1 30 282 525 60])
```

MATLAB returns

```
Transfer function:
- 4 s^4 + 48 s^3 - 18 s^2 + 250 s + 600
---------------------------------------
  s^4 + 30 s^3 + 282 s^2 + 525 s + 60
```

Type

```
nichols(H)
ngrid
```



See Also `nichols`

Purpose

Nichols plot of LTI models

Syntax

```
nichols
nichols(sys)
nichols(sys,w)
nichols(sys1,sys2,...,sysN,w)
nichols(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
[mag,phase,w] = nichols(sys)
[mag,phase] = nichols(sys,w)
```

Description

`nichols` computes the frequency response of an LTI model and plots it in the Nichols coordinates. Nichols plots are useful to analyze open- and closed-loop properties of SISO systems, but offer little insight into MIMO control loops. Use `ngrid` to superimpose a Nichols chart on an existing SISO Nichols plot.

`nichols(sys)` produces a Nichols plot of the LTI model `sys`. This model can be continuous or discrete, SISO or MIMO. In the MIMO case, `nichols` produces an array of Nichols plots, each plot showing the response of one particular I/O channel. The frequency range and gridding are determined automatically based on the system poles and zeros.

`nichols(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. Frequencies should be specified in radians/sec.

`nichols(sys1,sys2,...,sysN)` or `nichols(sys1,sys2,...,sysN,w)` superimposes the Nichols plots of several LTI models on a single figure. All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous- and discrete-time systems. You can also specify a distinctive color, linestyle, and/or marker for each system plot with the syntax

```
nichols(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
```

See `bode` for an example.

When invoked with left-hand arguments,

```
[mag,phase,w] = nichols(sys)
[mag,phase] = nichols(sys,w)
```

return the magnitude and phase (in degrees) of the frequency response at the frequencies `w` (in rad/sec). The outputs `mag` and `phase` are 3-D arrays similar to those produced by `bode` (see the `bode` reference page). They have dimensions

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

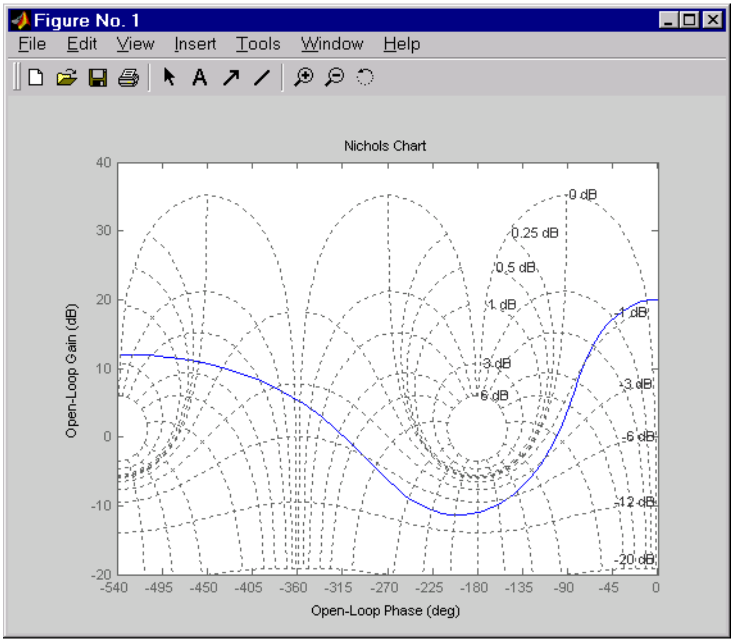
Example

Plot the Nichols response of the system

$$H(s) = \frac{-4s^4 + 48s^3 - 18s^2 + 250s + 600}{s^4 + 30s^3 + 282s^2 + 525s + 60}$$

```
num = [-4 48 -18 250 600];
den = [1 30 282 525 60];
H = tf(num,den)

nichols(H); ngrid
```



The right-click menu for Nichols plots includes the **Tight** option under **Zoom**. You can use this to clip unbounded branches of the Nichols plot.

Algorithm

See `bode`.

See Also

`bode`, `evalfr`, `freqresp`, `ltiview`, `ngrid`, `nyquist`, `sigma`

nicholsplot

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 LTI model `sys` (created with `tf`, `zpk`, `ss`, or `frd`). 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/s).

`nicholsplot(sys,w)` uses the user-supplied vector `w` of frequencies, in radians/second, 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 LTI models `sys1,sys2,...` on a single plot. The frequency vector `w` is optional. You can also specify a color, line style, and marker for each system, as in

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

`nicholsplot(AX,...)` plots into the axes with handle `AX`.

`nicholsplot(..., plotoptions)` plots the Nichols plot with the options specified in `plotoptions`. Type

```
help nicholsoptions
```

for more details.

Example

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`, `setoptions`

Purpose

Compute LTI model norm

Syntax

```
norm
norm(sys,inf)
norm(sys,inf,tol)
[ninf,fpeak] = norm(sys,inf)
```

Description

`norm` computes the H_2 or L_∞ norm of a continuous- or discrete-time LTI model.

H2 Norm

The H_2 norm of a stable continuous system with transfer function $H(s)$, is the root-mean-square of its impulse response, or equivalently

$$|H|_2 = \sqrt{\frac{1}{2\pi}\int_{-\infty}^{\infty} \mathrm{Trace}(H(j\omega)^H H(j\omega))\, d\omega}$$

This norm measures the steady-state covariance (or power) of the output response $y = Hw$ to unit white noise inputs w.

$$\|H\|_2^2 = \lim_{t \to \infty} E\{y(t)^T y(t)\} \quad , \qquad E(w(t)w(\tau)^T) = \delta(t-\tau)I$$

Infinity Norm

The infinity norm is the peak gain of the frequency response, that is,

$$|H(s)|_\infty = \max_{\omega} |H(j\omega)| \qquad \text{(SISO case)}$$

$$\|H(s)\|_\infty = \max_{\omega} \sigma_{\max}(H(j\omega)) \qquad \text{(MIMO case)}$$

where $\sigma_{\max}(\cdot)$ denotes the largest singular value of a matrix.

The discrete-time counterpart is

$$\|H(z)\|_\infty = \max_{\theta \in [0, \pi]} \sigma_{\max}(H(e^{j\theta}))$$

Usage

`norm(sys)` or `norm(sys,2)` both return the H_2 norm of the TF, SS, or ZPK model `sys`. This norm is infinite in the following cases:

- `sys` is unstable.
- `sys` is continuous and has a nonzero feedthrough (that is, nonzero gain at the frequency $\omega = \infty$).

Note that `norm(sys)` produces the same result as

```
sqrt(trace(covar(sys,1)))
```

`norm(sys,inf)` computes the infinity norm of any type of LTI model `sys`. This norm is infinite if `sys` has poles on the imaginary axis in continuous time, or on the unit circle in discrete time.

`norm(sys,inf,tol)` sets the desired relative accuracy on the computed infinity norm (the default value is `tol=1e-2`).

`[ninf,fpeak] = norm(sys,inf)` also returns the frequency `fpeak` where the gain achieves its peak value.

Example

Consider the discrete-time transfer function

$$H(z) = \frac{z^3 - 2.841z^2 + 2.875z - 1.004}{z^3 - 2.417z^2 + 2.003z - 0.5488}$$

with sample time 0.1 second. Compute its H_2 norm by typing

```
H = tf([1 -2.841 2.875 -1.004],[1 -2.417 2.003 -0.5488],0.1)
norm(H)
ans =
    1.2438
```

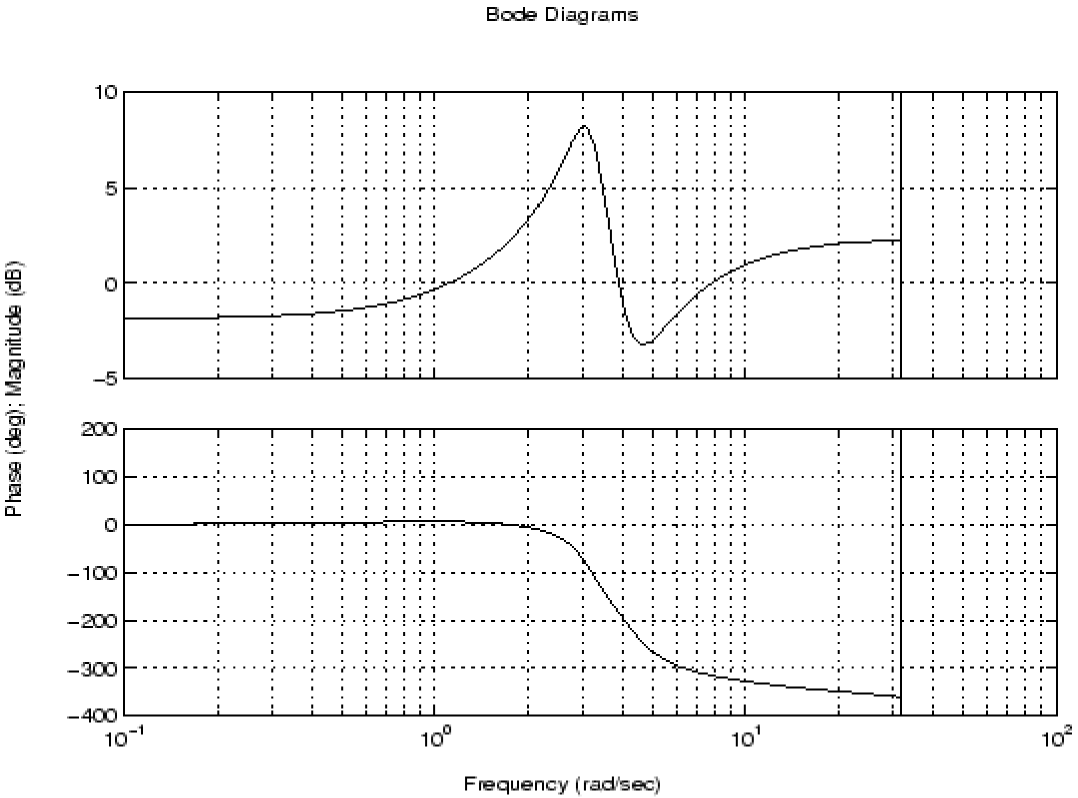
Compute its infinity norm by typing

```
[ninf,fpeak] = norm(H,inf)
ninf =
    2.5488

fpeak =
    3.0844
```

These values are confirmed by the Bode plot of $H(z)$.

```
bode(H)
```



The gain indeed peaks at approximately 3 rad/sec and its peak value in dB is found by typing

```
20*log10(ninf)
```

MATLAB returns

```
ans =
    8.1268
```

Algorithm

`norm` uses the same algorithm as `covar` for the H_2 norm, and the algorithm of [1] for the infinity norm. `sys` is first converted to state space.

References

[1] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

See Also

`bode`, `freqresp`, `sigma`

nyquist

Purpose Nyquist plot of LTI models

Syntax

```
nyquist
nyquist(sys)
nyquist(sys,w)
nyquist(sys1,sys2,...,sysN)
nyquist(sys1,sys2,...,sysN,w)
nyquist(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
[re,im,w] = nyquist(sys)
[re,im] = nyquist(sys,w)
```

Description

`nyquist` calculates the Nyquist frequency response of LTI models. When invoked without left-hand arguments, `nyquist` produces a Nyquist plot on the screen. Nyquist plots are used to analyze system properties including gain margin, phase margin, and stability.

`nyquist(sys)` plots the Nyquist response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. In the MIMO case, `nyquist` produces an array of Nyquist plots, each plot showing the response of one particular I/O channel. The frequency points are chosen automatically based on the system poles and zeros.

`nyquist(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. Frequencies should be specified in rad/sec.

`nyquist(sys1,sys2,...,sysN)` or `nyquist(sys1,sys2,...,sysN,w)` superimposes the Nyquist plots of several LTI models on a single figure. All systems must have the same number of inputs and outputs, but may otherwise be a mix of continuous- and discrete-time systems. You can also specify a distinctive color, linestyle, and/or marker for each system plot with the syntax

```
nyquist(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
```

See `bode` for an example.

When invoked with left-hand arguments

```
[re,im,w] = nyquist(sys)
[re,im] = nyquist(sys,w)
```

return the real and imaginary parts of the frequency response at the frequencies `w` (in rad/sec). `re` and `im` are 3-D arrays (see "Arguments" below for details).

Arguments

The output arguments `re` and `im` are 3-D arrays with dimensions

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

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 $\omega_k = \mathrm{w(k)}$.

$$\mathrm{re(1,1,k)} = \mathbf{Re}(h(j\omega_k))$$

$$\mathrm{im(1,1,k)} = \mathbf{Im}(h(j\omega_k))$$

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,

$$\mathrm{re(i,j,k)} = \mathbf{Re}(h_{ij}(j\omega_k))$$

$$\mathrm{im(i,j,k)} = \mathbf{Im}(h_{ij}(j\omega_k))$$

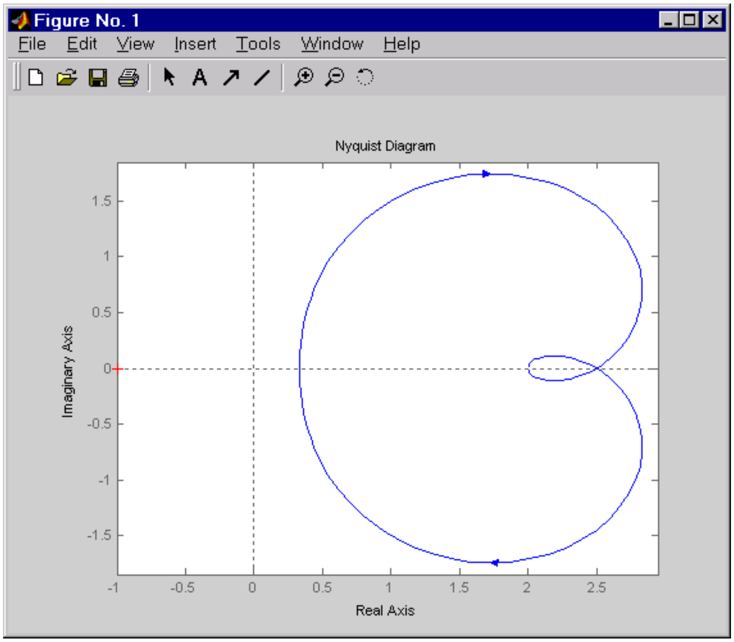
where h_{ij} is the transfer function from input j to output i.

Example

Plot the Nyquist response of the system

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

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



The nyquist function has support for M-circles, which are the contours of the constant closed-loop magnitude. M-circles are defined as the locus of complex numbers where

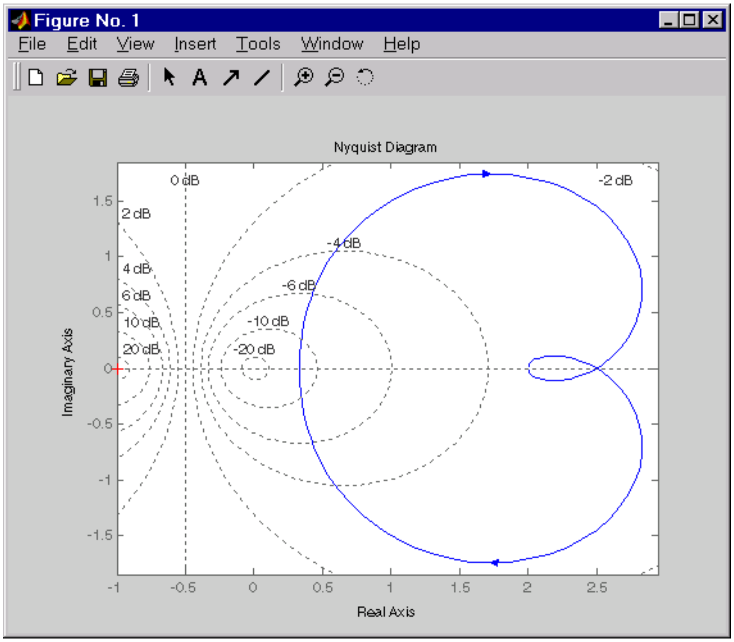
$$T(j\omega) = \left|\frac{G(j\omega)}{1+G(j\omega)}\right|$$

is a constant value. In this equation, ω is the frequency in radians/second, and *G* is the collection of complex numbers that satisfy the constant magnitude requirement.

To activate the grid, select **Grid** from the right-click menu or type

```
grid
```

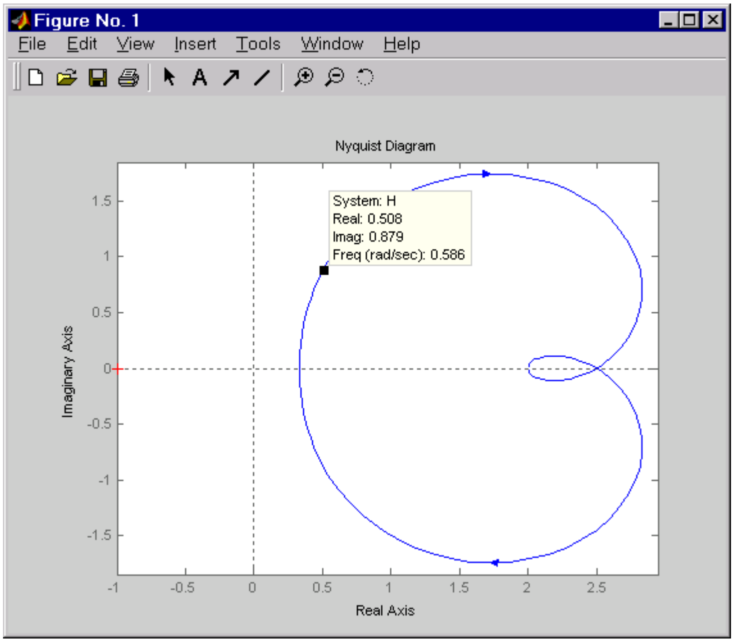
at the MATLAB prompt. This figure shows the M circles for transfer function *H*.



You have two zoom options available from the right-click menu that apply specifically to Nyquist plots:

- **Tight** —Clips unbounded branches of the Nyquist plot, but still includes the critical point (-1, 0)
- **On (-1,0)** — Zooms around the critical point (-1,0)

Also, click anywhere on the curve to activate data markers that display the real and imaginary values at a given frequency. This figure shows the nyquist plot with a data marker.



See Also

`bode`, `evalfr`, `freqresp`, `ltiview`, `nichols`, `sigma`

Purpose

Plot Nyquist frequency responses and return plot handle

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 LTI model `sys` (created with `tf`, `zpk`, `ss`, or `frd`). 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/s).

`nyquistplot(sys,w)` uses the user-supplied vector `w` of frequencies (in rad/s) 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 LTI 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.

Example 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');
```

See Also `getoptions`, `nyquist`, `setoptions`

Purpose

Observability matrix

Syntax

```
obsv
Ob = obsv(sys)
```

Description

`obsv` computes the observability matrix for state-space systems. For an n-by-n matrix `A` and a p-by-n matrix `C`, `obsv(A,C)` returns the observability matrix

$$Ob = \begin{bmatrix} C \\ CA \\ CA^2 \\ : \\ CA^{n-1} \end{bmatrix}$$

with n columns and np rows.

`Ob = obsv(sys)` calculates the observability matrix of the state-space model `sys`. This syntax is equivalent to executing

```
Ob = obsv(sys.A,sys.C)
```

The model is observable if `Ob` has full rank n.

Example

Determine if the pair

```
A =
       1       1
       4      -2

C =
       1       0
       0       1
```

is observable. Type

```
Ob = obsv(A,C);

% Number of unobservable states
unob = length(A)-rank(Ob)
```

MATLAB responds with

```
unob =
     0
```

Caveat

`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://lawww.epfl.ch/webdav/site/la/users/105941/public/NumCompCtrl.pdf

See Also

`obsvf`

Purpose

Compute observability staircase form

Syntax

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

Description

If the observability matrix of `(A,C)` has rank $r \leq n$, where n is the size of `A`, then there exists a similarity transformation such that

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

where T is unitary and the transformed system has a *staircase* form with the unobservable modes, if any, in the upper left corner.

$$\bar{A} = \begin{bmatrix} A_{no} & A_{12} \\ 0 & A_o \end{bmatrix}, \qquad \bar{B} = \begin{bmatrix} B_{no} \\ B_o \end{bmatrix}, \qquad \bar{C} = \begin{bmatrix} 0 & C_o \end{bmatrix}$$

where (C_o, A_o) is observable, and the eigenvalues of A_{no} are the unobservable modes.

`[Abar,Bbar,Cbar,T,k] = obsvf(A,B,C)` decomposes the state-space system with matrices `A`, `B`, and `C` into the observability staircase form `Abar`, `Bbar`, and `Cbar`, as described above. `T` is the similarity transformation matrix and `k` is a vector of length n, where n is the number of states in `A`. Each entry of `k` represents the number of observable states factored out during each step of the transformation matrix calculation [1]. The number of nonzero elements in `k` indicates how many iterations were necessary to calculate `T`, and `sum(k)` is the number of states in A_o, the observable portion of `Abar`.

`obsvf(A,B,C,tol)` uses the tolerance `tol` when calculating the observable/unobservable subspaces. When the tolerance is not specified, it defaults to `10*n*norm(a,1)*eps`.

Example

Form the observability staircase form of

```
A =
```

```
        1     1
        4    -2

  B =
        1    -1
        1    -1

  C =
        1     0
        0     1
```

by typing

```
  [Abar,Bbar,Cbar,T,k] = obsvf(A,B,C)
  Abar =
        1     1
        4    -2
  Bbar =
        1     1
        1    -1
  Cbar =
        1     0
        0     1
  T =
        1     0
        0     1
  k =
        2     0
```

Algorithm

`obsvf` is an M-file that implements the Staircase Algorithm of [1] by calling `ctrbf` and using duality.

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 `wn` (ω_n) and damping factor `z` (ζ). Use `ss` to turn this description into a state-space object.

`[num,den] = ord2(wn,z)` returns the numerator and denominator of the second-order transfer function. Use `tf` to form the corresponding transfer function object.

Example

To generate an LTI model of the second-order transfer function with damping factor $\zeta = 0.4$ and natural frequency $\omega_n = 2.4 \text{ rad/sec.}$, 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`

lti/order

Purpose

LTI model order

Syntax

```
NS = order(sys)
```

Description

`NS = order(sys)` returns the model order `NS`. The order of an LTI model is the number of poles (for proper transfer functions) or the number of states (for state-space models). For improper transfer functions, the order is defined as the minimum number of states needed to build an equivalent state-space model (ignoring pole/zero cancellations).

`order(sys)` is an overloaded method that accepts SS, TF, and ZPK models. For LTI arrays, `NS` is an array of the same size listing the orders of each model in `sys`.

Caveat

`order` does not attempt to find minimal realizations of MIMO systems. For example, consider this 2-by-2 MIMO system:

```
s=tf('s');
h = [1, 1/(s*(s+1)); 1/(s+2), 1/(s*(s+1)*(s+2))];
order(h)
ans =

      6
```

Although `h` has a 3rd order realization, `order` returns 6. Use

```
order(ss(h,'min')
```

to find the minimal realization order.

See Also

`pole`, `balred`, `ltimodels`

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 LTI models. Such approximations are useful to model time delay effects such as transport and computation delays within the context of continuous-time systems. The Laplace transform of an time delay of T seconds is $\exp(-sT)$. This exponential transfer function is approximated by a rational transfer function using Padé approximation formulas [1].

`[num,den] = pade(T,N)` returns the `N`th-order Padé approximation of the continuous-time I/O delay $\exp(-sT)$ in transfer function form. The row vectors `num` and `den` contain the numerator and denominator coefficients in descending powers of s. Both are `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 Time Delays for details on LTI models with 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

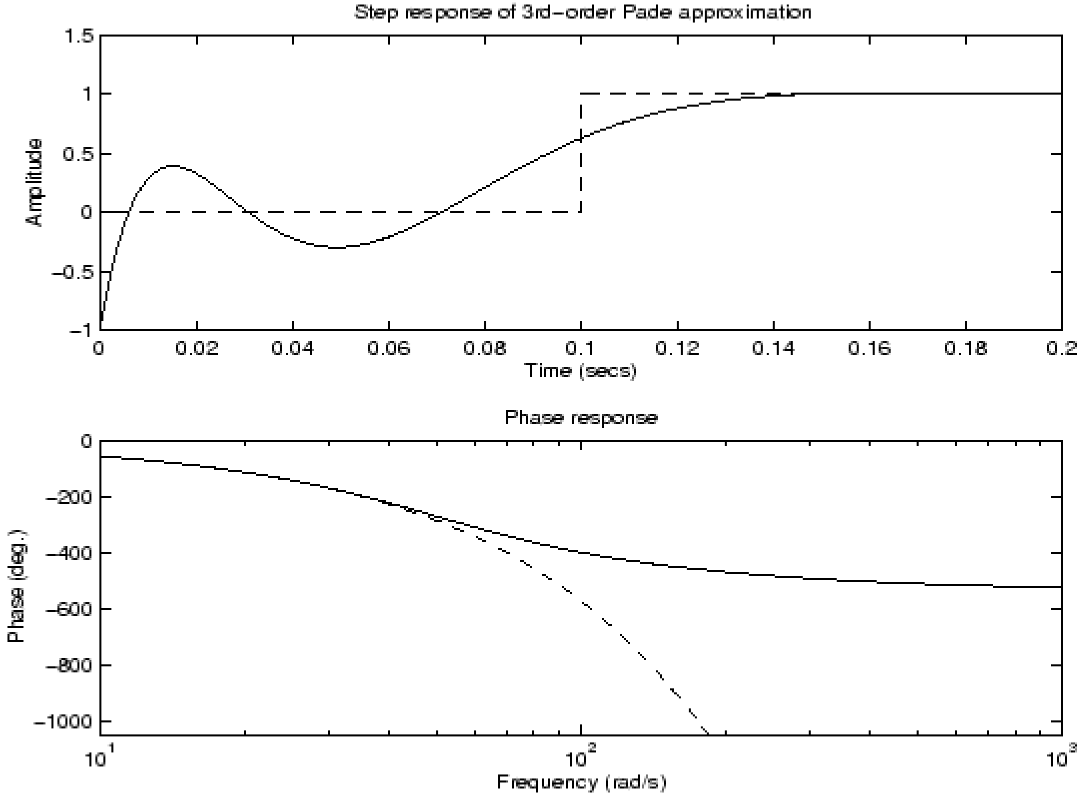
- `NINT` is the approximation order for I/O delays (TF or ZPK models) or internal delays (state-space models)

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.

Example

Compute a third-order Padé approximation of a 0.1 second I/O delay and compare the time and frequency responses of the true delay and its approximation. To do this, type

```
pade(0.1,3)
```



Limitations High-order Padé approximations produce transfer functions with clustered poles. Because such pole configurations tend to be very sensitive to perturbations, Padé approximations with order `N>10` should be avoided.

References [1] Golub, G. H. and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989, pp. 557-558.

See Also `c2d`, `delay2z`, `ltimodels`, `ltiprops`

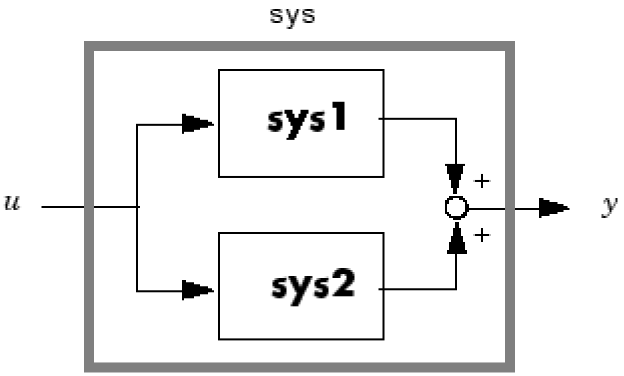
Purpose Parallel connection of two LTI models

Syntax

```
parallel
sys = parallel(sys1,sys2)
sys = parallel(sys1,sys2,inp1,inp2,out1,out2)
```

Description `parallel` connects two LTI models in parallel. This function accepts any type of LTI model. The two systems must be either both continuous or both discrete with identical sample time. Static gains are neutral and can be specified as regular matrices.

`sys = parallel(sys1,sys2)` forms the basic parallel connection shown below.

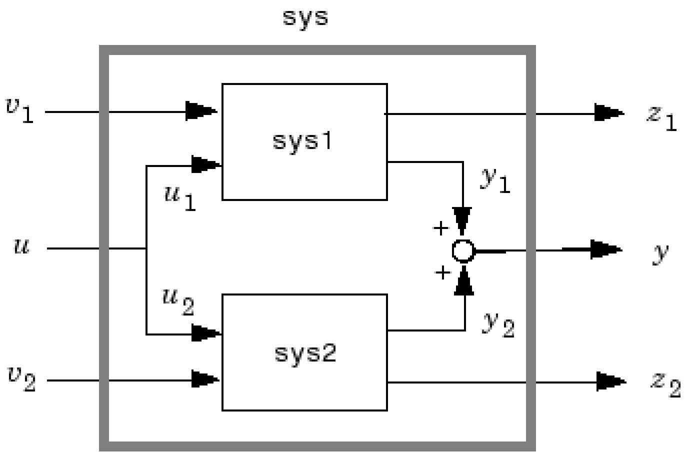


This command is equivalent to the direct addition

```
sys = sys1 + sys2
```

(See Addition and Subtraction for details on LTI system addition.)

`sys = parallel(sys1,sys2,inp1,inp2,out1,out2)` forms the more general parallel connection.



The index vectors `inp1` and `inp2` specify which inputs u_1 of `sys1` and which inputs u_2 of `sys2` are connected. Similarly, the index vectors `out1` and `out2` specify which outputs y_1 of `sys1` and which outputs y_2 of `sys2` are summed. The resulting model `sys` has $[v_1 \; ; u \; ; v_2]$ as inputs and $[z_1 \; ; y \; ; z_2]$ as outputs.

Example See Kalman Filtering for an example.

See Also `append`, `feedback`, `series`

place

Purpose

Pole placement design

Syntax

```
K = place(A,B,p)
[K,prec,message] = place(A,B,p)
```

Description

Given the single- or multi-input system

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

and a vector `p` of desired self-conjugate closed-loop pole locations, `place` computes a gain matrix `K` such that the state feedback $u = -Kx$ places the closed-loop poles at the locations `p`. In other words, the eigenvalues of $A - BK$ match the entries of `p` (up to the ordering).

`K = place(A,B,p)` computes a feedback gain matrix `K` that achieves the desired closed-loop pole locations `p`, assuming all the inputs of the plant are control inputs. The length of `p` must match the row size of `A`. `place` works for multi-input systems and is based on the algorithm from [1]. This algorithm uses the extra degrees of freedom to find a solution that minimizes the sensitivity of the closed-loop poles to perturbations in A or B.

`[K,prec,message] = place(A,B,p)` also returns `prec`, an estimate of how closely the eigenvalues of $A - BK$ match the specified locations `p` (`prec` measures the number of accurate decimal digits in the actual closed-loop poles). If some nonzero closed-loop pole is more than 10% off from the desired location, `message` contains a warning message.

You can also use `place` for estimator gain selection by transposing the `A` matrix and substituting `C'` for `B`.

```
l = place(A',C',p).'
```

Example

Consider a state-space system `(a,b,c,d)` with two inputs, three outputs, and three states. You can compute the feedback gain matrix needed to place the closed-loop poles at `p = [1.1 23 5.0]` by

```
p = [1 1.23 5.0];
K = place(a,b,p)
```

Algorithm

`place` uses the algorithm of [1] which, for multi-input systems, optimizes the choice of eigenvectors for a robust solution. We recommend `place` rather than `acker` even for single-input systems.

In high-order problems, some choices of pole locations result in very large gains. The sensitivity problems attached with large gains suggest caution in the use of pole placement techniques. See [2] for results from numerical testing.

References

[1] Kautsky, J. and N.K. Nichols, "Robust Pole Assignment in Linear State Feedback," *Int. J. Control,* 41 (1985), pp. 1129-1155.

[2] Laub, A.J. and M. Wette, *Algorithms and Software for Pole Assignment and Observers*, UCRL-15646 Rev. 1, EE Dept., Univ. of Calif., Santa Barbara, CA, Sept. 1984.

See Also

`acker`, `lqr`, `rlocus`

pole

Purpose

Compute poles of LTI system

Syntax

`pole`

Description

`pole` computes the poles `p` of the SISO or MIMO LTI model `sys`.

Algorithm

For state-space models, the poles are the eigenvalues of the A matrix, or the generalized eigenvalues of $A - \lambda E$ in the descriptor case.

For SISO transfer functions or zero-pole-gain models, the poles are simply the denominator roots (see `roots`).

For MIMO transfer functions (or zero-pole-gain models), the poles are computed as the union of the poles for each SISO entry. If some columns or rows have a common denominator, the roots of this denominator are counted only once.

Limitations

Multiple poles are numerically sensitive and cannot be computed to high accuracy. A pole λ with multiplicity m typically gives rise to a cluster of computed poles distributed on a circle with center λ and radius of order

$$\rho \approx \varepsilon^{1/m}$$

where ε is the relative machine precision (`eps`).

See Also

`damp`, `esort`, `dsort`, `pzmap`, `zero`

Purpose

Compute pole-zero map of LTI models

Syntax

```
pzmap(sys)
pzmap(sys1,sys2,...,sysN)
[p,z] = pzmap(sys)
```

Description

`pzmap(sys)` plots the pole-zero map of the continuous- or discrete-time LTI model `sys`. For SISO systems, `pzmap` plots the transfer function poles and zeros. For MIMO systems, it plots the system poles and transmission zeros. The poles are plotted as `x`'s and the zeros are plotted as `o`'s.

`pzmap(sys1,sys2,...,sysN)` plots the pole-zero map of several LTI models on a single figure. The LTI models can have different numbers of inputs and outputs and can be a mix of continuous and discrete systems.

When invoked with left-hand arguments,

```
[p,z] = pzmap(sys)
```

returns the system poles and (transmission) zeros in the column vectors `p` and `z`. No plot is drawn on the screen.

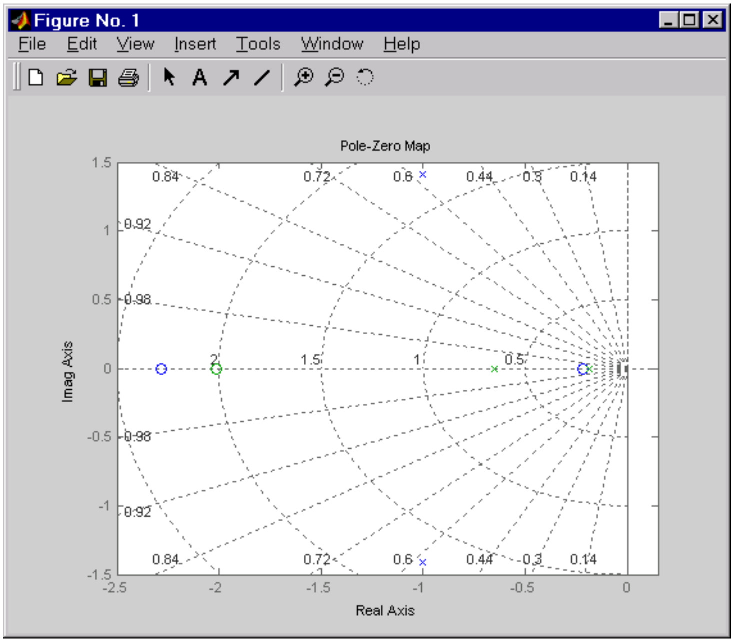
You can use the functions `sgrid` or `zgrid` to plot lines of constant damping ratio and natural frequency in the s- or z-plane.

Example

Plot the poles and zeros of the continuous-time system.

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

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



Algorithm

`pzmap` uses a combination of `pole` and `zero`.

See Also

`damp`, `esort`, `dsort`, `pole`, `rlocus`, `sgrid`, `zgrid`, `zero`

Purpose

Plot pole-zero map of LTI model and return plot handle

Syntax

```
h = pzplot(sys)
pzplot(sys1,sys2,...)
pzplot(AX,...)
pzplot(..., plotoptions)
```

Description

`h = pzplot(sys)` computes the poles and (transmission) zeros of the LTI 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 LTI 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 LTI models, `pzmap` plots the poles and zeros of each model in the array on the same diagram.

Example

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`

Purpose Real part of frequency response for FRD model

Syntax `realfrd = real(sys)`

Description `realfrd = real(sys)` computes the real part of the frequency response contained in the FRD model `sys`, including the contribution of input, output, and I/O delays. The output `realfrd` is an FRD object containing the values of the real part across frequencies.

See Also `frd/abs`, `frd/imag`

reg

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.

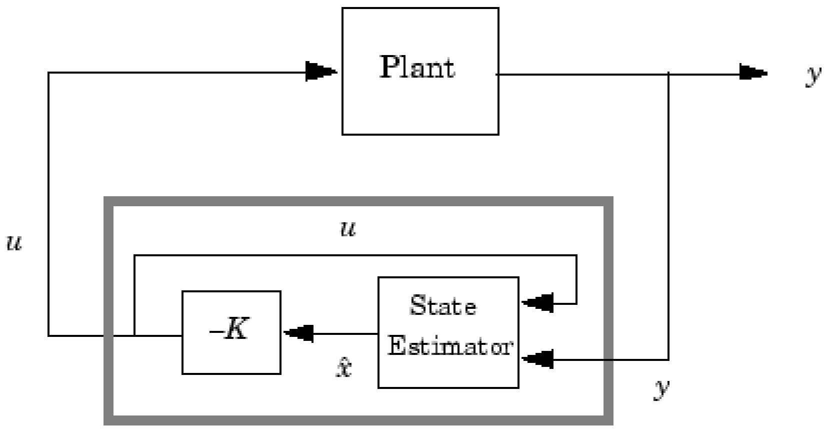
This syntax assumes that all inputs of `sys` are controls, and all outputs are measured. The regulator `rsys` is obtained by connecting the state-feedback law $u = -Kx$ and the state estimator with gain matrix `L` (see `estim`). For a plant with equations

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

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.

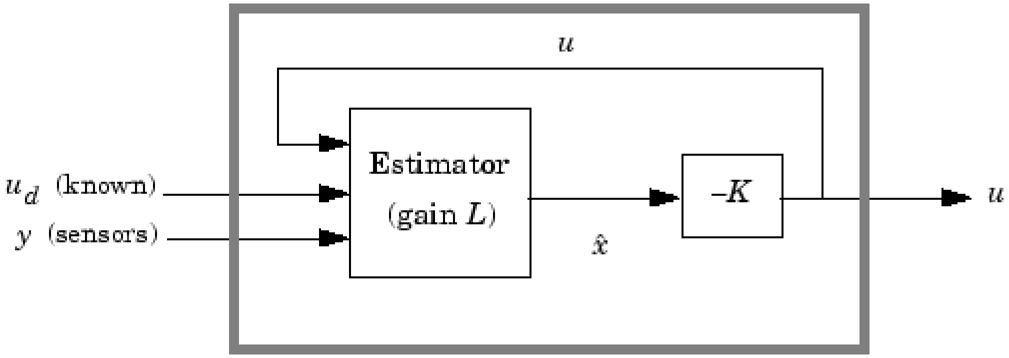


Regulator

`rsys = reg(sys,K,L,sensors,known,controls)` handles more general regulation problems where:

- The plant inputs consist of controls u, known inputs u_d, and stochastic inputs w.
- Only a subset y of the plant outputs is measured.

The index vectors `sensors`, `known`, and `controls` specify y, u_d, and u as subsets of the outputs and inputs of `sys`. The resulting regulator uses $[u_d\ ;\ y]$ as inputs to generate the commands u (see figure below).



Regulator `rsys`

Example

Given a continuous-time state-space model

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

with seven outputs and four inputs, suppose you have designed:

- A state-feedback controller gain `K` using inputs 1, 2, and 4 of the plant as control inputs
- A state estimator with gain `L` using outputs 4, 7, and 1 of the plant as sensors, and input 3 of the plant as an additional known input

You can then connect the controller and estimator and form the complete regulation system by

```
controls = [1,2,4];
sensors = [4,7,1];
known = [3];
regulator = reg(sys,K,L,sensors,known,controls)
```

See Also

`estim`, `kalman`, `lqgreg`, `lqr`, `dlqr`, `place`

Purpose

Change shape of LTI array

Syntax

```
sys = reshape(sys,s1,s2,...,sk)
sys = reshape(sys,[s1 s2 ... sk])
```

Description

`sys = reshape(sys,s1,s2,...,sk)` (or, equivalently, `sys = reshape(sys,[s1 s2 ...  sk])`) reshapes the LTI array sys into an `s1`-by-`s2`-by...-`sk` array of LTI models. Equivalently, `sys = reshape(sys,[s1 s2 ...  sk])` reshapes the LTI array sys into an `s1`-by-`s2`-by...-`sk` array of LTI models. With either syntax, there must be `s1*s2*...*sk` models in sys to begin with.

Example

```
sys = rss(4,1,1,2,3);
size(sys)
2x3 array of state-space models
Each model has 1 output, 1 input, and 4 states.
sys1 = reshape(sys,6);
size(sys1)
6x1 array of state-space models
Each model has 1 output, 1 input, and 4 states.
```

See Also

`ndims`, `size`

Purpose

Evans root locus

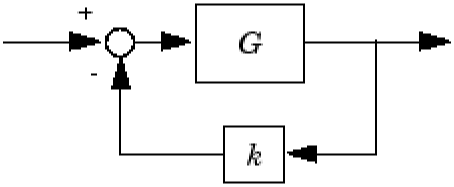
Syntax

```
rlocus
rlocus(sys)
rlocus(sys1,sys2,...)
[r,k] = rlocus(sys)
r = rlocus(sys,k)
```

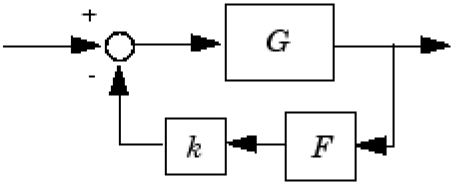
Description

`rlocus` computes the Evans root locus of a SISO open-loop model. The root locus gives the closed-loop pole trajectories as a function of the feedback gain k (assuming negative feedback). Root loci are used to study the effects of varying feedback gains on closed-loop pole locations. In turn, these locations provide indirect information on the time and frequency responses.

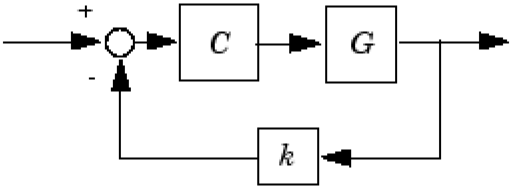
`rlocus(sys)` calculates and plots the root locus of the open-loop SISO model `sys`. This function can be applied to any of the following *negative* feedback loops by setting `sys` appropriately.



`sys = G`



`sys = F * G`



`sys = G * C`

If `sys` has transfer function

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

the closed-loop poles are the roots of

$$d(s) + k\ n(s) = 0$$

`rlocus` adaptively selects a set of positive gains k to produce a smooth plot. Alternatively,

```
rlocus(sys,k)
```

uses the user-specified vector `k` of gains to plot the root locus.

`rlocus(sys1,sys2,...)` draws the root loci of multiple LTI models `sys1, sys2,...` on a single plot. You can specify a color, line style, and marker for each model, as in

```
rlocus(sys1,'r',sys2,'y:',sys3,'gx').
```

When invoked with output arguments,

```
[r,k] = rlocus(sys)
r = rlocus(sys,k)
```

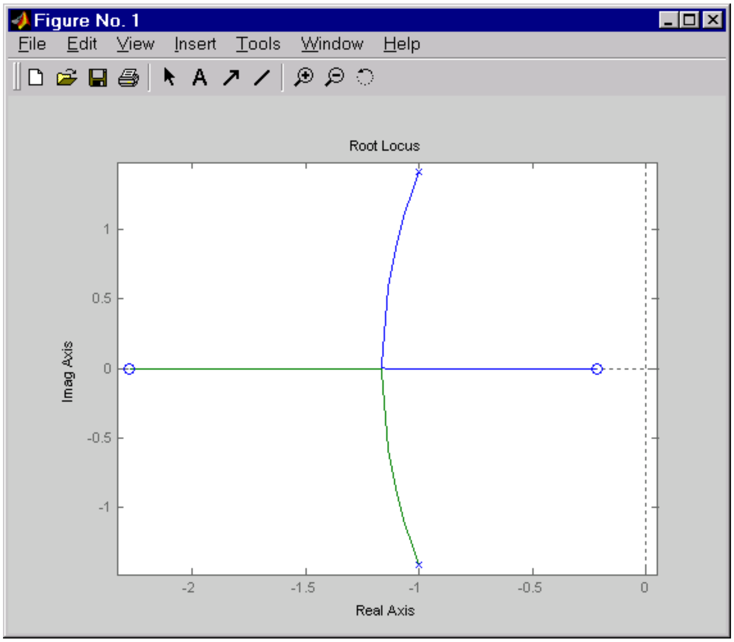
return the vector `k` of selected gains and the complex root locations `r` for these gains. The matrix `r` has `length(k)` columns and its `j`th column lists the closed-loop roots for the gain `k(j)`.

Example

Find and plot the root-locus of the following system.

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

```
h = tf([2 5 1],[1 2 3]);
rlocus(h)
```



You can use the right-click menu for rlocus to add grid lines, zoom in or out, and invoke the Property Editor to customize the plot. Also, click anywhere on the curve to activate a data marker that displays the gain value, pole, damping, overshoot, and frequency at the selected point.

See Also

`pole`, `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.

Example 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`, `setoptions`

Purpose

Generate stable random continuous test model

Syntax

```
rss(n)
rss(n,p)
rss(n,p,m,s1,...,sn)produces
```

Description

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

`rss(n,p)` produces a random `nth` order stable model with one input and `p` outputs, and `rss(n,p,m)` produces a random `n-th` order stable model with `m` inputs and `p` outputs. The output `sys` is always a state-space model.

`rss(n,p,m,s1,...,sn)produces` an s1-by-...-by-sn array of random n-th order stable state-space models with `m` inputs and `p` outputs.

Use `tf`, `frd`, or `zpk` to convert the state-space object `sys` to transfer function, frequency response, or zero-pole-gain form.

Example

Obtain a stable random continuous LTI model with three states, two inputs, and two outputs by typing

```
sys = rss(3,2,2)
a =
                          x1          x2          x3
              x1    -0.54175     0.09729     0.08304
              x2     0.09729    -0.89491     0.58707
              x3     0.08304     0.58707    -1.95271

b =
                          u1          u2
              x1    -0.88844    -2.41459
              x2           0    -0.69435
              x3    -0.07162    -1.39139

c =
                          x1          x2          x3
```

```
           y1       0.32965       0.14718             0
           y2       0.59854      -0.10144       0.02805


d =
                         u1            u2
           y1      -0.87631      -0.32758
           y2             0             0

Continuous-time system.
```

See Also `drss`, `frd`, `tf`, `zpk`

Purpose

Series connection of two LTI models

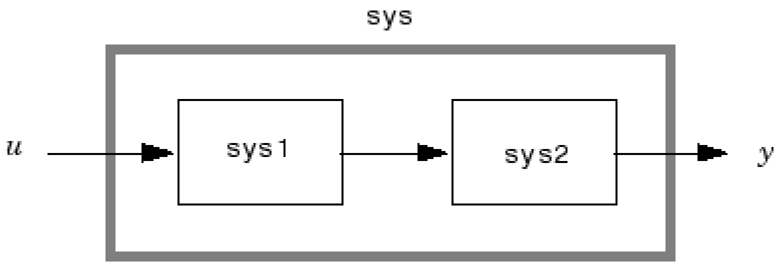
Syntax

```
series
sys = series(sys1,sys2)
sys = series(sys1,sys2,outputs1,inputs2)
```

Description

`series` connects two LTI models in series. This function accepts any type of LTI model. The two systems must be either both continuous or both discrete with identical sample time. Static gains are neutral and can be specified as regular matrices.

`sys = series(sys1,sys2)` forms the basic series connection shown below.

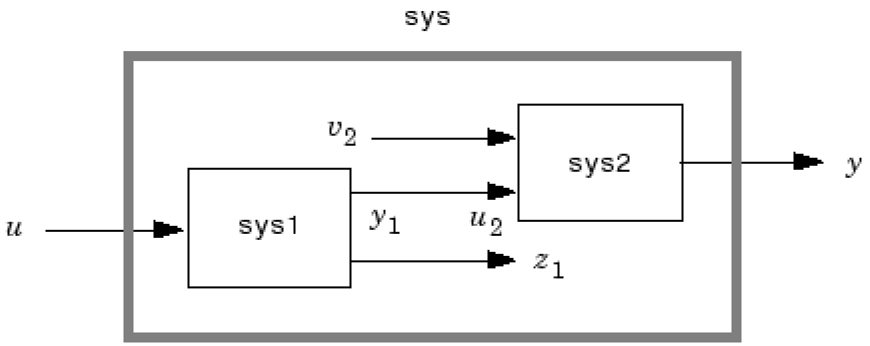


This command is equivalent to the direct multiplication

```
sys = sys2 * sys1
```

See Multiplication for details on multiplication of LTI models.

`sys = series(sys1,sys2,outputs1,inputs2)` forms the more general series connection.



The index vectors `outputs1` and `inputs2` indicate which outputs y_1 of `sys1` and which inputs u_2 of `sys2` should be connected. The resulting model `sys` has u as input and y as output.

Example

Consider a state-space system `sys1` with five inputs and four outputs and another system `sys2` with two inputs and three outputs. Connect the two systems in series by connecting outputs 2 and 4 of `sys1` with inputs 1 and 2 of `sys2`.

```
outputs1 = [2 4];
inputs2 = [1 2];
sys = series(sys1,sys2,outputs1,inputs2)
```

See Also

`append`, `feedback`, `parallel`

Purpose

Set or modify LTI model properties

Syntax

```
set
set(sys,'Property',Value)
set(sys,'Property1',Value1,'Property2',Value2,...)
set(sys,'Property')
set(sys)
```

Description

`set` is used to set or modify the properties of an LTI model (see LTI Properties for background on LTI properties). Like its Handle Graphics counterpart, `set` uses property name/property value pairs to update property values.

`set(sys,'Property',Value)` assigns the value `Value` to the property of the LTI model `sys` specified by the string `'Property'`. This string can be the full property name (for example, `'UserData'`) or any unambiguous case-insensitive abbreviation (for example, `'user'`). The specified property must be compatible with the model type. For example, if `sys` is a transfer function, `Variable` is a valid property but `StateName` is not (see Model-Specific Properties for details).

`set(sys,'Property1',Value1,'Property2',Value2,...)` sets multiple property values with a single statement. Each property name/property value pair updates one particular property.

`set(sys,'Property')` displays admissible values for the property specified by `'Property'`. See “Property Values” on page 2-248 below for an overview of legitimate LTI property values.

`set(sys)` displays all assignable properties of `sys` and their admissible values.

Example

Consider the SISO state-space model created by

```
sys = ss(1,2,3,4);
```

You can add an input delay of 0.1 second, label the input as `torque`, reset the D matrix to zero, and store its DC gain in the `'Userdata'` property by

```
set(sys,'inputd',0.1,'inputn','torque','d',0,'user',dcgain(sys)
)
```

Note that `set` does not require any output argument. Check the result with `get` by typing

```
get(sys)
                  a: 1
                  b: 2
                  c: 3
                  d: 0
                  e: []
          StateName: {''}
      InternalDelay: [0x1 double]
                 Ts: 0
         InputDelay: 0.1
        OutputDelay: 0
          InputName: {'torque'}
         OutputName: {''}
         InputGroup: [1x1 struct]
        OutputGroup: [1x1 struct]
               Name: ''
              Notes: {}
           UserData: -2
```

Property Values

The following table lists the admissible values for each LTI property. N_u and N_y denotes the number of inputs and outputs of the underlying LTI model. For *K*-dimensional LTI arrays, let S_1, S_2, …, S_K denote the array dimensions.

LTI Properties

Property Name	Admissible Property Values
`Ts`	• `0` (zero) for continuous-time systems • Sample time in seconds for discrete-time systems • `-1` or `[]` for discrete systems with unspecified sample time **Note:** Resetting the sample time property does not alter the model data. Use `c2d`, `d2c`, or `d2d` for discrete/continuous and discrete/discrete conversions.
`InputDelay`	Input delays specified with • Nonnegative real numbers for continuous-time models (seconds) • Integers for discrete-time models (number of sample periods) • Scalar when $N_u = 1$ or system has uniform input delay • Vector of length N_u to specify independent delay times for each input channel • Array of size N_y-by- N_u-by- S_1-by-. . .-by-S_n to specify different input delays for each model in an LTI array.

Property Name	Admissible Property Values
`OutputDelay`	delays specified withOutput - Nonnegative real numbers for continuous-time models (seconds) - Integers for discrete-time models (number of sample periods) - Scalar when $N_y = 1$ or system has uniform output delay - Vector of length N_y to specify independent delay times for each output channel - Array of size N_y-by- N_u-by- S_1-by-. . .-by-S_n to specify different output delays for each model in an LTI array.
`InputName`	- String for single-input systems, for example, `'thrust'` - Cell vector of strings for multi-input systems (with as many cells as inputs), for example, `{'u';'w'}` for a two-input system - Padded array of strings with as many rows as inputs, for example, `['rudder ' ; 'aileron']`
`OutputDelay`	Same as `InputDelay`
`Notes`	String, array of strings, or cell array of strings
`UserData`	Arbitrary MATLAB variable

State-Space Model Properties

Property Name	Admissible Property Values
`StateName`	Same as `InputName` (with `Input` replaced by `State`)
`a`, `b`, `c`, `d`, `e`	Real- or complex-valued state-space matrices (multidimensional arrays, in the case of LTI arrays) with compatible dimensions for the number of states, inputs, and outputs. See The Size of LTI Array Data for SS Models.
`InternalDelay`	This property contains internal representations of delays in state-space. Internal delays in SS objects are creating when converting from ZPK or TF objects with I/O delays. We do not recommmend using `set` to modify this property. See Time Delays for more information.

TF Model Properties

Property Name	Admissible Property Values
`num`, `den`	• Real- or complex-valued row vectors for the coefficients of the numerator or denominator polynomials in the SISO case. List the coefficients in *descending* powers of the variable s or z by default, and in *ascending* powers of $q = z^{-1}$ when the `Variable` property is set to `'q'` or `'z^-1'` (see note below). • N_y-by-N_u cell arrays of real- or complex-valued row vectors in the MIMO case, for example, `{[1 2];[1 0 3]}` for a two-output/one-input transfer function • N_y-by-N_u-by-S_1-by-...-by-S_K-dimensional real- or complex-valued cell arrays for MIMO LTI arrays

Property Name	Admissible Property Values
`Variable`	• String `'s'` (default) or `'p'` for continuous-time systems • String `'z'` (default), `'q'`, or `'z^-1'` for discrete-time systems
`ioDelay`	• An matrix of dimension N_y-by-N_u, where N_y is the number of outputs and N_u is the number of inputs. • If you have an LTI array, using an N_y-by-N_u matrix populates all the LTI models in the LTI array with the specified `ioDelay` matrix. To specify I/O delays for individual models in the LTI array, use an N_y-by-N_u-by-S_1-by-$\ldots$-by-S_K-array, where S_1, ..., S_K are the dimensions of the LTI array.

ZPK Model Properties

Property Name	Admissible Property Values
`z`, `p`	• Vectors of zeros and poles (either real- or complex-valued) in SISO case • N_y-by-N_u cell arrays of vectors (entries are real- or complex valued) in MIMO case, for example, `z = {[],[-1 0]}` for a model with two inputs and one output • N_y-by-N_u-by-S_1-by-...-by-S_K-dimensional cell arrays for MIMO LTI arrays
`ioDelay`	• A matrix of dimension N_y-by-N_u , where N_y is the number of outputs and N_u is the number of inputs. • If you have an LTI array, using an N_y-by-N_u matrix populates all the LTI models in the LTI array with the specified `ioDelay` matrix. To specify I/O delays for individual models in the LTI array, use an N_y-by-N_u-by-S_1-by-...-by-S_K- array, where S_1, ..., S_K are the dimensions of the LTI array.
`Variable`	• String `'s'` (default) or `'p'` for continuous-time systems • String `'z'` (default), `'q'`, or `'z^-1'` for discrete-time systems

FRD Model Properties

`Property Name`	**Admissible Property Values**
`Frequency`	Real-valued vector of length N_f-by-1, where N_f is the number of frequencies
`Response`	• N_y-by-N_u-by-N_f-dimensional array of complex data for single LTI models • N_y-by-N_u-by-N_f-by-S_1-by-$\ldots$-by-S_K-dimensional array for LTI arrays
`Units`	String `'rad/s'` (default), or `'Hz'`
`ioDelay`	• An matrix of dimension N_y-by-N_u , where N_y is the number of outputs and N_u is the number of inputs. • If you have an LTI array, using an N_y-by-N_u matrix populates all the LTI models in the LTI array with the specified `ioDelay` matrix. To specify I/O delays for individual models in the LTI array, use an N_y-by-N_u-by-S_1-by-$\ldots$-by-S_K- array, where S_1, ..., S_K are the dimensions of the LTI array.

Remark

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'` (or `'q'`) by

```
set(h,'Variable','z^-1'),
```

the same command

```
set(h,'num',[1 2],'den',[1 3 4])
```

now interprets the row vectors `[1 2]` and `[1 3 4]` as the polynomials $1 + 2z^{-1}$ and $1 + 3z^{-1} + 4z^{-2}$ and produces:

$$\bar{h}(z^{-1}) = \frac{1+2z^{-1}}{1+3z^{-1}+4z^{-2}} = zh(z)$$

Note Because the resulting transfer functions are different, make sure to use the convention consistent with your choice of variable.

See Also

`get`, `frd`, `ss`, `tf`, `zpk`

Purpose

Create internal delays of state-space model

Syntax

```
sys = setdelaymodel(A,B1,B2,C1,C2,D11,D12,D21,D22,tau)
sys = setdelaymodel(H,tau)
```

Description

`setdelaymodel` is the converse of getdelaymodel. You can use it to directly specify the internal representation of state-space models with internal delays. See `getdelaymodel` for more details on this internal representation. `setdelaymodel` is an advanced operation and is not the natural way to construct models with internal delays. See Time Delays for recommended ways of creating internal delays.

`sys = setdelaymodel(A,B1,B2,C1,C2,D11,D12,D21,D22,tau)` constructs the state-space model `sys` defined by the matrices `A`,`B1`,`B2`, ... and the vector of internal delays `TAU`. The resulting model is continuous and can be made discrete by modifying its sample time.

`sys = setdelaymodel(H,tau)` constructs the state-space model `sys` obtained by LFT interconnection of the state-space model `H` with the bank of internal delays `tau`.

See Also

`getdelaymodel`

setoptions

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 properties and values are available, type `help <`*`function`*`>options`. For example, for Bode plots type

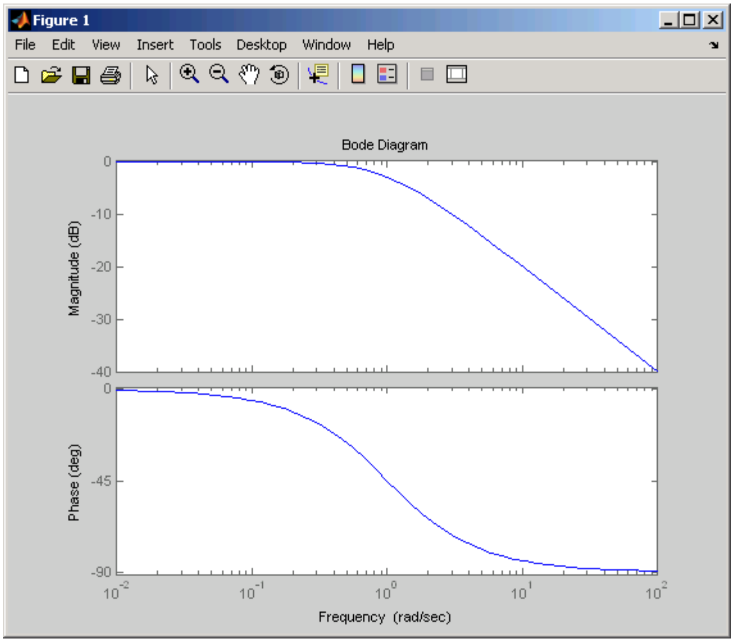
```
help bodeoptions
```

`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

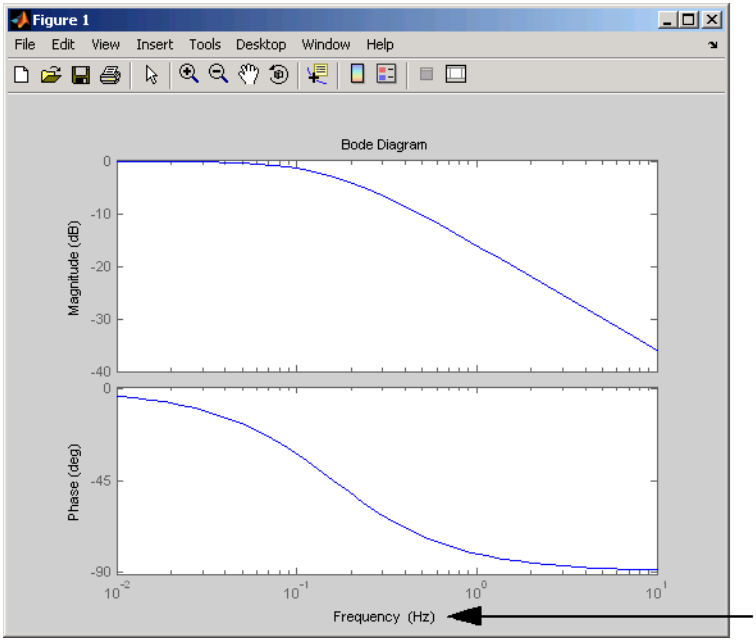
To change frequency units, first create a Bode plot.

```
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`

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.

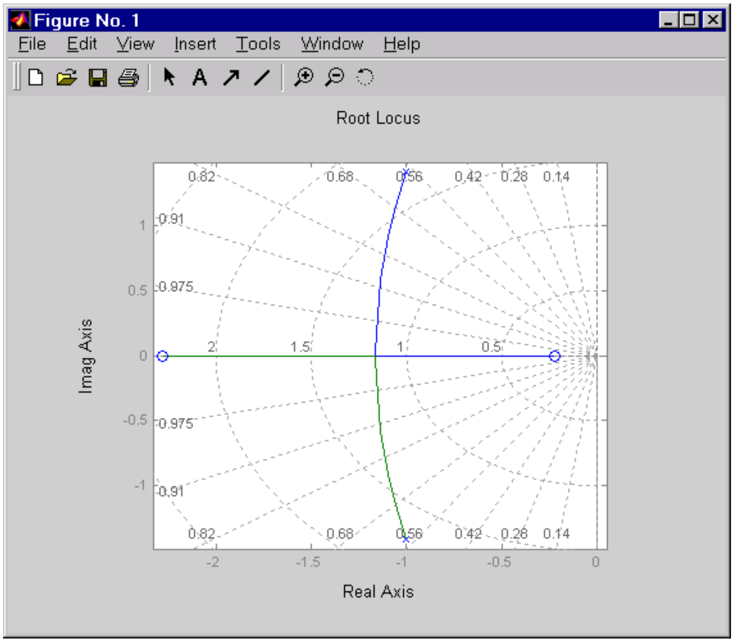
Example

Plot *s*-plane grid lines on the root locus for the following system.

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

You can do this by typing

```
H = tf([2 5 1],[1 2 3])
Transfer function:
2 s^2 + 5 s + 1
---------------
 s^2 + 2 s + 3
rlocus(H)
sgrid
```



See Also `pzmap`, `rlocus`, `zgrid`

Purpose

Plot singular values of LTI models

Syntax

```
sigma
sigma(sys)
sigma(sys,w)
sigma(sys,[],type)
sigma(sys,w,type)
```

Description

`sigma` calculates the singular values of the frequency response of an LTI model. For an FRD model, `sys`, `sigma` computes the singular values of `sys.Response` at the frequencies, `sys.frequency`. For continuous-time TF, SS, or ZPK models with transfer function $H(s)$, `sigma` computes the singular values of $H(j\omega)$ as a function of the frequency ω. For discrete-time TF, SS, or ZPK models with transfer function $H(z)$ and sample time T_s, `sigma` computes the singular values of

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

for frequencies ω between 0 and the Nyquist frequency $\omega_N = \pi/T_s$.

The singular values of the frequency response extend the Bode magnitude response for MIMO systems and are useful in robustness analysis. The singular value response of a SISO system is identical to its Bode magnitude response. When invoked without output arguments, `sigma` produces a singular value plot on the screen.

`sigma(sys)` plots the singular values of the frequency response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. The frequency points are chosen automatically based on the system poles and zeros, or from `sys.frequency` if `sys` is an FRD.

`sigma(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the corresponding vector of frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. The frequencies must be specified in rad/sec.

`sigma(sys,[],type)` or `sigma(sys,w,type)` plots the following modified singular value responses:

`type = 1`	Singular values of the frequency response H^{-1}, where H is the frequency response of `sys`.
`type = 2`	Singular values of the frequency response $I + H$.
`type = 3`	Singular values of the frequency response $I + H^{-1}$.

These options are available only for square systems, that is, with the same number of inputs and outputs.

To superimpose the singular value plots of several LTI models on a single figure, use

```
sigma(sys1,sys2,...,sysN)
sigma(sys1,sys2,...,sysN,[],type)  % modified SV plot
sigma(sys1,sys2,...,sysN,w)       % specify frequency range/grid
```

The models `sys1,sys2,...,sysN` need not have the same number of inputs and outputs. Each model can be either continuous- or discrete-time. You can also specify a distinctive color, linestyle, and/or marker for each system plot with the syntax

```
sigma(sys1,'PlotStyle1',...,sysN,'PlotStyleN')
```

See `bode` for an example.

When invoked with output arguments,

```
[sv,w] = sigma(sys)
sv = sigma(sys,w)
```

return the singular values `sv` of the frequency response at the frequencies `w`. For a system with `Nu` input and `Ny` outputs, the array `sv` has `min(Nu,Ny)` rows and as many columns as frequency points (length of `w`). The singular values at the frequency `w(k)` are given by `sv(:,k)`.

Example

Plot the singular value responses of

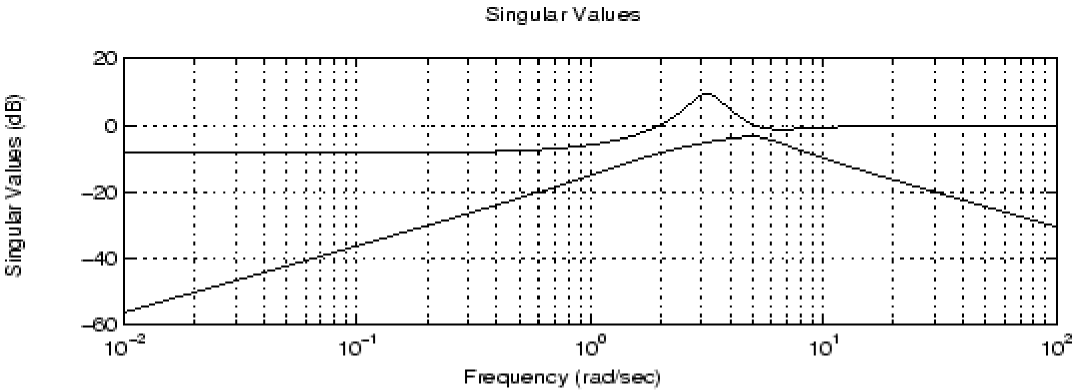
$$H(s) = \begin{bmatrix} 0 & \dfrac{3s}{s^2 + s + 10} \\ \dfrac{s+1}{s+5} & \dfrac{2}{s+6} \end{bmatrix}$$

and $I + H(s)$.

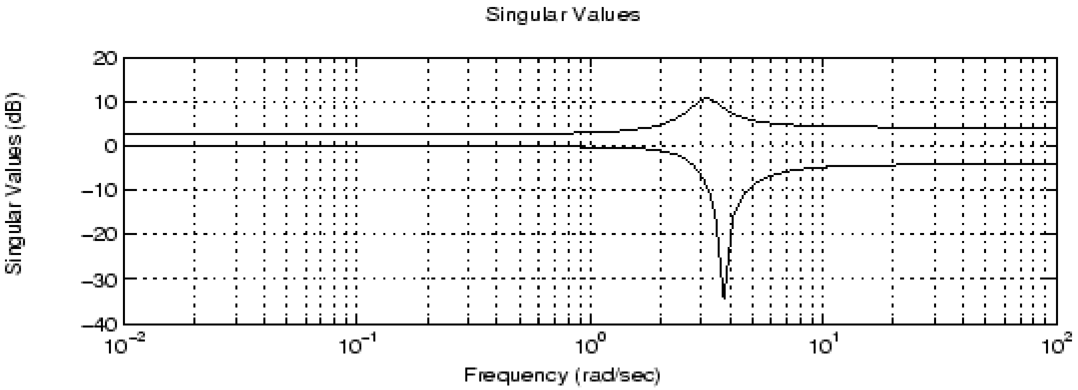
You can do this by typing

```
H = [0 tf([3 0],[1 1 10]) ; tf([1 1],[1 5]) tf(2,[1 6])]

subplot(211)
sigma(H)
subplot(212)
sigma(H,[],2)
```



Singular Values



Singular Values

Algorithm

`sigma` uses the `svd` function in MATLAB to compute the singular values of a complex matrix.

See Also

`bode`, `evalfr`, `freqresp`, `ltiview`, `nichols`, `nyquist`

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)
```

Discussion

`h = sigmaplot(sys)` produces a singular value (SV) plot of the frequency response of the LTI model `sys` (created with `tf`, `zpk`, `ss`, or `frd`). 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/s).

`sigmaplot(sys,w)` uses the user-supplied vector `w` of frequencies, in rad/s, 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.

Example

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`

Purpose

Configure SISO Design Tool at startup

Syntax

`T = sisoinit(CONFIG)`

Description

`T = sisoinit(CONFIG)` returns a template T for initializing Graphical Tuning window of the SISO Design Tool with a particular control system configuration `CONFIG`. Available configurations include:

- `CONFIG=1` — 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

This figure shows the six configurations in order.

For each configuration, you can specify the plant models G,H, initialize the compensator C and prefilter F, and configure the open- and closed-loop views by filling the corresponding fields of the structure T. Then use `sisotool(T)` to start the SISO Design Tool in the specified configuration.

Output argument `T` is an object with object properties. These tables list the block and loop properties.

Block Properties

Block	Properties	Values

`F`	`Name`
	`Description`
	`Value`
		String
		String
		LTI object

`G`	`Name` `Value`	String LTI object

Block	Properties	Values
`H`	`Name` `Value`	String LTI object

`C`	`Name`
	`Descripton`
	`Value`
		String
		String
		LTI object

Loop Properties

Loops	Properties	Values

`OL1`	`Name`
	`Description`
	`View`
		String
		String
		`'rlocus'  'bode'`

`CL1`	`Name`
	`Description`
	`View`
		String
		String
		`'bode'`

Example

```
T = sisoinit(2);          % Single-loop configuration with
                          % C in the feedback path
T.G.Value = rss(3);       % Model for plant G
T.C.Value = tf(1,[1 2]);  % Initial compensator value
T.OL1.View = {'rlocus','nichols'}; % Views for tuning Open-Loop
                                   % OL1
% Now launch SISO Design Tool using configuration T
sisotool(T)
```

See Also

`sisotool`

Purpose

Initialize SISO Design Tool

Syntax

```
sisotool(plant)
sisotool(plant,comp)
sisotool(plant,comp,sensor,prefilt)
sisotool(views)
sisotool(views,plant,comp)
sisotool(initdata)
sisotool(sessiondata)
```

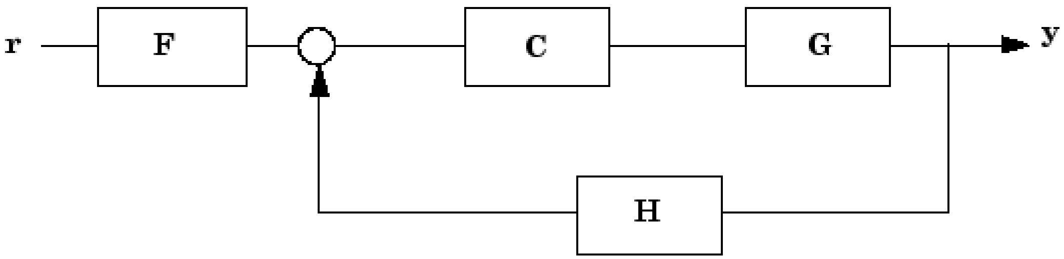
Description

When invoked without input arguments, `sisotool` opens a SISO Design GUI for interactive compensator design. This GUI allows you to design a single-input/single-output (SISO) compensator using root locus, Bode diagram, Nicholse and Nyquist techniques. You can also have the SISO Design Tool automatically design a compensator.

By default, the SISO Design Tool:

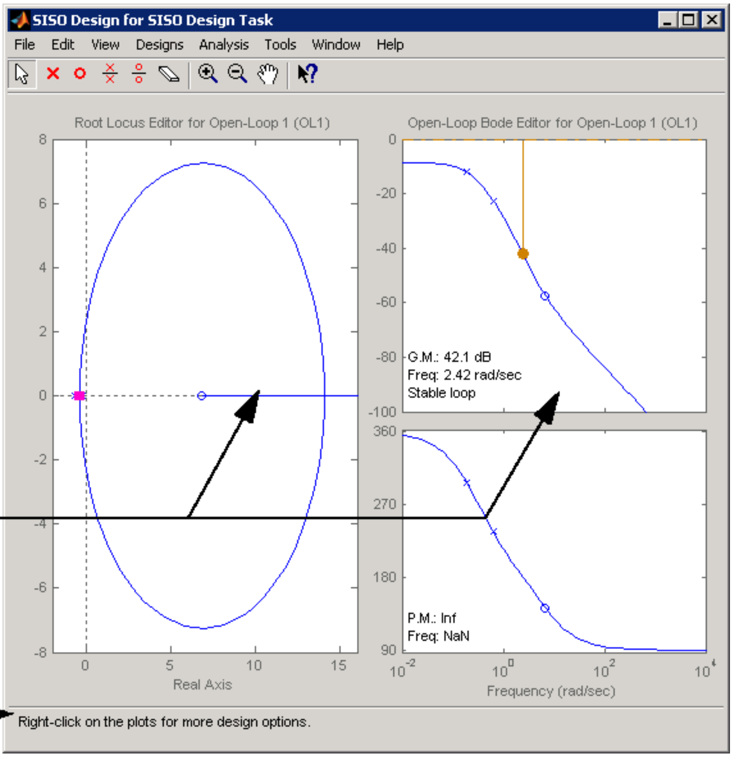
- Opens the Controls and Estimation Tools Manager with a default SISO Design Task node.
- Opens the Graphical Tuning editor with root locus and open-loop Bode diagrams.
- Places the compensator, **C**, in the forward path in series with the plant, **G**.
- Assumes the prefilter, **F**, and the sensor, **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 four control architectures available. See `sisoinit` for more information.

This picture shows the SISO Design Graphical editor.



Right-click on the plots for more design options.

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`. The workspace variable `plant` can be any SISO LTI model created with `ss`, `tf`, or `zpk`.

`sisotool(plant,comp)` initializes the plant model **G** to `plant`, the compensator **C** to `comp`.

`sisotool(plant,comp,sensor,prefilt)` initializes the plant **G** to `plant`, compensator C to `comp`, sensor H to `sensor`, and the prefilter F to `prefilt`. All arguments must be SISO LTI objects.

`sisotool(views)` or `sisotool(`*`views`*`,plant,comp)` specifies the initial configuration of the SISO Design Tool. The argument *`views`* can be any of the following strings (or combination thereof):

- `'rlocus'` — Root Locus plot
- `'bode'` — Bode diagrams of the open-loop response
- `'nichols'` — Nichols plot
- `'filter'` — Bode diagrams of the prefilter **F** and the closed-loop response from the command into **F** to the output of the compensator **G** (see the feedback structure figure below)

For example

```
sisotool('bode')
```

opens a SISO Design Tool with only the Bode Diagrams. Note that if there is more than one view, the views are stored in a cell array.

`sisotool(initdata)` initializes the SISO Design Tool with more general control system configurations. Use `sisoinit` to build 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.

For more details on the SISO Design Tool, see Designing Compensators in the *Control System Toolbox Getting Started* guide.

See Also `bode`, `ltiview`, `rlocus`, `nichols`

Purpose

Provide output/input/array dimensions of LTI model, model order of TF, SS, and ZPK model, and number of frequencies of FRD model

Syntax

```
d = size(sys)
Ny = size(sys,1)
Nu = size(sys,2)
Sk = size(sys,2+k)
Ns = size(sys,'order')
Nf = size(sys,'frequency')
```

Description

When invoked without output arguments, `size(sys)` returns a vector of the number of outputs and inputs for a single LTI model. The lengths of the array dimensions are also included in the response to size when `sys` is an LTI array. `size` is the overloaded version of the MATLAB function `size` for LTI objects.

`d = size(sys)` returns:

- The row vector `d = [Ny Nu]` for a single LTI model `sys` with `Ny` outputs and `Nu` inputs
- The row vector `d = [Ny Nu S1 S2 ... Sp]` for an `S1`-by-`S2`-by-...-by-`Sp` array of LTI models with `Ny` outputs and `Nu` inputs

`Ny = size(sys,1)` returns the number of outputs of `sys`.

`Nu = size(sys,2)` returns the number of inputs of `sys`.

`Sk = size(sys,2+k)` returns the length of the `k`-th array dimension when `sys` is an LTI array.

`Ns = size(sys,'order')` returns the model order of a TF, SS, or ZPK model. This is the same as the number of states for state-space models. When `sys` is an LTI array, ns is the maximum order of all of the models in the LTI array.

`Nf = size(sys,'frequency')` returns the number of frequencies when `sys` is an FRD. This is the same as the length of `sys.frequency`.

Example

Consider the random LTI array of state-space models

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

Its dimensions are obtained by typing

```
size(sys)
3x1 array of state-space models
Each model has 3 outputs, 2 inputs, and 5 states.
```

See Also

`isempty`, `issiso`, `ndims`

Purpose Perform model reduction based on structure

Syntax

```
msys = sminreal(sys)
```

Description `msys = sminreal(sys)` eliminates the states of the state-space model `sys` that don't affect the input/output response. All of the states of the resulting state-space model `msys` are also states of `sys` and the input/output response of `msys` is equivalent to that of `sys`.

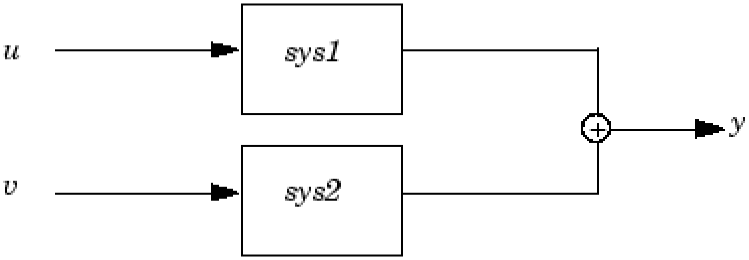
`sminreal` eliminates only structurally non minimal states, i.e., states that can be discarded by looking only at hard zero entries in the A, B, and C matrices. Such structurally nonminimal states arise, for example, when linearizing a Simulink model that includes some unconnected state-space or transfer function blocks.

Remark The model resulting from `sminreal(sys)` is not necessarily minimal, and may have a higher order than one resulting from `minreal(sys)`. However, `sminreal(sys)` retains the state structure of `sys`, while, in general, `minreal(sys)` does not.

Example Suppose you concatenate two SS models, `sys1` and `sys2`.

```
sys = [sys1,sys2];
```

This operation is depicted in the diagram below.



If you extract the subsystem `sys1` from `sys`, with

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

all of the states of `sys`, including those of `sys2` are retained. To eliminate the unobservable states from `sys2`, while retaining the states of `sys1`, type

```
sminreal(sys(1,1))
```

See Also `minreal`

Purpose

Specify state-space models or convert LTI model to state space

Syntax

```
ss
sys = ss(a,b,c,d)
sys = ss(a,b,c,d,Ts)
sys = ss(d)
sys = ss(a,b,c,d,ltisys)
sys_ss = ss(sys)
```

Description

`ss` is used to create real- or complex-valued state-space models (SS objects) or to convert transfer function or zero-pole-gain models to state space.

Creation of State-Space Models

`sys = ss(a,b,c,d)` creates the continuous-time state-space model

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

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

- `a` is an `Nx`-by-`Nx` real- or complex-valued matrix.
- `b` is an `Nx`-by-`Nu` real- or complex-valued matrix.
- `c` is an `Ny`-by-`Nx` real- or complex-valued matrix.
- `d` is an `Ny`-by-`Nu` real- or complex-valued matrix.

The output `sys` is an SS model that stores the model data (see "State-Space Models" on page 2-14). If $D = 0$, you can simply set `d` to the scalar `0` (zero), regardless of the dimension.

`sys = ss(a,b,c,d,Ts)` creates the discrete-time model

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

with sample time `Ts` (in seconds). Set `Ts = -1` or `Ts = []` to leave the sample time unspecified.

`sys = ss(d)` specifies a static gain matrix D and is equivalent to

```
sys = ss([],[],[],d)
```

`sys = ss(a,b,c,d,ltisys)` creates a state-space model with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See "Generic Properties" on page 2-26 for an overview of generic LTI properties.

See "Building LTI Arrays" on page 4-12 for information on how to build arrays of state-space models.

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

```
'PropertyName',PropertyValue
```

Each pair specifies a particular LTI property of the model, for example, the input names or some notes on the model history. See `set` and the example below for details. Note that

```
sys = ss(a,b,c,d,'Property1',Value1,...,'PropertyN',ValueN)
```

is equivalent to the sequence of commands.

```
sys = ss(a,b,c,d)
set(sys,'Property1',Value1,...,'PropertyN',ValueN)
```

Conversion to State Space

`sys_ss = ss(sys)` converts an arbitrary TF or ZPK model `sys` to state space. The output `sys_ss` is an equivalent state-space model (SS object). This operation is known as *state-space realization*.

`sys_ss = ss(sys,'minimal')` produces a state-space realization with no uncontrollable or unobservable states. This is equivalent to `sys_ss = minreal(ss(sys))`.

Algorithm

In the case of 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.

In the case of 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 Schafer 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
```

See the `balance` or `ssbal` documentation for details.

Examples

Example 1

The command

```
sys = ss(A,B,C,D,0.05,'statename',{'position' 'velocity'},...
                          'inputname','force',...
                          'notes','Created 10/15/96')
```

creates a discrete-time model with matrices A, B, C, D and sample time 0.05 second. This model has two states labeled `position` and `velocity`, and one input labeled `force` (the dimensions of A, B, C, D should be consistent with these numbers of states and inputs). Finally, a note is attached with the date of creation of the model.

Example 2

Compute a state-space realization of the transfer function

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

by typing

```
H = [tf([1 1],[1 3 3 2]) ; tf([1 0 3],[1 1 1])];
sys = ss(H);
size(sys)
State-space model with 2 outputs, 1 input, and 5 states.
```

Note that the number of states is equal to the cumulative order of the SISO entries of $H(s)$.

To obtain a minimal realization of $H(s)$, type

```
sys = ss(H,'min');
size(sys)
State-space model with 2 outputs, 1 input, and 3 states.
```

The resulting state-space model order has order three, the minimum number of states needed to represent $H(s)$. This can be seen directly by factoring $H(s)$ as the product of a first order system with a second order one.

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

See Also `dss`, `frd`, `get`, `set`, `ssdata`, `tf`, `zpk`

Purpose State coordinate transformation for state-space model

Syntax

```
sysT = ss2ss(sys,T)
```

Description Given a state-space model `sys` with equations

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

(or their discrete-time counterpart), `ss2ss` performs the similarity transformation $\bar{x} = Tx$ on the state vector x and produces the equivalent state-space model `sysT` with equations.

$$\begin{aligned} \dot{\bar{x}} &= TAT^{-1}\bar{x} + TBu \\ y &= CT^{-1}\bar{x} + Du \end{aligned}$$

`sysT = ss2ss(sys,T)` returns the transformed state-space model `sysT` given `sys` and the state coordinate transformation `T`. The model `sys` must be in state-space form and the matrix `T` must be invertible. `ss2ss` is applicable to both continuous- and discrete-time models.

Example Perform a similarity transform to improve the conditioning of the A matrix.

```
T = balance(sys.a)
sysb = ss2ss(sys,inv(T))
```

See `ssbal` for a more direct approach.

See Also `balreal`, `canon`, `ssbal`

Purpose

Balance state-space model using diagonal similarity transformation

Syntax

```
[sysb,T] = ssbal(sys,condT)
ssbal
```

Description

Given a state-space model `sys` with matrices (A, B, C, D),

```
[sysb,T] = ssbal(sys)
```

computes a diagonal similarity transformation T and a scalar α such that

$$\begin{bmatrix} TAT^{-1} & TB/\alpha \\ \alpha CT^{-1} & 0 \end{bmatrix}$$

has approximately equal row and column norms. `ssbal` returns the balanced model `sysb` with matrices

$$(TAT^{-1}, TB/\alpha, \alpha CT^{-1}, D)$$

and the state transformation $\bar{x} = Tx$ where $\bar{x}$ is the new state.

`[sysb,T] = ssbal(sys,condT)` specifies an upper bound `condT` on the condition number of T. Since balancing with ill-conditioned T can inadvertently magnify rounding errors, `condT` gives control over the worst-case roundoff amplification factor. The default value is `condT=Inf`.

`ssbal` returns an error if the state-space model `sys` has varying state dimensions.

Example

Consider the continuous-time state-space model with the following data.

$$A = \begin{bmatrix} 1 & 10^4 & 10^2 \\ 0 & 10^2 & 10^5 \\ 10 & 1 & 0 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \qquad C = \begin{bmatrix} 0.1 & 10 & 100 \end{bmatrix}$$

```
a = [1 1e4 1e2;0 1e2 1e5;10 1 0];
b = [1;1;1];
c = [0.1 10 1e2];
sys = ss(a,b,c,0)
```

Balance this model with `ssbal` by typing

```
ssbal(sys)
a =
                         x1           x2           x3
           x1             1         2500      0.39063
           x2             0          100       1562.5
           x3          2560           64            0


b =
                         u1
           x1         0.125
           x2           0.5
           x3            32


c =
                         x1           x2           x3
           y1           0.8           20        3.125


d =
                         u1
```

```
                     y1            0

   Continuous-time system.
```

Direct inspection shows that the range of numerical values has been compressed by a factor 100 and that the B and C matrices now have nearly equal norms.

Algorithm

`ssbal` uses the MATLAB function `balance` to compute T and α.

See Also

`balreal`, `ss2ss`

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-Specific Properties for more information on the format of state-space model data.

If `sys` is in descriptor form (non-emply `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 system with delay-free dynamics).

`[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]=stapsep
[G1,GNS] = stabsep(G,'abstol'ATOL,'reltol',RTOL)
[G1,G2]=stabsep(G, ...,'Mode', MODE,'Offset', ALPHA)
```

Description

`[GS,GNS]=stapsep` decomposes the LTI model 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 reponses 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`..

Mode	Continuous Time Region	Discrete Time Region
1	`Re(s)<-ALPHA*max(1,|Im(s)|)`	`1 |z| < 1-ALPHA`
2	`Re(s)> ALPHA*max(1,|Im(s)|)`	`2 |z| > 1+ALPHA`

The default values are `MODE=1` and `ALPHA=0`.

Example

Compute a stable/unstable decomposition with absolute error no larger than 1e-5 and offset 0.1:

```
h = zpk(1,[-2 -1 1 -0.001],0.1)
[hs,hns] = stabsep(h,'AbsTol',1e-5,'Offset',0.1);
```

The stable part of the decomposition has poles at -1 and -2.

```
hshs

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

`modsep`

Purpose

Build LTI array by stacking LTI models or LTI arrays along array dimensions

Syntax

```
sys = stack(arraydim,sys1,sys2,...)
```

Description

`sys = stack(arraydim,sys1,sys2,...)` produces an array of LTI models `sys` by stacking (concatenating) the LTI models (or LTI arrays) `sys1,sys2,...` along the array dimension `arraydim`. All models must have the same number of inputs and outputs (the same I/O dimensions), but the number of states can vary. The I/O dimensions are not counted in the array dimensions. See Dimensions, Size, and Shape of an LTI Array and Building LTI Arrays Using the stack Function for more information.

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

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

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

Example

If `sys1` and `sys2` are two LTI models:

- `stack(1,sys1,sys2)` produces a 2-by-1 LTI array.
- `stack(2,sys1,sys2)` produces a 1-by-2 LTI array.
- `stack(3,sys1,sys2)` produces a 1-by-1-by-2 LTI array.

Purpose Step response of LTI systems

Syntax

```
step
step(sys)
step(sys,t)
```

Description `step` calculates the unit step response of a linear system. Zero initial state is assumed in the state-space case. When invoked with no output arguments, this function plots the step response on the screen.

`step(sys)` plots the step response of an arbitrary LTI model `sys`. This model can be continuous or discrete, and SISO or MIMO. The step response of multi-input systems is the collection of step responses for each input channel. The duration of simulation is determined automatically based on the system poles and zeros.

`step(sys,t)` sets the simulation horizon explicitly. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form

```
t = 0:dt:Tfinal
```

For discrete systems, the spacing `dt` should match the sample period. For continuous systems, `dt` becomes the sample time of the discretized simulation model (see “Algorithm” on page 2-294), so make sure to choose `dt` small enough to capture transient phenomena.

To plot the step responses of several LTI models `sys1`,..., `sysN` on a single figure, use

```
step(sys1,sys2,...,sysN)
step(sys1,sys2,...,sysN,t)
```

All systems must have the same number of inputs and outputs but may otherwise be a mix of continuous- and discrete-time systems. This syntax is useful to compare the step responses of multiple systems.

You can also specify a distinctive color, linestyle, and/or marker for each system. For example,

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

plots the step response of `sys1` with a dotted yellow line and the step response of `sys2` with a green dashed line.

When invoked with output arguments,

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

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x` (for state-space models only). No plot is drawn on the screen. For single-input systems, `y` has as many rows as time samples (length of `t`), and as many columns as outputs. In the multi-input case, the step responses of each input channel are stacked up along the third dimension of `y`. The dimensions of `y` are then

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

and `y(:,:,j)` gives the response to a unit step command injected in the `j`th input channel. Similarly, the dimensions of `x` are

$$(\text{length of } \mathtt{t}) \times (\text{number of states}) \times (\text{number of inputs})$$

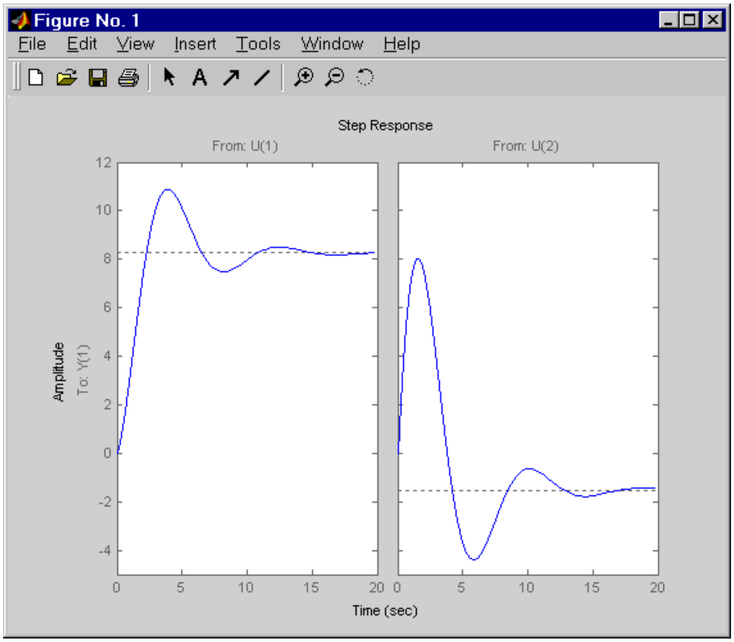
Example Plot the step response of the following second-order state-space model.

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

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

```
a = [-0.5572   -0.7814;0.7814  0];
b = [1 -1;0 2];
c = [1.9691  6.4493];
```

```
sys = ss(a,b,c,0);
step(sys)
```



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

Algorithm

Continuous-time models are converted to state space and discretized using zero-order hold on the inputs. The sampling period is chosen automatically based on the system dynamics, except when a time vector `t = 0:dt:Tf` is supplied (`dt` is then used as sampling period).

See Also

`impulse`, `initial`, `lsim`, `ltiview`

Purpose

Compute 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) - yfinal|` 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`.

Example

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])

S =

        RiseTime: 7.4519
    SettlingTime: 13.9326
     SettlingMin: 2.3737
     SettlingMax: 2.5203
       Overshoot: 0.8112
      Undershoot: 0
            Peak: 2.5203
        PeakTime: 15.2640
```

See Also

`step`, `lsiminfo`, `ltimodels`

Purpose

Plot step response of LTI systems and return plot handle

Syntax

```
h = stepplot(sys)
stepplot(sys,Tfinal)
stepplot(sys,t)
stepplot(sys1,sys2,...,t)
stepplot(AX,...)
stepplot(..., plotoptions)
```

Description

`h = stepplot(sys)` plots the step response of the LTI model `sys` (created with either `tf`, `zpk`, or `ss`). It also returns the plot handle `h`. You can use this handle to customize the plot with the `getoptions` and `setoptions` commands. Type

```
help timeoptions
```

for a list of available plot options.

For multiinput models, independent step commands are applied to each input channel. The time range and number of points are chosen automatically.

`stepplot(sys,Tfinal)` simulates the step response from `t=0` to the final time `t=Tfinal`. For discrete-time models with unspecified sampling time, `Tfinal` is interpreted as the number of samples.

`stepplot(sys,t)` uses the user-supplied time vector `t` for simulation. 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 for the discrete approximation to the continuous system. The step input is always assumed to start at `t=0` (regardless of `Ti`).

`stepplot(sys1,sys2,...,t)` plots the step responses of multiple LTI models `sys1`,`sys2`,... on a single plot. The time vector `t` is optional. 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.

Example

Use the plot handle to normalize the responses on a step plot.

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

See Also

`getoptions`, `setoptions`, `step`

Purpose

Create or convert to transfer function model

Syntax

```
tf
sys = tf(num,den)
sys = tf(num,den,Ts)
sys = tf(M)
sys = tf(num,den,ltisys)
tfsys = tf(sys)
tfsys = tf(sys,'inv')
```

Description

`tf` is used to create real- or complex-valued transfer function models (TF objects) or to convert state-space or zero-pole-gain models to transfer function form.

Creation of Transfer Functions

`sys = tf(num,den)` creates a continuous-time transfer function with numerator(s) and denominator(s) specified by `num` and `den`. The output `sys` is a TF object storing the transfer function data (see "Transfer Function Models" on page 2-8).

In the SISO case, `num` and `den` are the real- or complex-valued row vectors of numerator and denominator coefficients ordered in *descending* powers of s. These two vectors need not have equal length and the transfer function need not be proper. For example, `h = tf([1 0],1)` specifies the pure derivative $h(s) = s$.

To create MIMO transfer functions, specify the numerator and denominator of each SISO entry. In this case:

- `num` and `den` are cell arrays of row vectors with as many rows as outputs and as many columns as inputs.
- The row vectors `num{i,j}` and `den{i,j}` specify the numerator and denominator of the transfer function from input `j` to output `i` (with the SISO convention).

If all SISO entries of a MIMO transfer function have the same denominator, you can set `den` to the row vector representation of this common denominator. See "Examples" for more details.

`sys = tf(num,den,Ts)` creates a discrete-time transfer function with sample time `Ts` (in seconds). Set `Ts = -1` or `Ts = []` to leave the sample time unspecified. The input arguments `num` and `den` are as in the continuous-time case and must list the numerator and denominator coefficients in *descending* powers of z.

`sys = tf(M)` creates a static gain `M` (scalar or matrix).

`sys = tf(num,den,ltisys)` creates a transfer function with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See "Generic Properties" on page 2-26 for an overview of generic LTI properties.

There are several ways to create LTI arrays of transfer functions. To create arrays of SISO or MIMO TF models, either specify the numerator and denominator of each SISO entry using multidimensional cell arrays, or use a `for` loop to successively assign each TF model in the array. See "Building LTI Arrays" on page 4-12 for more information.

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

```
'Property',Value
```

Each pair specifies a particular LTI property of the model, for example, the input names or the transfer function variable. See `set` entry and the example below for details. Note that

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

is a shortcut for

```
sys = tf(num,den)
set(sys,'Property1',Value1,...,'PropertyN',ValueN)
```

Transfer Functions as Rational Expressions in s or z

You can also use real- or complex-valued rational expressions to create a TF model. To do so, first type either:

- `s = tf('s')` to specify a TF model using a rational function in the Laplace variable, `s`.
- `z = tf('z',Ts)` to specify a TF model with sample time `Ts` using a rational function in the discrete-time variable, `z`.

Once you specify either of these variables, you can specify TF models directly as rational expressions in the variable `s` or `z` by entering your transfer function as a rational expression in either `s` or `z`.

Conversion to Transfer Function

`tfsys = tf(sys)` converts an arbitrary SS or ZPK LTI model `sys` to transfer function form. The output `tfsys` (TF object) is the transfer function of `sys`. By default, `tf` uses `zero` to compute the numerators when converting a state-space model to transfer function form. Alternatively,

```
tfsys = tf(sys,'inv')
```

uses inversion formulas for state-space models to derive the numerators. This algorithm is faster but less accurate for high-order models with low gain at $s = 0$.

Examples

Example 1

Create the two-output/one-input transfer function

$$H(p) = \begin{bmatrix} \dfrac{p+1}{p^2+2p+2} \\ \dfrac{1}{p} \end{bmatrix}$$

with input `current` and outputs `torque` and `ang velocity`.

To do this, type

```
num = {[1 1] ; 1}
den = {[1 2 2] ; [1 0]}
H = tf(num,den,'inputn','current',...
                      'outputn',{'torque' 'ang. velocity'},...
                          'variable','p')
Transfer function from input "current" to output...
              p + 1
 torque:  -------------
          p^2 + 2 p + 2

                 1
 ang. velocity:  -
                 p
```

Note how setting the `'variable'` property to `'p'` causes the result to be displayed as a transfer function of the variable p.

Example 2

To use a rational expression to create a SISO TF model, type

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

This produces the same transfer function as

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

Example 3

Specify the discrete MIMO transfer function

$$H(z) = \begin{bmatrix} \frac{1}{z+0.3} & \frac{z}{z+0.3} \\ \frac{-z+2}{z+0.3} & \frac{3}{z+0.3} \end{bmatrix}$$

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

Compute the transfer function of the state-space model with the following data.

$$A = \begin{bmatrix} -2 & -1 \\ 1 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1 \\ 2 & -1 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

To do this, type

```
sys = ss([-2 -1;1 -2],[1 1;2 -1],[1 0],[0 1])
tf(sys)
Transfer function from input 1 to output:
      s
-------------
s^2 + 4 s + 5

Transfer function from input 2 to output:
s^2 + 5 s + 8
-------------
s^2 + 4 s + 5
```

Example 5

You can use a `for` loop to specify a 10-by-1 array of SISO TF models.

```
s = tf('s')
H = tf(zeros(1,1,10));
 for k=1:10,
  H(:,:,k) = k/(s^2+s+k);
 end
```

The first statement pre-allocates the TF array and fills it with zero transfer functions.

Discrete-Time Conventions

The control and digital signal processing (DSP) communities tend to use different conventions to specify discrete transfer functions. Most control engineers use the z variable and order the numerator and denominator terms in descending powers of z, for example,

$$h(z) = \frac{z^2}{z^2 + 2z + 3}$$

The polynomials z^2 and $z^2 + 2z + 3$ are then specified by the row vectors `[1 0 0]` and `[1 2 3]`, respectively. By contrast, DSP engineers prefer to write this transfer function as

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

and specify its numerator as `1` (instead of `[1 0 0]`) and its denominator as `[1 2 3]`.

`tf` switches convention based on your choice of variable (value of the `'Variable'` property).

Variable	Convention
`'z'` (default)	Use the row vector `[ak ... a1 a0]` to specify the polynomial $a_k z^k + \ldots + a_1 z + a_0$ (coefficients ordered in *descending* powers of z).
`'z^-1'`, `'q'`	Use the row vector `[b0 b1 ... bk]` to specify the polynomial $b_0 + b_1 z^{-1} + \ldots + b_k z^{-k}$ (coefficients in *ascending* powers of z^{-1} or q).

For example,

```
g = tf([1 1],[1 2 3],0.1)
```

specifies the discrete transfer function

$$g(z) = \frac{z+1}{z^2 + 2z + 3}$$

because z is the default variable. In contrast,

```
h = tf([1 1],[1 2 3],0.1,'variable','z^-1')
```

uses the DSP convention and creates

$$h(z^{-1}) = \frac{1 + z^{-1}}{1 + 2z^{-1} + 3z^{-2}} = zg(z)$$

See also `filt` for direct specification of discrete transfer functions using the DSP convention.

Note that `tf` stores data so that the numerator and denominator lengths are made equal. Specifically, `tf` stores the values

```
num = [0 1 1]; den = [1 2 3]
```

for `g` (the numerator is padded with zeros on the left) and the values

```
num = [1 1 0]; den = [1 2 3]
```

for `h` (the numerator is padded with zeros on the right).

Algorithm

`tf` uses the MATLAB function `poly` to convert zero-pole-gain models, and the functions `zero` and `pole` to convert state-space models.

See Also

`filt`, `frd`, `get`, `set`, `ss`, `tfdata`, `zpk`

Purpose

Access transfer function data

Syntax

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

Description

`[num,den] = tfdata(sys)` returns the numerator(s) and denominator(s) of the transfer function for the TF, SS or ZPK model (or LTI array of TF, SS or ZPK models) `sys`. For single LTI models, the outputs `num` and `den` of `tfdata` are cell arrays with the following characteristics:

- `num` and `den` have as many rows as outputs and as many columns as inputs.
- The `(i,j)` entries `num{i,j}` and `den{i,j}` are row vectors specifying the numerator and denominator coefficients of the transfer function from input `j` to output `i`. These coefficients are ordered in *descending* powers of s or z.

For arrays `sys` of LTI models, `num` and `den` are multidimensional cell arrays with the same sizes as `sys`.

If `sys` is a state-space or zero-pole-gain model, it is first converted to transfer function form using `tf`. See LTI Properties on page 2-249 for more information on the format of transfer function model data.

For SISO transfer functions, the syntax

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

forces `tfdata` to return the numerator and denominator directly as row vectors rather than as cell arrays (see example below).

`[num,den,Ts] = tfdata(sys)` also returns the sample time `Ts`.

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

```
sys.Ts
```

```
sys.variable
```

Example

Given the SISO transfer function

```
h = tf([1 1],[1 2 5])
```

you can extract the numerator and denominator coefficients by typing

```
[num,den] = tfdata(h,'v')
num =
     0     1     1

den =
     1     2     5
```

This syntax returns two row vectors.

If you turn `h` into a MIMO transfer function by typing

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

the command

```
[num,den] = tfdata(H)
```

now returns two cell arrays with the numerator/denominator data for each SISO entry. Use `celldisp` to visualize this data. Type

```
celldisp(num)
```

and MATLAB returns the numerator vectors of the entries of `H`.

```
num{1} =
     0     1     1

num{2} =
     0     1
```

Similarly, for the denominators, type

```
celldisp(den)
den{1} =
     1     2     5

den{2} =
     1     1
```

See Also `get`, `ssdata`, `tf`, `zpkdata`

totaldelay

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 (see `set` or `ltiprops` for details on these properties).

Delays are expressed in seconds for continuous-time models, and as integer multiples of the sample period for discrete-time models. To obtain the delay times in seconds, multiply `td` by the sample time `sys.Ts`.

Example

```
sys = tf(1,[1 0]);   % TF of 1/s
sys.inputd = 2;    % 2 sec input delay
sys.outputd = 1.5;   % 1.5 sec output delay
td = totaldelay(sys)
td =
    3.5000
```

The resulting I/O map is

$$e^{-2s} \times \frac{1}{s} e^{-1.5s} = e^{-3.5s} \frac{1}{s}$$

This is equivalent to assigning an I/O delay of 3.5 seconds to the original model `sys`.

See Also `delay2z`, `hasdelay`

Purpose

Transmission zeros of LTI model

Syntax

```
zero
z = zero(sys)
[z,gain] = zero(sys)
```

Description

`zero` computes the zeros of SISO systems and the transmission zeros of MIMO systems. For a MIMO system with matrices (A, B, C, D), the transmission zeros are the complex values λ for which the normal rank of

$$\begin{bmatrix} A - \lambda I & B \\ C & D \end{bmatrix}$$

drops.

`z = zero(sys)` returns the (transmission) zeros of the LTI model `sys` as a column vector.

`[z,gain] = zero(sys)` also returns the gain (in the zero-pole-gain sense) if `sys` is a SISO system.

Algorithm

`zero` is based on SLICOT routine AB08NX. Also use LAPACK routines DGEEV and DGEGV (and their complex counterparts) for eigenvalue computation.

The transmission zeros are computed using the algorithm in [1].

References

[1] Emami-Naeini, A. and P. Van Dooren, "Computation of Zeros of Linear Multivariable Systems," *Automatica*, 18 (1982), pp. 415-430.

See Also

`pole`, `pzmap`

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 π in steps of $\pi/10$, and plots the grid over the current axis. If the current axis contains a discrete z-plane root locus diagram or pole-zero map, `zgrid` draws the grid over the plot without altering the current axis limits.

`zgrid(z,wn)` plots a grid of constant damping factor and natural frequency lines for the damping factors and normalized natural frequencies in the vectors `z` and `wn`, respectively. If the current axis contains a discrete z-plane root locus diagram or pole-zero map, `zgrid(z,wn)` draws the grid over the plot. The frequency lines for unnormalized (true) frequencies can be plotted using

```
zgrid(z,wn/Ts)
```

where `Ts` is the sample time.

`zgrid([],[])` draws the unit circle.

Alternatively, you can select **Grid** from the right-click menu to generate the same z-plane grid.

Example

Plot z-plane grid lines on the root locus for the system

$$H(z) = \frac{2z^2 - 3.4z + 1.5}{z^2 - 1.6z + 0.8}$$

by typing

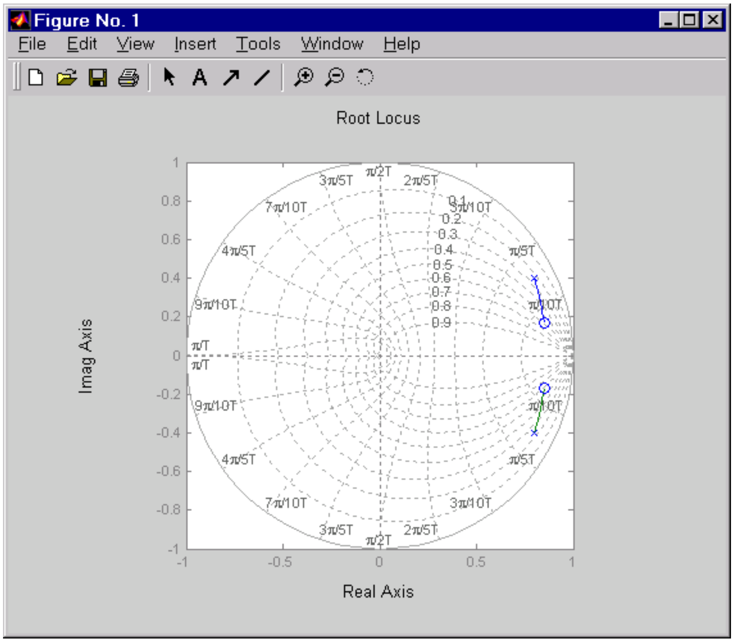
```
H = tf([2 -3.4 1.5],[1 -1.6 0.8],-1)
```

```
Transfer function:
2 z^2 - 3.4 z + 1.5
-------------------
 z^2 - 1.6 z + 0.8

Sampling time: unspecified
```

To see the z-plane grid on the root locus plot, type

```
rlocus(H)
zgrid
axis('square')
```



See Also

`pzmap`, `rlocus`, `sgrid`

Purpose Create or convert to zero-pole-gain model

Syntax

```
zpk
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)
```

Description `zpk` is used to create zero-pole-gain models (ZPK objects) or to convert TF or SS models to zero-pole-gain form.

Creation of Zero-Pole-Gain Models

`sys = zpk(z,p,k)` creates a continuous-time zero-pole-gain model with zeros `z`, poles `p`, and gain(s) `k`. The output `sys` is a ZPK object storing the model data (see "LTI Objects" on page 2-3).

In the SISO case, `z` and `p` are the vectors of real- or complex-valued zeros and poles, and `k` is the real- or complex-valued scalar gain.

$$h(s) = k\,\frac{(s-z(1))(s-z(2))\ldots(s-z(m))}{(s-p(1))(s-p(2))\ldots(s-p(n))}$$

Set `z` or `p` to `[]` for systems without zeros or poles. These two vectors need not have equal length and the model need not be proper (that is, have an excess of poles).

You can also use rational expressions to create a ZPK model. To do so, use either:

- `s = zpk('s')` to specify a ZPK model from a rational transfer function of the Laplace variable, `s`.
- `z = zpk('z',Ts)` to specify a ZPK model with sample time `Ts` from a rational transfer function of the discrete-time variable, `z`.

Once you specify either of these variables, you can specify ZPK models directly as real- or complex-valued rational expressions in the variable `s` or `z`.

To create a MIMO zero-pole-gain model, specify the zeros, poles, and gain of each SISO entry of this model. In this case:

- `z` and `p` are cell arrays of vectors with as many rows as outputs and as many columns as inputs, and `k` is a matrix with as many rows as outputs and as many columns as inputs.
- The vectors `z{i,j}` and `p{i,j}` specify the zeros and poles of the transfer function from input `j` to output `i`.
- `k(i,j)` specifies the (scalar) gain of the transfer function from input `j` to output `i`.

See below for a MIMO example.

`sys = zpk(z,p,k,Ts)` creates a discrete-time zero-pole-gain model with sample time `Ts` (in seconds). Set `Ts = -1` or `Ts = []` to leave the sample time unspecified. The input arguments `z`, `p`, `k` are as in the continuous-time case.

`sys = zpk(M)` specifies a static gain `M`.

`sys = zpk(z,p,k,ltisys)` creates a zero-pole-gain model with generic LTI properties inherited from the LTI model `ltisys` (including the sample time). See "Generic Properties" on page 2-26 for an overview of generic LTI properties.

To create an array of ZPK models, use a `for` loop, or use multidimensional cell arrays for `z` and `p`, and a multidimensional array for `k`.

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

```
'PropertyName',PropertyValue
```

Each pair specifies a particular LTI property of the model, for example, the input names or the input delay time. See `set` entry and the example below for details. Note that

```
sys = zpk(z,p,k,'Property1',Value1,...,'PropertyN',ValueN)
```

is a shortcut for the following sequence of commands.

```
sys = zpk(z,p,k)
set(sys,'Property1',Value1,...,'PropertyN',ValueN)
```

Zero-Pole-Gain Models as Rational Expressions in s or z

You can also use rational expressions to create a ZPK model. To do so, first type either:

- `s = zpk('s')` to specify a ZPK model using a rational function in the Laplace variable, `s`.
- `z = zpk('z',Ts)` to specify a ZPK model with sample time `Ts` using a rational function in the discrete-time variable, `z`.

Once you specify either of these variables, you can specify ZPK models directly as rational expressions in the variable `s` or `z` by entering your transfer function as a rational expression in either `s` or `z`.

Conversion to Zero-Pole-Gain Form

`zsys = zpk(sys)` converts an arbitrary LTI model `sys` to zero-pole-gain form. The output `zsys` is a ZPK object. By default, `zpk` uses `zero` to compute the zeros when converting from state-space to zero-pole-gain. Alternatively,

```
zsys = zpk(sys,'inv')
```

uses inversion formulas for state-space models to compute the zeros. This algorithm is faster but less accurate for high-order models with low gain at $s = 0$.

Variable Selection

As for transfer functions, you can specify which variable to use in the display of zero-pole-gain models. Available choices include s (default) and p for continuous-time models, and z (default), z^{-1}, or $q = z^{-1}$ for discrete-time models. Reassign the `'Variable'` property to override the defaults. Changing the variable affects only the display of zero-pole-gain models.

Example

Example 1

Specify the following zero-pole-gain model.

$$H(z) = \begin{bmatrix} \dfrac{1}{z-0.3} \\ \dfrac{2(z+0.5)}{(z-0.1+j)(z-0.1-j)} \end{bmatrix}$$

To do this, type

```
z = {[] ; -0.5}
p = {0.3 ; [0.1+i 0.1-i]}
k = [1 ; 2]
H = zpk(z,p,k,-1)    % unspecified sample time
```

Example 2

Convert the transfer function

```
h = tf([-10 20 0],[1 7 20 28 19 5])
Transfer function:
             -10 s^2 + 20 s
----------------------------------------
s^5 + 7 s^4 + 20 s^3 + 28 s^2 + 19 s + 5
```

to zero-pole-gain form by typing

```
zpk(h)
Zero/pole/gain:
```

```
     -10 s (s-2)
----------------------
(s+1)^3 (s^2 + 4s + 5)
```

Example 3

Create a discrete-time ZPK model from a rational expression in the variable `z`, by typing

```
z = zpk('z',0.1);
H = (z+.1)*(z+.2)/(z^2+.6*z+.09)
Zero/pole/gain:
(z+0.1) (z+0.2)
---------------
   (z+0.3)^2

Sampling time: 0.1
```

Algorithm

`zpk` uses the MATLAB function `roots` to convert transfer functions and the functions `zero` and `pole` to convert state-space models.

See Also

`frd`, `get`, `set`, `ss`, `tf`, `zpkdata`

Purpose

Access zero-pole-gain data

Syntax

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

Description

`[z,p,k] = zpkdata(sys)` returns the zeros `z`, poles `p`, and gain(s) `k` of the zero-pole-gain model `sys`. The outputs `z` and `p` are cell arrays with the following characteristics:

- `z` and `p` have as many rows as outputs and as many columns as inputs.
- The `(i,j)` entries `z{i,j}` and `p{i,j}` are the (column) vectors of zeros and poles of the transfer function from input `j` to output `i`.

The output `k` is a matrix with as many rows as outputs and as many columns as inputs such that `k(i,j)` is the gain of the transfer function from input `j` to output `i`. If `sys` is a transfer function or state-space model, it is first converted to zero-pole-gain form using `zpk`. See "LTI Properties" in the *Control System Toolbox User's Guide* for more information on the format of state-space model data.

For SISO zero-pole-gain models, the syntax

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

forces `zpkdata` to return the zeros and poles directly as column vectors rather than as cell arrays (see example below).

`[z,p,k,Ts,Td] = zpkdata(sys)` also returns the sample time `Ts` and the input delay data `Td`. For continuous-time models, `Td` is a row vector with one entry per input channel (`Td(j)` indicates by how many seconds the `j`th input is delayed). For discrete-time models, `Td` is the empty matrix `[]` (see `d2d` for delays in discrete systems).

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

```
sys.Ts
```

```
sys.inputname
```

Example

Given a zero-pole-gain model with two outputs and one input

```
H = zpk({[0];[-0.5]},{[0.3];[0.1+i 0.1-i]},[1;2],-1)
Zero/pole/gain from input to output...
         1
 #1:  -------
      (z-0.3)

           2 (z+0.5)
 #2:  -------------------
      (z^2 - 0.2z + 1.01)

Sampling time: unspecified
```

you can extract the zero/pole/gain data embedded in `H` with

```
[z,p,k] = zpkdata(H)
z =
    [      0]
    [-0.5000]
p =
    [    0.3000]
    [2x1 double]
k =
     1
     2
```

To access the zeros and poles of the second output channel of `H`, get the content of the second cell in `z` and `p` by typing

```
z{2,1}
ans =
   -0.5000
p{2,1}
ans =
```

```
          0.1000+ 1.0000i
          0.1000- 1.0000i
```

See Also

`get`, `ssdata`, `tfdata`, `zpk`

3

Block Reference

Introduction

The Control System Toolbox provides the LTI System block for use with Simulink. Its reference page contains the following information:

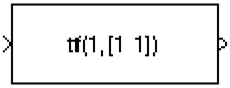
- The block name and icon
- The purpose of the block
- A description of the block
- The block parameters and dialog box including a brief description of each parameter

LTI System

Purpose

Import LTI System

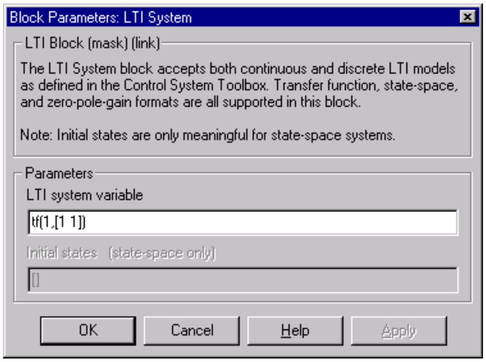
Description



The LTI System block imports linear, time-invariant (LTI) systems into Simulink.

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



Block Parameters: LTI System

LTI Block (mask) (link)

The LTI System block accepts both continuous and discrete LTI models as defined in the Control System Toolbox. Transfer function, state-space, and zero-pole-gain formats are all supported in this block.

Note: Initial states are only meaningful for state-space systems.

Parameters

LTI system variable

tf(1,[1 1])

Initial states (state-space only)

[]

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.

Index