Robust Control Toolbox™

Reference

Gary Balas
Richard Chiang
Andy Packard
Michael Safonov

MATLAB®

R2019**b**

MathWorks®

Revision History

September 2005	First printing	New for Version 3.0.2 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1 (Release 2006a)
September 2006	Online only	Revised for Version 3.1.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.4.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.5 (Release 2010b)
April 2011	Online only	Revised for Version 3.6 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.2 (Release 2012b)
March 2013	Online only	Revised for Version 4.3 (Release 2013a)
September 2013	Online only	Revised for Version 5.0 (Release 2013b)
March 2014	Online only	Revised for Version 5.1 (Release 2014a)
October 2014	Online only	Revised for Version 5.2 (Release 2014b)
March 2015	Online only	Revised for Version 5.3 (Release 2015a)
September 2015	Online only	Revised for Version 6.0 (Release 2015b)
March 2016	Online only	Revised for Version 6.1 (Release 2016a)
September 2016	Online only	Revised for Version 6.2 (Release 2016b)
March 2017	Online only	Revised for Version 6.3 (Release 2017a)
September 2017	Online only	Revised for Version 6.4 (Release 2017b)
March 2018	Online only	Revised for Version 6.4.1 (Release 2018a)
September 2018	Online only	Revised for Version 6.5 (Release 2018b)
March 2019	Online only	Revised for Version 6.6 (Release 2019a)
September 2019	Online only	Revised for Version 6.7 (Release 2019b)

Contents

1

Alphabetical List

actual2normalized

Transform actual values to normalized values

Syntax

```
NV = actual2normalized(uElement,AV)
[NV,ndist] = actual2normalized(uElement,AV)
```

Description

`NV = actual2normalized(uElement,AV)` transforms the values `AV` of the uncertain element `uElement` into normalized values `NV`. If `AV` is the nominal value of `uElement`, `NV` is 0. Otherwise, `AV` values inside the uncertainty range of `uElement` map to the unit ball `||NV|| <= 1`, and values outside the uncertainty range map to `||NV|| > 1`. The argument `AV` can contain a single value or an array of values. `NV` has the same dimensions as `AV`.

`[NV,ndist] = actual2normalized(uElement,AV)` also returns the normalized distance `ndist` between the values `AV` and the nominal value of `uElement`. This distance is the norm of `NV`. Therefore, `ndist <= 1` for values inside the uncertainty range of `uElement`, and `ndist > 1` for values outside the range. If `AV` is an array of values, then `ndist` is an array of normalized distances.

The robustness margins computed by`robstab` and `robgain` serve as bounds for the normalized distances in `ndist`. For example, if an uncertain system has a stability margin of 1.4, this system is stable for all uncertain element values whose normalized distance from the nominal is less than 1.4.

Examples

Uncertain Real Parameter with Symmetric Range

For uncertain real parameters whose range is symmetric about their nominal value, the normalized distance is intuitive, scaling linearly with the numerical difference from the uncertain real parameter's nominal value.

Create uncertain real parameters with a range that is symmetric about the nominal value, where each end point is 1 unit from the nominal. Points that lie inside the range are less than 1 unit from the nominal, while points that lie outside the range are greater than 1 unit from the nominal.

```
a = ureal('a',3,'range',[1 5]);
NV = actual2normalized(a,[1 3 5])
```

```
NV = 1×3

   -1.0000         0    1.0000
```

```
NV = actual2normalized(a,[2 4])
```

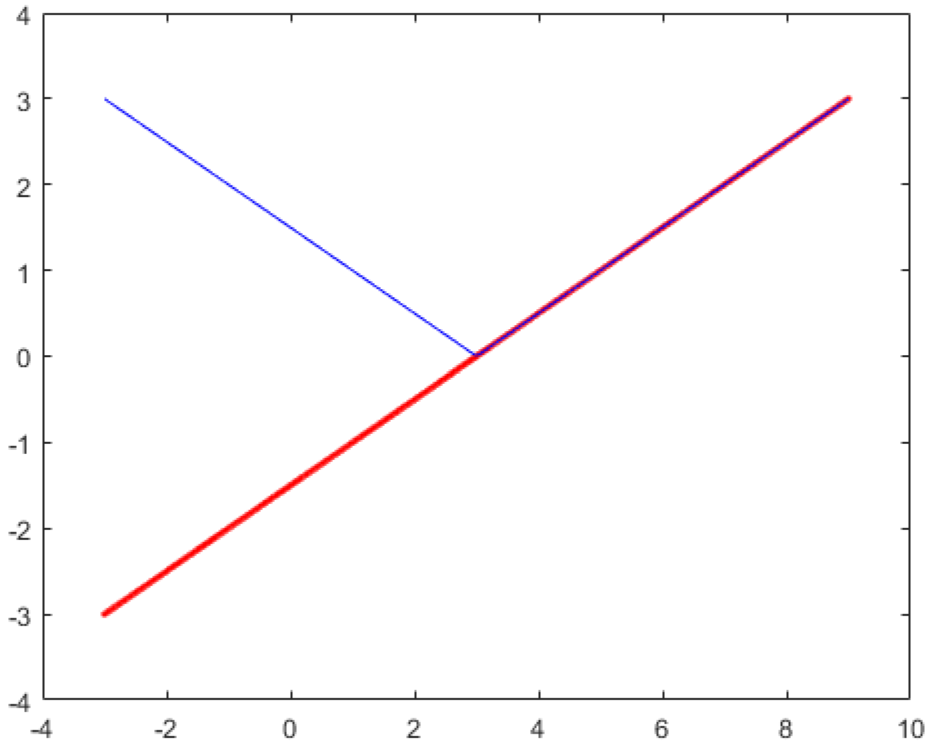
```
NV = 1×2

   -0.5000    0.5000
```

```
NV = actual2normalized(a,[0 6])
```

```
NV = 1×2

   -1.5000    1.5000
```

Plot the normalized values and normalized distance for several values.

```
values = linspace(-3,9,250);
[nv,ndist] = actual2normalized(a,values);
plot(values,nv,'r.',values,ndist,'b-')
```



Uncertain Real Parameter with Nonsymmetric Range

Create a nonsymmetric parameter. The end points are 1 normalized unit from nominal, and the nominal is 0 normalized units from nominal. Moreover, points inside the range are less than 1 unit from nominal, and points outside the range are greater than 1 unit from nominal. However, the relationship between the normalized distance and numerical difference is nonlinear.

```
au = ureal('ua',4,'range',[1 5]);
NV = actual2normalized(au,[1 4 5])
NV = 1×3

    -1     0     1
```

```
NV = actual2normalized(au,[2 4.5])
```

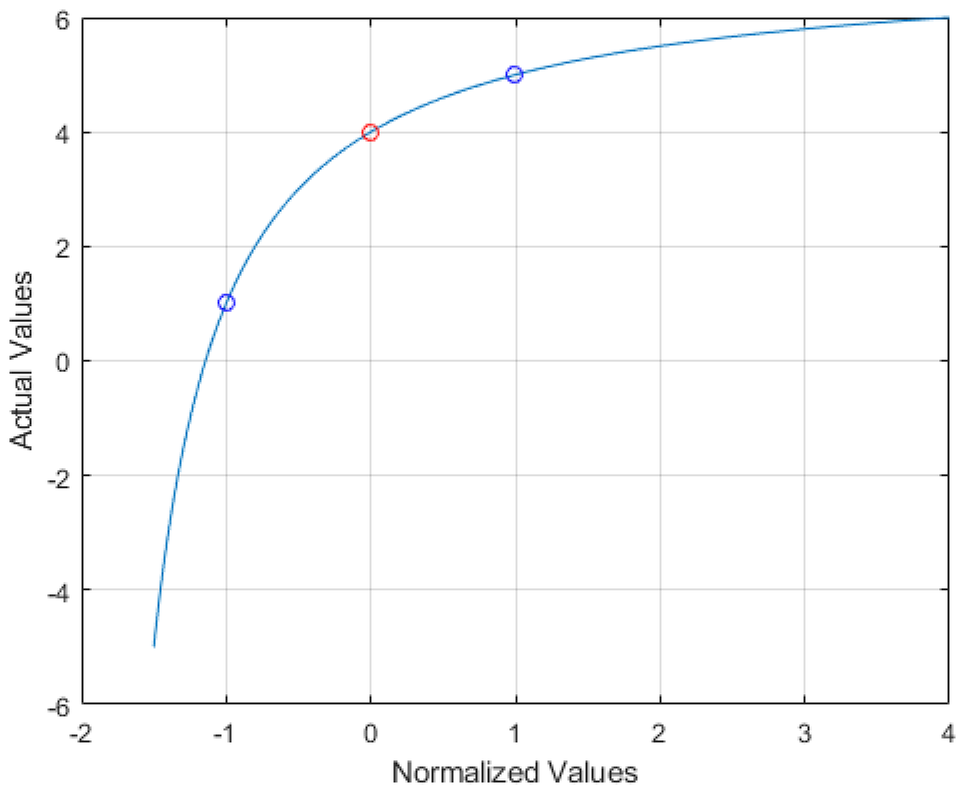
```
NV = 1×2

   -0.8000    0.4000
```

```
NV = actual2normalized(au,[0 6])
```

```
NV = 1×2

   -1.1429    4.0000
```

Graph the relationship between actual and normalized values. The relationship is very nonlinear.

```
AV = linspace(-5,6,250);
NV = actual2normalized(au,AV);

plot(NV,AV,0,au.NominalValue,'ro',-1,au.Range(1),'bo',1,au.Range(2),'bo')
grid, xlabel('Normalized Values'), ylabel('Actual Values')
```



The red circle shows the nominal value (normalized value = 0). The blue circles show the values at the edges of the uncertainty range (normalized values = -1, 1).

Algorithms

For details on the normalize distance, see “Normalizing Functions for Uncertain Elements”.

See Also

`getLimits` | `normalized2actual` | `robgain` | `robstab`

Introduced before R2006a

aff2pol

Convert affine parameter-dependent models to polytopic models

Syntax

```
polsys = aff2pol(affsys)
```

Description

`aff2pol` derives a polytopic representation `polsys` of the *affine* parameter-dependent system

$$E(p)\dot{x} = A(p)x + B(p)u \qquad (1\text{-}1)$$

$$y = C(p)x + D(p)u \qquad (1\text{-}2)$$

where $p = (p_1,\ldots, p_n)$ is a vector of uncertain or time-varying real parameters taking values in a box or a polytope. The description `affsys` of this system should be specified with `psys`.

The vertex systems of `polsys` are the instances of “Equation 1-1” and “Equation 1-2” at the vertices p_{ex} of the parameter range, i.e., the `SYSTEM` matrices

$$\begin{pmatrix} A(p_{ex}) + jE(p_{ex}) & B(p_{ex}) \\ C(p_{ex}) & D(p_{ex}) \end{pmatrix}$$

for all corners p_{ex} of the parameter box or all vertices p_{ex} of the polytope of parameter values.

See Also

`psys` | `pvec` | `uss`

Introduced before R2006a

augw

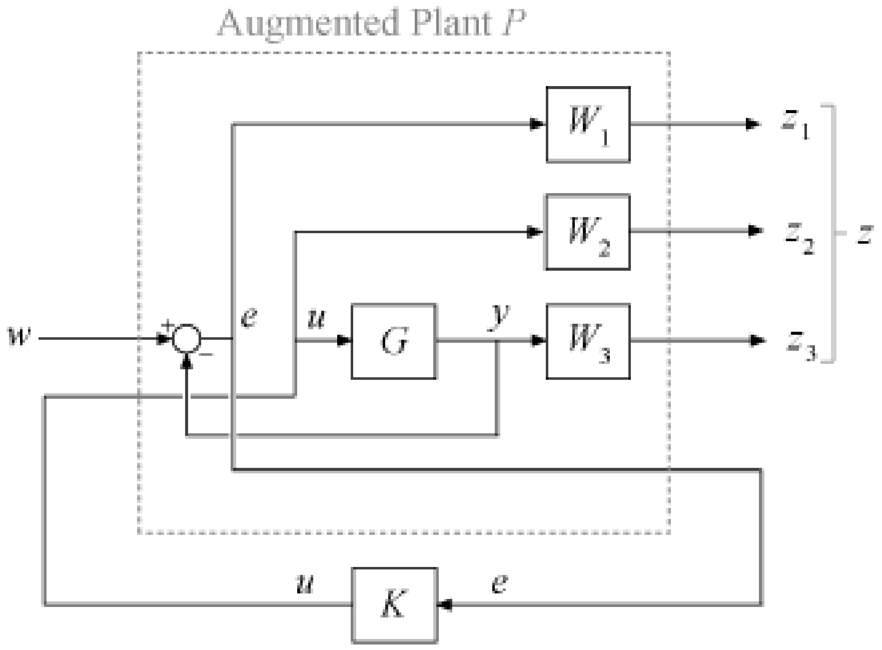
Plant augmentation for weighted mixed-sensitivity H_∞ and H_2 loop-shaping design

Syntax

```
P = augw(G,W1,W2,W3)
```

Description

`P = augw(G,W1,W2,W3)` computes a state-space model of an augmented LTI plant $P(s)$ with the weighting functions $W_1(s)$, $W_2(s)$, and $W_3(s)$ penalizing the error signal, control signal, and output signal, respectively. `P` is the augmented plant of the following diagram.

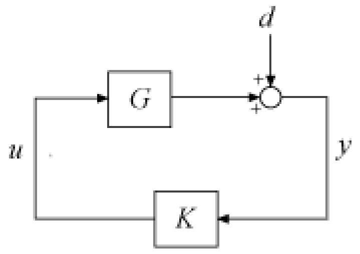


This control structure is used in mixed H_∞ synthesis, which lets you design an H_∞ controller by simultaneously shaping the frequency responses for tracking and disturbance rejection, noise reduction and robustness, and controller effort. For more information, see “Mixed-Sensitivity Loop Shaping”.

Examples

Create Augmented Plant for H-Infinity Synthesis

Suppose you want to synthesize a stabilizing robust controller for the system of the following diagram. The controller must also reject disturbances injected at the plant output.



The plant, G, is an unstable first-order system.

```
G = tf(1,[1 -1]);
```

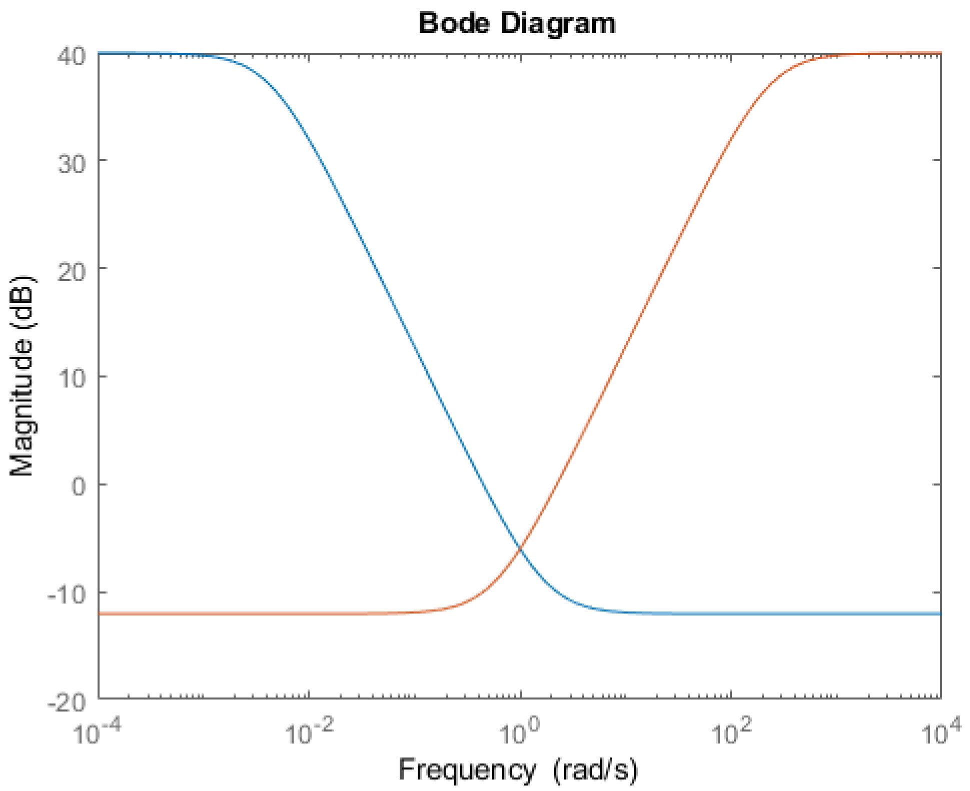
To set up this problem for hinfsyn, insert a weighting function W1 that captures the disturbance rejection goal, and another weighting function W3 to enforce robustness. Specify these weighting functions as the inverses of the desired loop shapes for the sensitivity S and complementary sensitivity T, respectively. (See “Mixed-Sensitivity Loop Shaping”.)

For this example, choose W1 with:

- Low-frequency gain of 100 (40 dB)
- 0 dB crossover at 0.5 rad/s
- High-frequency gain of 0.25 (–12 dB)

Choose W3 to have the opposite low-frequency and high-frequency gains.

```
W1 = makeweight(100,[1 0.5],0.25);
W3 = makeweight(0.25,[1 0.5],100);
bodemag(W1,W3)
```



For this example, do not specify a `W2` (no restriction on control effort). Construct the augmented plant, `P`.

```
P = augw(G,W1,[],W3);
```

G has one input and one output. The augmented plant has an additional input for the control signal, and additional outputs for each of the weights.

```
size(P)
```

```
State-space model with 3 outputs, 2 inputs, and 3 states.
```

The inputs and outputs of `P` are grouped to keep track of the disturbance and control inputs and the error and measurement outputs. For example, example the output groups.

Group `Y1` contains the two error outputs z, and group `Y2` contains the single measurement output.

```
P.OutputGroup
```

```
ans = struct with fields:
    Y1: [1 2]
    Y2: 3
```

You can now use `P` for control design. For example, use `hinfsyn` to design an H_∞ optimal controller that meets the design requirements specified by `W1` and `W3`.

```
[K,CL,gamma] = hinfsyn(P);
gamma
```

```
gamma = 0.9946
```

Input Arguments

`G` — Plant

dynamic system model

Plant, specified as a dynamic system model such as a state-space (`ss`) model. `G` can be any LTI model. If `G` is a generalized state-space model with uncertain or tunable control design blocks, then `mixsyn` uses the nominal or current value of those elements.

`W1,W2,W3` — Weighting functions

dynamic system model | `[]`

Weighting functions, specified as dynamic system models. Choose the weighting functions `W1,W2,W3` to shape the frequency responses for tracking and disturbance rejection, controller effort, and noise reduction and robustness. Typically:

- For good reference-tracking and disturbance-rejection performance, choose `W1` large inside the control bandwidth to obtain small S.
- For robustness and noise attenuation, choose `W3` large outside the control bandwidth to obtain small T.
- To limit control effort in a particular frequency band, increase the magnitude of W_2 in this frequency band to obtain small KS.

If one of the weights is not needed, set it to `[]`. For instance, if you do not want to restrict control effort, use `W2 = []`.

Use `makeweight` to create weighting functions with the desired gain profiles. For details about choosing weighting functions, see “Mixed-Sensitivity Loop Shaping”.

If `G` has N_U inputs and N_Y outputs, then `W1,W2,W3` must be either SISO or square systems of size N_Y, N_U, and N_Y, respectively.

Because $S + T = I$, `mixsyn` cannot make both S and T small (less than 0 dB) in the same frequency range. Therefore, when you specify weights for loop shaping, there must be a frequency band in which both `W1` and `W3` are below 0 dB.

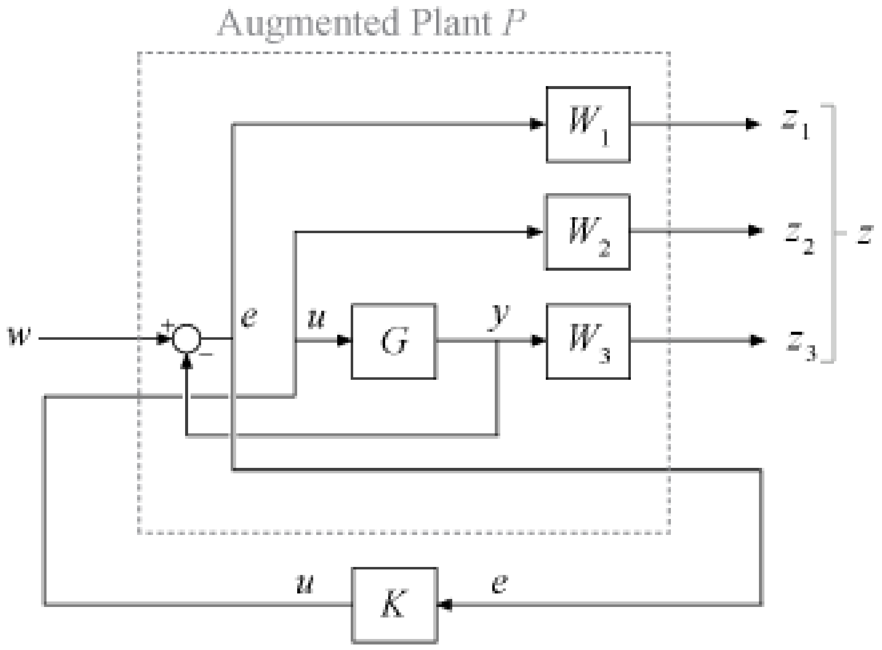
Output Arguments

P — Augmented plant
dynamic system model

Augmented plant, returned as a state-space (`ss`) model. `P` can be any LTI model with inputs $[w;u]$ and outputs $[z;y]$. `augw` groups the inputs and outputs of `P` using the `ss` properties `InputGroup` and `OutputGroup` such that:

- `P.InputGroup` has field `U1` containing the inputs corresponding to w, and field `U2` containing the inputs corresponding to u.
- `P.OutputGroup` has field `Y1` containing the outputs corresponding to z, and group `Y2` containing the outputs corresponding to e.

Here, $\{w;u\}$ and $\{z;e\}$ are the inputs and outputs of `P` in the following control system.



Tips

- For H_∞ or H_2 synthesis, the models `G` and `W1,W2,W3` must be proper. In other words, they must be bounded as $s \to \infty$ (for continuous-time transfer functions) or $z \to \infty$ (for discrete-time transfer functions). Additionally, `W1,W2,W3` must be stable. The plant `G` must be stabilizable and detectable. Otherwise, the resulting `P` is not stabilizable by any controller.

Algorithms

`augw` produces the augmented plant $P(s)$ given by:

$$P(s) = \begin{bmatrix} W_1 & -W_1G \\ 0 & W_2 \\ 0 & W_3G \\ I & -G \end{bmatrix}$$

The partitioning is embedded using `P = mktito(P,NY,NU)`, which sets the `P.InputGroup` and `P.OutputGroup` properties as follows.

```
[r,c] = size(P);
P.InputGroup  = struct('U1',1:c-NU,'U2',c-NU+1:c);
P.OutputGroup = struct('Y1',1:r-NY,'Y2',r-NY+1:r);
```

See Also

`h2syn` | `hinfsyn` | `makeweight` | `mixsyn`

Topics

"Mixed-Sensitivity Loop Shaping"

Introduced before R2006a

balancmr

Balanced model truncation via square root method

Syntax

```
GRED = balancmr(G)
```

```
GRED = balancmr(G,order)
```

```
[GRED,redinfo] = balancmr(G,key1,value1,...)
```

```
[GRED,redinfo] = balancmr(G,order,key1,value1,...)
```

Description

`balancmr` returns a reduced order model `GRED` of `G` and a struct array `redinfo` containing the error bound of the reduced model and Hankel singular values of the original system.

The error bound is computed based on Hankel singular values of `G`. For a stable system these values indicate the respective state energy of the system. Hence, reduced order can be directly determined by examining the system Hankel singular values, $\sigma\iota$.

With only one input argument `G`, the function will show a Hankel singular value plot of the original model and prompt for model order number to reduce.

This method guarantees an error bound on the infinity norm of the *additive error* ‖ `G-GRED` ‖ ∞ for well-conditioned model reduced problems [1]:

$$\|G - Gred\|_\infty \le 2 \sum_{k+1}^{n} \sigma_i$$

This table describes input arguments for `balancmr`.

Argument	Description
`G`	LTI model to be reduced. Without any other inputs, `balancmr` will plot the Hankel singular values of `G` and prompt for reduced order
`ORDER`	(Optional) Integer for the desired order of the reduced model, or optionally a vector packed with desired orders for batch runs

A batch run of a serial of different reduced order models can be generated by specifying `order = x:y`, or a vector of positive integers. By default, all the anti-stable part of a system is kept, because from control stability point of view, getting rid of unstable state(s) is dangerous to model a system.

`'MaxError'` can be specified in the same fashion as an alternative for `'Order'`. In this case, reduced order will be determined when the sum of the tails of the Hankel singular values reaches the `'MaxError'`.

This table lists the input arguments `'key'` and its `'value'`.

Argument	Value	Description
`'MaxError'`	Real number or vector of different errors	Reduce to achieve H_∞ error. When present, `'MaxError'` overrides `ORDER` input.

Argument	Value	Description
`'Weights'`	`{Wout,Win}` cell array	Optional 1-by-2 cell array of LTI weights `Wout` (output) and `Win` (input). The weights must be stable, minimum phase and invertible. When you supply these weights, `balancmr` finds the reduced model that minimizes the Hankel norm of $W_{out}^{-1}(G - G_{red})W_{in}^{-1}$. You can use weighting functions to make the model reduction algorithm focus on frequency bands of interest. See: • "Reduction with Focus on Particular Frequency Band" on page 1-26 • "Model Reduction With Frequency-Dependent Error Profile" on page 1-28 As an alternative, you can use `balred` to focus model reduction on a particular frequency band without defining a weighting function. Using `balancmr` and providing your own weighting functions allows more precise control over the error profile. Default weights are both identity.
`'Display'`	`'on''` or `'off'`	Display Hankel singular plots (default `'off'`).
`'Order'`	Integer, vector or cell array	Order of reduced model. Use only if not specified as 2nd argument.

This table describes output arguments.

Argument	Description
GRED	LTI reduced order model. Becomes multidimensional array when input is a serial of different model order array
REDINFO	A STRUCT array with three fields: • REDINFO.ErrorBound (bound on ‖ *G-GRED* ‖∞) • REDINFO.StabSV (Hankel SV of stable part of G) • REDINFO.UnstabSV (Hankel SV of unstable part of G)

G can be stable or unstable, continuous or discrete.

Examples

Choose Order of Reduced Model

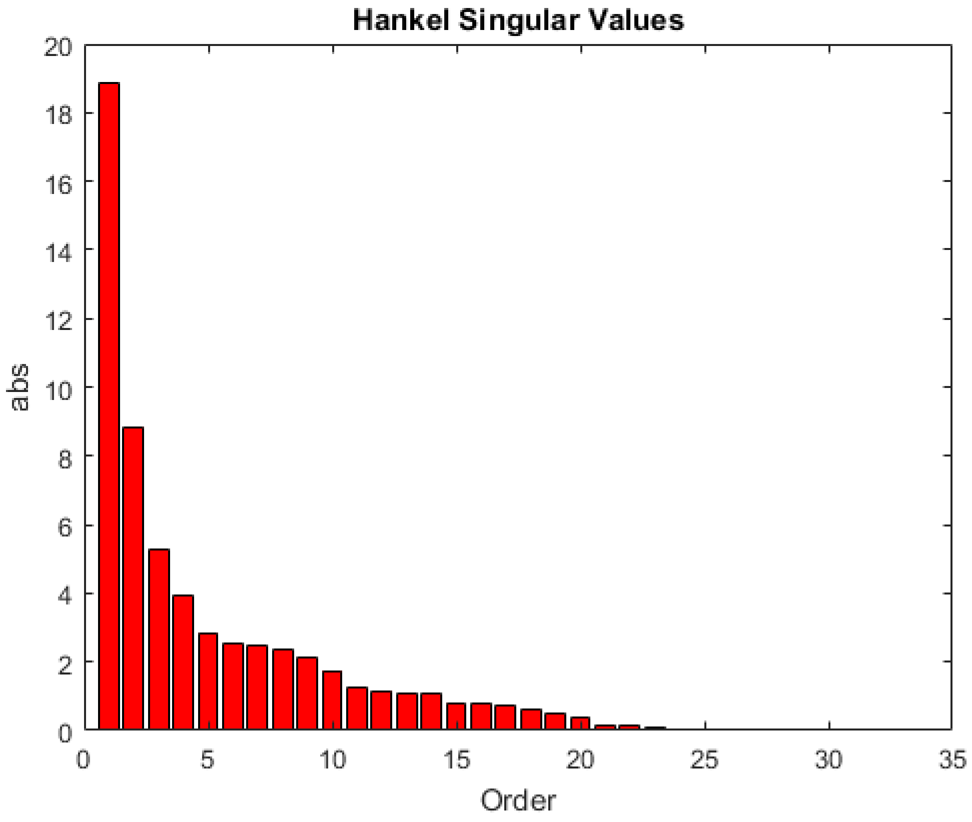
If you do not specify any target order for the reduced model, balancmr displays the Hankel singular values of the model and prompts you to choose a reduced-model order.

For this example, use a random 30th-order state-space model.

```
rng(1234,'twister');     % fix random seed for example repeatability
G = rss(30,5,4);

G1 = balancmr(G)
```

```
Please enter the desired order: (>=0)
```

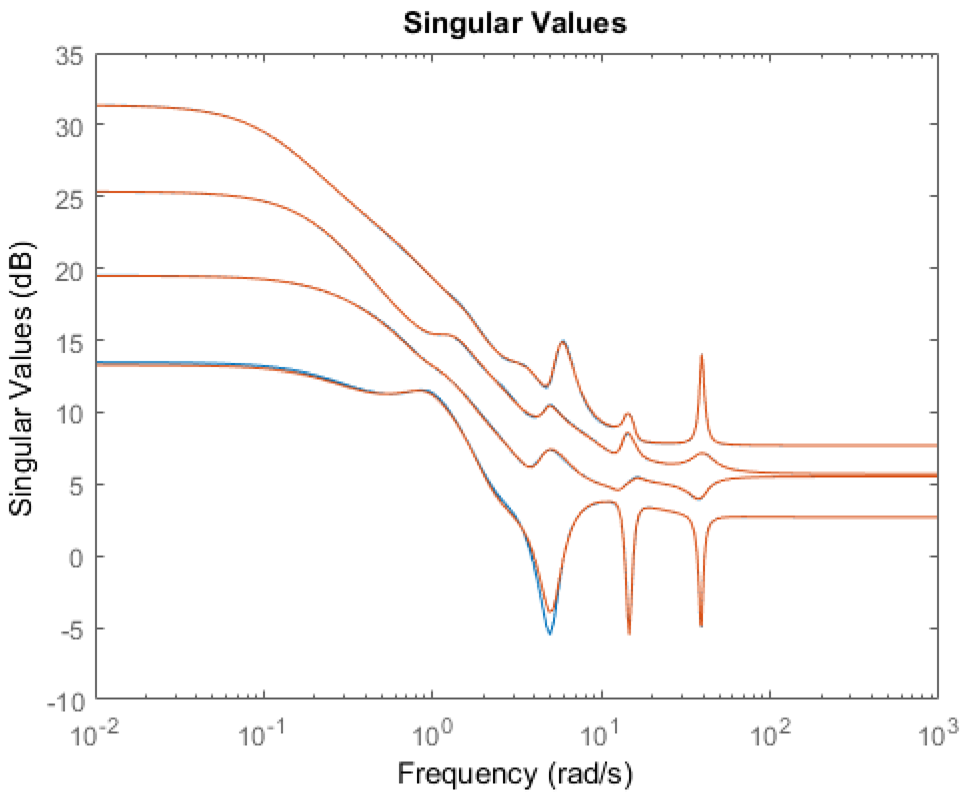
Examine the Hankel singular value plot.



The plot shows that most of the energy of the system can be captured in a 20th-order approximation. In the command window, enter 20. `balancmr` returns `G1`.

Examine the response of the original and reduced models.

```
sigma(G,G1)
```



The 20th-order approximation matches the dynamics of the original 30th-order model fairly well.

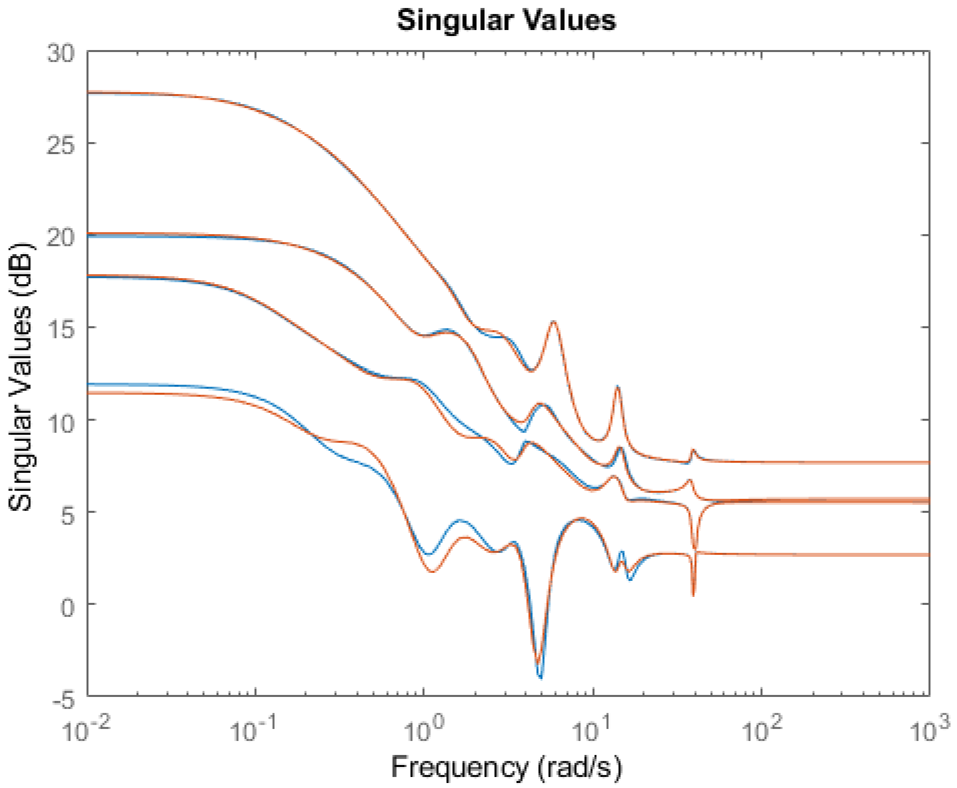
Model Reduction to Specified Order

When you have particular target order or orders in mind, you can use balancmr to reduce a high-order model to those orders. For this example, use a random 30th-order state-space model.

```
rng(1234,'twister');      % fix random seed for example repeatability
G = rss(30,5,4);
```

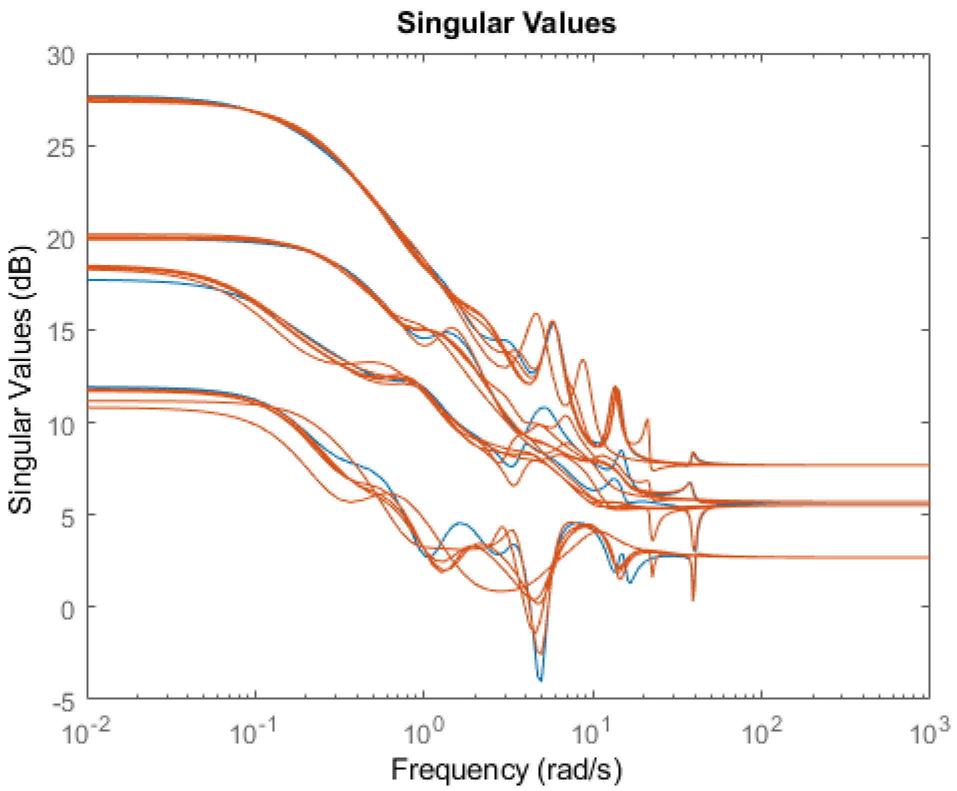
Use a scalar input argument to reduce the model to a single order. For example, compute a 20th-order approximation.

```
[G1,info1] = balancmr(G,20);
sigma(G,G1)
```



Use a vector to generate several approximations. The following command returns an array of models of even orders from 10 to 18.

```
[G2,info2] = balancmr(G,[10:2:18]);
sigma(G,G2)
```



Model Reduction to Specified Maximum Error

Obtain the lowest-order approximation such that the sum of the Hankel singular values of the truncated states does not exceed a specified value. For this example, use a random 30th-order state-space model.

```
rng(1234,'twister');     % fix random seed for example repeatability
G = rss(30,5,4);
```

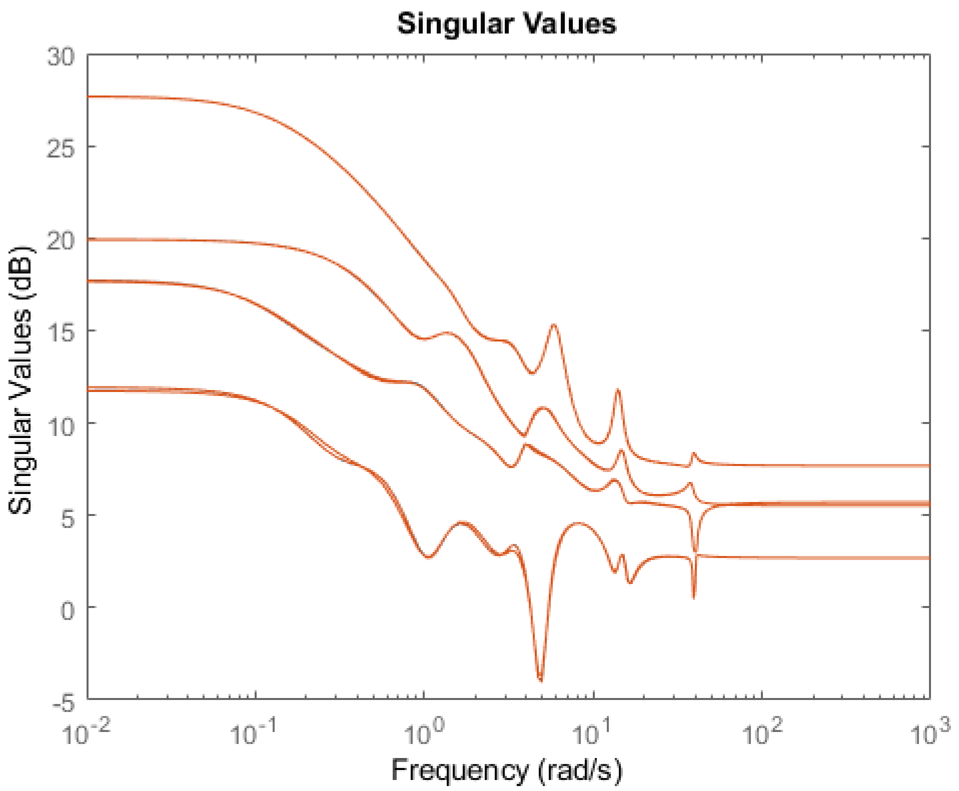
Compute two approximate models, one for which the error does not exceed 0.1, and a second for which the error does not exceed 0.5. To do so, provide these values in an array. `balancmr` returns an array of approximate models.

```
Gr = balancmr(G,'MaxError',[0.1 0.5]);
size(Gr)
```

```
2x1 array of state-space models.
Each model has 5 outputs, 4 inputs, and between 24 and 26 states.
```

Examine the results.

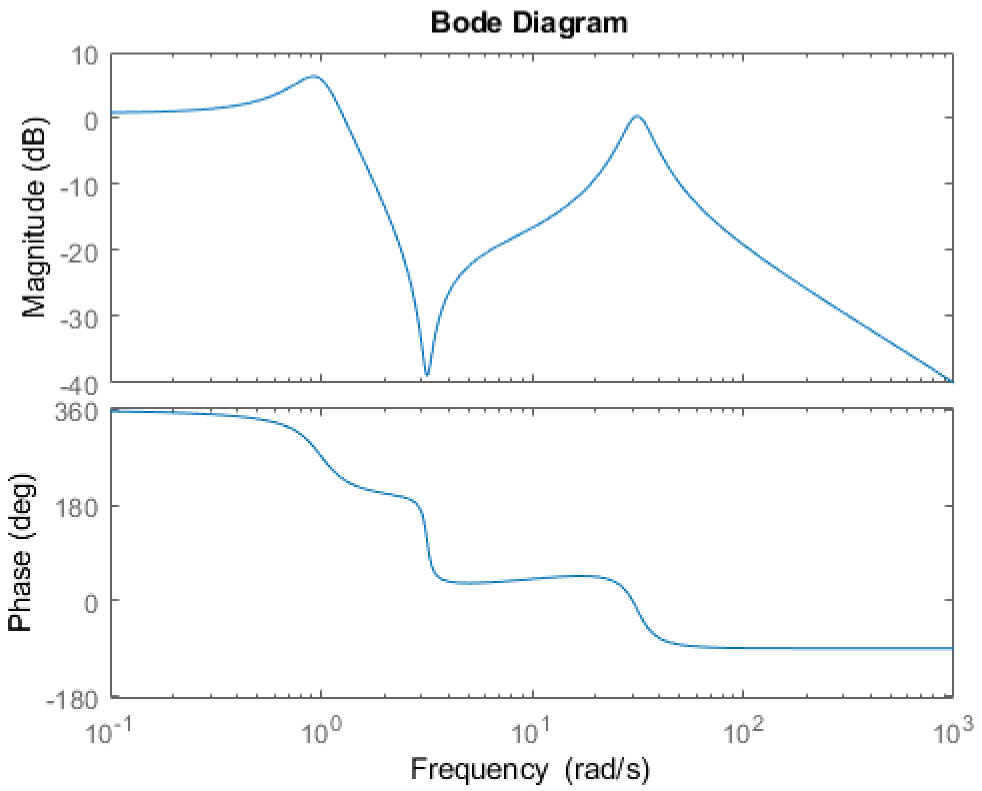
```
sigma(G,Gr)
```



Reduction with Focus on Particular Frequency Band

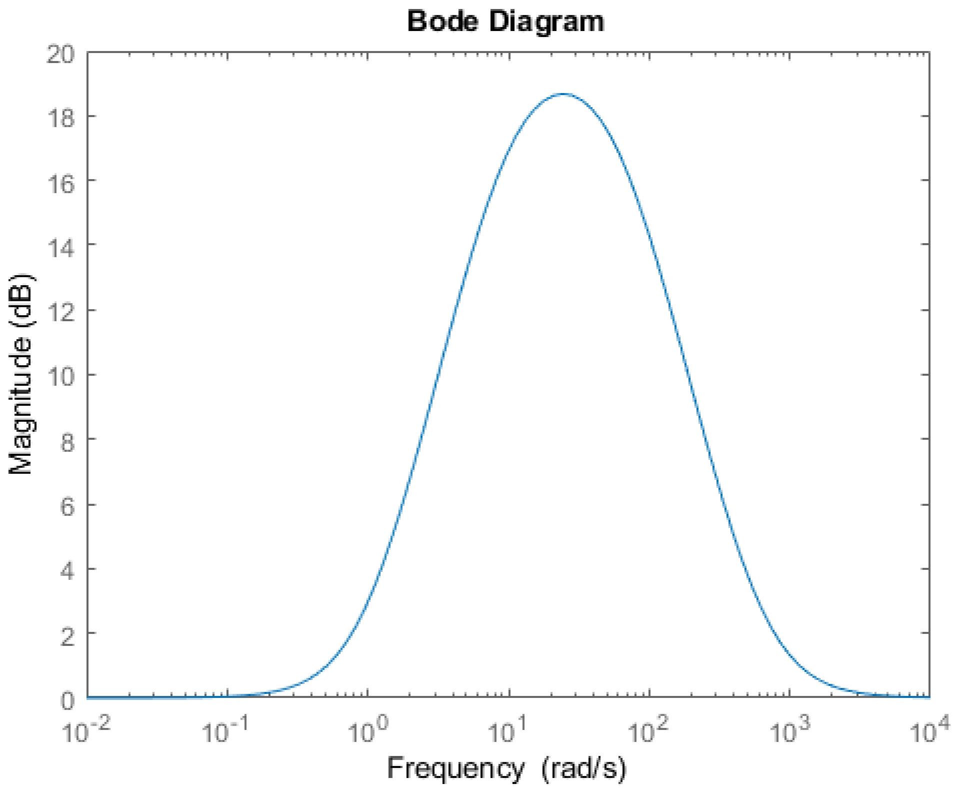
Reduce a 4th-order system to a second-order approximation with emphasis on the frequency band 10 rad/s - 100 rad/s. Consider the following system.

```
sys = tf(1,[1 0.5 1]) + tf(100*[1/10 1],[1 10 1000]);
bode(sys)
```



To focus the model-reduction algorithm on the higher-frequency dynamics, specify a function with a bandpass profile.

```
s = tf('s');
w1 = (s+1)/(s/10+1)/(s/60+1)*(s/600+1);
bodemag(w1)
```

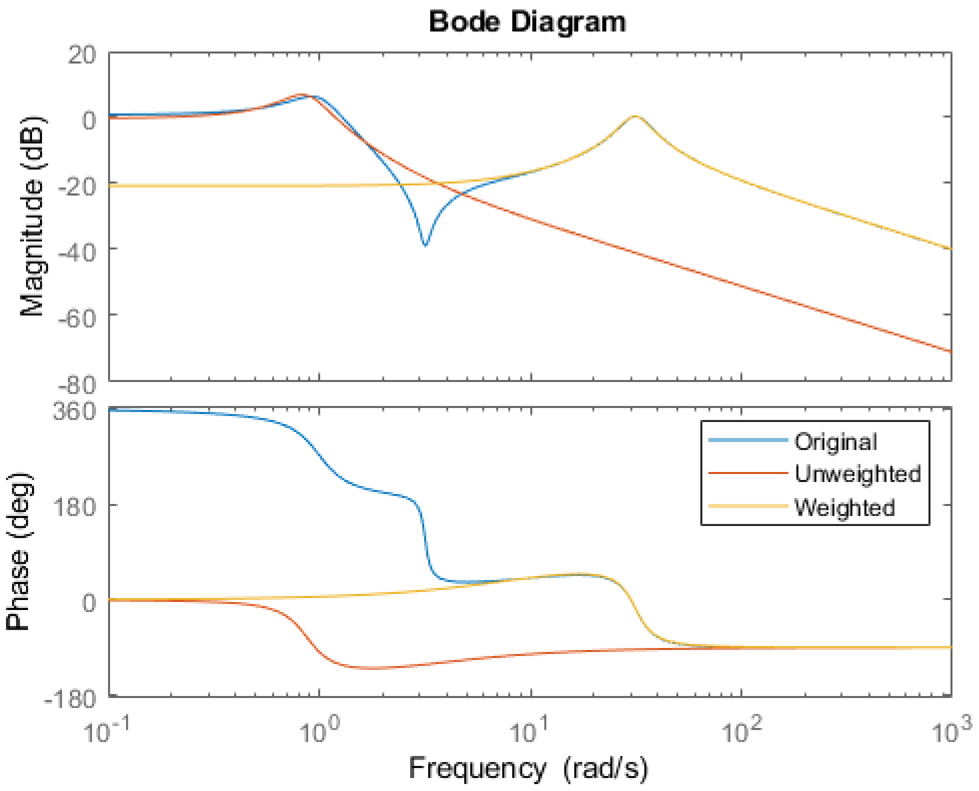


The plot confirms that the weighting function `w1` has the desired profile, peaking between 10 rad/s and 100 rad/s. To perform the reduction, specify the inverse of this profile as the output weight, using the `'Weights'` option of `balancmr`.

```
weight = {1/w1,1};
wrsys = balancmr(sys,2,'Weights',weight);
```

Compare the result with a second-order model obtained without the weighting.

```
rsys = balancmr(sys,2);
bode(sys,rsys,wrsys)
legend('Original','Unweighted','Weighted')
```



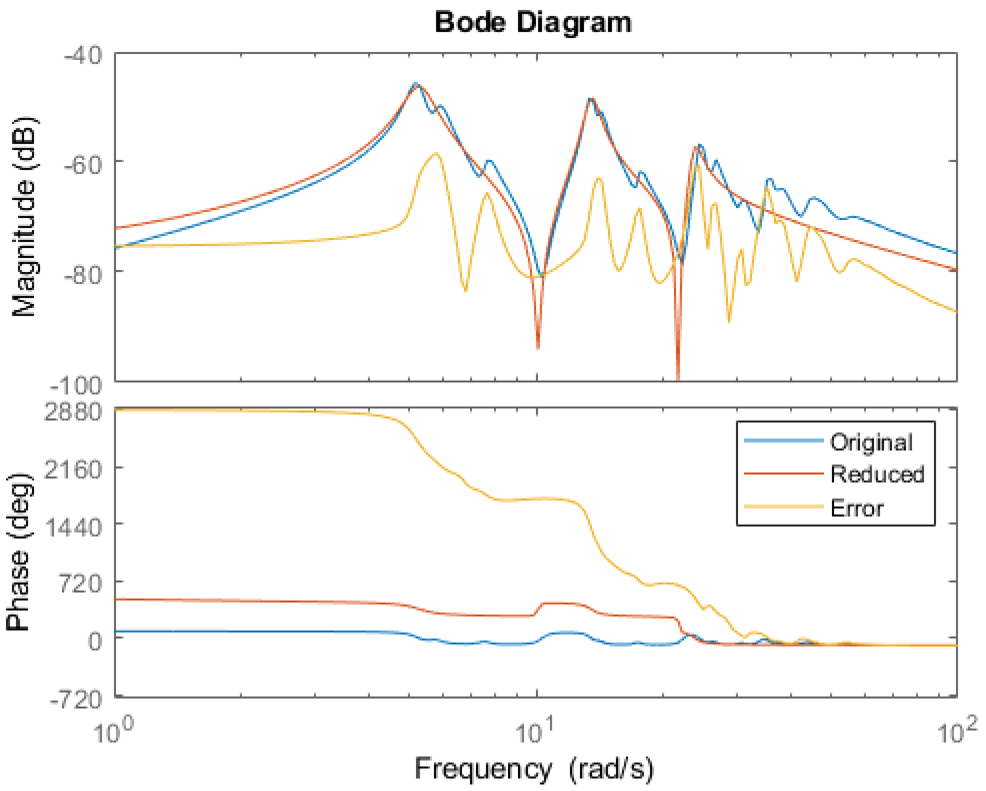
The model obtained with the weighting function provides a better match for the dynamics in the frequency band 10 rad/s - 100 rad/s.

Model Reduction With Frequency-Dependent Error Profile

Use a weighting function to control the frequency dependence of the error between the original and reduced models.

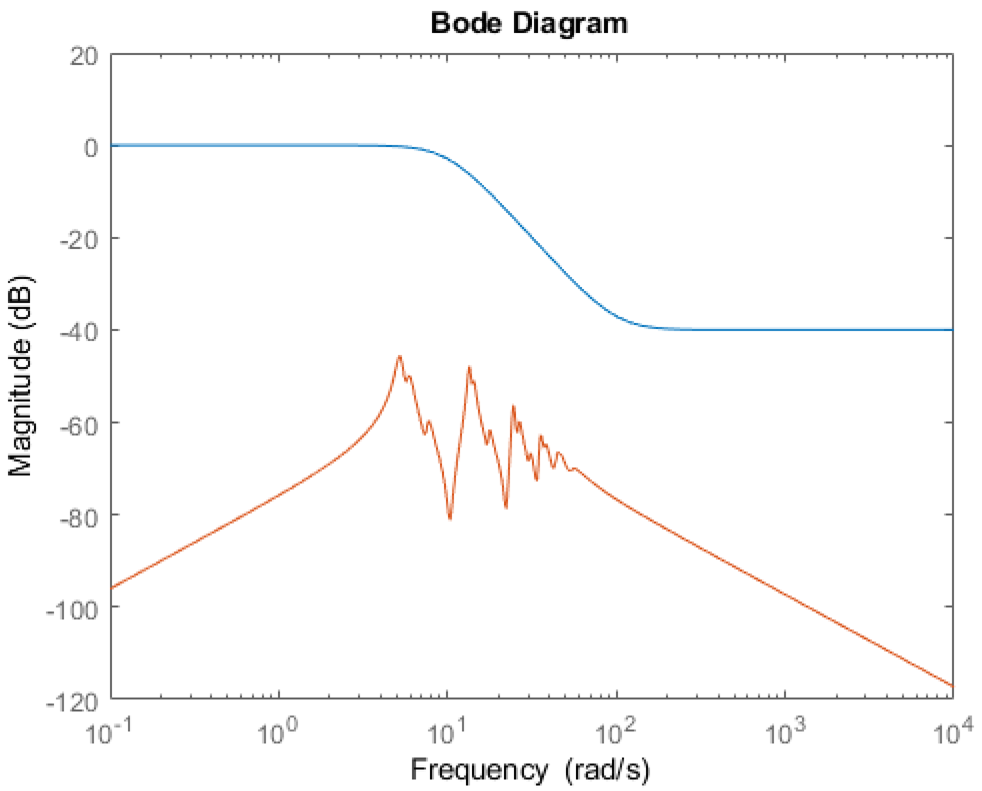
For this example, load a 48-state SISO model and reduce it to 6th order.

```
load(fullfile(matlabroot,'examples','robust','balancmrData.mat'),'bplant')
bplant6 = balancmr(bplant,6);
bode(bplant,bplant6,bplant-bplant6,logspace(0,2,200))
legend('Original','Reduced','Error')
```



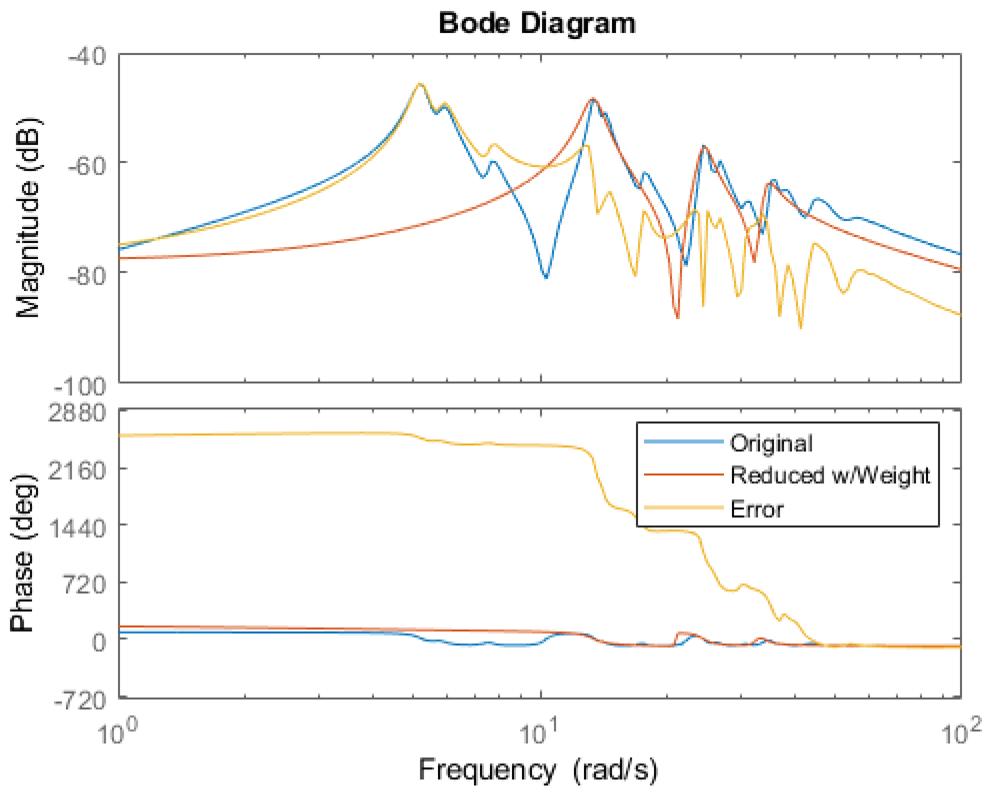
The error is fairly uniformly distributed in frequency. Create a weighting function that allows for a larger error at frequencies below 100 rad/s. In this case, use a biproper function that has unit gain at low frequency, but drops to -40 dB at higher frequencies.

```
W = tf(0.01*[1 1.4e2 1e4],[1 14 100]);
bodemag(W,bplant)
```



Reduce the model to 6th order again, using this weighting function. Because `bplant` is a SISO model, you can use the function as either input or output weight.

```
bplant6W = balancmr(bplant,6,'Weights',{W,1});
bode(bplant,bplant6W,bplant-bplant6W,logspace(0,2,200))
legend('Original','Reduced w/Weight','Error')
```



The error is now larger at low frequencies, with a correspondingly better match at high frequencies between the original and reduced models.

Algorithms

Given a state space (A,B,C,D) of a system and k, the desired reduced order, the following steps will produce a similarity transformation to truncate the original state-space system to the k^{th} order reduced model.

1 Find the SVD of the controllability and observability grammians

$$P = U_p \Sigma_p V_p^T$$

$$Q = U_q \Sigma_q V_q^T$$

2 Find the square root of the grammians (left/right eigenvectors)

$$L_p = U_p \Sigma_p^{½}$$

$$L_o = U_q \Sigma_q^{½}$$

3 Find the SVD of ($L_o^T L_p$)

$$L_o^T L_p = U \Sigma V^T$$

4 Then the left and right transformation for the final k^{th} order reduced model is

$$S_{L,BIG} = L_o U(:,1:k) \Sigma(1;k,1:k))^{-½}$$

$$S_{R,BIG} = L_p V(:,1:k) \Sigma(1;k,1:k))^{-½}$$

5 Finally,

$$\begin{bmatrix} \widehat{A} & \widehat{B} \\ \widehat{C} & \widehat{D} \end{bmatrix} = \begin{bmatrix} S_{L,BIG}^T A S_{R,BIG} & S_{L,BIG}^T B \\ C S_{R,BIG} & D \end{bmatrix}$$

The proof of the square root balance truncation algorithm can be found in [2].

References

[1] Glover, K., “All Optimal Hankel Norm Approximation of Linear Multivariable Systems, and Their Lµ-error Bounds,“ Int. J. Control, Vol. 39, No. 6, 1984, p. 1145-1193

[2] Safonov, M.G., and R.Y. Chiang, “A Schur Method for Balanced Model Reduction,” *IEEE Trans. on Automat. Contr.*, Vol. 34, No. 7, July 1989, p. 729-733

See Also

`balred` | `bstmr` | `hankelmr` | `hankelsv` | `ncfmr` | `reduce` | `schurmr`

Introduced before R2006a

bilin

Multivariable bilinear transform of frequency (*s* or *z*)

Syntax

```
GT = bilin(G,VERS,METHOD,AUG)
```

Description

`bilin` computes the effect on a system of the frequency-variable substitution,

$$s = \frac{\alpha z + \delta}{\gamma z + \beta}$$

The variable `VERS` denotes the transformation direction:

`VERS= 1`, forward transform ($s \rightarrow z$) or ($s \rightarrow \tilde{s}$).

`VERS=-1`, reverse transform ($z \rightarrow s$) or ($\tilde{s} \rightarrow s$).

This transformation maps lines and circles to circles and lines in the complex plane. People often use this transformation to do sampled-data control system design [1] or, in general, to do shifting of $j\omega$ modes [2], [3], [4].

`Bilin` computes several state-space bilinear transformations such as backward rectangular, etc., based on the `METHOD` you select

Bilinear Transform Types

Method	Type of bilinear transform
`'BwdRec'`	backward rectangular: $$s = \frac{z-1}{Tz}$$ `AUG` $= T$, the sample time.
`'FwdRec'`	forward rectangular: $$s = \frac{z-1}{T}$$ `AUG` $= T$, the sample time.
`'S_Tust'`	shifted Tustin: $$s = \frac{2}{T}\left(\frac{z-1}{\frac{z}{h}+1}\right)$$ `AUG` $= [T\ h]$, is the “shift” coefficient.
`'S_ftjw'`	shifted $j\omega$-axis, bilinear pole-shifting, continuous-time to continuous-time: $$s = \frac{\tilde{s}+p_1}{1+\tilde{s}/p_2}$$ `AUG` $= [p_2\ p_1]$.
`'G_Bili'`	`METHOD = 'G_Bili'`, general bilinear, continuous-time to continuous-time: $$s = \frac{\alpha\tilde{s}+\delta}{\gamma\tilde{s}+\beta}$$ `AUG` $= [\alpha\ \beta\ \gamma\ \delta]$.

Examples

Tustin Continuous s-Plane to Discrete z-Plane Transforms

Consider the following continuous-time plant (sampled at 20 Hz):

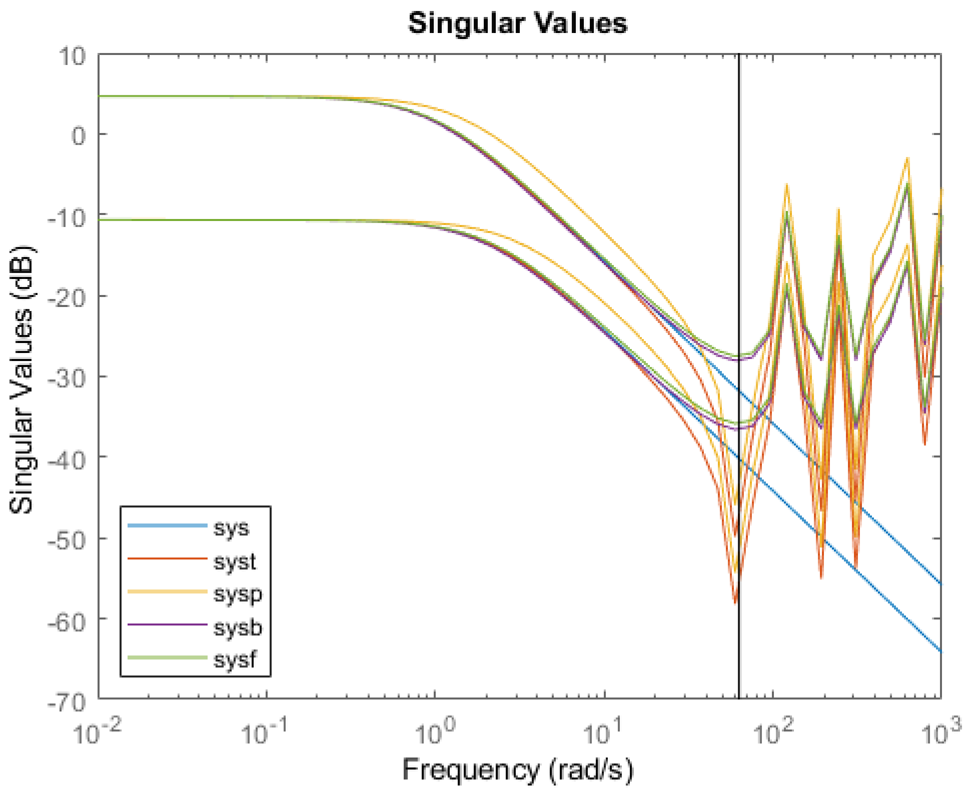
$$A = \begin{bmatrix} -1 & 1 \\ 0 & -2 \end{bmatrix},\ B = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},\ C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix};\ T_s = 0.05$$

Following is an example of four common “continuous to discrete” `bilin` transformations for the sampled plant:

```
A = [-1 1; 0 -2];
B = [1 0; 1 1];
C = [1 0; 0 1];
D = [0 0; 0 0];
sys = ss(A,B,C,D);                  % ANALOG
Ts = 0.05;  % sample time
syst = c2d(sys,Ts,'tustin');        % Tustin
sysp = c2d(sys,Ts,'prewarp',40);    % Pre-warped Tustin
sysb = bilin(sys,1,'BwdRec',Ts);    % Backward Rectangular
sysf = bilin(sys,1,'FwdRec',Ts);    % Forward Rectangular
```

Plot the response of the continuous-time plant and the transformed discrete-time plants.

```
w = logspace(-2,3,50); % frequencies to plot
sigma(sys,syst,sysp,sysb,sysf,w);
legend('sys','syst','sysp','sysb','sysf','Location','SouthWest')
```



Bilinear continuous to continuous pole-shifting

Design an H mixed-sensitivity controller for the ACC Benchmark plant

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

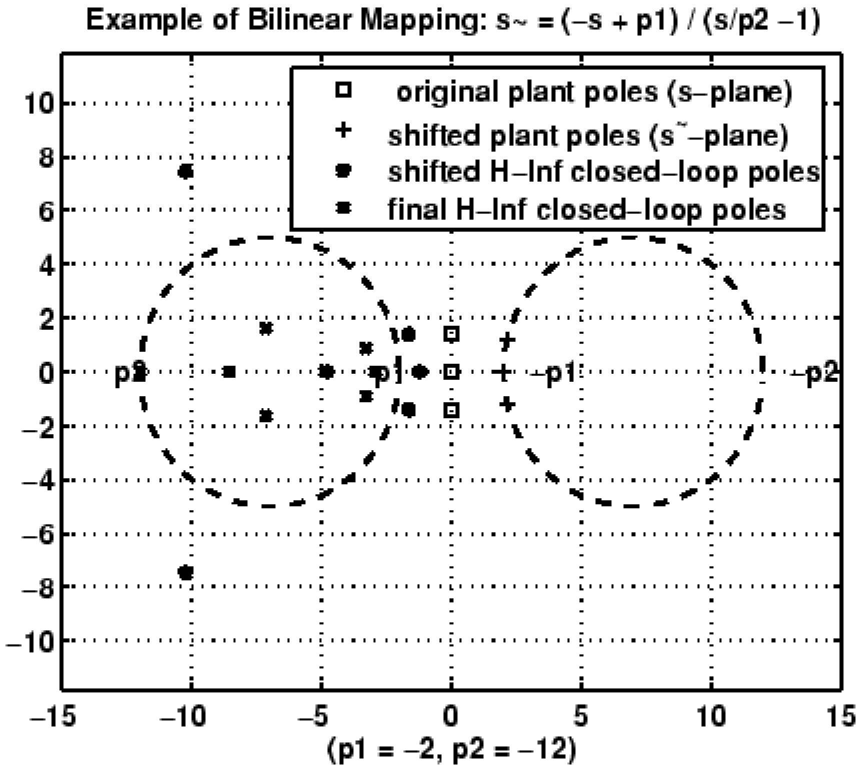
such that all closed-loop poles lie inside a circle in the left half of the s-plane whose diameter lies on between points [p1,p2]=[–12,–2]:

```
p1=-12; p2=-2; s=zpk('s');
G=ss(1/(s^2*(s^2+2)));          % original unshifted plant
Gt=bilin(G,1,'Sft_jw',[p1 p2]); % bilinear pole shifted plant Gt
```

```
Kt=mixsyn(Gt,1,[],1);          % bilinear pole shifted controller
K =bilin(Kt,-1,'Sft_jw',[p1 p2]); % final controller K
```

As shown in the following figure, closed-loop poles are placed in the left circle `[p1 p2]`. The shifted plant, which has its non-stable poles shifted to the inside the right circle, is

$$G_t(s) = 4.765 \times 10^{-5} \frac{(s-12)^4}{(s-2)^2(s^2 - 4.274s + 5.918)}$$



'S_ftjw' final closed-loop poles are inside the left [p1,p2] circle

Algorithms

`bilin` employs the state-space formulae in [3]:

$$\begin{bmatrix} A_b & B_b \\ C_b & D_b \end{bmatrix} = \begin{bmatrix} (\beta A - \delta I)(\alpha I + \gamma A)^{-1} & (\alpha\beta - \gamma\delta)(\alpha I - \gamma A)^{-1}B \\ C(\alpha I - \gamma A)^{-1} & D + \gamma C(\alpha I - \gamma A)^{-1}B \end{bmatrix}$$

References

[1] Franklin, G.F., and J.D. Powell, *Digital Control of Dynamics System*, Addison-Wesley, 1980.

[2] Safonov, M.G., R.Y. Chiang, and H. Flashner, “H_∞ Control Synthesis for a Large Space Structure,” *AIAA J. Guidance, Control and Dynamics*, 14, 3, p. 513-520, May/June 1991.

[3] Safonov, M.G., “Imaginary-Axis Zeros in Multivariable H_∞ Optimal Control”, in R.F. Curtain (editor), *Modelling, Robustness and Sensitivity Reduction in Control Systems*, p. 71-81, Springer-Varlet, Berlin, 1987.

[4] Chiang, R.Y., and M.G. Safonov, “H_∞ Synthesis using a Bilinear Pole Shifting Transform,” *AIAA, J. Guidance, Control and Dynamics*, vol. 15, no. 5, p. 1111-1117, September-October 1992.

See Also

`c2d` | `d2c` | `sectf`

Introduced before R2006a

bstmr

Balanced stochastic model truncation (BST) via Schur method

Syntax

```
GRED = bstmr(G)
```

```
GRED = bstmr(G,order)
```

```
[GRED,redinfo] = bstmr(G,key1,value1,...)
```

```
[GRED,redinfo] = bstmr(G,order,key1,value1,...)
```

Description

`bstmr` returns a reduced order model `GRED` of `G` and a struct array `redinfo` containing the error bound of the reduced model and Hankel singular values of the *phase matrix* of the original system [2].

The error bound is computed based on Hankel singular values of the phase matrix of `G`. For a stable system these values indicate the respective state energy of the system. Hence, reduced order can be directly determined by examining these values.

With only one input argument `G`, the function will show a Hankel singular value plot of the phase matrix of `G` and prompt for model order number to reduce.

This method guarantees an error bound on the infinity norm of the *multiplicative* ‖ `GRED`–1`(G-GRED)` ‖ ∞ or *relative error* ‖ `G`-–1`(G-GRED)` ‖ ∞ for well-conditioned model reduction problems [1]:

$$\left\| G^{-1}(G - Gred) \right\|_\infty \leq \prod_{k+1}^{n} \left(1 + 2\sigma_i(\sqrt{1 + \sigma_i^2} + \sigma_i)\right) - 1$$

This table describes input arguments for `bstmr`.

Argument	Description
`G`	LTI model to be reduced (without any other inputs will plot its Hankel singular values and prompt for reduced order)
`ORDER`	(Optional) an integer for the desired order of the reduced model, or a vector of desired orders for batch runs

A batch run of a serial of different reduced order models can be generated by specifying `order = x:y`, or a vector of integers. By default, all the anti-stable part of a system is kept, because from control stability point of view, getting rid of unstable state(s) is dangerous to model a system.

`'MaxError'` can be specified in the same fashion as an alternative for `'ORDER'`. In this case, reduced order will be determined when the accumulated product of Hankel singular values shown in the above equation reaches the `'MaxError'`.

Argument	Value	Description
`'MaxError'`	Real number or vector of different errors	Reduce to achieve H_∞ error. When present, `'MaxError'` overrides `ORDER` input.
`'Display'`	`'on'` or `'off'`	Display Hankel singular plots (default `'off'`).
`'Order'`	Integer, vector or cell array	Order of reduced model. Use only if not specified as 2nd argument.

This table describes output arguments.

Argument	Description
`GRED`	LTI reduced order model. Become multi-dimension array when input is a serial of different model order array.
`REDINFO`	A STRUCT array with three fields: • `REDINFO.ErrorBound` (bound on $\|G^{-1}(G\text{-}GRED)\|\infty$) • `REDINFO.StabSV` (Hankel SV of stable part of G) • `REDINFO.UnstabSV` (Hankel SV of unstable part of G)

`G` can be stable or unstable, continuous or discrete.

Examples

Given a continuous or discrete, stable or unstable system, `G`, the following commands can get a set of reduced order models based on your selections:

```
rng(1234,'twister');
G = rss(30,5,4);
G.D = zeros(5,4);
[g1, redinfo1] = bstmr(G); % display Hankel SV plot
                           % and prompt for order (try 15:20)
[g2, redinfo2] = bstmr(G,20);
[g3, redinfo3] = bstmr(G,[10:2:18]);
[g4, redinfo4] = bstmr(G,'MaxError',[0.01, 0.05]);
for i = 1:4
    figure(i)
    eval(['sigma(G,g' num2str(i) ');']);
end
```

Algorithms

Given a state space (A,B,C,D) of a system and k, the desired reduced order, the following steps will produce a similarity transformation to truncate the original state-space system to the k^{th} order reduced model.

1. Find the controllability grammian P and observability grammian Q of the left *spectral factor* $\Phi = \Gamma(\sigma)\Gamma^*(-\sigma) = \Omega^*(-\sigma)\Omega(\sigma)$ by solving the following Lyapunov and Riccati equations

$$AP + PA^T + BB^T = 0$$

$$B_W = PC^T + BD^T$$

$$QA + A^T Q + (QB_W - C^T)(-DD^T)(QB_W - C^T)^T = 0$$

2. Find the Schur decomposition for PQ in both ascending and descending order, respectively,

$$V_A^T PQV_A = \begin{bmatrix} \lambda_1 & \cdots & \cdots \\ 0 & \cdots & \cdots \\ 0 & 0 & \lambda_n \end{bmatrix}$$

$$V_D^T PQV_D = \begin{bmatrix} \lambda_n & \cdots & \cdots \\ 0 & \cdots & \cdots \\ 0 & 0 & \lambda_1 \end{bmatrix}$$

3 Find the left/right orthonormal eigen-bases of PQ associated with the k^{th} big Hankel singular values of the all-pass *phase matrix* $(W^*(s))^{-1}G(s)$.

$$k$$

$$V_A = [V_{R,SMALL}, \overset{k}{V}_{L,BIG}]$$
$$V_D = [V_{R,BIG}, V_{L,SMALL}]$$

4 Find the SVD of $(V^T{}_{L,BIG}V_{R,BIG}) = U\ \Sigma\ \varsigma\mathrm{T}$

5 Form the left/right transformation for the final k^{th} order reduced model

$$S_{L,BIG} = V_{L,BIG}\ U\ \Sigma(1{:}k,1{:}k)^{-1/2}$$

$$S_{R,BIG} = V_{R,BIG}\ V\ \Sigma(1{:}k,1{:}k)^{-1/2}$$

6 Finally,

$$\begin{bmatrix} \widehat{A} & \widehat{B} \\ \widehat{C} & \widehat{D} \end{bmatrix} = \begin{bmatrix} S_{L,BIG}^T A S_{R,BIG} & S_{L,BIG}^T B \\ C S_{R,BIG} & D \end{bmatrix}$$

The proof of the Schur BST algorithm can be found in [1].

Note The BST model reduction theory requires that the original model D matrix be full rank, for otherwise the Riccati solver fails. For any problem with strictly proper model, you can shift the $j\omega$-axis via `bilin` such that BST/REM approximation can be achieved up to a particular frequency range of interests. Alternatively, you can attach a small but full rank D matrix to the original problem but remove the D matrix of the reduced order model afterwards. As long as the size of D matrix is insignificant inside the control bandwidth, the reduced order model should be fairly close to the true model. By default, the `bstmr` program will assign a full rank D matrix scaled by 0.001 of the minimum eigenvalue of the original model, if its D matrix is not full rank to begin with. This serves the purpose for most problems if user does not want to go through the trouble of model pretransformation.

References

[1] Zhou, K., “Frequency-weighted model reduction with L∞ error bounds,” *Syst. Contr. Lett.*, Vol. 21, 115-125, 1993.

[2] Safonov, M.G., and R.Y. Chiang, “Model Reduction for Robust Control: A Schur Relative Error Method,” *International J. of Adaptive Control and Signal Processing,* Vol. 2, p. 259-272, 1988.

See Also

`balancmr` | `hankelmr` | `hankelsv` | `ncfmr` | `reduce` | `schurmr`

Introduced before R2006a

complexify

Replace ureal atoms by summations of ureal and ucomplex (or ultidyn) atoms

Syntax

```
MC = complexify(M,alpha)
```

```
MC = complexify(M,alpha,'ultidyn')
```

Description

The command complexify replaces ureal atoms with sums of ureal and ucomplex atoms using usubs. Optionally, the sum can consist of a ureal and ultidyn atom.

complexify is used to improve the conditioning of robust stability calculations (robstab) for situations when there are predominantly ureal uncertain elements.

MC = complexify(M,alpha) results in each ureal atom in MC having the same Name and NominalValue as the corresponding ureal atom in M. If Range is the range of one ureal atom from M, then the range of the corresponding ureal atom in MC is

```
[Range(1)+alpha*diff(Range)/2 Range(2)-alpha*diff(Range)/2]
```

The net effect is that the same real range is covered with a real and complex uncertainty. The real parameter range is reduced by equal amounts at each end, and alpha represents (in a relative sense) the reduction in the total range. The ucomplex atom will add this reduction in range back into MC, but as a ball with real and imaginary parts.

The ucomplex atom has NominalValue of 0, and Radius equal to alpha*diff(Range). Its name is the name of the original ureal atom, appended with the characters '_cmpxfy'.

MC = complexify(M,alpha,'ultidyn') is the same, except that gain-bounded ultidyn atoms are used instead of ucomplex atoms. The ultidyn atom has its Bound equal to alpha*diff(Range).

Examples

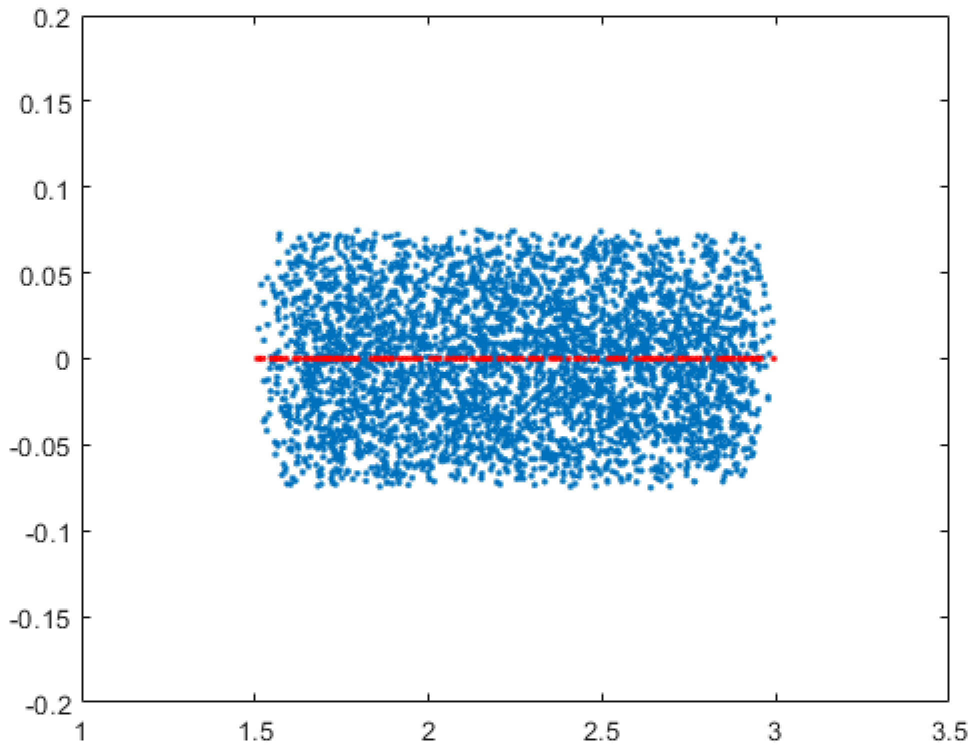
Complexified Uncertain Parameter

To illustrate complexification, create a uncertain real parameter, cast it to an uncertain matrix, and apply a 10% complexification.

```
a = umat(ureal('a',2.25,'Range',[1.5 3]));
b = complexify(a,.1);
as = usample(a,200);
bs = usample(b,4000);
```

Make a scatter plot of the values that the complexified matrix (scalar) can take, as well as the values of the original uncertain real parameter.

```
plot(real(bs(:)),imag(bs(:)),'.',real(as(:)),imag(as(:)),'r.')
axis([1 3.5 -0.2 0.2])
```



See Also

`icomplexify` | `robstab`

Topics

Getting Reliable Estimates of Robustness Margins

Introduced in R2007a

cmsclsyn

Approximately solve constant-matrix, upper bound µ-synthesis problem

Syntax

```
[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure);
```

```
[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure,opt);
```

```
[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure,opt,qinit);
```

```
[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure,opt,'random',N)
```

Description

`cmsclsyn` approximately solves the constant-matrix, upper bound µ-synthesis problem by minimization,

$$\min_{Q \in C^{r \times t}} \mu_{\Delta}(R + UQV)$$

for given matrices $R \in \mathbf{C}^{n} x_{m}$, $U \in \mathbf{C}^{n} x_{r}$, $V \in \mathbf{C}^{t} x_{m}$, and a set $\Delta \subset \mathbf{C}^{m} x_{n}$. This applies to constant matrix data in R, U, and V.

`[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure)` minimizes, by choice of Q. `QOPT` is the optimum value of Q, the upper bound of `mussv(R+U*Q*V,BLK), BND`. The matrices `R,U` and `V` are constant matrices of the appropriate dimension. `BlockStructure` is a matrix specifying the perturbation blockstructure as defined for `mussv`.

`[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure,OPT)` uses the options specified by `OPT` in the calls to `mussv`. See `mussv` for more information. The default value for `OPT` is `'cUsw'`.

`[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure,OPT,QINIT)` initializes the iterative computation from Q = `QINIT`. Because of the nonconvexity of the overall problem, different starting points often yield different final answers. If `QINIT` is an N-D array, then the iterative computation is performed multiple times - the `i`'th optimization is initialized at Q = `QINIT(:,:,i)`. The output arguments are associated with the best solution obtained in this brute force approach.

`[QOPT,BND] = cmsclsyn(R,U,V,BlockStructure,OPT,'random',N)` initializes the iterative computation from `N` random instances of `QINIT`. If `NCU` is the number of columns of `U`, and `NRV` is the number of rows of `V`, then the approximation to solving the constant matrix μ synthesis problem is two-fold: only the upper bound for μ is minimized, and the minimization is not convex, hence the optimum is generally not found. If `U` is full column rank, or `V` is full row rank, then the problem can (and is) cast as a convex problem, [Packard, Zhou, Pandey and Becker], and the global optimizer (for the upper bound for μ) is calculated.

Algorithms

The `cmsclsyn` algorithm is iterative, alternatively holding Q fixed, and computing the `mussv` upper bound, followed by holding the upper bound multipliers fixed, and minimizing the bound implied by choice of Q. If `U` or `V` is square and invertible, then the optimization is reformulated (exactly) as an linear matrix inequality, and solved directly, without resorting to the iteration.

References

Packard, A.K., K. Zhou, P. Pandey, and G. Becker, "A collection of robust control problems leading to LMI's," *30th IEEE Conference on Decision and Control,* Brighton, UK, 1991, p. 1245–1250.

See Also

`hinfsyn` | `mussv` | `musyn` | `robgain` | `robstab`

Introduced before R2006a

dcgainmr

Reduced order model

Syntax

```
[sysr,syse,gain] = dcgainmr(sys,ord)
```

Description

`[sysr,syse,gain] = dcgainmr(sys,ord)` returns a reduced order model of a continuous-time LTI system SYS by truncating modes with least DC gain.

Specify your LTI continuous-time system in sys. The order is specified in `ord`.

This function returns:

- `sysr`—The reduced order models (a multidimensional array if `sys` is an LTI array)
- `syse`—The difference between `sys` and `sysr` (`syse=sys-sysr`)
- `gain`—The g-factors (dc-gains)

The DC gain of a complex mode

```
(1/(s+p))*c*b'
```

is defined as

```
norm(b)*norm(c)/abs(p)
```

See Also

`reduce`

Introduced in R2008a

decay

Quadratic decay rate of polytopic or affine P-systems

Syntax

```
[drate,P] = decay(ps,options)
```

Description

For affine parameter-dependent systems

$$E(p)\dot{x} = A(p)x, \quad p(t) = (p_1(t), \ldots, p_n(t)),$$

or polytopic systems

$$E(t)\dot{x} = A(t)x, \quad (A, E) \in \mathrm{Co}\{(A_1, E_1), \ldots, (A_n, E_n)\},$$

`decay` returns the quadratic decay rate `drate`, i.e., the smallest $\alpha \in R$ such that

$$A^{T}QE + EQA^{T} < \alpha Q$$

holds for some Lyapunov matrix $Q > 0$ and all possible values of (A, E). Two control parameters can be reset via `options(1)` and `options(2)`:

- If `options(1)=0` (default), `decay` runs in fast mode, using the least expensive sufficient conditions. Set `options(1)=1` to use the least conservative conditions.
- `options(2)` is a bound on the condition number of the Lyapunov matrix P. The default is 109.

See Also

`pdlstab` | `psys` | `quadstab`

Introduced before R2006a

decinfo

Describe how entries of matrix variable X relate to decision variables

Syntax

```
decinfo(lmisys)
```

```
decX = decinfo(lmisys,X)
```

Description

The function decinfo expresses the entries of a matrix variable X in terms of the decision variables $x_1, \ldots, x_N$. Recall that the decision variables are the free scalar variables of the problem, or equivalently, the free entries of all matrix variables described in `lmisys`. Each entry of X is either a hard zero, some decision variable x_n, or its opposite $-x_n$.

If `X` is the identifier of X supplied by `lmivar`, the command

```
decX = decinfo(lmisys,X)
```

returns an integer matrix `decX` of the same dimensions as X whose (i, j) entry is

- 0 if $X(i, j)$ is a hard zero
- n if $X(i, j) = x_n$ (the n-th decision variable)
- $-n$ if $X(i, j) = -x_n$

`decX` clarifies the structure of X as well as its entry-wise dependence on $x_1, \ldots, x_N$. This is useful to specify matrix variables with atypical structures (see `lmivar`).

`decinfo` can also be used in interactive mode by invoking it with a single argument. It then prompts the user for a matrix variable and displays in return the decision variable content of this variable.

Examples

Example 1

Consider an LMI with two matrix variables *X* and *Y* with structure:

- $X = x\, I_3$ with *x* scalar
- *Y* rectangular of size 2-by-1

If these variables are defined by

```
setlmis([])
X = lmivar(1,[3 0])
Y = lmivar(2,[2 1])
    :
    :
lmis = getlmis
```

the decision variables in *X* and *Y* are given by

```
dX = decinfo(lmis,X)

dX =
    1     0     0
    0     1     0
    0     0     1

dY = decinfo(lmis,Y)

dY =
    2
    3
```

This indicates a total of three decision variables x_1, x_2, x_3 that are related to the entries of *X* and *Y* by

$$X = \begin{pmatrix} x_1 & 0 & 0 \\ 0 & x_1 & 0 \\ 0 & 0 & x_1 \end{pmatrix}, Y = \begin{pmatrix} x_2 \\ x3 \end{pmatrix}$$

Note that the number of decision variables corresponds to the number of free entries in *X* and *Y* when taking structure into account.

Example 2

Suppose that the matrix variable X is symmetric block diagonal with one 2-by-2 full block and one 2-by-2 scalar block, and is declared by

```
setlmis([])
X = lmivar(1,[2 1;2 0])
          :
lmis = getlmis
```

The decision variable distribution in X can be visualized interactively as follows:

```
decinfo(lmis)

There are 4 decision variables labeled x1 to x4 in this problem.

Matrix variable Xk of interest (enter k between 1 and 1, or 0 to quit):

?> 1

The decision variables involved in X1 are among {-x1,...,x4}.
Their entry-wise distribution in X1 is as follows
        (0,j>0,-j<0 stand for 0,xj,-xj, respectively):

X1 :

    1     2     0     0
    2     3     0     0
    0     0     4     0
    0     0     0     4

             *********

Matrix variable Xk of interest (enter k between 1 and 1, or 0 to quit):

?> 0
```

See Also

`dec2mat` | `lmivar` | `mat2dec`

Introduced before R2006a

decnbr

Total number of decision variables in system of LMIs

Syntax

```
ndec = decnbr(lmisys)
```

Description

The function `decnbr` returns the number `ndec` of decision variables (free scalar variables) in the LMI problem described in `lmisys`. In other words, `ndec` is the length of the vector of decision variables.

Examples

For an LMI system `lmis` with two matrix variables *X* and *Y* such that

- *X* is symmetric block diagonal with one 2-by-2 full block, and one 2-by-2 scalar block
- *Y* is 2-by-3 rectangular,

the number of decision variables is

```
ndec = decnbr(LMIs)

ndec =
       10
```

This is exactly the number of free entries in *X* and *Y* when taking structure into account (see `decinfo` for more details).

See Also

`dec2mat` | `decinfo` | `mat2dec`

Introduced before R2006a

dec2mat

Given values of decision variables, derive corresponding values of matrix variables

Syntax

```
valX = dec2mat(lmisys,decvars,X)
```

Description

Given a value `decvars` of the vector of decision variables, `dec2mat` computes the corresponding value `valX` of the matrix variable with identifier `X`. This identifier is returned by `lmivar` when declaring the matrix variable.

Recall that the decision variables are all free scalar variables in the LMI problem and correspond to the free entries of the matrix variables $X_1, \ldots, X_K$. Since LMI solvers return a feasible or optimal value of the vector of decision variables, `dec2mat` is useful to derive the corresponding feasible or optimal values of the matrix variables.

Examples

See the description of `feasp`.

See Also

`decinfo` | `decnbr` | `mat2dec`

Introduced before R2006a

defcx

Help specify $c^T x$ objectives for mincx solver

Syntax

```
[V1,...,Vk] = defcx(lmisys,n,X1,...,Xk)
```

Description

`defcx` is useful to derive the c vector needed by `mincx` when the objective is expressed in terms of the matrix variables.

Given the identifiers `X1,...,Xk` of the matrix variables involved in this objective, `defcx` returns the values `V1,...,Vk` of these variables when the *n*-th decision variable is set to one and all others to zero.

See Also

`decinfo` | `mincx`

Introduced before R2006a

dellmi

Remove LMI from system of LMIs

Syntax

```
newsys = dellmi(lmisys,n)
```

Description

`dellmi` deletes the n-th LMI from the system of LMIs described in `lmisys`. The updated system is returned in `newsys`.

The ranking `n` is relative to the order in which the LMIs were declared and corresponds to the identifier returned by `newlmi`. Since this ranking is not modified by deletions, it is safer to refer to the remaining LMIs by their identifiers. Finally, matrix variables that only appeared in the deleted LMI are removed from the problem.

Examples

Suppose that the three LMIs

$$A_1^T X_1 + X_1 A_1 + Q_1 < 0$$
$$A_2^T X_2 + X_2 A_2 + Q_2 < 0$$
$$A_3^T X_3 + X_3 A_3 + Q_3 < 0$$

have been declared in this order, labeled `LMI1`, `LMI2`, `LMI3` with `newlmi`, and stored in `lmisys`. To delete the second LMI, type

```
lmis = dellmi(lmisys,LMI2)
```

`lmis` now describes the system of LMIs

$$A_1^T X_1 + X_1 A_1 + Q_1 < 0$$
$$A_3^T X_3 + X_3 A_3 + Q_3 < 0$$

and the second variable X_2 has been removed from the problem since it no longer appears in the system.

To further delete `LMI3` from the system, type

```
lmis = dellmi(lmis,LMI3)
```

or equivalently

```
lmis = dellmi(lmis,3)
```

Note that the system has retained its original ranking after the first deletion.

See Also

`lmiedit` | `lmiinfo` | `newlmi`

Introduced before R2006a

delmvar

Remove one matrix variable from LMI problem

Syntax

```
newsys = delmvar(lmisys,X)
```

Description

`delmvar` removes the matrix variable *X* with identifier `X` from the list of variables defined in `lmisys`. The identifier `X` should be the second argument returned by `lmivar` when declaring *X*. All terms involving *X* are automatically removed from the list of LMI terms. The description of the resulting system of LMIs is returned in `newsys`.

Examples

Consider the LMI

$$0 < \begin{pmatrix} A^TY + B^TYA + Q & CX + D \\ X^TC^T + D^T & -(X + X^T) \end{pmatrix}$$

involving two variables *X* and *Y* with identifiers `X` and `Y`. To delete the variable *X*, type

```
lmisys = delmvar(lmisys,X)
```

Now `lmisys` describes the LMI

$$0 < \begin{pmatrix} A^TYB + B^TYA + Q & D \\ D^T & 0 \end{pmatrix}$$

with only one variable *Y*. Note that *Y* is still identified by the label `Y`.

See Also

`lmiinfo` | `lmivar` | `setmvar`

Introduced before R2006a

diag

Diagonalize vector of uncertain matrices and systems

Syntax

```
v = diag(x)
```

Description

If `x` is a vector of uncertain system models or matrices, `diag(x)` puts `x` on the main diagonal. If `x` is a matrix of uncertain system models or matrices, `diag(x)` is the main diagonal of `x`. `diag(diag(x))` is a diagonal matrix of uncertain system models or matrices.

Examples

The statement produces a diagonal system `mxg` of size 4-by-4. Given multivariable system `xx`, a vector of the diagonal elements of `xxg` is found using `diag`.

```
x = rss(3,4,1);
xg = frd(x,logspace(-2,2,80));
size(xg)

FRD model with 4 output(s) and 1 input(s), at 80 frequency point(s).

mxg = diag(xg);
size(mxg)
FRD model with 4 output(s) and 4 input(s), at 80 frequency point(s).

xxg = [xg(1:2,1) xg(3:4,1)];
m = diag(xxg);
size(m)
FRD model with 2 output(s) and 1 input(s), at 80 frequency point(s).
```

See Also
append

Introduced before R2006a

diskmargin

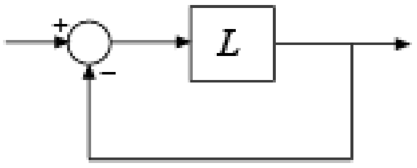
Disk-based stability margins of feedback loops

Syntax

```
[DM,MM] = diskmargin(L)
MMIO = diskmargin(P,C)
___ = diskmargin(___,E)
```

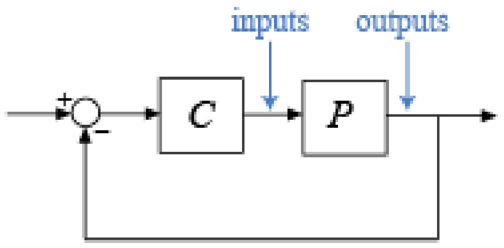
Description

`[DM,MM] = diskmargin(L)` computes the disk-based stability margins for the SISO or MIMO negative feedback loop `feedback(L,eye(N))`, where `N` is the number of inputs and outputs in `L`.



The `diskmargin` command returns loop-at-a-time stability margins in `DM` and multiloop margins in `MM`. Disk-based margin analysis provides a stronger guarantee of stability than the classical gain and phase margins. For general information about disk margins, see “Stability Analysis Using Disk Margins”.

`MMIO = diskmargin(P,C)` computes the stability margins when considering independent, concurrent variations at both the plant inputs and plant outputs the negative feedback loop of the following diagram.



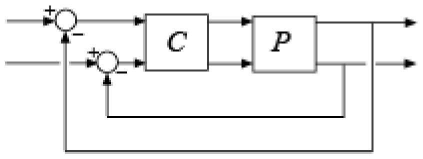
`____ = diskmargin(____,E)` specifies an additional eccentricity parameter that varies the shape of the uncertainty region used to compute the stability margins. You can use the eccentricity argument with any of the previous syntaxes.

Examples

Disk Margins for MIMO Feedback Loop

Use `diskmargin` to compute loop-at-a-time and multiloop disk margins. This example illustrates that loop-at-a-time margins can give an overly optimistic assessment of the true robustness of MIMO feedback loops. Margins of individual loops can be sensitive to small perturbations within other loops.

Consider the closed-loop system of the following illustration.



P and *C* are 2-by-2 (MIMO) systems. Construct *P* in state-space form, and compute the disk-based margins at the plant output.

```
a = [0 10;-10 0];
b = eye(2);
c = [1 8;-10 1];
d = zeros(2,2);
```

```
P = ss(a,b,c,d);
C = [1 -2;0 1];
L = P*C;
[DM,MM] = diskmargin(L);
```

Examine the loop-at-a-time disk margins, returned in the structure array `DM`. Each entry in this structure array contains the stability margins of the corresponding channel.

```
DM(1)
```

```
ans = struct with fields:
     GainMargin: [0 Inf]
    PhaseMargin: [-90 90]
     DiskMargin: 2
     LowerBound: 2
     UpperBound: 2
      Frequency: Inf
```

```
DM(2)
```

```
ans = struct with fields:
     GainMargin: [0 Inf]
    PhaseMargin: [-90 90]
     DiskMargin: 2
     LowerBound: 2
     UpperBound: 2
      Frequency: Inf
```

For each channel, the closed-loop system remains stable for any variation in gain or phase.

Here, gain and phase variations are a model of uncertainty in the plant. In practice, plant uncertainty affects both channels simultaneously. To estimate the stability margins with respect to such independent and concurrent uncertainty, examine the multiloop disk margins.

```
MM
```

```
MM = struct with fields:
     GainMargin: [0.6071 1.6472]
    PhaseMargin: [-27.4762 27.4762]
     DiskMargin: 0.4890
     LowerBound: 0.4890
     UpperBound: 0.4899
      Frequency: 0.2250
```

The result is a more stringent limit on tolerable variations (and thus tolerable uncertainty) than the loop-at-a-time margins. `MM.GainMargin` shows that if the loop gains in both channels are multiplied independently by values between about 0.6 and about 1.6, the closed-loop system is guaranteed to remain stable. Similarly, stability is preserved against independent phase variations in each channel of about ±27.5°. The frequency at which these smallest margins occur is at 22.5 rad/s.

It can be useful to compute margins at the plant inputs separately from those at the plant outputs, because there is typically uncertainty in both the actuators (inputs) and sensors (outputs). Using `L = P*C` computes margins at the outputs. Use `L = C*P` to compute margins at the inputs.

```
[DMI,MMI] = diskmargin(C*P);
DMI(1)
```

```
ans = struct with fields:
     GainMargin: [0 Inf]
    PhaseMargin: [-90 90]
     DiskMargin: 2
     LowerBound: 2
     UpperBound: 2
      Frequency: 0
```

```
DMI(2)
```

```
ans = struct with fields:
     GainMargin: [0.4750 2.1053]
    PhaseMargin: [-39.1846 39.1846]
     DiskMargin: 0.7119
     LowerBound: 0.7119
     UpperBound: 0.7119
      Frequency: 0
```

At the outputs, both feedback channels were stable against any variation. Here at the inputs, however, the second channel is guaranteed stable only for a limited range of variations. The multiloop margin at the plant input yields a more stringent estimate of the tolerance of the closed-loop system to variations at the input to the plant.

```
MMI
```

```
MMI = struct with fields:
     GainMargin: [0.7288 1.3721]
    PhaseMargin: [-17.8304 17.8304]
     DiskMargin: 0.3137
     LowerBound: 0.3137
     UpperBound: 0.3144
      Frequency: 0
```

Finally, compute the multiloop margin against simultaneous variations in gain (or phase) at both the plant inputs and plant outputs. This multiloop margin provides the most conservative guarantee of closed-loop stability.

```
MMIO = diskmargin(P,C)
```

```
MMIO = struct with fields:
     GainMargin: [0.8270 1.2092]
    PhaseMargin: [-10.8190 10.8190]
     DiskMargin: 0.1894
     LowerBound: 0.1894
     UpperBound: 0.1898
      Frequency: 0
```

This result shows that for independent and concurrent variations in both channels and at inputs and outputs, stability is guaranteed for changes in loop gain of a factor between about 0.83 and 1.2. Likewise, loop phase can vary only by about ±10.8°. The frequency at which the smallest margins occur is DC.

Disk Margins Based on Sensitivity and Complementary Sensitivity

`diskmargin` bases its computation on a shifted sensitivity function $S + (E - 1)I/2$, where $S = (I + L)^{-1}$ is the sensitivity function and E is the eccentricity parameter. By default, $E = 0$, which corresponds to the balanced sensitivity function $S - I/2 = (S - T)/2$, where $T = I - S$ is the complementary sensitivity function. Setting $E = 1$ computes disk margins based on S, while $E = -1$ computes the disk margins based on T. You can try different values of E and combine the results to obtain less conservative stability margins than you get from the balanced sensitivity function alone.

Compute margins based on the complementary sensitivity, balanced sensitivity, and sensitivity functions for the system with open-loop transfer function given by

$$L = \frac{25}{s^3 + 10s^2 + 10s + 10}.$$

```
L = tf(25,[1 10 10 10]);
DMt = diskmargin(L,-1);
DMb = diskmargin(L);
DMs = diskmargin(L,1);
```

Compare the resulting gain margins.

```
DMt.GainMargin
```

```
ans = 1×2

    0.5136    1.4864
```

```
DMb.GainMargin
```

```
ans = 1×2

    0.6273    1.5942
```

```
DMs.GainMargin
```

```
ans = 1×2

    0.7132    1.6726
```

The different *E* values give different ranges for the estimated gain margins. Each of these is a different estimate for the true gain margins, and each guarantees stability for gain variations within its range. Therefore, the closed-loop system is guaranteed to remain stable for all variations in the union of all three ranges, or variations in the open-loop gain of a factor between about 0.51 and about 1.67.

For some systems, larger positive or negative *E* values can yield an even larger range of guaranteed stability.

```
DMnegE = diskmargin(L,-100);
DMposE = diskmargin(L,100);
DMnegE.GainMargin
```

```
ans = 1×2
```

```
    0.0761    1.0100
```

```
DMposE.GainMargin
```

```
ans = 1×2

    0.9902    1.9109
```

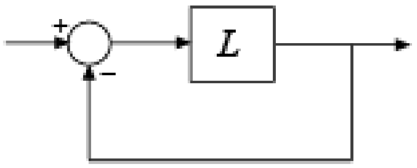
These values extend the range such that stability is guaranteed for gain variations of a factor between about 0.08 and about 1.91.

For more information about the variation of gain margin estimates with *E*, see “Stability Analysis Using Disk Margins”.

Input Arguments

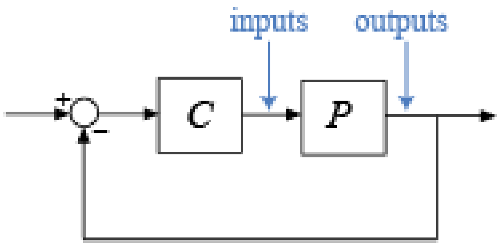
L — Open-loop response
dynamic system model | model array

Open-loop response, specified as a dynamic system model. `L` can be SISO or MIMO, as long as it has the same number of inputs and outputs. `diskmargin` computes the disk-based stability margins for the negative-feedback closed-loop system `feedback(L,eye(N))`.



To compute the disk margins of the positive feedback system `feedback(L,eye(N),+1)`, use `diskmargin(-L)`.

When you have a controller `P` and a plant `C`, you can compute the disk margins for gain (or phase) variations at the plant inputs or outputs, as in the following diagram.



- To compute margins at the plant outputs, set `L = P*C`.
- To compute margins at the plant inputs, set `L = C*P`.

`L` can be continuous time or discrete time. If `L` is a generalized state-space model (`genss` or `uss`) then `diskmargin` uses the current or nominal value of all control design blocks in `L`.

If `L` is a frequency-response data model (such as `frd`), then `diskmargin` computes the margins at each frequency represented in the model. The function returns the margins at the frequency with the smallest disk margin.

If `L` is a model array, then `diskmargin` computes margins for each model in the array.

P — Plant
dynamic system model

Plant, specified as a dynamic system model. `P` can be SISO or MIMO, as long as `P*C` has the same number of inputs and outputs. `diskmargin` computes the disk-based stability margins for a negative-feedback closed-loop system. To compute the disk margins of the system with positive feedback, use `diskmargin(P,-C)`.

`P` can be continuous time or discrete time. If `P` is a generalized state-space model (`genss` or `uss`) then `diskmargin` uses the current or nominal value of all control design blocks in `P`.

If `P` is a frequency-response data model (such as `frd`), then `diskmargin` computes the margins at each frequency represented in the model. The function returns the margins at the frequency with the smallest disk margin.

C — Controller
dynamic system model

Controller, specified as a dynamic system model. C can be SISO or MIMO, as long as P*C has the same number of inputs and outputs. diskmargin computes the disk-based stability margins for a negative-feedback closed-loop system. To compute the disk margins of the system with positive feedback, use diskmargin(P,-C).

C can be continuous time or discrete time. If C is a generalized state-space model (genss or uss) then diskmargin uses the current or nominal value of all control design blocks in C.

If C is a frequency-response data model (such as frd), then diskmargin computes the margins at each frequency represented in the model. The function returns the margins at the frequency with the smallest disk margin.

E — Eccentricity

0 (default) | real scalar

Eccentricity of uncertainty region used to compute the stability margins, specified as a real scalar value. Use this parameter to vary the shape of the uncertainty region used to model gain and phase variations. Varying the eccentricity parameter yields lower estimates of the true stability margins, letting you infer a larger region of guaranteed stability than that obtained using the default E = 0. Some special values of E include:

- 0 — Margins based on balanced sensitivity function
- 1 — Margins based on sensitivity function
- –1 — Margins based on complementary sensitivity function

For an example, see “Disk Margins Based on Sensitivity and Complementary Sensitivity” on page 1-68. For more detailed information about how the choice of E affects the margin computation, see “Stability Analysis Using Disk Margins”.

Output Arguments

DM — Disk margins for each feedback channel

structure | structure array

Disk margins for each feedback channel with all other loops closed, returned as a structure for SISO feedback loops, or an *N*-by-1 structure array for a MIMO loop with *N* feedback channels. The fields of DM(i) are:

- `GainMargin` — Disk-based gain margin of the corresponding feedback channel, returned as a vector of the form `[gmin,gmax]`. These values express in absolute units the amount by which the loop gain in that channel can decrease or increase while preserving stability. For example, if `DM(i).GainMargin = [0.8,1.25]` then the gain of the *i*th loop can be multiplied by any factor between 0.8 and 1.25 without causing instability. When `E` = 0, `gmin = 1/gmax`. If the closed-loop system is unstable, then `DM(i).GainMargin = [1 1]`.
- `PhaseMargin` — Disk-based phase margin of the corresponding feedback channel, returned as a vector of the form `[-pm,pm]` in degrees. These values express the amount by which the loop phase in that channel can decrease or increase while preserving stability. If the closed-loop system is unstable, then `DM(i).PhaseMargin = [0 0]`.
- `DiskMargin` — Maximum $|\Delta|$ compatible with closed-loop stability for the corresponding feedback channel. Δ parameterizes the uncertainty in the loop response (see “Algorithms” on page 1-75). If the closed-loop system is unstable, then `DM(i).DiskMargin = 0`.
- `LowerBound` — Lower bound on disk margin. This value is the same as `DiskMargin`.
- `UpperBound` — Upper bound on disk margin. This value represents an upper limit on the actual disk margin of the system. In other words, the disk margin is guaranteed to be no worse than `LowerBound` and no better than `UpperBound`.
- `Frequency` — Frequency at which the weakest margin occurs for the corresponding loop channel. This value is in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `L`.

When `L = P*C` is the open-loop response of a system comprising a controller and plant with unit negative feedback in each channel, `DM` contains the stability margins for variations at the plant outputs. To compute the stability margins for variations at the plant inputs, use `L = C*P`. To compute the stability margins for simultaneous, independent variations at both the plant inputs and outputs, use `MMIO = diskmargin(P,C)`.

When `L` is a model array, `DM` has additional dimensions corresponding to the array dimensions of `L`. For instance, if `L` is a 1-by-3 array of two-input, two-output models, then `DM` is a 2-by-3 structure array. `DM(j,k)` contains the margins for the jth feedback channel of the kth model in the array.

`MM` — Multiloop disk margins
structure

Multiloop disk margins, returned as a structure. The gain (or phase) margins quantify how much gain variation (or phase variation) the system can tolerate in all feedback channels at once while remaining stable. Thus, `MM` is a single structure regardless of the number of feedback channels in the system. (For SISO systems, `MM` = `DM`.) The fields of `MM` are:

- `GainMargin` — Multiloop gain margin, returned as a vector of the form `[gmin,gmax]`. These values express in absolute units the amount by which the loop gain can vary in all channels independently and concurrently while preserving stability. For example, if `MM.GainMargin = [0.8,1.25]` then the gain of all loops can be multiplied by any factor between 0.8 and 1.25 without causing instability. When `E` = 0, `gmin = 1/gmax`.
- `PhaseMargin` — Multiloop phase margin, returned as a vector of the form `[-pm,pm]` in degrees. These values express the amount by which the loop phase can vary in all channels independently and concurrently while preserving stability.
- `DiskMargin` — Maximum $|\Delta|$ compatible with closed-loop stability. Δ parameterizes the uncertainty in the loop response (see "Algorithms" on page 1-75).
- `LowerBound` — Lower bound on disk margin. This value is the same as `DiskMargin`.
- `UpperBound` — Upper bound on disk margin. This value represents an upper limit on the actual disk margin of the system. In other words, the disk margin is guaranteed to be no worse than `LowerBound` and no better than `UpperBound`.
- `Frequency` — Frequency at which the weakest margin occurs. This value is in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `L`.

When `L = P*C` is the open-loop response of a system comprising a controller and plant with unit negative feedback in each channel, `MM` contains the stability margins for variations at the plant outputs. To compute the stability margins for variations at the plant inputs, use `L = C*P`. To compute the stability margins for simultaneous, independent variations at both the plant inputs and outputs, use `MMIO = diskmargin(P,C)`.

When `L` is a model array, `MM` is a structure array with one entry for each model in `L`.

`MMIO` — Disk margins for independent variations in all input and output channels
structure

Disk margins for independent variations in all input and output channels of the plant `P`, returned as a structure having the same fields as `MM`.

Tips

- `diskmargin` assumes negative feedback. To compute the disk margins of a positive feedback system, use `diskmargin(-L)` or `diskmargin(P,-C)`.
- To compute disk margins for a system modeled in Simulink®, first linearize the model to obtain the open-loop response at a particular operating point. Then, use `diskmargin` to compute stability margins for the linearized system. For more information, see “Stability Margins of a Simulink Model”.
- To compute classical gain and phase margins, use `allmargin`.

Algorithms

For SISO L, the uncertainty model for disk-margin analysis incorporates a multiplicative complex uncertainty Δ into the loop transfer function as follows:

$$L \rightarrow L_{\Delta} = L\frac{1 + \Delta(1 - E)/2}{1 - \Delta(1 + E)/2} = L(1 + \delta_L), \quad |\Delta| < \alpha\,.$$

For Δ = 0, the multiplicative factor is 1, corresponding to the nominal L. As Δ varies in the ball |Δ| < α, the gain and phase of the multiplicative factor are a model for gain and phase variation in L. The eccentricity parameter E varies the shape of the applied uncertainty in the complex plane. The disk margin is the smallest radius α at which the closed-loop system becomes unstable[1]. From the disk margin α, `diskmargin` derives the minimum gain and phase margins.

For MIMO systems, `diskmargin` applies an analogous uncertainty model that allows the uncertainty to vary independently in each channel.

For further details about the computation and interpretation of disk margins, see “Stability Analysis Using Disk Margins”.

References

[1] Blight, J.D., R.L. Dailey, and D. Gangsaas. "Practical Control Law Design for Aircraft Using Multivariable Techniques." *International Journal of Control*. Vol. 59, Number 1, 1994, pp. 93–137.

See Also

`allmargin` | `diskmargin` | `margin` | `wcdiskmargin`

Topics

"Stability Analysis Using Disk Margins"
"Stability Margins of a Simulink Model"

Introduced in R2018b

dksyn

Robust controller design for discrete-time plants using µ-synthesis

Syntax

`[k,clp,bnd] = dksyn(p,nmeas,ncont)`

`[k,clp,bnd] = dksyn(p,nmeas,ncont,opt)`

`[k,clp,bnd,dkinfo] = dksyn(p,nmeas,ncont,...)`

`[k,clp,bnd,dkinfo] = dksyn(p,nmeas,ncont,prevdkinfo,opt)`

`[...] = dksyn(p)`

Description

`[k,clp,bnd] = dksyn(p,nmeas,ncont)` synthesizes a robust controller `k` for the uncertain open-loop plant model `p` via the D-K or D-G-K algorithm for µ-synthesis. `p` is an uncertain state-space `uss` model. The last `nmeas` outputs and `ncont` inputs of `p` are assumed to be the measurement and control channels. `k` is the controller, `clp` is the closed-loop model and `bnd` is the robust closed-loop performance bound. `p`, `k`, `clp`, and `bnd` are related as follows:

```
clp = lft(p,k);
bnd1 = dksynperf(clp);
bnd = 1/bnd1.LowerBound;
```

`[k,clp,bnd] = dksyn(p,nmeas,ncont,opt)` specifies user-defined options `opt` for the D-K or D-K-G algorithm. Use `dksynOptions` to create `opt`.

`[k,clp,bnd,dkinfo] = dksyn(p,nmeas,ncont,...)` returns a log of the algorithm execution in `dkinfo`. `dkinfo` is an *N*-by-1 cell array where N is the total number of iterations performed. The `ith` cell contains a structure with the following fields:

Field	Description
`K`	Controller at `i`th iteration, a `ss` object
`Bnds`	Robust performance bound on the closed-loop system (`double`)
`DL`	Left D-scale, an `ss` object
`DR`	Right D-scale, an `ss` object
`GM`	Offset G-scale, an `ss` object
`GR`	Right G-scale, an `ss` object
`GFC`	Center G-scale, an `ss` object
`MussvBnds`	Upper and lower μ bounds, an `frd` object
`MussvInfo`	Structure returned from `mussv` at each iteration.

`[k,clp,bnd,dkinfo] = dksyn(p,nmeas,ncont,prevdkinfo,opt) allows you to use information from a previous dksyn iteration.` `prevdkinfo` is a structure from a previous attempt at designing a robust controller using `dksyn`. `prevdkinfo` is used when the `dksyn` starting iteration is not 1 (`opt.StartingIterationNumber = 1`) to determine the correct D-scalings to initiate the iteration procedure.

`[...] = dksyn(p)` takes `p` as a `uss` object that has two-input/two-output partitioning as defined by `mktito`.

Examples

The following statements create a robust performance control design for an unstable, uncertain single-input/single-output plant model. The nominal plant model, G, is an unstable first order system $\frac{s}{s-1}$.

```
G = tf(1,[1 -1]);
```

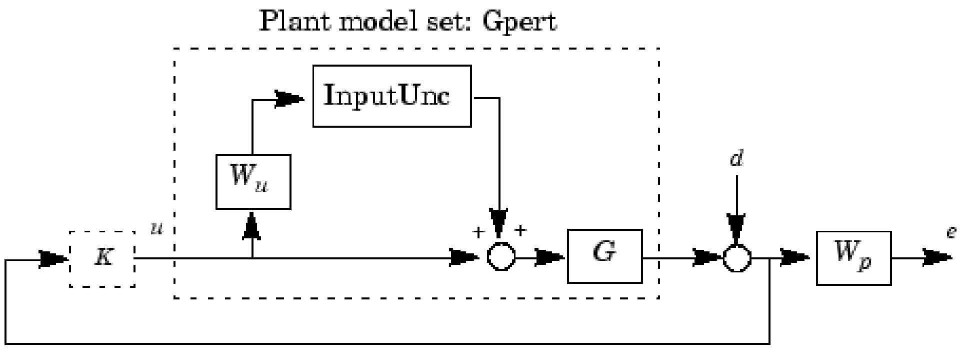
The model itself is uncertain. At low frequency, below 2 rad/s, it can vary up to 25% from its nominal value. Around 2 rad/s the percentage variation starts to increase and reaches 400% at approximately 32 rad/s. The percentage model uncertainty is represented by the weight `Wu` which corresponds to the frequency variation of the model uncertainty and the uncertain LTI dynamic object `InputUnc`.

```
Wu = 0.25*tf([1/2 1],[1/32 1]);
InputUnc = ultidyn('InputUnc',[1 1]);
```

The uncertain plant model `Gpert` represents the model of the physical system to be controlled.

```
Gpert = G*(1+InputUnc*Wu);
```

The robust stability objective is to synthesize a stabilizing LTI controller for all the plant models parameterized by the uncertain plant model, `Gpert`. The performance objective is defined as a weighted sensitivity minimization problem. The control interconnection structure is shown in the following figure.



The sensitivity function, S, is defined as

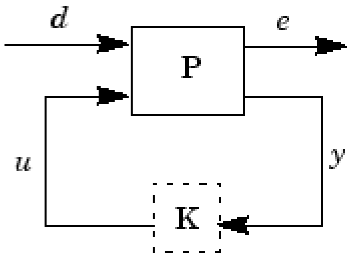
$$S = \frac{1}{1 + PK}$$

where `P` is the plant model and `K` is the controller. A weighted sensitivity minimization problem selects a weight `Wp`, which corresponds to the *inverse* of the desired sensitivity function of the closed-loop system as a function of frequency. Hence the product of the sensitivity weight `Wp` and actual closed-loop sensitivity function is less than 1 across all frequencies. The sensitivity weight `Wp` has a gain of 100 at low frequency, begins to decrease at 0.006 rad/s, and reaches a minimum magnitude of 0.25 after 2.4 rad/s.

```
Wp = tf([1/4 0.6],[1 0.006]);
```

The defined sensitivity weight `Wp` implies that the desired disturbance rejection should be at least 100:1 disturbance rejection at DC, rise slowly between 0.006 and 2.4 rad/s, and allow the disturbance rejection to increase above the open-loop level, 0.25, at high frequency.

When the plant model is uncertain, the closed-loop performance objective is to achieve the desired sensitivity function for all plant models defined by the uncertain plant model, `Gpert`. The performance objective for an uncertain system is a robust performance objective. A block diagram of this uncertain closed-loop system illustrating the performance objective (closed-loop transfer function from *d*→*e*) is shown.



From the definition of the robust performance control objective, the weighted, uncertain control design interconnection model, which includes the robustness and performance objectives, can be constructed and is denoted by `P`. The robustness and performance weights are selected such that if the robust performance structure singular value, `bnd`, of the closed-loop uncertain system, `clp`, is less than 1 then the performance objectives have been achieved for all the plant models in the model set.

You can form the uncertain transfer matrix `P` from `[d; u]` to `[e; y]` using the following commands.

```
P = [Wp; 1 ]*[1 Gpert];
[K,clp,bnd] = dksyn(P,1,1);
bnd
```

```
bnd =
    0.6806
```

The controller `K` achieves a robust performance µ value `bnd` of about 0.68. Therefore you have achieved the robust performance objectives for the given problem.

You can use the `robgain` command to analyze the closed-loop robust performance of `clp`.

```
[rpmarg,rpmargunc] = robgain(clp,1);
```

Limitations

There are two shortcomings of the D-K iteration control design procedure:

- Calculation of the structured singular value $\mu_\Delta(\cdot)$ is approximated by its upper bound. This is not a serious problem because the value of μ and its upper bound are often close.
- The D-K iteration is not guaranteed to converge to a global, or even local minimum. This is a serious problem, and represents the biggest limitation of the design procedure.

In spite of these drawbacks, the D-K iteration control design technique appears to work well on many engineering problems. It has been applied to a number of real-world applications with success. These applications include vibration suppression for flexible structures, flight control, chemical process control problems, and acoustic reverberation suppression in enclosures.

Tutorials

Control of Spring-Mass-Damper Using Mixed mu-Synthesis

Algorithms

`dksyn` synthesizes a robust controller via D-K iteration. The D-K iteration procedure is an approximation to μ-synthesis control design. The objective of μ-synthesis is to minimize the structure singular value μ of the corresponding robust performance problem associated with the uncertain system `p`. The uncertain system `p` is an open-loop interconnection containing known components including the nominal plant model, uncertain parameters, `ucomplex`, and unmodeled LTI dynamics, `ultidyn`, and performance and uncertainty weighting functions. You use weighting functions to include magnitude and frequency shaping information in the optimization. The control objective is to synthesize a stabilizing controller `k` that minimizes the robust performance μ value, which corresponds to `bnd`.

The D-K iteration procedure involves a sequence of minimizations, first over the controller variable *K* (holding the *D* variable associated with the scaled μ upper bound fixed), and then over the *D* variable (holding the controller *K* variable fixed). The D-K iteration procedure is not guaranteed to converge to the minimum μ value, but often works well in practice.

`dksyn` automates the D-K iteration procedure and the options object `dksynOptions` allows you to customize its behavior. Internally, the algorithm works with the generalized scaled plant model `P`, which is extracted from a `uss` object using the command `lftdata`.

The following is a list of what occurs during a single, complete step of the D-K iteration.

1 (In the first iteration, this step is skipped.) The μ calculation (from the previous step) provides a frequency-dependent scaling matrix, D_f. The fitting procedure fits these scalings with rational, stable transfer function matrices. After fitting, plots of

$$\bar{\sigma}\left(\widehat{D}_f(j\omega)F_L(P,K)(j\omega)D_f^{-1}(j\omega)\right)$$

and

$$\bar{\sigma}\left(\widehat{D}_f(j\omega)F_L(P,K)(j\omega)\widehat{D}_f^{-1}(j\omega)\right)$$

are shown for comparison.

(In the first iteration, this step is skipped.) The rational $\widehat{D}$ is absorbed into the open-loop interconnection for the next controller synthesis. Using either the previous frequency-dependent D's or the just-fit rational $\widehat{D}$, an estimate of an appropriate value for `the` H_∞ norm is made. This is simply a conservative value of the scaled closed-loop H_∞ `norm`, using the most recent controller and either a frequency sweep (using the frequency-dependent D's) or a state-space calculation (with the rational D's).

2 (The first iteration begins at this point.) A controller is designed using H_∞ synthesis on the scaled open-loop interconnection. If you set the `DisplayWhileAutoIter` field in `dksynOptions` to `'on'`, the following information is displayed:

   a The progress of the γ-iteration is displayed.

   b The singular values of the closed-loop frequency response are plotted.

   c You are given the option to change the frequency range. If you change it, all relevant frequency responses are automatically recomputed.

   d You are given the option to rerun the H_∞ synthesis with a set of modified parameters if you set the `AutoIter` field in `dksynOptions` to `'off'`. This is convenient if, for instance, the bisection tolerance was too large, or if `maximum gamma value` was too small.

3 The structured singular value of the closed-loop system is calculated and plotted.

4 An iteration summary is displayed, showing all the controller order, as well as the peak value of μ of the closed-loop frequency responses.

5 The choice of stopping or performing another iteration is given.

Subsequent iterations proceed along the same lines without the need to reenter the iteration number. A summary at the end of each iteration is updated to reflect data from all previous iterations. This often provides valuable information about the progress of the robust controller synthesis procedure.

Interactive Fitting of D-Scalings

Setting the `AutoIter` field in `dksynOptions` to `'off'` requires that you interactively fit the *D*-scales each iteration. During step 2 of the D-K iteration procedure, you are prompted to enter your choice of options for fitting the *D*-scaling data. You press return after, the following is a list of your options.

```
Enter Choice (return for list):
  Choices:
        nd    Move to Next D-scaling
        nb    Move to Next D-Block
        i    Increment Fit Order
        d    Decrement Fit Order
        apf    Auto-PreFit
        mx 3    Change Max-Order to 3
        at 1.01    Change Auto-Prefit Tol to 1.01
        0    Fit with zeroth order
        2    Fit with second order
        n    Fit with n'th order
        e    Exit with Current Fittings
        s    See Status
```

- `nd` and `nb` allow you to move from one *D*-scale data to another. `nd` moves to the next scaling, whereas `nb` moves to the next scaling block. For scalar *D*-scalings, these are identical operations, but for problems with full *D*-scalings, (perturbations of the form δ*I*) they are different. In the (1,2) subplot window, the title displays the *D*-scaling block number, the row/column of the scaling that is currently being fitted, and the order of the current fit (with `d` for data when no fit exists).

- You can increment or decrement the order of the current fit (by 1) using `i` and `d`.
- `apf` automatically fits each *D*-scaling data. The default maximum state order of individual *D*-scaling is 5. The `mx` variable allows you to change the maximum *D*-scaling state order used in the automatic prefitting routine. `mx` must be a positive, nonzero integer. `at` allows you to define how close the rational, scaled μ upper bound is to approximate the actual μ upper bound in a norm sense. Setting `at` to `1` would require an exact fit of the *D*-scale data, and is not allowed. Allowable values for `at` are greater than 1. This setting plays a role (mildly unpredictable, unfortunately) in determining where in the (*D*,*K*) space the D-K iteration converges.
- Entering a positive integer at the prompt will fit the current *D*-scale data with that state order rational transfer function.
- `e` exits the *D*-scale fitting to continue the D-K iteration.
- The variable `s` displays a status of the current and fits.

References

[1] Balas, G.J., and J.C. Doyle, "Robust control of flexible modes in the controller crossover region," *AIAA Journal of Guidance, Dynamics and Control*, Vol. 17, no. 2, March-April, 1994, p. 370-377.

[2] Balas, G.J., A.K. Packard, and J.T. Harduvel, "Application of μ-synthesis techniques to momentum management and attitude control of the space station," *AIAA Guidance, Navigation and Control Conference*, New Orleans, August 1991.

[3] Doyle, J.C., K. Lenz, and A. Packard, "Design examples using μ-synthesis: Space shuttle lateral axis FCS during reentry," *NATO ASI Series, Modelling, Robustness, and Sensitivity Reduction in Control Systems*, vol. 34, Springer-Verlag, Berlin 1987.

[4] Packard, A., J. Doyle, and G. Balas, "Linear, multivariable robust control with a μ perspective," *ASME Journal of Dynamic Systems, Measurement and Control*, 50th Anniversary Issue, Vol. 115, no. 2b, June 1993, p. 310-319.

[5] Stein, G., and J. Doyle, "Beyond singular values and loopshapes," *AIAA Journal of Guidance and Control*, Vol. 14, No. 1, January, 1991, p. 5-16.

See Also

`dksynOptions` | `dksynperf` | `h2syn` | `hinfsyn` | `mktito` | `mussv` | `robgain` | `robstab` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

dksynOptions

Set options for dksyn

Syntax

```
opt = dksynOptions
opt = dksynOptions(Name,Value)
```

Description

`opt = dksynOptions` returns the default options for `dksyn`.

`opt = dksynOptions(Name,Value)` returns an option set with additional options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`dksynOptions` takes the following `Name` arguments:

`FrequencyVector`

Frequencies for mu-analysis, specified as a vector. When empty, `dksyn` automatically chooses the frequency range and number of points.

Default: `[]`

`InitialController`

Controller for initializing first iteration, specified as a state-space (`ss`) model.

Default: `[]`

`AutoIter`

Automated mu-synthesis mode, specified as either `'on'` or `'off'`. When automated mu-synthesis mode is off, `dksyn` performs an interactive D-K iteration procedure. You are prompted to fit the D-scale data and provide input on the control design process.

Default: `'on'`

`DisplayWhileAutoIter`

Status of display in automated mu-synthesis mode, specified as either `'off'` or `'on'`. When the display is on, and automated mu-synthesis mode is active, `dksyn` displays the iteration progress during the synthesis computation.

Default: `'off'`

`StartingIterationNumber`

Iteration number for initiating iteration procedure, specified as a positive integer. Use this option when you provide the `prevdkinfo` argument to `dksyn` to use information from a previous `dksyn` calculation. In this case, specify the starting iteration number from which to resume the iteration procedure.

Default: 1

`NumberOfAutoIterations`

Number of iterations to perform in automatic mu-synthesis mode, specified as a positive integer.

Default: 10

`MixedMU`

Flag indicating whether to perform mixed real/complex mu-synthesis when real parameters are present, specified as either `'off'` or `'on'`. Mixed mu-synthesis accounts for uncertain real parameters directly in the synthesis process. Setting `'MixedMU'` to 'on' when you have uncertain real parameters can result in improved robust performance of the synthesized controller.

Default: 'off'

AutoScalingOrder

State order for fitting *D*-scaling and *G*-scaling data for real/complex mu-synthesis, specified as a vector of the form [dorder,gorder].

Default: [5 2] (5th-order *D*-scalings and 2nd-order *G*-scalings)

AutoIterSmartTerminate

Automatic termination mode, specified as either 'on' or 'off'. When AutoIterSmartTerminate is 'on', the iteration procedure terminates based on the progress of the design iteration. Set the tolerance for automatic termination using AutoIterSmartTerminateTol.

In automatic termination mode, the iteration procedure terminates when a stopping criterion is satisfied. The stopping criterion involves the objective value (peak value, across frequency, of the upper bound for μ) in the current iteration, denoted v_0. The stopping criterion also involves the objective value in the previous two iterations, denoted v_{-1} and v_{-2}. The stopping criterion is satisfied for lack of progress if:

$$|v_0 - v_{-1}| < AutoIterSmartTerminateTol * v_0,$$

and

$$|v_{-1} - v_{-2}| < AutoIterSmartTerminateTol * v_0 .$$

The stopping criteria is also satisfied for an undesirable significant increase in the objective value if:

$$v_0 > v_{-1} + 20 * AutoIterSmartTerminateTol * v_0 .$$

Default: 'on'

AutoIterSmartTerminateTol

Tolerance for AutoIterSmartTerminate mode.

Default: 0.005

Output Arguments

`options`

Option set containing the specified options for the `dksyn` command.

Examples

Create Options Set for dksyn

Create an options set for a `dksyn` run using a logarithmic distribution of frequency points for analysis and performing 24 iterations.

```
options = dksynOptions('FrequencyVector',logspace(-2,3,80),...
                       'NumberOfAutoIterations',24);
```

Alternatively, use dot notation to set the values of `options`.

```
options = dksynOptions;
options.FrequencyVector = logspace(-2,3,80);
options.NumberOfAutoIterations = 24;
```

See Also

`dksyn`

Introduced in R2013a

dksynperf

Robust H_∞ performance optimized by `dksyn`

Syntax

```
[gamma,wcu] = dksynperf(clp)
[gamma,wcu] = dksynperf(clp,w)
[gamma,wcu] = dksynperf( ___ ,opts)
[gamma,wcu,info] = dksynperf( ___ )
```

Description

The robust H_∞ performance (or robust H_∞ norm) of an uncertain closed-loop system is the smallest value γ such that the I/O gain of the system stays below γ for all modeled uncertainty up to size $1/\gamma$ (in normalized units). The `dksyn` function synthesizes a robust controller by minimizing this quantity over all possible choices of controller. `dksynperf` computes this quantity for a specified uncertain model.

`[gamma,wcu] = dksynperf(clp)` calculates the robust H_∞ performance for an uncertain closed-loop system, `clp`. The robust H_∞ performance is the smallest value γ for which the peak I/O gain stays below γ for all modeled uncertainty up to $1/\gamma$, in normalized units. For example, a value of $\gamma = 1.125$ implies the following:

- The I/O gain of `clp` remains less than 1.125 as long as the uncertain elements stay within 0.8 normalized units of their nominal values. In other words, for uncertain element values within 0.8 normalized units, the largest possible H_∞ norm is 1.125.
- There is a perturbation of size 0.8 normalized units that drives the peak I/O gain to 1.125.

The peak I/O gain is the maximum I/O gain over all inputs, which is also the peak of the largest singular value over all frequencies and uncertainties. In other words, if Δ represents all possible values of the uncertain parameters in the closed-loop transfer function $CLP(j\omega)$, then

$$\gamma = \max_{\Delta} \max_{\omega} \sigma_{max}(CLP(j\omega)) .$$

The output structure `gamma` contains upper and lower bounds on the robust H_∞ performance and the critical frequency at which the I/O gain of `clp` reaches the lower bound. The structure `wcu` contains the uncertain-element values that drive the peak I/O gain to the lower bound.

`[gamma,wcu] = dksynperf(clp,w)` computes the robust H_∞ performance at the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `dksynperf` restricts the computation to the interval between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `dksynperf` computes the H_∞ performance at the specified frequencies only.

`[gamma,wcu] = dksynperf( ___ ,opts)` specifies additional options for the computation. Use `robOptions` to create `opts`. You can use this syntax with any of the previous input-argument combinations.

`[gamma,wcu,info] = dksynperf( ___ )` returns a structure with additional information about the H_∞ performance values and the perturbations that drive the I/O gain to γ. See `info` for details about this structure. You can use this syntax with any of the previous input-argument combinations.

Examples

Reduce Synthesized Controller While Preserving Robust Performance

When you use `dksyn` to synthesize an unstructured robust controller, the resulting controller often is of higher order than is necessary to achieve the desired robust performance. One way to mitigate this problem is to perform model reduction, using `dksynperf` to test the robust performance of the reduced-order controller.

Synthesize a controller for the system described in the `dksyn` example.

```
G = tf(1,[1 -1]);
Wu = 0.25*tf([1/2 1],[1/32 1]);
InputUnc = ultidyn('InputUnc',[1 1]);
Gpert = G*(1+InputUnc*Wu);
Wp = tf([1/4 0.6],[1 0.006]);
P = [Wp; 1 ]*[1 Gpert];
```

```
[K,clp,clperf] = dksyn(P,1,1);
N = order(K)
```

```
N = 7
```

`dksyn` returns a 7th-order controller, the closed-loop system with that controller, `clp`, and the robust H_∞ performance of that system, `clperf`. Compute reduced-order controllers for orders ranging from 1 to 7.

```
Kred = reduce(K,1:N);
```

Find the lowest-order controller `Klow` with performance no worse than 1.05*`clperf`, or 5% degradation compared to the full-order controller.

```
for k=1:N
   Klow = Kred(:,:,k);
   clp = lft(P,Klow);
   [gamma,~] = dksynperf(clp);
   if gamma.UpperBound < 1.05*clperf
      break
   end
end
order(Klow)
```

```
ans = 3
```

To validate the reduced-order controller, compare the robust H_∞ performance of the system using the simplified controller with that of the system using the full-order controller. The latter value is `clperf`, returned by `dksyn` when synthesizing the controller.

```
clplow = lft(P,Klow);
dksynperf(clplow)
```

```
ans = struct with fields:
           LowerBound: 0.7116
           UpperBound: 0.7131
    CriticalFrequency: 0.7408
```

```
clperf
```

```
clperf = 0.6816
```

The third-order controller achieves almost the same robust H_∞ performance as the 7th-order controller returned by `dksyn`.

Input Arguments

`clp` — Closed-loop uncertain system

`uss` | `ufrd` | `genss` | `genfrd`

Closed-loop uncertain system, specified as a `uss`, `ufrd`, `genss`, or `genfrd` model that contains uncertain elements. For `genss` or `genfrd` models, `dksynperf` uses the current value of any tunable blocks and folds them into the known (not uncertain) part of the model.

`w` — Frequencies

`{wmin,wmax}` | vector

Frequencies at which to compute robust H_∞ performance, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the H_∞ performance at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the H_∞ performance at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

`opts` — Options for H_∞ performance computation

`robOptions` object

Options for computation of robust H_∞ performance, specified as an object you create with `robOptions`. The available options include settings that let you:

- Extract frequency-dependent H_∞ performance values.
- Examine the sensitivity of the H_∞ performance to each uncertain element.
- Improve the results of the calculation by setting certain options for the underlying `mussv` calculation. In particular, setting the option `'MussvOptions'` to `'mN'` can reduce the gap between the lower bound and upper bound. `N` is the number of restarts.

For more information about all available options, see `robOptions`.

Example: `robOptions('Sensitivity','on','MussvOptions','m3')`

Output Arguments

gamma — Robust H_∞ performance and critical frequency

structure

Robust H_∞ performance and critical frequency, returned as a structure containing the following fields:

Field	Description
`LowerBound`	Lower bound on the actual robust H_∞ performance γ, returned as a scalar value. The exact value of γ is guaranteed to be no smaller than `LowerBound`. In other words, there exist some uncertain-element values of magnitude 1/`LowerBound` for which the I/O gain of `clp` reaches `LowerBound`. The function returns one such instance in `wcu`.
`UpperBound`	Upper bound on the actual robust H_∞ performance, returned as a scalar value. The exact value is guaranteed to be no larger than `UpperBound`. In other words, for all modeled uncertainty with normalized magnitude up to 1/`UpperBound`, the peak I/O gain of `clp` is less than `UpperBound`.
`CriticalFrequency`	Frequency at which the I/O gain reaches `LowerBound`, in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `clp`.

Use `normalized2actual` to convert the normalized uncertainty values 1/`LowerBound` or 1/`UpperBound` to actual deviations from nominal values.

wcu — Perturbations driving I/O gain to gamma.LowerBound

structure

Perturbations driving I/O gain to `gamma.LowerBound`, returned as a structure whose fields are the names of the uncertain elements of `clp`. Each field contains the actual value of the corresponding uncertain element. For example, if `clp` includes an uncertain matrix

M and SISO uncertain dynamics delta, then wcu.M is a numeric matrix and wcu.delta is a SISO state-space model.

Use usubs(clp,wcu) to substitute these values for the uncertain elements in clp and obtain the corresponding dynamic system. This system has peak gain gamma.LowerBound.

Use actual2normalized to convert these actual uncertainty values to the normalized units in which 1/gamma.LowerBound or 1/gamma.UpperBound are expressed.

info — Additional information about γ values

structure

Additional information about the γ values, returned as a structure with the following fields:

Field	Description
`Frequency`	Frequency points at which `dksynperf` returns γ values, returned as a vector. • If the `'VaryFrequency'` option of `robOptions` is `'off'`, then `info.Frequency` is the critical frequency, the frequency at which the I/O gain reaches `gamma.LowerBound`. If the smallest lower bound and the smallest upper bound on γ occur at different frequencies, then `info.Frequency` is a vector containing these two frequencies. • If the `'VaryFrequency'` option of `robOptions` is `'on'`, then `info.Frequency` contains the frequencies selected by `dksynperf`. These frequencies are guaranteed to include the frequency at which the peak gain occurs. • If you specify a vector of frequencies `w` at which to compute γ, then `info.Frequency = w`. When you specify a frequency vector, these frequencies are not guaranteed to include the frequency at which the peak gain occurs. The `'VaryFrequency'` option is meaningful only for `uss` and `genss` models. `dksynperf` ignores the option for `ufrd` and `genfrd` models.
`Bounds`	Lower and upper bounds on the actual γ values, returned as an array. `info.Bounds(:,1)` contains the lower bound at each corresponding frequency in `info.Frequency`, and `info.Bounds(:,2)` contains the corresponding upper bounds.

Field	Description
`WorstPerturbation`	Smallest perturbations at each frequency point in `info.Frequency`, returned as a structure array. The fields of `info.WorstPerturbation` are the names of the uncertain elements in `clp`. Each field contains the value of the corresponding element that drives the I/O gain to the corresponding lower bound at each frequency. For example, if `clp` includes an uncertain parameter `p` and SISO uncertain dynamics `delta`, then `info.WorstPerturbation.p` is a collection of numeric values and `info.WorstPerturbation.delta` is a collection of SISO state-space models.
`Sensitivity`	Sensitivity of γ to each uncertain element, returned as a structure when the `'Sensitivity'` option of `robOptions` is `'on'`. The fields of `info.Sensitivity` are the names of the uncertain elements in `clp`. Each field contains a percentage that measures how much the uncertainty in the corresponding element affects γ. For example, if `info.Sensitivity.p` is 50, then a given fractional change in the uncertainty range of `p` causes half as much fractional change in γ. If the `'Sensitivity'` option of `robOptions` is off (the default setting), then `info.Sensitivity` is `NaN`.

See Also

`actual2normalized` | `dksyn` | `normalized2actual` | `robOptions` | `robgain` | `robstab` | `wcgain`

Introduced in R2017a

dmplot

Interpret disk gain and phase margins

Syntax

```
dmplot
```

```
dmplot(diskgm)
```

```
[dgm,dpm] = dmplot
```

Description

`dmplot` brings up a plot that illustrates the disk gain margin (`dgm`) and disk phase margin (`dpm`) for a sample system. Both margins are derived from the largest disk that

- Contains the critical point (–1,0)
- Does not intersect the Nyquist plot of the open-loop response *L*

`diskgm` is the radius of this disk and a lower bound on the classical gain margin.

`dmplot(diskgm)` plots the maximum allowable phase variation as a function of the actual gain variation for a given disk gain margin `diskgm` (the maximum gain variation being `diskgm`). The closed-loop system is guaranteed to remain stable for all combined gain/phase variations inside the plotted ellipse.

`[dgm,dpm] = dmplot` returns the data used to plot the gain/phase variation ellipse.

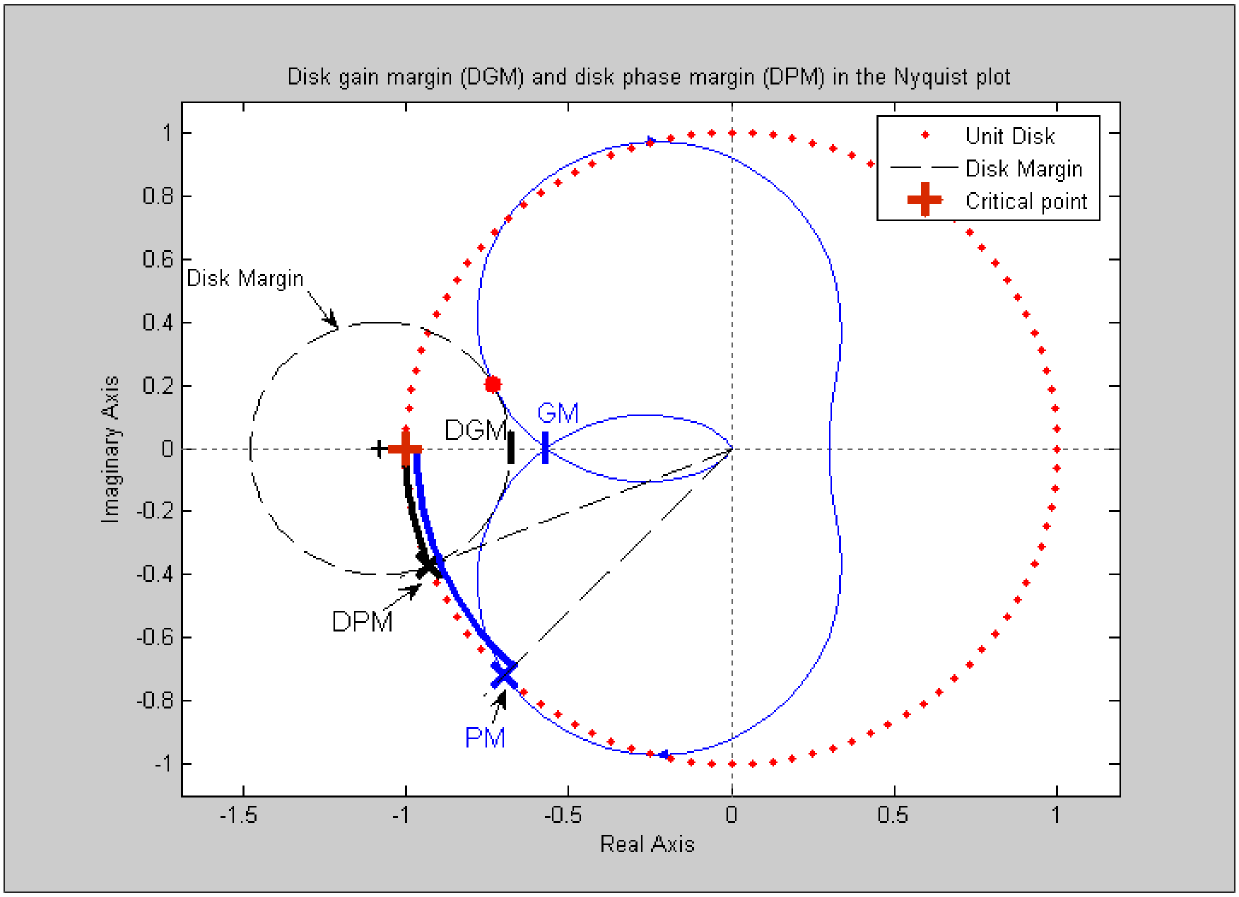
Examples

Disk Gain and Phase Margins

When you call `dmplot` without an argument, the resulting plot and text shows a comparison of a disk margin analysis with the classical notations of gain and phase margins. The Nyquist plot is of the loop transfer function L(s):

$$L(s) = \frac{\frac{s}{30} + 1}{(s+1)(s^2 + 1.6s + 16)}.$$

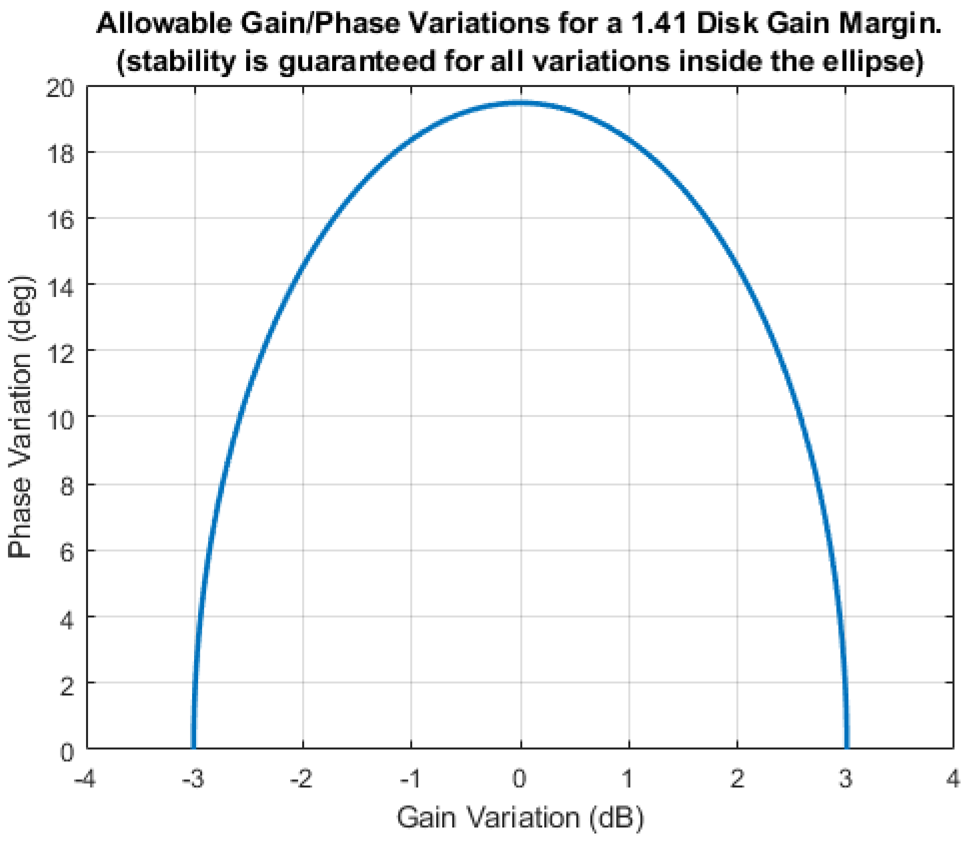
```
dmplot
```



```
This figure shows a comparison of a disk margin analysis
with the classical notations of gain and phase margins.
The Nyquist plot is of the loop transfer function

         L = 4(s/30 + 1)/((s+1)*(s^2 + 1.6s + 16))

 - The Nyquist plot of L corresponds to the blue line
 - The unit disk corresponds to the dotted red line
 - GM and PM indicate the location of the classical gain
    and phase margins for the system L.
 - DGM and DPM correspond to the disk gain and phase
   margins. The disk margins provide a lower bound on
   classical gain and phase margins.
 - The disk margin circle corresponds to the dashed black
   line. The disk margin corresponds to the largest disk
   centered at (GMD + 1/GMD)/2 that just touches the
   loop transfer function L. This location is indicated
   by the red dot.
```

The *x*-axis corresponds to the gain variation in dB and the *y*-axis corresponds to the allowable phase variation in degrees. For a disk gain margin corresponding to 3 dB (1.414), the closed-loop system is stable for all phase and gain variations inside the blue ellipse. For example, the closed-loop system can simultaneously tolerate +/– 2 dB gain variation and +/– 14 deg phase variations. To see the allowable variations for a given disk gain margin, use the given value as an input to `dmplot`.

```
figure           % new figure window
dmplot(1.414)
```



References

Barrett, M.F., Conservatism with robustness tests for linear feedback control systems, Ph.D. Thesis. Control Science and Dynamical Systems, University of Minnesota, 1980.

Blight, J.D., R.L. Dailey, and Gangsass, D., “Practical control law design for aircraft using multivariable techniques,” *International Journal of Control*, Vol. 59, No. 1, 1994, 93-137.

Bates, D., and I. Postlethwaite, Robust Multivariable Control of Aerospace Systems, Delft University Press, Delft, The Netherlands, ISBN: 90-407-2317-6, 2002.

See Also

`wcdiskmargin`

Introduced before R2006a

evallmi

Given particular instance of decision variables, evaluate all variable terms in system of LMIs

Syntax

```
evalsys = evallmi(lmisys,decvars)
```

Description

`evallmi` evaluates all LMI constraints for a particular instance `decvars` of the vector of decision variables. Recall that `decvars` fully determines the values of the matrix variables $X_1, . . ., X_K$. The “evaluation” consists of replacing all terms involving $X_1, . . ., X_K$ by their matrix value. The output `evalsys` is an LMI system containing only constant terms.

The function `evallmi` is useful for validation of the LMI solvers' output. The vector returned by these solvers can be fed directly to `evallmi` to evaluate all variable terms. The matrix values of the left and right sides of each LMI are then returned by `showlmi`.

Observation

`evallmi` is meant to operate on the output of the LMI solvers. To evaluate all LMIs for particular instances of the matrix variables $X_1, . . ., X_K$, first form the corresponding decision vector x with `mat2dec` and then call `evallmi` with x as input.

Examples

Consider the feasibility problem of finding $X > 0$ such that

$$A^T X A - X + I < 0$$

where

$$A = \begin{pmatrix} 0.5 & -0.2 \\ 0.1 & -0.7 \end{pmatrix}.$$

This LMI system is defined by:

```
setlmis([])
X = lmivar(1,[2 1])     % full symmetric X

lmiterm([1 1 1 X],A',A)     % LMI #1: A'*X*A
lmiterm([1 1 1 X],-1,1)         % LMI #1: -X
lmiterm([1 1 1 0],1)     % LMI #1: I
lmiterm([-2 1 1 X],1,1)     % LMI #2: X
lmis = getlmis
```

To compute a solution `xfeas`, call `feasp` by

```
[tmin,xfeas] = feasp(lmis)
```

The result is

```
tmin =
    -4.7117e+00

xfeas' =
    1.1029e+02      -1.1519e+01      1.1942e+02
```

The LMI constraints are therefore feasible since `tmin` < 0. The solution *X* corresponding to the feasible decision vector `xfeas` would be given by `X = dec2mat(lmis,xfeas,X)`.

To check that `xfeas` is indeed feasible, evaluate all LMI constraints by typing

```
evals = evallmi(lmis,xfeas)
```

The left and right sides of the first and second LMIs are then given by

```
[lhs1,rhs1] = showlmi(evals,1)
[lhs2,rhs2] = showlmi(evals,2)
```

and the test

```
eig(lhs1-rhs1)
ans =
    -8.2229e+01
    -5.8163e+01
```

confirms that the first LMI constraint is satisfied by `xfeas`.

See Also

`dec2mat` | `mat2dec` | `setmvar` | `showlmi`

Introduced before R2006a

feasp

Compute solution to given system of LMIs

Syntax

```
[tmin,xfeas] = feasp(lmisys,options,target)
```

Description

The function `feasp` computes a solution `xfeas` (if any) of the system of LMIs described by `lmisys`. The vector `xfeas` is a particular value of the decision variables for which all LMIs are satisfied.

Given the LMI system

$$N^T LxN \leq M^T R(x)M, \tag{1-3}$$

`xfeas` is computed by solving the auxiliary convex program:

Minimize t subject to $N^TL(x)N-M^TR(\mathrm{x})M \leq tI$.

The global minimum of this program is the scalar value `tmin` returned as first output argument by `feasp`. The LMI constraints are feasible if `tmin` ≤ 0 and strictly feasible if `tmin` < 0. If the problem is feasible but not strictly feasible, `tmin` is positive and very small. Some post-analysis may then be required to decide whether `xfeas` is close enough to feasible.

The optional argument `target` sets a target value for `tmin`. The optimization code terminates as soon as a value of *t* below this target is reached. The default value is `target` = 0.

Note that `xfeas` is a solution in terms of the decision variables and not in terms of the matrix variables of the problem. Use `dec2mat` to derive feasible values of the matrix variables from `xfeas`.

Control Parameters

The optional argument options gives access to certain control parameters for the optimization algorithm. This five-entry vector is organized as follows:

- options(1) is not used.
- options(2) sets the maximum number of iterations allowed to be performed by the optimization procedure (100 by default).
- options(3) resets the *feasibility radius*. Setting options(3) to a value $R > 0$ further constrains the decision vector $x = (x_1, . . ., x_N)$ to lie within the ball

  $$\sum_{i=1}^{N} x_i^2 < R^2$$

  In other words, the Euclidean norm of xfeas should not exceed R. The feasibility radius is a simple means of controlling the magnitude of solutions. Upon termination, feasp displays the *f-radius saturation*, that is, the norm of the solution as a percentage of the feasibility radius R.

  The default value is $R = 10^9$. Setting options(3) to a negative value activates the “flexible bound” mode. In this mode, the feasibility radius is initially set to 10^8, and increased if necessary during the course of optimization
- options(4) helps speed up termination. When set to an integer value $J > 0$, the code terminates if t did not decrease by more than one percent in relative terms during the last J iterations. The default value is 10. This parameter trades off speed vs. accuracy. If set to a small value (< 10), the code terminates quickly but without guarantee of accuracy. On the contrary, a large value results in natural convergence at the expense of a possibly large number of iterations.
- options(5) = 1 turns off the trace of execution of the optimization procedure. Resetting options(5) to zero (default value) turns it back on.

Setting option(i) to zero is equivalent to setting the corresponding control parameter to its default value. Consequently, there is no need to redefine the entire vector when changing just one control parameter. To set the maximum number of iterations to 10, for instance, it suffices to type

```
options=zeros(1,5)      % default value for all parameters
options(2)=10
```

Memory Problems

When the least-squares problem solved at each iteration becomes ill conditioned, the `feasp` solver switches from Cholesky-based to QR-based linear algebra (see “Memory Problems” on page 1-326 for details). Since the QR mode typically requires much more memory, MATLAB® may run out of memory and display the message

```
??? Error using ==> feaslv
Out of memory. Type HELP MEMORY for your options.
```

You should then ask your system manager to increase your swap space or, if no additional swap space is available, set `options(4) = 1`. This will prevent switching to QR and `feasp` will terminate when Cholesky fails due to numerical instabilities.

Examples

Solve System of LMIs

Consider the problem of finding $P > I$ such that:

$$A_1^T P + P A_1 < 0,$$

$$A_2^T P + P A_2 < 0,$$

$$A_3^T P + P A_3 < 0,$$

with data

$$A_1 = \begin{pmatrix} -1 & 2 \\ 1 & -3 \end{pmatrix}, A_2 = \begin{pmatrix} -0.8 & 1.5 \\ 1.3 & -2.7 \end{pmatrix}, A_3 = \begin{pmatrix} -1.4 & 0.9 \\ 0.7 & -2.0 \end{pmatrix}.$$

This problem arises when studying the quadratic stability of the polytope of the matrices, $\mathrm{Co}\{A_1, A_2, A_3\}$.

To assess feasibility using `feasp`, first enter the LMIs.

```
setlmis([])
p = lmivar(1,[2 1]);
```

```
A1 = [-1 2;1 -3];
A2 = [-0.8 1.5; 1.3 -2.7];
A3 = [-1.4 0.9;0.7 -2.0];

lmiterm([1 1 1 p],1,A1,'s');     % LMI #1
lmiterm([2 1 1 p],1,A2,'s');     % LMI #2
lmiterm([3 1 1 p],1,A3,'s');     % LMI #3
lmiterm([-4 1 1 p],1,1);         % LMI #4: P
lmiterm([4 1 1 0],1);            % LMI #4: I
lmis = getlmis;
```

Call `feasp` to a find a feasible decision vector.

```
[tmin,xfeas] = feasp(lmis);
```

```
 Solver for LMI feasibility problems L(x) < R(x)
    This solver minimizes  t  subject to  L(x) < R(x) + t*I
    The best value of t should be negative for feasibility

 Iteration   :    Best value of t so far 

     1                        0.972718
     2                        0.870460
     3                       -3.136305

 Result:  best value of t:    -3.136305
          f-radius saturation:  0.000% of R =  1.00e+09
```

The result `tmin = -3.1363` means that the problem is feasible. Therefore, the dynamical system $\dot{x} = A(t)x$ is quadratically stable for $A(t) \in \mathrm{Co}\{A_1, A_2, A_3\}$.

To obtain a Lyapunov matrix `P` proving the quadratic stability, use `dec2mat`.

```
P = dec2mat(lmis,xfeas,p)
```

```
P = 2×2

  270.8553  126.3999
  126.3999  155.1336
```

It is possible to add further constraints on this feasibility problem. For instance, the following command bounds the Frobenius norm of `P` by 10 while asking `tmin` to be less than or equal to –1.

```
options = [0,0,10,0,0];
[tmin,xfeas] = feasp(lmis,options,-1);
```

```
 Solver for LMI feasibility problems L(x) < R(x)
    This solver minimizes  t  subject to  L(x) < R(x) + t*I
    The best value of t should be negative for feasibility

 Iteration   :    Best value of t so far

     1                        0.988505
     2                        0.872239
     3                       -0.476638
     4                       -0.920574
     5                       -0.920574
***                 new lower bound:    -3.726964
     6                       -1.011130
***                 new lower bound:    -1.602398

 Result:  best value of t:    -1.011130
          f-radius saturation:  91.385% of R =  1.00e+01
```

The third entry of `options` sets the feasibility radius to 10 while the third argument to `feasp`, `-1`, sets the target value for `tmin`. This constraint yields `tmin = -1.011` and a matrix `P` with largest eigenvalue $\lambda_{max}(P) = 8.4653$.

```
P = dec2mat(lmis,xfeas,p);
e = eig(P)
```

```
e = 2×1

    3.8875
    8.4653
```

References

The feasibility solver `feasp` is based on Nesterov and Nemirovski's Projective Method described in:

Nesterov, Y., and A. Nemirovski, *Interior Point Polynomial Methods in Convex Programming: Theory and Applications*, SIAM, Philadelphia, 1994.

Nemirovski, A., and P. Gahinet, “The Projective Method for Solving Linear Matrix Inequalities,” *Proc. Amer. Contr. Conf.*, 1994, Baltimore, Maryland, p. 840–844.

The optimization is performed by the C-MEX file `feaslv.mex`.

See Also

`dec2mat` | `gevp` | `mincx`

Introduced before R2006a

fitfrd

Fit frequency response data with state-space model

Syntax

```
B = fitfrd(A,N)
```

```
B = fitfrd(A,N,RD)
```

```
B = fitfrd(A,N,RD,WT)
```

Description

`B = fitfrd(A,N)` is a state-space object with state dimension N, where `A` is an `frd` object and *N* is a nonnegative integer. The frequency response of `B` closely matches the *D*-scale frequency response data in `A`.

`A` must have either 1 row or 1 column, although it need not be 1-by-1. `B` will be the same size as `A`. In all cases, `N` should be a nonnegative scalar.

`B = fitfrd(A,N,RD)` forces the relative degree of `B` to be `RD`. `RD` must be a nonnegative integer. The default value for `RD` is 0. If `A` is a row (or column) then `RD` can be a vector of the same size as well, specifying the relative degree of each entry of `B`. If `RD` is a scalar, then it specifies the relative degree for all entries of `B`. You can specify the default value for `RD` by setting `RD` to an empty matrix.

`B = fitfrd(A,N,RD,WT)` uses the magnitude of `WT` to weight the optimization fit criteria. `WT` can be a `double, ss` or `frd`. If `WT` is a scalar, then it is used to weight all entries of the error criteria (`A-B`). If `WT` is a vector, it must be the same size as `A`, and each individual entry of `WT` acts as a weighting function on the corresponding entry of (`A-B`).

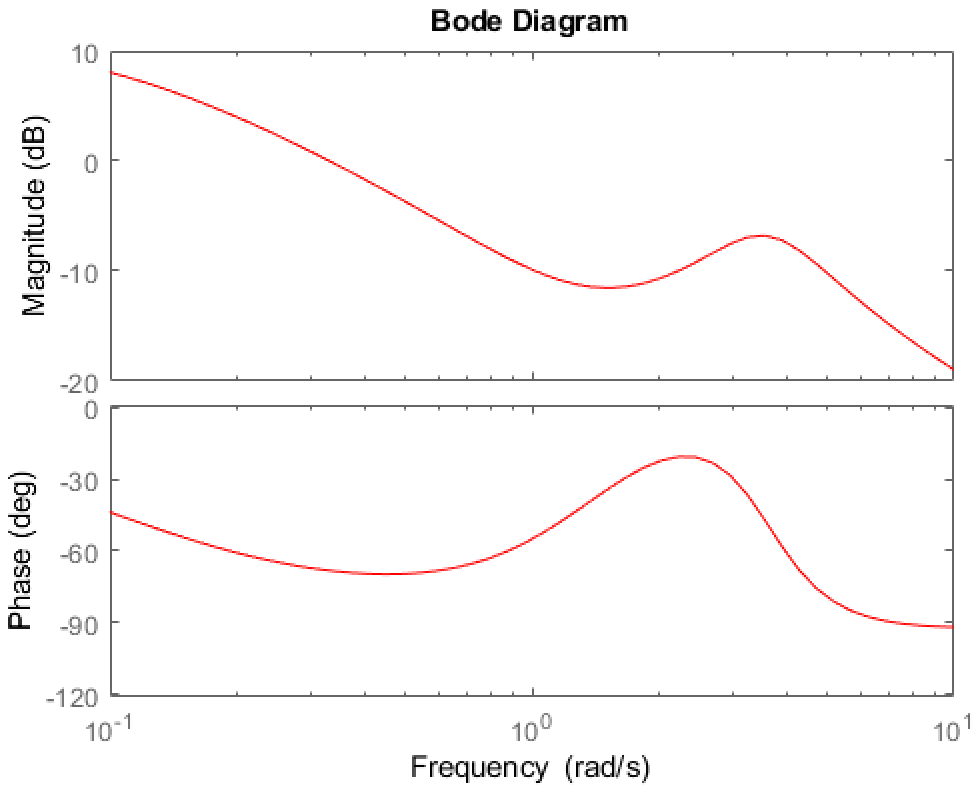
Examples

Fit D-scale Data

Use the `fitfrd` command to fit *D*-scale data.

Create *D*-scale frequency response data from a fifth-order system.

```
sys = tf([1 2 2],[1 2.5 1.5])*tf(1,[1 0.1]);
sys = sys*tf([1 3.75 3.5],[1 2.5 13]);
omeg = logspace(-1,1);
sysg = frd(sys,omeg);
bode(sysg,'r-');
```

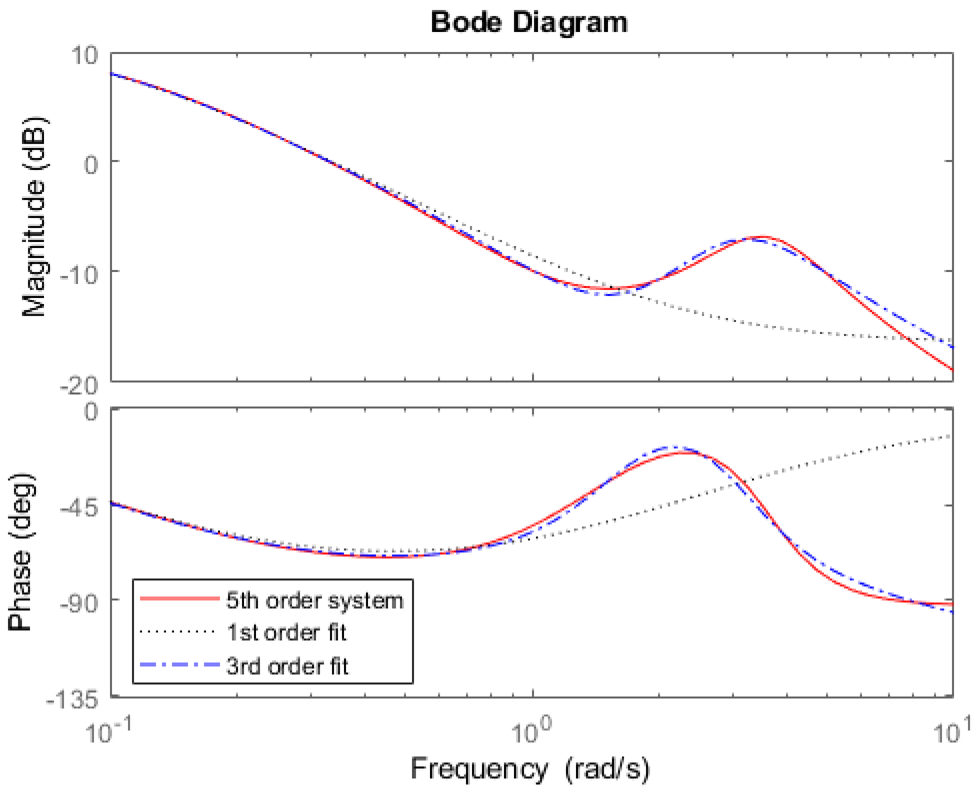


You can try to fit the frequency response *D*-scale data `sysg` with a first-order system, `b1`. Similarly, you can fit the *D*-scale data with a third-order system, `b3`.

```
b1 = fitfrd(sysg,1);
b3 = fitfrd(sysg,3);
```

Compare the original *D*-scale data `sysg` with the frequency responses of the first and third-order models calculated by `fitfrd`.

```
b1g = frd(b1,omeg);
b3g = frd(b3,omeg);
bode(sysg,'r-',b1g,'k:',b3g,'b-.')
legend('5th order system','1st order fit','3rd order fit','Location','Southwest')
```



Limitations

Numerical conditioning problems arise if the state order of the fit `N` is selected to be higher than required by the dynamics of `A`.

See Also

`fitmagfrd`

Introduced before R2006a

fitmagfrd

Fit frequency response magnitude data with minimum-phase state-space model using log-Chebyshev magnitude design

Syntax

```
B = fitmagfrd(A,N)
```

```
B = fitmagfrd(A,N,RD)
```

```
B = fitmagfrd(A,N,RD,WT)
```

```
B = fitmagfrd(A,N,RD,WT,C)
```

Description

`B = fitmagfrd(A,N)` is a stable, minimum-phase `ss` object, with state-dimension `N`, whose frequency response magnitude closely matches the magnitude data in `A`. `A` is a 1-by-1 `frd` object, and `N` is a nonnegative integer.

`B = fitmagfrd(A,N,RD)` forces the relative degree of `B` to be `RD`. `RD` must be a nonnegative integer whose default value is `0`. You can specify the default value for `RD` by setting `RD` to an empty matrix.

`B = fitmagfrd(A,N,RD,WT)` uses the magnitude of `WT` to weight the optimization fit criteria. `WT` can be a `double`, `ss` or `frd`. If `WT` is a scalar, then it is used to weight all entries of the error criteria `(A-B)`. If `WT` is a vector, it must be the same size as `A`, and each individual entry of `WT` acts as a weighting function on the corresponding entry of (`A-B`). The default value for `WT` is 1, and you can specify it by setting `WT` to an empty matrix.

`B = fitmagfrd(A,N,RD,WT,C)` enforces additional magnitude constraints on `B`, specified by the values of `C.LowerBound` and `C.UpperBound`. These can be empty, `double` or `frd` (with `C.Frequency` equal to `A.Frequency`). If `C.LowerBound` is non-empty, then the magnitude of `B` is constrained to lie above `C.LowerBound`. No lower bound is enforced at frequencies where `C.LowerBound` is equal to -inf. Similarly, the `UpperBound` field can be used to specify an upper bound on the magnitude of `B`. If `C` is a `double` or `frd` (with `C.Frequency` equal to `A.Frequency`), then the upper and lower bound constraints on `B` are taken directly from `A` as:

- if $C(w)$ == –1, then enforce abs($B(w)$) <= abs($A(w)$)
- if $C(w)$ == 1, then enforce abs($B(w)$) >= abs($A(w)$)
- if $C(w)$ == 0, then no additional constraint

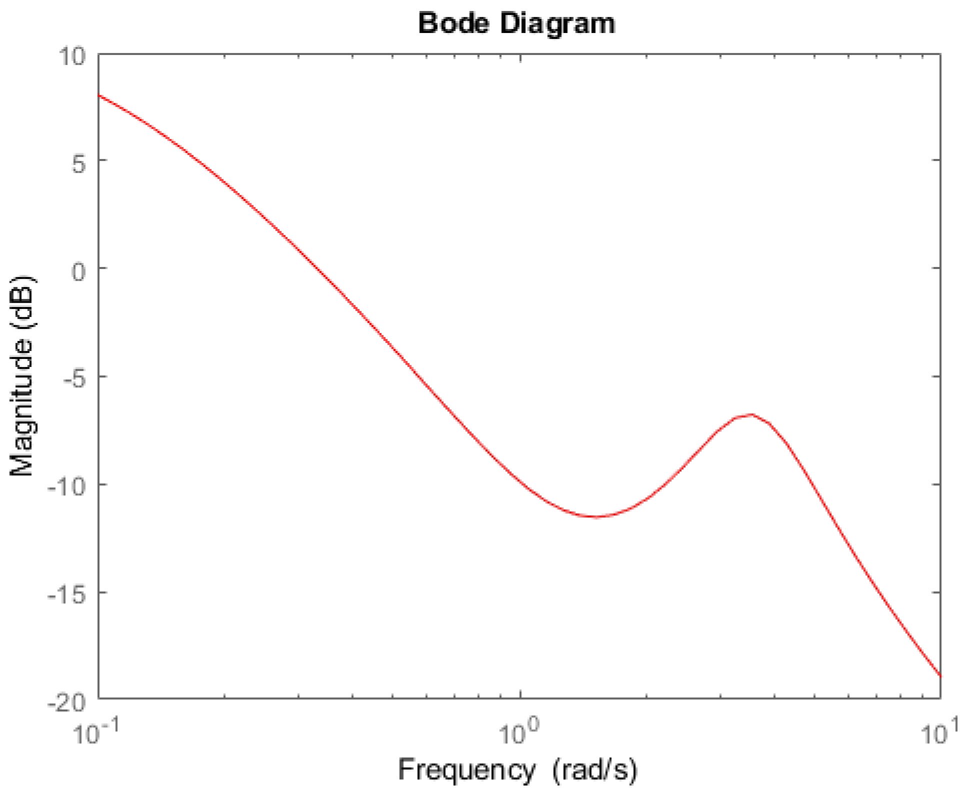
where `w` denotes the frequency.

Examples

Fit Frequency Response Data With Stable Minimum-Phase State-Space Model

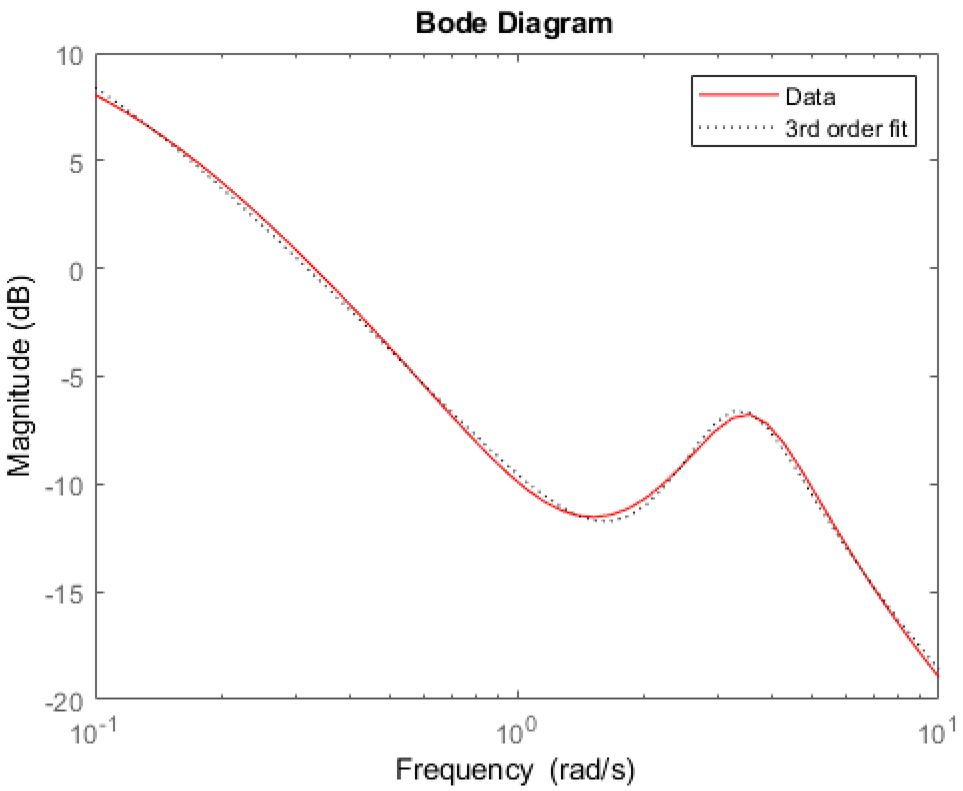
Create frequency response magnitude data from a fifth-order system.

```
sys = tf([1 2 2],[1 2.5 1.5])*tf(1,[1 0.1]);
sys = sys*tf([1 3.75 3.5],[1 2.5 13]);
omega = logspace(-1,1);
sysg = abs(frd(sys,omega));
bodemag(sysg,'r');
```



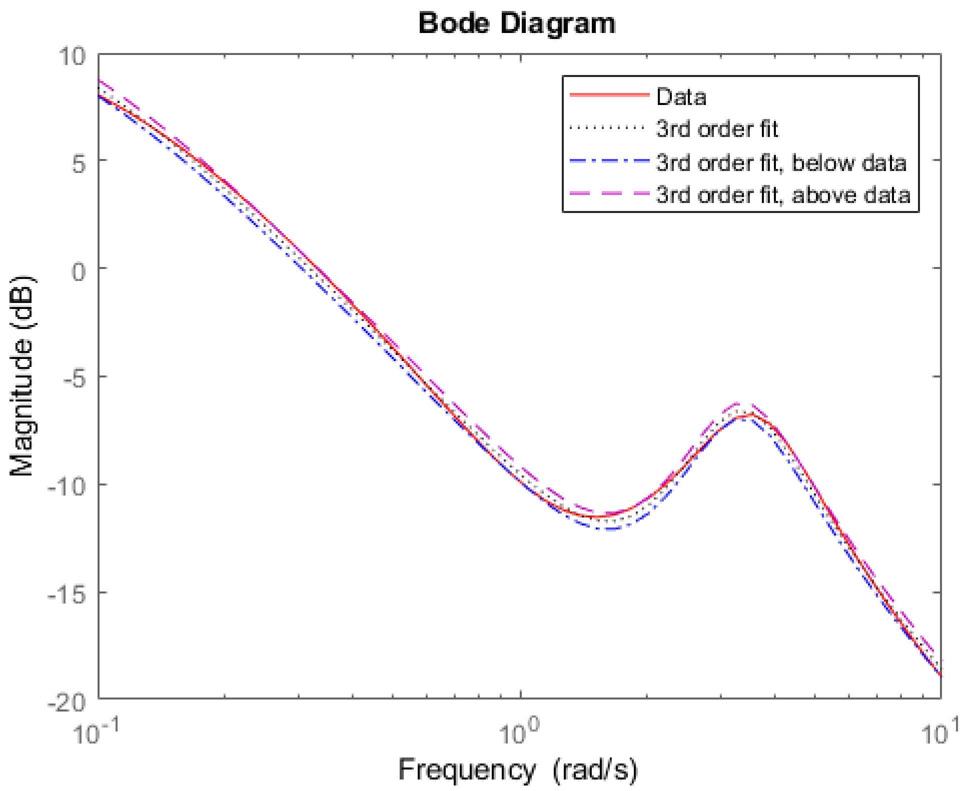
Fit the magnitude data with a minimum-phase, stable third-order system.

```
ord = 3;
b1 = fitmagfrd(sysg,ord);
b1g = frd(b1,omega);
bodemag(sysg,'r',b1g,'k:');
legend('Data','3rd order fit');
```



Fit the magnitude data with a third-order system constrained to lie below and above the given data.

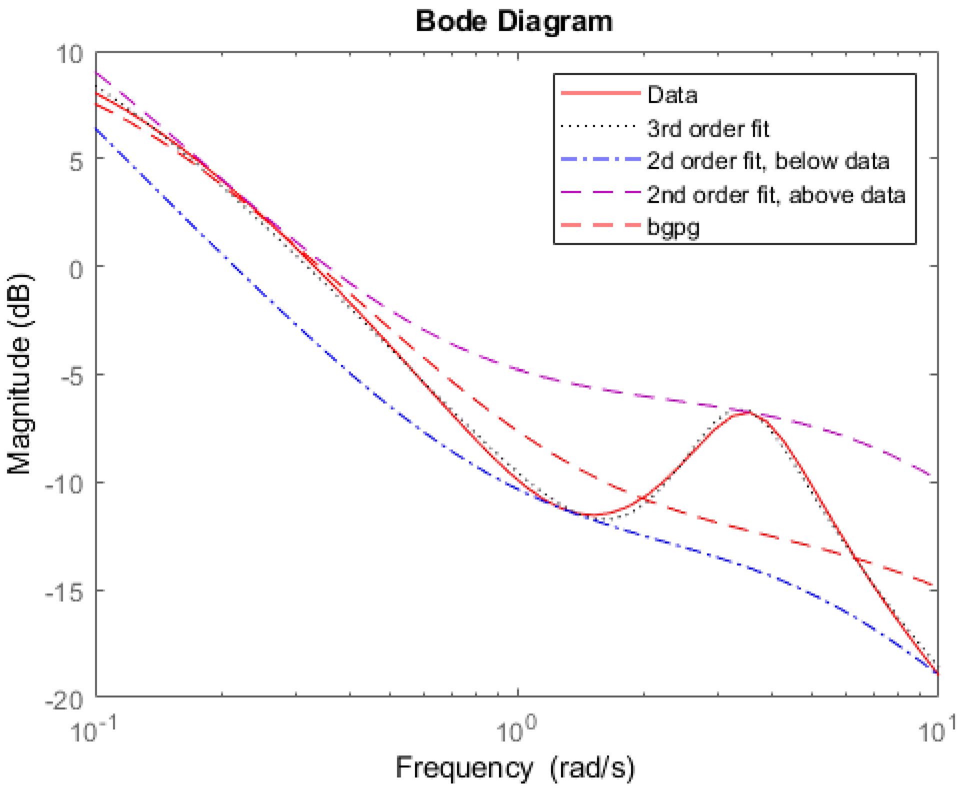
```
C2.UpperBound = sysg;
C2.LowerBound = [];
b2 = fitmagfrd(sysg,ord,[],[],C2);
b2g = frd(b2,omega);
C3.UpperBound = [];
C3.LowerBound = sysg;
b3 = fitmagfrd(sysg,ord,[],[],C3);
b3g = frd(b3,omega);
bodemag(sysg,'r',b1g,'k:',b2g,'b-.',b3g,'m--')
legend('Data','3rd order fit','3rd order fit, below data',...
       '3rd order fit, above data')
```



Fit the magnitude data with a second-order system constrained to lie below and above the given data.

```
ord = 2;
C2.UpperBound = sysg;
C2.LowerBound = [];
b2 = fitmagfrd(sysg,ord,[],sysg,C2);
b2g = frd(b2,omega);
C3.UpperBound = [];
C3.LowerBound = sysg;
b3 = fitmagfrd(sysg,ord,[],sysg,C3);
b3g = frd(b3,omega);
bgp = fitfrd(genphase(sysg),ord);
bgpg = frd(bgp,omega);
```

```
bodemag(sysg,'r',b1g,'k:',b2g,'b-.',b3g,'m--',bgpg,'r--')
legend('Data','3rd order fit','2d order fit, below data',...
       '2nd order fit, above data','bgpg')
```



Limitations

This input `frd` object must be either a scalar 1-by-1 object or, a row, or column vector.

Algorithms

`fitmagfrd` uses a version of log-Chebyshev magnitude design, solving

```
min f     subject to (at every frequency point in A):
        |d|^2 /(1+ f/WT) < |n|^2/A^2 < |d|^2*(1 + f/WT)
```

plus additional constraints imposed with `C`. `n, d` denote the numerator and denominator, respectively, and `B = n/d`. `n` and `d` have orders (`N-RD`) and `N`, respectively. The problem is solved using linear programming for fixed `f and` bisection to minimize `f`. An alternate approximate method, which cannot enforce the constraints defined by `C`, is `B = fitfrd(genphase(A),N,RD,WT)`.

References

Oppenheim, A.V., and R.W. Schaffer, *Digital Signal Processing,* Prentice Hall, New Jersey, 1975, p. 513.

Boyd, S. and Vandenberghe, L., *Convex Optimization*, Cambridge University Press, 2004.

See Also

`fitfrd`

Introduced before R2006a

gapmetric

Gap metric and Vinnicombe (nu-gap) metric for distance between two systems

Syntax

```
[gap,nugap] = gapmetric(P1,P2)
[gap,nugap] = gapmetric(P1,P2,tol)
```

Description

`[gap,nugap] = gapmetric(P1,P2)` computes the gap and Vinnicombe (ν-gap) metrics for the distance between dynamic systems `P1` and `P2`. The gap metric on page 1-128 values satisfy 0 ≤ `nugap` ≤ `gap` ≤ 1. Values close to zero imply that any controller that stabilizes `P1` also stabilizes `P2` with similar closed-loop gains.

`[gap,nugap] = gapmetric(P1,P2,tol)` specifies a relative accuracy for calculating the gaps.

Examples

Compute Gap Metrics for Stable and Unstable Plant Models

Create two plant models. One plant, `P1`, is an unstable first-order system with transfer function 1/(s–0.001). The other plant, `P2`, is stable, with transfer function 1/(s +0.001).

```
P1 = tf(1,[1 -0.001]);
P2 = tf(1,[1 0.001]);
```

Despite the fact that one plant is unstable and the other is stable, these plants are close as measured by the `gap` and `nugap` metrics.

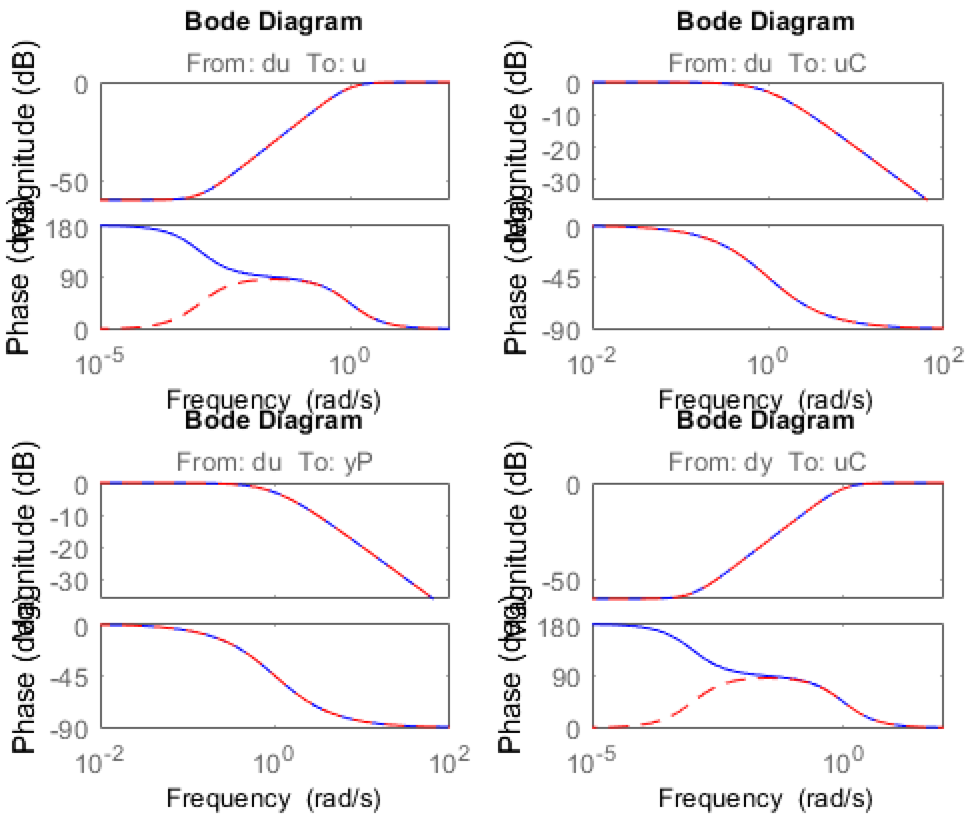
```
[gap,nugap] = gapmetric(P1,P2)
```

```
gap = 0.0021
```

```
nugap = 0.0020
```

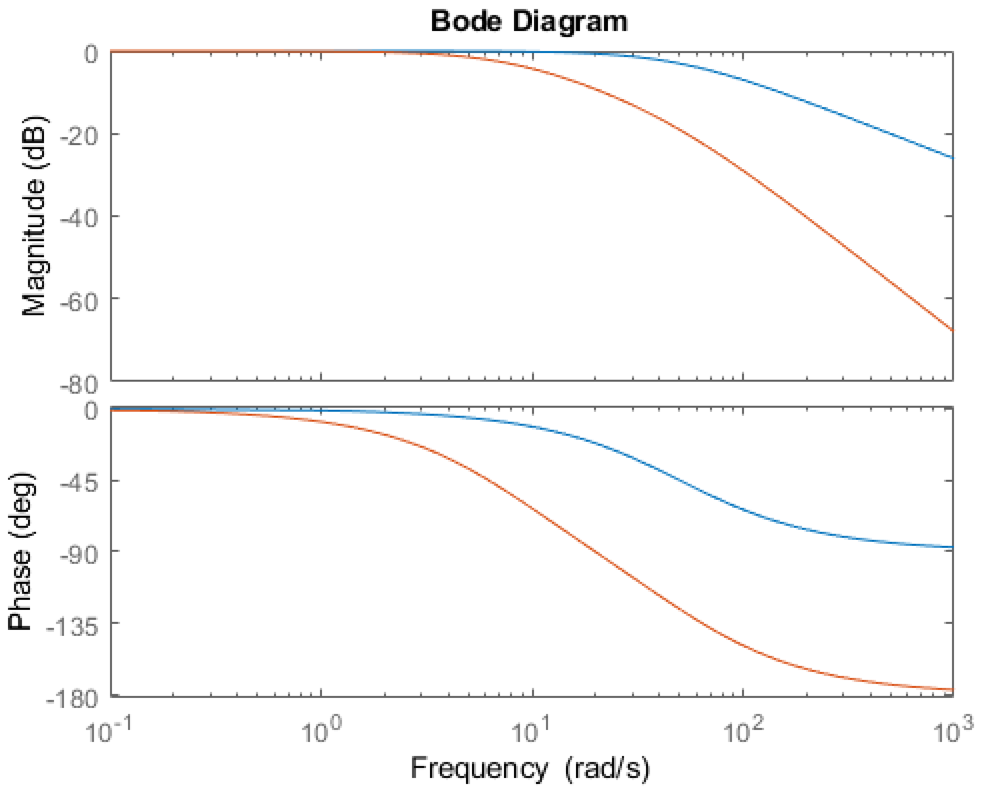
The gap is very small compared to 1. Thus a controller that yields a stable closed-loop system with P2 also tends to stabilize P1. For instance, the feedback controller `C = 1` stabilizes both plants and renders nearly identical closed-loop gains. To see this, examine the sensitivity functions of the two closed-loop systems.

```
C = 1;
H1 = loopsens(P1,C);
H2 = loopsens(P2,C);
subplot(2,2,1); bode(H1.Si,'-',H2.Si,'r--');
subplot(2,2,2); bode(H1.Ti,'-',H2.Ti,'r--');
subplot(2,2,3); bode(H1.PSi,'-',H2.PSi,'r--');
subplot(2,2,4); bode(H1.CSo,'-',H2.CSo,'r--');
```



Next, consider two stable plant models that differ by a first-order system. One plant, P3, is the transfer function 50/(*s*+50), and the other plant, P4, is the transfer function [50/(*s*+50)]*8/(*s*+8).

```
P3 = tf(50,[1 50]);
P4 = tf(8,[1 8])*P3;
figure
bode(P3,P4)
```



Although the two systems have similar high-frequency dynamics and the same unity gain at low frequency, by the gap and nugap metrics, the plants are fairly far apart.

```
[gap,nugap] = gapmetric(P3,P4)
```

```
gap = 0.6148
```

```
nugap = 0.6147
```

Compute Gap Metric and Stability Margin

Consider a plant and a stabilizing controller.

```
P1 = tf([1 2],[1 5 10]);
C = tf(4.4,[1 0]);
```

Compute the stability margin for this plant and controller.

```
b1 = ncfmargin(P1,C)
```

```
b1 = 0.1961
```

Next, compute the gap between `P1` and the perturbed plant, `P2`.

```
P2 = tf([1 1],[1 3 10]);
[gap,nugap] = gapmetric(P1,P2)
```

```
gap = 0.1391
```

```
nugap = 0.1390
```

Because the stability margin `b1 = b(P1,C)` is greater than the gap between the two plants, `C` also stabilizes `P2`. As discussed in “Gap Metrics and Stability Margins” on page 1-128, the stability margin `b2 = b(P2,C)` satisfies the inequality `asin(b(P2,C)) ≥ asin(b1)-asin(gap)`. Confirm this result.

```
b2 = ncfmargin(P2,C);
[asin(b2) asin(b1)-asin(gap)]
```

```
ans = 1×2

    0.0997    0.0579
```

Input Arguments

`P1,P2` — Input systems

dynamic system models

Input systems, specified as dynamic system models. `P1` and `P2` must have the same input and output dimensions. If `P1` or `P2` is a generalized state-space model (`genss` or `uss`) then `gapmetric` uses the current or nominal value of all control design blocks.

`tol` — Relative accuracy

0.001 (default) | positive scalar

Relative accuracy for computing the gap metrics, specified as a positive scalar. If gap_{actual} is the true value of the gap (or the Vinnicombe gap), the returned value `gap` (or `nugap`) is guaranteed to satisfy

$|1 - \texttt{gap}/gap_{actual}| < \texttt{tol}$.

Output Arguments

`gap` — Gap between P1 and P2

scalar in [0,1]

Gap on page 1-128 between `P1` and `P2`, returned as a scalar in the range [0,1]. A value close to zero implies that any controller that stabilizes `P1` also stabilizes `P2` with similar closed-loop gains. A value close to 1 means that `P1` and `P2` are far apart. A value of 0 means that the two systems are identical.

`nugap` — Vinnicombe gap (ν-gap) between P1 and P2

scalar in [0,1]

Vinnicombe gap on page 1-128 (ν-gap) between `P1` and `P2`, returned as a scalar value in the range [0,1]. As with `gap`, a value close to zero implies that any controller that stabilizes `P1` also stabilizes `P2` with similar closed-loop gains. A value close to 1 means that `P1` and `P2` are far apart. A value of 0 means that the two systems are identical. Because $0 \le \texttt{nugap} \le \texttt{gap} \le 1$, the ν-gap can provide a more stringent test for robustness as described in “Gap Metrics and Stability Margins” on page 1-128.

More About

Gap Metric

For plants P_1 and P_2, let $P_1 = N_1 M_1^{-1}$ and $P_2 = N_2 M_2^{-1}$ be right normalized coprime factorizations (see `rncf`). Then the gap metric δ_g is given by:

$$\delta_g(P_1,P_2) = \max\left\{\vec{\delta}_g(P_1,P_2), \vec{\delta}_g(P_2,P_1)\right\}.$$

Here, $\vec{\delta}_g(P_1,P_2)$ is the directed gap, given by

$$\vec{\delta}_g(P_1,P_2) = \min_{\text{stable } Q(s)} \left\| \begin{bmatrix} M_1 \\ N_1 \end{bmatrix} - \begin{bmatrix} M_2 \\ N2 \end{bmatrix} Q \right\|_\infty .$$

For more information, see Chapter 17 of [1].

Vinnicombe Gap Metric

For P_1 and P_2, the Vinnicombe gap metric is given by

$$\delta_\nu(P_1,P_2) = \max_\omega \left\| \left(I + P_2 P_2^*\right)^{-1/2} (P_1 - P_2) \left(I + P_1 P_1^*\right)^{-1/2} \right\|_\infty ,$$

provided that $\det\left(I + P_2^* P_1\right)$ has the right winding number. Here, * denotes the conjugate (see `ctranspose`). This expression is a weighted difference between the two frequency responses $P_1(j\omega)$ and $P_2(j\omega)$. For more information, see Chapter 17 of [1].

Gap Metrics and Stability Margins

The gap and ν-gap metrics give a numerical value $\delta(P_1,P_2)$ for the distance between two LTI systems. For both metrics, the following robust performance result holds:

$$\arcsin b(P_2,C_2) \geq \arcsin b(P_1,C_1) - \arcsin \delta(P_1,P_2) - \arcsin \delta(C_1,C_2),$$

where the stability margin b (see `ncfmargin`), assuming negative-feedback architecture, is given by

$$b(P,C) = \left\| \begin{bmatrix} I \\ C \end{bmatrix} (I + PC)^{-1} [I \;\; P] \right\|_\infty^{-1} = \left\| \begin{bmatrix} I \\ P \end{bmatrix} (I + CP)^{-1} [I \;\; C] \right\|_\infty^{-1} .$$

To interpret this result, suppose that a nominal plant P_1 is stabilized by controller C_1 with stability margin $b(P_1,C_1)$. Then, if P_1 is perturbed to P_2 and C_1 is perturbed to C_2, the stability margin is degraded by no more than the above formula. For an example, see “Compute Gap Metric and Stability Margin” on page 1-126.

The ν-gap is always less than or equal to the gap, so its predictions using the above robustness result are tighter.

The quantity $b(P,C)^{-1}$ is the signal gain from disturbances on the plant input and output to the input and output of the controller.

Gap Metrics in Robust Design

To make use of the gap metrics in robust design, you must introduce weighting functions. In the robust performance formula, replace P by W_2PW_1, and replace C by $W_1^{-1}CW_2^{-1}$. You can make similar substitutions for P_1, P_2, C_1 and C_2. This form makes the weighting functions compatible with the weighting structure in the H_∞ loop shaping control design procedure used by functions such as `loopsyn` and `ncfsyn`.

References

[1] Zhou, K., Doyle, J.C., *Essentials of Robust Control*. London, UK: Pearson, 1997.

See Also

`loopsyn` | `ncfmargin` | `ncfsyn` | `robstab` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

genphase

Fit single-input/single-output magnitude data with real, rational, minimum-phase transfer function

Syntax

```
resp = genphase(d)
```

Description

`genphase` uses the complex-cepstrum algorithm to generate a complex frequency response `resp` whose magnitude is equal to the real, positive response `d`, but whose phase corresponds to a stable, minimum-phase function. The input, `d`, and output, `resp`, are `frd` objects.

References

Oppenheim, A.V., and R.W. Schaffer, *Digital Signal Processing,* Prentice Hall, New Jersey, 1975, p. 513.

See Also

`fitfrd` | `fitmagfrd`

Introduced before R2006a

getLimits

Validity range for uncertain real (`ureal`) parameters

Syntax

```
[ActLims,NormLims] = getLimits(ublk)
```

Description

When the uncertainty range of a `ureal` parameter is not centered at its nominal value, there are restrictions on the range of values the parameter can take. For robust stability analysis, these restrictions mean that the smallest destabilizing perturbation of the parameter may be out of the reach of the specified `ureal` model. Use `getLimits` to find out the range of actual and normalized values that a ureal parameter can take.

`[ActLims,NormLims] = getLimits(ublk)` computes the intervals of actual and normalized values that an uncertain real parameter can take. For meaningful analysis results, the actual and normalized values of `ublk` must remain in these intervals. Values outside these intervals are essentially meaningless. In other words, `ActLims` and `NormLims` are the ranges of validity of the uncertainty model for real parameters.

Examples

Validity Range for Uncertain Parameters

Create a `ureal` uncertain parameter with range centered at the nominal value.

```
ublk = ureal('a',1,'range',[-1 3])
```

```
ublk =

  Uncertain real parameter "a" with nominal value 1 and range [-1,3].
```

For such a parameter, $b = 0$ (see “Algorithms” on page 1-133), so there is no constraint on the values that the actual uncertainty (`ublk`) and the normalized uncertainty (Δ) can take. Use `getLimits` to confirm the ranges of the actual and normalized uncertainty.

```
[ActLims,NormLims] = getLimits(ublk)
```

```
ActLims = 1×2

  -Inf   Inf


NormLims = 1×2

  -Inf   Inf
```

Skew the uncertainty range to the right of the nominal value ($DL < DR$).

```
ublk.PlusMinus = [-1 2]
```

```
ublk = 

  Uncertain real parameter "a" with nominal value 1 and range [0,3].
```

Now, the values that `ublk` and Δ can take for analysis purposes are limited.

```
[ActLims,NormLims] = getLimits(ublk)
```

```
ActLims = 1×2

   -3.0000       Inf


NormLims = 1×2

  -Inf     3
```

Input Arguments

ublk — Uncertain real parameter
`ureal`

Uncertain real parameter, specified as a `ureal` object.

Output Arguments

`ActLims` — Limits on actual uncertainty
2-element row vector

Limits on the actual uncertainty range taken by `ublk` for analysis purposes, returned as a 2-element vector of the form `[min,max]`. When the uncertainty range specified in `ublk` is centered on the nominal value, `ActLims = -Inf,Inf`.

`NormLims` — Limits on normalized uncertainty
2-element row vector

Limits on the normalized uncertainty range of `ublk` used for analysis purposes, returned as a 2-element vector of the form `[min,max]`. When the uncertainty range specified in `ublk` is centered on the nominal value, `NormLims = -Inf,Inf`.

Algorithms

Analysis functions such as `robstab` and `robgain` model uncertain real parameters as:

$$u = u_{nom} + \frac{a\Delta}{1 - b\Delta}, \quad a > 0,$$

where u is the actual value, u_{nom} is the nominal value, and Δ is the normalized value. When the uncertainty range is centered at the nominal value, there are no restrictions on the values u or Δ can take. However, when the uncertainty range is skewed, there are limitations on these values. To ensure continuity, the analysis functions restrict the values Δ and u to the ranges:

$$\Delta < \frac{1}{|b|},\ u > \left(u_{nom} - \left|\frac{a}{b}\right|\right), \text{ for } DL < DR$$

$$\Delta > -\frac{1}{|b|},\ u < \left(u_{nom} + \left|\frac{a}{b}\right|\right), \text{ for } DL < DR,$$

where DL and DR define the uncertainty range of u, $[u_{nom}–DL,u_{nom}+DR]$. Note that b and $DR–DL$ always have the same sign.

See Also

`actual2normalized` | `normalized2actual` | `ureal`

Introduced in R2018a

getlmis

Internal description of LMI system

Syntax

lmisys = getlmis

Description

After completing the description of a given LMI system with `lmivar` and `lmiterm`, its internal representation lmisys is obtained with the command

```
lmisys = getlmis
```

This MATLAB representation of the LMI system can be forwarded to the LMI solvers or any other LMI-Lab function for subsequent processing.

See Also

`lmiterm` | `lmivar` | `newlmi` | `setlmis`

Introduced before R2006a

getNominal

Nominal value of uncertain model

Syntax

```
Mnom = getNominal(M)
```

Description

`Mnom = getNominal(M)` replaces all uncertain elements in `M` with their nominal values. All other control design blocks in `M` are unchanged.

Examples

Nominal Value of Uncertain Models

Create a model of a mass-spring-damper system in which the mass, spring constant, and damping constant are all uncertain.

```
m = ureal('m',3,'percent',40);
k = ureal('k',2,'percent',30);
c = ureal('c',1,'percent',20);
G = tf(1,[m,c,k])
```

```
G =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    c: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    k: Uncertain real, nominal = 2, variability = [-30,30]%, 1 occurrences
    m: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "G.NominalValue" to see the nominal value, "get(G)" to see all properties, and "G
```

`G` is a `uss` model. Extract its nominal value.

```
Gnom = getNominal(G);
```

Because G has only uncertain control design blocks, Gnom is a numeric state-space (ss) model.

Combine G with a tunable PID controller.

```
C = tunablePID('C','pid');
T = feedback(G*C,1)
```

```
T =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, anc
    C: Parametric PID controller, 1 occurrences.
    c: Uncertain real, nominal = 1, variability = [-20,20]%, 1 occurrences
    k: Uncertain real, nominal = 2, variability = [-30,30]%, 1 occurrences
    m: Uncertain real, nominal = 3, variability = [-40,40]%, 1 occurrences

Type "ss(T)" to see the current value, "get(T)" to see all properties, and "T.Blocks" t
```

T is a generalized state-space (genss) model that has both tunable and uncertain blocks. Extract the nominal value of T.

```
Tnom = getNominal(T)
```

```
Tnom =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, anc
    C: Parametric PID controller, 1 occurrences.

Type "ss(Tnom)" to see the current value, "get(Tnom)" to see all properties, and "Tnom
```

Extracting the nominal value of T preserves the tunable control design block, resulting in another genss model.

Input Arguments

M — Uncertain model or matrix

dynamic system model | static model

Uncertain model or matrix, specified as a dynamic system model or static model. Typically, `M` is a model that contains uncertainty, such as a `uss`, uncertain `genss`, or `umat` model.

Output Arguments

`Mnom` — Nominal model or matrix
dynamic system model | static model

Nominal value of `M`, returned as a dynamic system model or static model. `Mnom` has no uncertain blocks.

The model type of `Mnom` depends on the type of `M`. For example, if `M` is a `genss` model with uncertain blocks and tunable blocks, then `Mnom` is a `genss` model with tunable blocks.

If `M` contains no control design blocks other than uncertain blocks, then `Mnom` is a state-space (`ss`) model, an `frd` model, or a numeric array, depending on the type of `M`. For example, if `M` is a `uss` model, then `Mnom` is a `ss` model. If `M` is a `umat`, then `Mnom` is a numeric array.

If `M` has no uncertain blocks, then `Mnom = M`.

See Also

`genss` | `umat` | `uss`

Introduced in R2015b

gevp

Generalized eigenvalue minimization under LMI constraints

Syntax

```
[lopt,xopt] = gevp(lmisys,nlfc,options,linit,xinit,target)
```

Description

`gevp` solves the generalized eigenvalue minimization problem of minimizing λ, subject to:

$$C(x) < D(x) \tag{1-4}$$

$$0 < B(x) \tag{1-5}$$

$$A(x) < \lambda B(x) \tag{1-6}$$

where $C(x) < D(x)$ and $A(x) < \lambda B(x)$ denote systems of LMIs. Provided that “Equation 1-4” and “Equation 1-5” are jointly feasible, `gevp` returns the global minimum `lopt` and the minimizing value `xopt` of the vector of decision variables x. The corresponding optimal values of the matrix variables are obtained with `dec2mat`.

The argument `lmisys` describes the system of LMIs “Equation 1-4” to “Equation 1-6” for $\lambda = 1$. The LMIs involving λ are called the *linear-fractional constraints* while “Equation 1-4” and “Equation 1-5” are regular LMI constraints. The number of linear-fractional constraints “Equation 1-6” is specified by `nlfc`. All other input arguments are optional. If an initial feasible pair (λ_0, x_0) is available, it can be passed to `gevp` by setting `linit` to λ_0 and `xinit` to x_0. Note that `xinit` should be of length `decnbr(lmisys)` (the number of decision variables). The initial point is ignored when infeasible. Finally, the last argument `target` sets some target value for λ. The code terminates as soon as it has found a feasible pair (λ, x) with $\lambda \leq$ target.

Caution

When setting up your `gevp` problem, be cautious to

- Always specify the linear-fractional constraints “Equation 1-6” *last* in the LMI system. `gevp` systematically assumes that the last `nlfc` LMI constraints are linear fractional.
- Add the constraint $B(x) > 0$ or any other LMI constraint that enforces it (see Remark below). This positivity constraint is required for regularity and good formulation of the optimization problem.

Control Parameters

The optional argument `options` lets you access control parameters of the optimization code. In `gevp`, this is a five-entry vector organized as follows:

- `options(1)` sets the desired relative accuracy on the optimal value `lopt` (default = 10^{-2}).
- `options(2)` sets the maximum number of iterations allowed to be performed by the optimization procedure (100 by default).
- `options(3)` sets the feasibility radius. Its purpose and usage are the same as for `feasp`.
- `options(4)` helps speed up termination. If set to an integer value $J > 0$, the code terminates when the progress in λ over the last J iterations falls below the desired relative accuracy. Progress means the amount by which λ decreases. The default value is 5 iterations.
- `options(5) = 1` turns off the trace of execution of the optimization procedure. Resetting `options(5)` to zero (default value) turns it back on.

Setting `option(i)` to zero is equivalent to setting the corresponding control parameter to its default value.

Examples

Given

$$A1 = \begin{pmatrix} -1 & 2 \\ 1 & -3 \end{pmatrix},\ A2 = \begin{pmatrix} -0.8 & 1.5 \\ 1.3 & -2.7 \end{pmatrix},\ A3 = \begin{pmatrix} -1.4 & 0.9 \\ 0.7 & -2.0 \end{pmatrix},$$

consider the problem of finding a single Lyapunov function $V(x) = x^T P x$ that proves stability of

$$\dot{x} = A_i x \ (i = 1, 2, 3)$$

and maximizes the decay rate $\frac{dV(x)}{dt}$. This is equivalent to minimizing

α subject to

$$I < P \tag{1-7}$$

$$A_1^T P + PA_1 < \alpha P \tag{1-8}$$

$$A_2^T P + PA_2 < \alpha P \tag{1-9}$$

$$A_3^T P + PA_3 < \alpha P \tag{1-10}$$

To set up this problem for `gevp`, first specify the LMIs “Equation ” to “Equation ”with α = 1:

```
setlmis([]);
p = lmivar(1,[2 1])

lmiterm([1 1 1 0],1)     % P > I : I
lmiterm([-1 1 1 p],1,1)     % P > I : P
lmiterm([2 1 1 p],1,a1,'s')     % LFC # 1 (lhs)
lmiterm([-2 1 1 p],1,1)     % LFC # 1 (rhs)
lmiterm([3 1 1 p],1,a2,'s')     % LFC # 2 (lhs)
lmiterm([-3 1 1 p],1,1)     % LFC # 2 (rhs)
lmiterm([4 1 1 p],1,a3,'s')     % LFC # 3 (lhs)
lmiterm([-4 1 1 p],1,1)     % LFC # 3 (rhs)
lmis = getlmis
```

Note that the linear fractional constraints are defined last as required. To minimize α subject to “Equation ” to “Equation ”, call `gevp` by

```
[alpha,popt]=gevp(lmis,3)
```

This returns `alpha = -0.122` as the optimal value (the largest decay rate is therefore 0.122). This value is achieved for:

$$P = \begin{pmatrix} 5.58 & -8.35 \\ -8.35 & 18.64 \end{pmatrix}$$

Tips

Generalized eigenvalue minimization problems involve standard LMI constraints "Equation 1-4" and linear fractional constraints "Equation 1-6". For well-posedness, the positive definiteness of $B(x)$ must be enforced by adding the constraint $B(x) > 0$ to the problem. Although this could be done automatically from inside the code, this is not desirable for efficiency reasons. For instance, the set of constraints "Equation 1-5" may reduce to a single constraint as in the example above. In this case, the single extra LMI "$P > I$ " is enough to enforce positivity of *all* linear-fractional right sides. It is therefore left to the user to devise the least costly way of enforcing this positivity requirement.

References

The solver `gevp` is based on Nesterov and Nemirovski's Projective Method described in

Nesterov, Y., and A. Nemirovski, *Interior Point Polynomial Methods in Convex Programming: Theory and Applications*, SIAM, Philadelphia, 1994.

The optimization is performed by the C MEX-file `fpds.mex`.

See Also

`dec2mat` | `decnbr` | `feasp` | `mincx`

Introduced before R2006a

gridureal

Grid `ureal` parameters uniformly over their range

Syntax

```
B = gridureal(A,N)
[B,SampleValues] = gridureal(A,N)
[B,SampleValues] = gridureal(A,NAMES,N)
[B,SampleValues] = gridureal(A,NAMES1,N1,NAMES2,N2,...)
```

Description

`B = gridureal(A,N)` substitutes `N` uniformly-spaced samples of the uncertain real parameters in `A`. The samples are chosen to cut “diagonally” across the cube of real parameter uncertainty space. The array `B` has size equal to `[size(A) N]`. For example, suppose `A` has 3 uncertain real parameters, say `X`, `Y` and `Z`. Let (`x1, x2 , , and xN`) denote `N` uniform samples of `X` across its range. Similar for `Y` and `Z`. Then sample `A` at the points `(x1, y1, z1)`, `(x2, y2, z2)`, and `(xN, yN, zN)` to obtain the result `B`.

If `A` depends on additional uncertain objects, then `B` will be an uncertain object.

`[B,SampleValues] = gridureal(A,N)` additionally returns the specific sampled values (as a `structure` whose fieldnames are the names of `A`'s uncertain elements) of the uncertain reals. Hence, `B` is the same as `usubs(A,SampleValues)`.

`[B,SampleValues] = gridureal(A,NAMES,N)` samples only the uncertain reals listed in the `NAMES` variable (`cell`, or `char` array). Any entries of `NAMES` that are not elements of `A` are simply ignored. Note that `gridureal(A, fieldnames(A.Uncertainty),N)` is the same as `gridureal(A,N)`.

`[B,SampleValues] = gridureal(A,NAMES1,N1,NAMES2,N2,...)` takes `N1` samples of the uncertain real parameters listed in `NAMES1`, and `N2` samples of the uncertain real parameters listed in `NAMES2` and so on. `size(B)` will equal `[size(A) N1 N2 ...]`.

Examples

Grid Open-Loop and Closed-Loop Responses of Uncertain System

Create two uncertain real parameters `gamma` and `tau`. The nominal value of `gamma` is 4 and its range is 3 to 5. The nominal value of `tau` is 0.5 and its value can vary by +/- 30 percent.

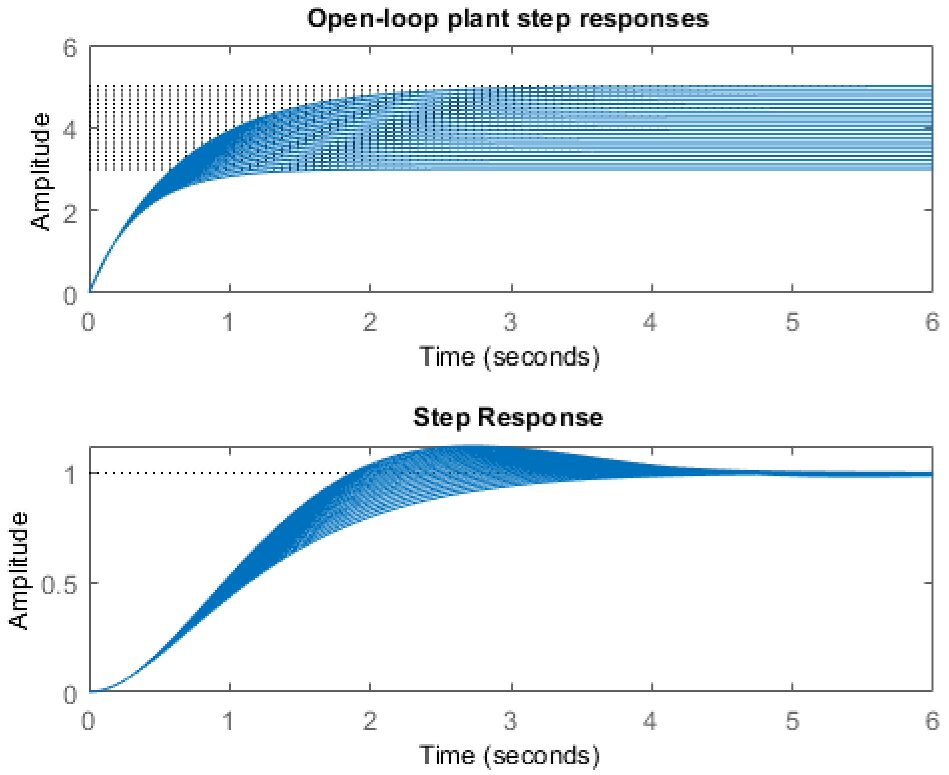
```
gamma = ureal('gamma',4);
tau = ureal('tau',.5,'Percentage',30);
```

These uncertain parameters are used to construct an uncertain transfer function `p`. An integral controller, `c`, is synthesized for the plant `p` based on the nominal values of `gamma` and `tau`. The uncertain closed-loop system `clp` is formed.

```
p = tf(gamma,[tau 1]);
KI = 1/(2*tau.Nominal*gamma.Nominal);
c = tf(KI,[1 0]);
clp = feedback(p*c,1);
```

The figure below shows the open-loop unit step response (top plot) and closed-loop response (bottom plot) for a grid of 20 values of `gamma` and `tau`.

```
subplot(2,1,1); step(gridureal(p,20),6)
title('Open-loop plant step responses')
subplot(2,1,2); step(gridureal(clp,20),6)
```



The plot illustrates the low-frequency closed-loop insensitivity achieved by the PI control system.

Grid Over Multi-Dimensional Parameter Spaces

This example illustrates the different options in gridding high-dimensional (e.g., `n` greater than 2) parameter spaces.

Construct an uncertain matrix, `m`, from four uncertain real parameters, `a`, `b`, `c`, and `d`, each making up the individual entries in `m`.

```
a = ureal('a',1);
b = ureal('b',2);
c = ureal('c',3);
d = ureal('d',4);
m = [a b;c d];
```

First, grid the `(a,b)` space at five places, and the `(c,d)` space at three places.

```
m1 = gridureal(m,{'a';'b'},5,{'c';'d'},3);
```

`gridureal` evaluates the uncertain matrix `m` at these 15 grid points, resulting in the numerical matrix `m1`.

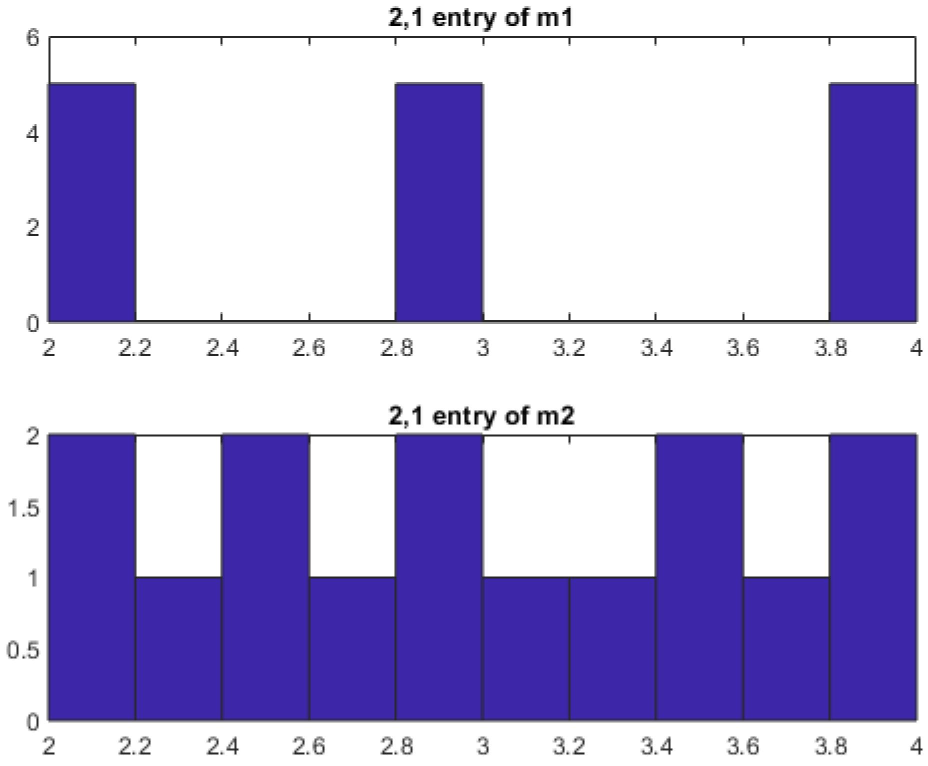
Next, grid the `(a,b,c,d)` space at 15 places.

```
m2 = gridureal(m,{'a';'b';'c';'d'},15);
```

`gridureal` samples the uncertain matrix `m` at these 15 points, resulting in the numerical matrix `m2`.

The (2,1) entry of `m` is just the uncertain real parameter `c`. Plot the histograms of the (2,1) entry of both `m1` and `m2`. The (2,1) entry of `m1` only takes on three distinct values, while the (2,1) entry of `m2` takes on 15 distinct values uniformly through its range.

```
subplot(2,1,1)
hist(squeeze(m1(2,1,:)))
title('2,1 entry of m1')
subplot(2,1,2)
hist(squeeze(m2(2,1,:)))
title('2,1 entry of m2')
```



See Also

`usample` | `usubs`

Introduced before R2006a

h2hinfsyn

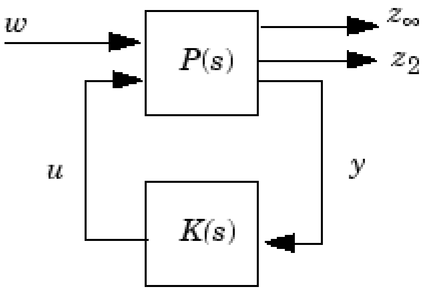
Mixed H_2/H_∞ synthesis with regional pole placement constraints

Syntax

```
[K,CL,normz,info] = h2hinfsyn(P,Nmeas,Ncon,Nz2,Wz,Name,Value)
```

Description

`[K,CL,normz,info] = h2hinfsyn(P,Nmeas,Ncon,Nz2,Wz,Name,Value)` employs LMI techniques to compute an output-feedback control law $u = K(s)y$ for the control problem of the following illustration.



The LTI plant `P` has partitioned state-space form given by

$$\begin{aligned} \dot{x} &= Ax + B_1 w + B_2 u, \\ z_\infty &= C_1 x + D_{11} w + D_{12} u, \\ z_2 &= C_2 x + D_{21} w + D_{22} u, \\ y &= C_y x + D_{y1} w + D_{y2} u\,. \end{aligned}$$

The resulting controller `K`:

- Keeps the H_∞ norm G of the transfer function from w to z_∞ below the value you specify using the `Name,Value` argument `'HINFMAX'`.

- Keeps the H_2 norm H of the transfer function from w to z_2 below the value you specify using the `Name,Value` argument `'H2MAX'`.
- Minimizes a trade-off criterion of the form

  $$W_1G^2 + W_2H^2,$$

  where W_1 and W_2 are the first and second entries in the vector `Wz`.
- Places the closed-loop poles in the LMI region that you specify using the `Name,Value` argument `'REGION'`.

Use the input arguments `Nmeas`, `Ncon`, and `Nz2` to specify the number of signals in y, u, and z_2, respectively. You can use additional `Name,Value` pairs to specify additional options for the computation.

Examples

Controller Design with Pole-Placement Constraint

Given a plant, design a controller such that the poles of the closed-loop system lie in the half-plane defined by Re(s) < –1.

You can define this region for pole-placement using the interactive `lmireg` command. To do so,

1. Enter `region = lmireg` at the MATLAB® command line.
2. Enter `h` to specify a half-plane constraint.
3. Enter `l` to specify the left half-plane.
4. Enter `-1` to specify that the cutoff for the region is `x0` = –1.
5. Enter `q` to exit and create the LMI region.

The region created by this process is equivalent to the following commands. (For more information, see the `lmireg` reference page.)

```
RealPart = -1;
region = [-2*RealPart + 1i 1];
```

Specify the plant model. For this example, use a three-input, two-output partitioned plant.

```
A = [1 0;2 1];
B = [1 1;1 0];
C = [1 1;1 1;1 1];
D = zeros(3,2);
P = ss(A,B,C,D);
```

Compute a controller for `P` using the LMI region to restrict the closed-loop pole locations. Assume the partitioned plant contains one control signal and one measurement signal (`nmeas = ncont = 1`). Apply an H_2 norm constraint to one signal (`Nz2 = 1`), and give the H_2 and H_∞ norms equal weight.

```
ncont = 1;
nmeas = 1;
Nz2 = 1 ;
Wz = [0 0];
[K,CL] = h2hinfsyn(P,nmeas,ncont,Nz2,Wz,'Region',region);
```

```
 Solver for LMI feasibility problems L(x) < R(x)
    This solver minimizes  t  subject to  L(x) < R(x) + t*I
    The best value of t should be negative for feasibility

 Iteration   :    Best value of t so far 

     1                        7.368392
     2                      -95.362851

 Result:  best value of t:   -95.362851
          f-radius saturation:  0.009% of R =  1.00e+08
```

Confirm that the poles of the closed-loop system have Re(*s*) < –1.

```
pole(CL)
```

```
ans = 4×1 complex

  -1.6786 + 3.2056i
  -1.6786 - 3.2056i
  -1.5563 + 1.6678i
  -1.5563 - 1.6678i
```

You can push the closed-loop eigenvalues further left by changing `RealPart`. Or you can define other pole-placement regions. For instance, place the poles such that Re(*s*) falls in a strip of the complex plane –5 < Re(s) < –3. To define this region, use `lmireg` interactively to create `reg1` specifying Re(s) > –5, and `reg2` specifying Re(s) < –3. Then, enter `region = lmireg(reg1,reg2)` to define the intersection of these two regions. The following code is equivalent.

```
LeftRealPart = -5;
RightRealPart = -3;
region = [-2*RightRealPart + 1i 0 1 0;
           0 2*LeftRealPart + 1i 0 -1];
```

Compute the new controller and confirm the locations of the closed-loop poles.

```
[K,CL] = h2hinfsyn(P,nmeas,ncont,Nz2,Wz,'Region',region);
```

```
 Solver for LMI feasibility problems L(x) < R(x)
    This solver minimizes  t  subject to  L(x) < R(x) + t*I
    The best value of t should be negative for feasibility

 Iteration   :    Best value of t so far

     1                       17.688394
     2                        1.074621
     3                      -13.502955

 Result:  best value of t:   -13.502955
          f-radius saturation:  0.048% of R =  1.00e+08
```

```
pole(CL)
```

```
ans = 4×1 complex

  -3.7864 + 4.9210i
  -3.7864 - 4.9210i
  -3.7752 + 3.6186i
  -3.7752 - 3.6186i
```

Input Arguments

P — Plant
LTI model

Plant, specified as an LTI model such as a `tf` or `ss` model. `P` must be a continuous-time model.

`Nmeas` — Number of measurement signals
positive integer

Number of measurement signals, specified as a positive integer. This value is the number of signals in *y*.

`Ncon` — Number of control signals
positive integer

Number of control signals, specified as a positive integer. This value is the number of signals in *u*.

`Nz2` — Number of signals subject to *H2* constraint
positive integers

Number of signals subject to the constraint on the H_2 norm, specified as a positive integer. This value is the number of signals in z_2. If the total number of outputs of `P` is `Nout`, then the first `Nout - Nz2 - Nmeas` outputs of `P` are subject to the constraint on the H_∞ norm.

`Wz` — Weights for H_∞ and H_2 performance
1-by-2 vector

Weights for H_∞ and H_2 performance, specified as a 1-by-2 vector of the form `[Winf,W2]`.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'REGION',reg,'H2MAX',1,'HINFMAX',1,'DISPLAY','on'`

`REGION` — Pole placement region
`[]` (default) | matrix

Pole placement region, specified as a comma-separated pair consisting of `'REGION'` and a matrix of the form `[L,M]`. This matrix specifies the pole placement region as:

$$\left\{ z : L + zM + \bar{z}M^T < 0 \right\}.$$

Generate the matrix using `lmireg`. The default LMI region for pole placement, specified by the empty matrix `[]`, is the open left-half plane, enforcing closed-loop stability only.

`H2MAX` — Upper bound on H_2 norm

`Inf` (default) | positive scalar

Upper bound on the H_2 norm of the transfer function from w to z_2, specified as a comma-separated pair consisting of `'H2MAX'` and a positive scalar value or `Inf`. The default value `Inf` is equivalent to setting the limit to zero, and causes `h2hinfsyn` to minimize the H_2 norm subject to the trade-off criterion.

Example: `'H2MAX',1`

`HINFMAX` — Upper bound on H_∞ norm

`Inf` (default) | positive scalar

Upper bound on the H_∞ norm of the transfer function from w to z_∞, specified as a comma-separated pair consisting of `'HINFMAX'` and a positive scalar value or `Inf`. The default value `Inf` is equivalent to setting the limit to zero, and causes `h2hinfsyn` to minimize the H_∞ norm subject to the trade-off criterion.

Example: `'HINFMAX',1`

`DKMAX` — Bound on controller feedthrough

100 (default) | nonnegative scalar

Bound on the norm on the feedthrough matrix D_K of the controller, specified as a comma-separated pair consisting of `'DKMAX'` and a nonnegative scalar value. To make the controller `K` strictly proper, set `'DKMAX'` to 0.

Example: `'DKMAX',0`

`TOL` — Relative accuracy of trade-off criterion

0.01 (default) | positive scalar

Desired relative accuracy on the optimal value of the trade-off criterion, specified as a comma-separated pair consisting of `'TOL'` and a positive scalar value.

`DISPLAY` — Toggle for screen display

`'off'` (default) | `'on'`

Toggle for screen display of synthesis information, specified as a comma-separated pair consisting of `'DISPLAY'` and either `'on'` or `'off'`.

Output Arguments

`K` — Optimal output-feedback controller
state-space model

Optimal output-feedback controller, returned as a state-space (`ss`) model with `Nmeas` inputs and `Ncon` outputs.

`CL` — Closed-loop system
state-space model

Closed-loop system with synthesized controller, returned as a state-space (`ss`) model. The closed-loop system is `CL = lft(P,K)`.

`normz` — Closed-loop norms
1-by-2 vector

Closed-loop norms, returned as a 1-by-2 vector. The entries in this vector, respectively, are:

- The H_∞ norm of the closed-loop transfer function from w to z_∞.
- The H_2 norm of the closed-loop transfer function from w to z_2.

`info` — Solutions of LMI solvability conditions
structure

Solutions of LMI solvability conditions, returned as a structure containing the following fields:

- `R` — Solution R of LMI solvability condition
- `S` — Solution S of LMI solvability condition

References

[1] Chilali, M., and P. Gahinet, "H_∞ Design with Pole Placement Constraints: An LMI Approach," *IEEE Trans. Aut. Contr.*, 41 (1995), pp. 358–367.

[2] Scherer, C., “Mixed H2/H-infinity Control,” *Trends in Control: A European Perspective*, Springer-Verlag (1995), pp.173–216.

See Also

`lmireg` | `msfsyn`

Introduced before R2006a

h2syn

Compute H_2 optimal controller

Syntax

```
[K,CL,gamma] = h2syn(P,nmeas,ncont)
[K,CL,gamma] = h2syn(P,nmeas,ncont,opts)
[K,CL,gamma,info] = h2syn( ___ )
```

Description

`[K,CL,gamma] = h2syn(P,nmeas,ncont)` computes a stabilizing H_2-optimal controller `K` for the plant `P`. The plant has a partitioned form

$$\begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} w \\ u \end{bmatrix},$$

where:

- w represents the disturbance inputs.
- u represents the control inputs.
- z represents the error outputs to be kept small.
- y represents the measurement outputs provided to the controller.

`nmeas` and `ncont` are the number of signals in y and u, respectively. y and u are the last outputs and inputs of `P`, respectively. `h2syn` returns a controller `K` that stabilizes `P` and has the same number of states. The closed-loop system `CL = lft(P,K)` achieves the performance level `gamma`, which is the H_2 norm of `CL` (see `norm`).

`[K,CL,gamma] = h2syn(P,nmeas,ncont,opts)` specifies additional computation options. To create `opts`, use `h2synOptions`.

`[K,CL,gamma,info] = h2syn( ___ )` returns a structure containing additional information about the H_2 synthesis computation. You can use this argument with any of the previous syntaxes.

Examples

Stabilizing Controller for MIMO Plant

Stabilize a 5-by-4 unstable plant with three states, two measurement signals, and one control signal.

In practice, P is an augmented plant that you have constructed by combining a model of the system to control with appropriate H_2 weighting functions. For this example, use the following model.

```
A = [5    6    -6
     6    0     5
    -6    5     4];
B = [0     4     0     0
     1     1    -2    -2
     4     0     0    -3];
C = [-6     0     8
     0     5     0
    -2     1    -4
     4    -6    -5
     0   -15     7];
D = [0     0     0     0
     0     0     0     1
     0     0     0     0
     0     0     3     6
     8     0    -7     0];
P = ss(A,B,C,D);
```

Confirm that P is unstable by examining its poles, some of which lie in the right half-plane.

```
pole(P)
```

```
ans = 3×1

   -8.5648
    6.8612
   10.7036
```

Design the stabilizing controller. `h2syn` assumes that the `nmeas` measurement signals and the `ncont` control signals are the last outputs and last inputs of P, respectively.

```
nmeas = 2;
ncont = 1;
[K,CL,gamma] = h2syn(P,nmeas,ncont);
```

Examine the closed-loop system to confirm that the controller `K` stabilizes the plant.

```
pole(CL)
```

```
ans = 6×1 complex

 -31.6236 + 0.0000i
 -12.6460 + 3.8045i
 -12.6460 - 3.8045i
  -9.6073 + 0.0000i
  -9.2393 + 0.0000i
  -8.6939 + 0.0000i
```

Mixed-Sensitivity H2 Loop Shaping

Shape the singular value plots of the sensitivity $S = (I + GK)^{-1}$ and complementary sensitivity $T = GK(I + GK)^{-1}$.

To do so, find a stabilizing controller `K` that minimizes the H_2 norm of:

$$T_{y_1u_1} \triangleq \begin{bmatrix} W_1S \\ (W_2/G)T \\ W_3T \end{bmatrix}.$$

Assume the following plant and weights:

$$G(s) = \frac{s-1}{s-2}, W_1 = \frac{0.1}{100s+1}, W_2 = 0.1, W_3 = 0.$$

Using those values, construct the augmented plant `P`, as illustrated in the `mixsyn` reference page.

```
s = zpk('s');
G = 10*(s-1)/(s+1)^2;
```

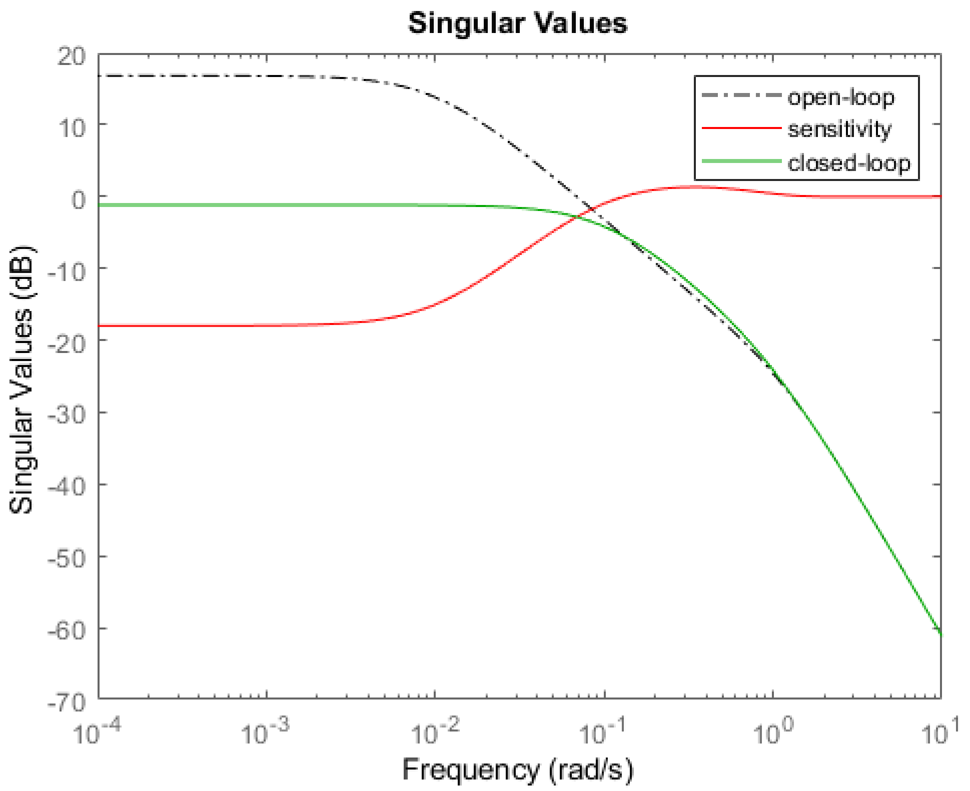
```
G.u = 'u2';
G.y = 'y';

W1 = 0.1/(100*s+1);
W1.u = 'y2';
W1.y = 'y11';

W2 = tf(0.1);
W2.u = 'u2';
W2.y = 'y12';

S = sumblk('y2 = u1 - y');

P = connect(G,S,W1,W2,{'u1','u2'},{'y11','y12','y2'});
```

Use `h2syn` to generate the controller. This system has one measurement signal and one control signal, which are the last output and input of `P`, respectively.

```
[K,CL,gamma] = h2syn(P,1,1);
```

Examine the resulting loop shapes.

```
L = G*K;
S = inv(1+L);
T = 1-S;
sigmaplot(L,'k-.',S,'r',T,'g')
legend('open-loop','sensitivity','closed-loop')
```



Input Arguments

P — Plant
dynamic system model

Plant, specified as a dynamic system model such as a state-space (`ss`) model. `P` can be any LTI model with inputs [*w*;*u*] and outputs [*z*;*y*], where:

- *w* represents the disturbance inputs.
- *u* represents the control inputs.

- z represents the error outputs to be kept small.
- y represents the measurement outputs provided to the controller.

Construct P such that measurement outputs y are the last outputs, and the control inputs u are the last inputs.

The function converts P to a state-space model of the form:

$$\begin{aligned} dx &= Ax + B_1 w + B_2 u \\ z &= C_1 x + D_{11} w + D_{12} u \\ y &= C_2 x + D_{21} w + D_{22} u\,. \end{aligned}$$

If P is a generalized state-space model with uncertain or tunable control design blocks, then the function uses the nominal or current value of those elements.

Conditions on P

For the H_2 synthesis problem to be solvable, (A,B_2) must be stabilizable, and (A,C_2) must be detectable. The plant is further restricted in that P_{12} and P_{21} must have no zeros on the imaginary axis (continuous-time plants) or the unit circle (discrete-time plants). In continuous time, this restriction means that

$$\begin{bmatrix} A - j\omega & B_2 \\ C_1 & D_{12} \end{bmatrix}$$

has full column rank for all frequencies ω. By default, `h2syn` automatically adds extra disturbances and errors to the plant to ensure that the restriction on P_{12} and P_{21} is met. This process is called regularization. If you are certain your plant meets the conditions, you can turn off regularization using the `Regularize` option of `h2synOptions`.

`nmeas` — Number of measurement outputs

1 (default) | nonnegative integer

Number of measurement output signals in the plant, specified as a nonnegative integer. The function takes the last `nmeas` plant outputs as the measurements y. The returned controller K has `nmeas` inputs.

`ncont` — Number of control inputs

1 (default) | nonnegative integer

Number of control input signals in the plant, specified as a nonnegative integer. The function takes the last `ncont` plant inputs as the controls *u*. The returned controller `K` has `ncont` outputs.

`opts` — Options
`h2synOptions` object

Additional options for the computation, specified as an options set you create using `h2synOptions`. Available options include turning off automatic scaling and regularization. For more information, see `h2synOptions`.

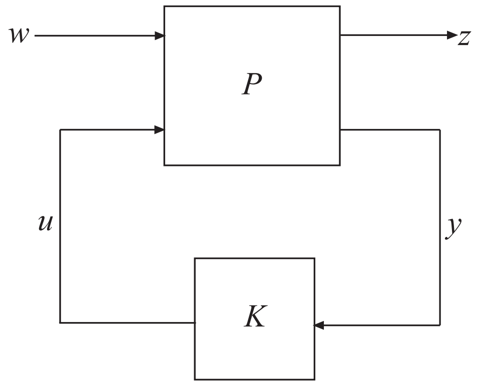
Output Arguments

`K` — Controller
`ss` model object

Controller, returned as a state-space (`ss`) model object. The controller stabilizes `P` and has the same number of states as `P`. The controller has `nmeas` inputs and `ncont` outputs.

`CL` — Closed-loop transfer function
`ss` model object | `[]`

Closed-loop transfer function, returned as a state-space (`ss`) model object or `[]`. The closed-loop transfer function is `CL = lft(P,K)` as in the following diagram.



`gamma` — Controller performance
nonnegative scalar

Controller performance, returned as a nonnegative scalar value. This value is the performance achieved using the returned controller `K`, and is the H_2 norm of `CL` (see `norm`).

`info` — Synthesis data

structure

Additional synthesis data, returned as a structure. `info` has the following fields.

Field	Description
`X`	Solution of state-feedback Riccati equation, returned as a matrix.
`Y`	Solution of observer Riccati equation, returned as a matrix.
`Ku`	State feedback gain of in the observer form of controller `K` returned as a matrix. For more information about the observer-form controller, see “Tips” on page 1-164.
`Lx,Lu`	Observer gains of the observer form of controller `K`, returned as matrices. For more information about the observer-form controller, see “Tips” on page 1-164.
`Preg`	Regularized plant used for `h2syn` computation, returned as a state-space (`ss`) model object. By default, `h2syn` automatically adds extra disturbances and errors to the plant to ensure that it meets certain conditions (see the input argument `P`). The field `info.Preg` contains the resulting plant model.
`NORMS`	Costs for the synthesized controller, returned in a vector of the form `[FI OE DF FC]`, where: • `FI` is the full-information control cost. • `OE` is the output-estimation cost. • `DF` is the disturbance-feedforward cost. • `FC` is full control cost. These quantities are related by `FI^2 + OE^2 = DF^2 + FC^2 = gamma^2`. For more details on these norms, see sections 14.8 and 14.9 of [1].

Field	Description
KFI	Full-information state-feedback gain, returned as a matrix. The full-information problem assumes full knowledge of the state `x` and disturbance `w`, and looks for an optimal state-feedback control of the form: • `u(t) = KFI*[x(t);w(t)]` in continuous time. In continuous time, `u` depends only on `x`. The entries in `KFI` corresponding to `w` are zero. • `u[k] = KFI*[x[k];w[k]]` in discrete time. For more information, see section 14.8 of [1].
GFI	Full-information closed-loop transfer from w to z with the controller `KFI`, returned as a state-space (`ss`) model. The H_2 norm of `GFI` is `FI`.
HAMX,HAMY	X Hamiltonian matrix (state feedback) and Y Hamiltonian matrix (Kalman filter). These values are provided for reference, but `h2syn` does not use them to compute the Riccati solutions. Instead, `h2syn` uses the implicit solvers `icare` and `idare`.

Tips

- `h2syn` gives you state-feedback gain and observer gains that you can use to express the controller in observer form. The observer form of the controller `K` is:

  $$\begin{aligned} dx_e &= Ax_e + B_2u + L_xe \\ u &= K_ux_e + L_ue\,. \end{aligned}$$

  Here, the innovation term e is:

  $$e = y - C_2x_e - D_{22}u\,.$$

  `h2syn` returns the state-feedback gain K_u and the observer gains L_x and L_u as fields in the `info` output argument.

  You can use this form of the controller for gain scheduling in Simulink. To do so, tabulate the plant matrices and the controller gain matrices as a function of the scheduling variables using the Matrix Interpolation block. Then, use the observer form of the controller to update the controller variables as the scheduling variables change.

Algorithms

`h2syn` uses the methods described in Chapter 14 of [1].

References

[1] Zhou, K., Doyle, J., Glover, K, *Robust and Optimal Control*. Upper Saddle River, NJ: Prentice Hall, 1996.

See Also

`h2synOptions` | `hinfsyn` | `loopsyn` | `mixsyn` | `ncfsyn` | `norm`

Introduced before R2006a

h2synOptions

Option set for `h2syn`

Syntax

```
opts = h2synOptions
opts = h2synOptions(Name,Value)
```

Description

`opts = h2synOptions` creates the default options set for the `h2syn` command.

`opts = h2synOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Examples

Turn Off Regularization for H2 Synthesis

Create an option set for the `h2syn` command that turns off automatic regularization of the plant. Turning off regularization can speed up the computation when you know your problem is far from singular.

You can use `Name,Value` pairs to create the option set.

```
opts = h2synOptions('Regularize','off');
```

Alternatively, create a default options set and use dot notation to change the option value.

```
opts = h2synOptions;
opts.Regularize = 'off';
```

You can now use `opts` as an input argument to `h2syn`.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'AutoScale','on','Regularize','off'`

`AutoScale` — Automatic plant scaling
`'on'` (default) | `'off'`

Automatic plant scaling, specified as the comma-separated pair consisting of `'AutoScale'` and one of the following:

- `'on'` — `h2syn` automatically scales the plant states, controls, and measurements to improve numerical accuracy. `h2syn` always returns the controller `K` in the original unscaled coordinates.
- `'off'` — `h2syn` does not change the plant scaling. Turning off scaling when you know your plant is well scaled can speed up the computation.

Example: `opts = h2synOptions('AutoScale','off')` creates an option set for `h2syn` that turns off automatic scaling.

`Regularize` — Automatic regularization
`'on'` (default) | `'off'`

Automatic regularization of the plant, specified as the comma-separated pair consisting of `'Regularize'` and one of the following:

- `'on'` — `h2syn` automatically regularizes the plant to enforce requirements on P_{12} and P_{21} (see `h2syn`). Regularization is a process of adding extra disturbances and errors to handle singular problems.
- `'off'` — `h2syn` does not regularize the plant. Turning off regularization can speed up the computation when you know your problem is far enough from singular.

Example: `opts = h2synOptions('Regularize','off')` creates an option set for `h2syn` that turns off regularization.

Output Arguments

opts — Options for h2syn
h2syn options object

Options for the h2syn computation, returned as an h2syn options object. Use the object as an input argument to h2syn. For example:

```
[K,CL,gamma,info] = h2syn(P,nmeas,ncont,opts);
```

See Also
h2syn

Introduced in R2019a

hankelmr

Hankel minimum degree approximation (MDA) without balancing

Syntax

```
GRED = hankelmr(G)
```

```
GRED = hankelmr(G,order)
```

```
[GRED,redinfo] = hankelmr(G,key1,value1,...)
```

```
[GRED,redinfo] = hankelmr(G,order,key1,value1,...)
```

Description

`hankelmr` returns a reduced order model `GRED` of `G` and a struct array `redinfo` containing the error bound of the reduced model and Hankel singular values of the original system.

The error bound is computed based on Hankel singular values of `G`. For a stable system Hankel singular values indicate the respective state energy of the system. Hence, reduced order can be directly determined by examining the system Hankel SV's, *σι*.

With only one input argument `G`, the function will show a Hankel singular value plot of the original model and prompt for model order number to reduce.

This method guarantees an error bound on the infinity norm of the *additive error* `‖G-GRED‖ ∞` for well-conditioned model reduced problems [1]:

$$\|G - Gred\|_{\infty} \leq 2 \sum_{k+1}^{n} \sigma_i$$

Note It seems this method is similar to the additive model reduction routines `balancmr` and `schurmr`, but actually it can produce more reliable reduced order model when the desired reduced model has nearly controllable and/or observable states (has Hankel singular values close to machine accuracy). `hankelmr` will then select an optimal reduced system to satisfy the error bound criterion regardless the order one might naively select at the beginning.

This table describes input arguments for `hankelmr`.

Argument	Description
`G`	LTI model to be reduced (without any other inputs will plot its Hankel singular values and prompt for reduced order)
`ORDER`	(Optional) an integer for the desired order of the reduced model, or optionally a vector packed with desired orders for batch runs

A batch run of a serial of different reduced order models can be generated by specifying `order = x:y, or a vector of integers`. By default, all the anti-stable part of a system is kept, because from control stability point of view, getting rid of unstable state(s) is dangerous to model a system.

`'MaxError'` can be specified in the same fashion as an alternative for `'ORDER'`. In this case, reduced order will be determined when the sum of the tails of the Hankel sv's reaches the '`MaxError`'.

Argument	Value	Description
`'MaxError'`	Real number or vector of different errors	Reduce to achieve H_∞ error. When present, `'MaxError'` overrides `ORDER` input.
`'Weights'`	`{Wout,Win}` cell array	Optimal 1x2 cell array of LTI weights `Wout` (output) and `Win` (input). Default for both is identity. Weights must be invertible.
`'Display'`	`'on'` or `'off'`	Display Hankel singular plots (default `'off'`).
`'Order'`	Integer, vector or cell array	Order of reduced model. Use only if not specified as 2nd argument.

Weights on the original model input and/or output can make the model reduction algorithm focus on some frequency range of interests. But weights have to be stable, minimum phase and invertible.

This table describes output arguments.

Argument	Description
GRED	LTI reduced order model. Become multi-dimensional array when input is a serial of different model order array.
REDINFO	A STRUCT array with 4 fields: • REDINFO.ErrorBound (bound on ‖ *G-GRED* ‖∞) • REDINFO.StabSV (Hankel SV of stable part of G) • REDINFO.UnstabSV (Hankel SV of unstable part of G) • REDINFO.Ganticausal (Anti-causal part of Hankel MDA)

G can be stable or unstable, continuous or discrete.

Note If size(GRED) is not equal to the order you specified. The optimal Hankel MDA algorithm has selected the best Minimum Degree Approximate it can find within the allowable machine accuracy.

Examples

Given a continuous or discrete, stable or unstable system, G, the following commands can get a set of reduced order models based on your selections:

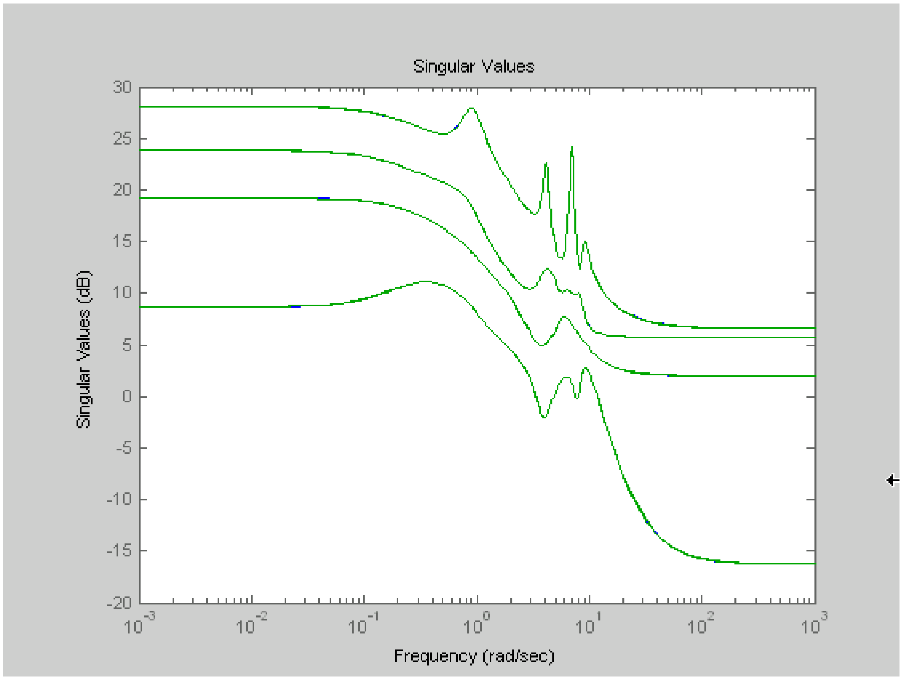
```
rng(1234,'twister');
G = rss(30,5,4);
[g1, redinfo1] = hankelmr(G); % display Hankel SV plot
                             % and prompt for order (try 15:20)
[g2, redinfo2] = hankelmr(G,20);
[g3, redinfo3] = hankelmr(G,[10:2:18]);
[g4, redinfo4] = hankelmr(G,'MaxError',[0.01, 0.05]);
for i = 1:4
    figure(i); eval(['sigma(G,g' num2str(i) ');']);
end
```

“Singular Value Bode Plot of G (30-state, 5 outputs, 4 inputs)” on page 1-172 shows a singular value Bode plot of a random system G with 20 states, 5 output and 4 inputs. The error system between G and its *Zeroth order Hankel MDA* has it infinity norm equals to an all pass function, as shown in “All-Pass Error System Between G and Zeroth Order G Anticausal” on page 1-173.

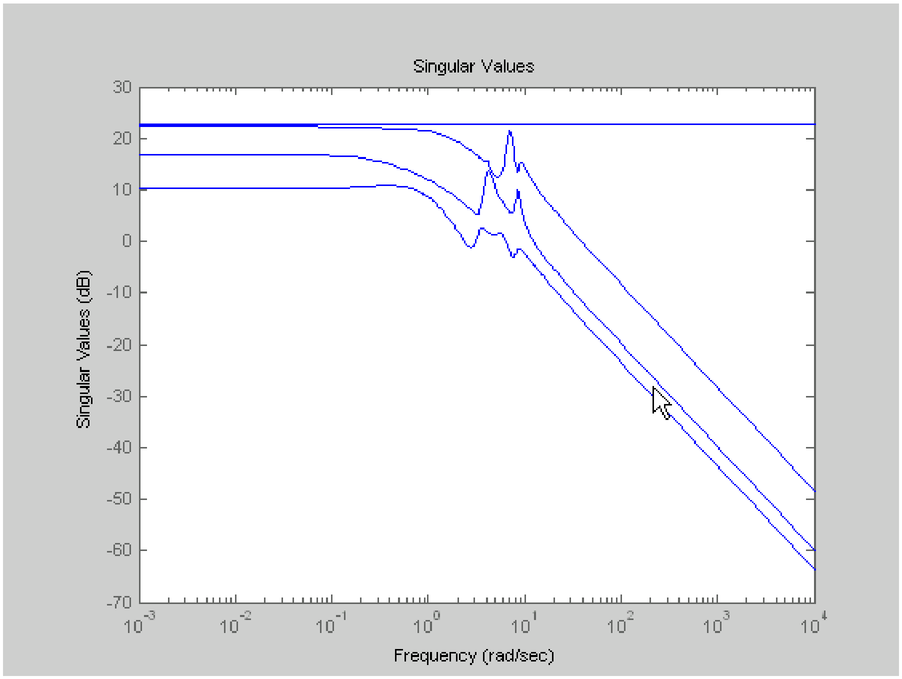
The *Zeroth order Hankel MDA* and its error system sigma plot are obtained via commands

```
[g0,redinfo0] = hankelmr(G,0);
sigma(G-redinfo0.Ganticausal)
```

This interesting all-pass property is unique in Hankel MDA model reduction.



Singular Value Bode Plot of G (30-state, 5 outputs, 4 inputs)



All-Pass Error System Between G and Zeroth Order G Anticausal

Algorithms

Given a state-space (A,B,C,D) of a system and k, the desired reduced order, the following steps will produce a similarity transformation to truncate the original state-space system to the k^{th} order reduced model.

1 Find the controllability and observability grammians P and Q.

2 Form the descriptor

$$E = QP - \rho^2 I$$

where $\sigma_k > \rho \geq \sigma_{k+1}$, and descriptor state-space

Take SVD of descriptor E and partition the result into k^{th} order truncation form

$$\begin{bmatrix} Es - \bar{A} & \bar{B} \\ \bar{C} & \bar{D} \end{bmatrix} = \begin{bmatrix} \rho^2 A^T + QAP & QB \\ CP & D \end{bmatrix}$$

$$E = [U_{E1}, U_{E2}] \begin{bmatrix} \Sigma_E 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{E1}^T \\ V_{E2}^T \end{bmatrix}$$

3 Apply the transformation to the descriptor state-space system above we have

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} U_{E1}^T \\ U_{E2}^T \end{bmatrix} (\rho^2 A^T + QAP)[V_{E1} \ V_{E2}]$$

$$\begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} U_{E1}^T \\ U_{E2}^T \end{bmatrix} \begin{bmatrix} QB & -C^T \end{bmatrix}$$

$$[C_1 \ C_2] = \begin{bmatrix} CP \\ -\rho B^T \end{bmatrix} [V_{E1} \ V_{E2}]$$

$$D_1 = D$$

4 Form the equivalent state-space model.

$$\begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{C} & \tilde{D} \end{bmatrix} = \begin{bmatrix} \sum_E^{-1} (A_{11} - A_{12} A_{22}{}^{\dagger} A_{21}) & \sum_E^{-1} (B_1 - A_{12} A_{22}{}^{\dagger} B_2) \\ C_1 - C_2 A_{22}{}^{\dagger} A_{21} & D_1 - C_2 A_{22}{}^{\dagger} B_2 \end{bmatrix}$$

The final k^{th} order Hankel MDA is the stable part of the above state-space realization. Its anticausal part is stored in `redinfo.Ganticausal`.

The proof of the Hankel MDA algorithm can be found in [2]. The error system between the original system G and the *Zeroth Order Hankel MDA* G_0 is an all-pass function [1].

References

[1] Glover, K., “All Optimal Hankel Norm Approximation of Linear Multivariable Systems, and Their L_∞-error Bounds,” *Int. J. Control*, vol. 39, no. 6, pp. 1145-1193, 1984.

[2] Safonov, M.G., R.Y. Chiang, and D.J.N. Limebeer, “Optimal Hankel Model Reduction for Nonminimal Systems,” *IEEE Trans. on Automat. Contr.*, vol. 35, no. 4, April 1990, pp. 496-502.

See Also
`balancmr` | `bstmr` | `hankelsv` | `ncfmr` | `reduce` | `schurmr`

Introduced before R2006a

hankelsv

Compute Hankel singular values for stable/unstable or continuous/discrete system

Syntax

```
hankelsv(G)
```

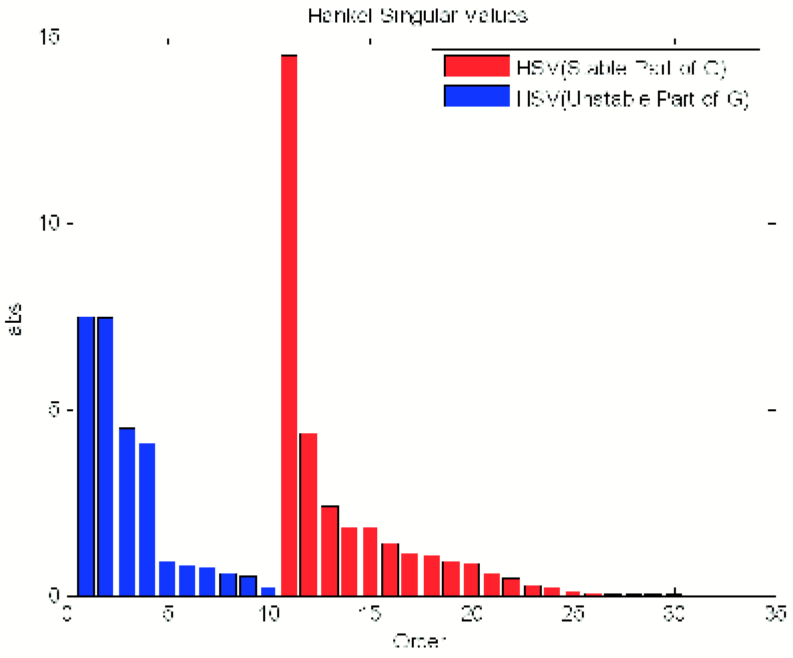
```
hankelsv(G,ErrorType,style)
```

```
[sv_stab,sv_unstab]=hankelsv(G,ErrorType,style)
```

Description

`[sv_stab,sv_unstab]=hankelsv(G,ErrorType,style)` returns a column vector `SV_STAB` containing the Hankel singular values of the stable part of `G` and `SV_UNSTAB` of anti-stable part (if it exists). The Hankel SV's of anti-stable part `ss(a,b,c,d)` is computed internally via `ss(-a,-b,c,d)`. Discrete model is converted to continuous one via the bilinear transform.

`hankelsv(G)` with no output arguments draws a bar graph of the Hankel singular values such as the following:



This table describes optional input arguments for `hankelsvd`.

Argument	Value	Description
`ERRORTYPE`	*`'add'`*	Regular Hankel SV's of G
	`'mult'`	Hankel SV's of phase matrix
	`'ncf'`	Hankel SV's of coprime factors
`STYLE`	*`'abs'`*	Absolute value
	`'log'`	logarithm scale

Algorithms

If`ErrorType = 'add'`, then `hankelsv` implements the numerically robust square root method to compute the Hankel singular values [1]. Its algorithm goes as follows:

Given a stable model G, with controllability and observability grammians P and Q, compute the SVD of P and Q:

```
[Up,Sp,Vp] = svd(P);
[Uq,Sq,Vq] = svd(Q);
```

Then form the square roots of the grammians:

```
Lr = Up*diag(sqrt(diag(Sp)));
Lo = Uq*diag(sqrt(diag(Sq)));
```

The Hankel singular values are simply:

σ_H `=svd(Lo'*Lr);`

This method not only takes the advantages of robust SVD algorithm, but also ensure the computations stay well within the “square root” of the machine accuracy.

If `ErrorType = 'mult'`, then `hankelsv` computes the Hankel singular value of the phase matrix of `G` [2].

If `ErrorType = 'ncf'`, then `hankelsv` computes the Hankel singular value of the normalized coprime factor pair of the model [3].

References

[1] Safonov, M.G., and R.Y. Chiang, “A Schur Method for Balanced Model Reduction,” *IEEE Trans. on Automat. Contr.*, vol. AC-2, no. 7, July 1989, pp. 729-733.

[2] Safonov, M.G., and R.Y. Chiang, “Model Reduction for Robust Control: A Schur Relative Error Method,” *International J. of Adaptive Control and Signal Processing, Vol. 2, pp. 259-272, 1988.*

[3] Vidyasagar, M., *Control System Synthesis - A Factorization Approach.* London: The MIT Press, 1985.

See Also

`balancmr` | `bstmr` | `hankelmr` | `ncfmr` | `reduce` | `schurmr`

Introduced before R2006a

hinffc

Full-control H-infinity synthesis

Syntax

```
[K,CL,gamma] = hinffc(P,nmeas)
[K,CL,gamma] = hinffc(P,nmeas,gamTry)
[K,CL,gamma] = hinffc(P,nmeas,gamRange)
[K,CL,gamma] = hinffc( ___ ,opts)
[K,CL,gamma,info] = hinffc( ___ )
```

Description

Full-control synthesis assumes the controller can directly affect both the state vector x and the error signal z. Synthesis with `hinffc` is the dual of the full-information problem covered by `hinffi`. For general H_∞ synthesis, use `hinfsyn`.

`[K,CL,gamma] = hinffc(P,nmeas)` computes the H_∞-optimal control law

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = Ky$$

for the plant `P`. The plant is described by the state-space equations:

$$\begin{aligned} dx &= Ax + B_1 w + u_1 \\ z &= C_1 x + D_{11} w + u_2 \\ y &= C_2 x + D_{21} w . \end{aligned}$$

Here,

- w represents the disturbance inputs
- u_1 represents the inputs that affect the state vector
- u_2 represents the inputs that affect the error

- z represents the error outputs to be kept small
- y represents the measurement outputs

`nmeas` is the number of measurements y, which must be the last outputs of `P`. The gain matrix `K` minimizes the H_∞ norm of the closed-loop transfer function `CL` from the disturbance signals w to the error signals z.

`[K,CL,gamma] = hinffc(P,nmeas,gamTry)` calculates a gain matrix for the target performance level `gamTry`. Specifying `gamTry` can be useful when the optimal achievable performance is better than you need for your application. In that case, a less-than-optimal solution can have smaller gains and be more numerically well-conditioned. If `gamTry` is not achievable, `hinffc` returns `[]` for `K` and `CL`, and `Inf` for `gamma`.

`[K,CL,gamma] = hinffc(P,nmeas,gamRange)` searches the range `gamRange` for the best achievable performance. Specify the range with a vector of the form `[gmin,gmax]`. Limiting the search range can speed up computation by reducing the number of iterations performed to test different performance levels.

`[K,CL,gamma] = hinffc(___,opts)` specifies additional computation options. To create `opts`, use `hinfsynOptions`. Specify `opts` after all other input arguments.

`[K,CL,gamma,info] = hinffc(___)` returns a structure containing additional information about the H_∞ synthesis computation. You can use this argument with any of the previous syntaxes.

Input Arguments

P — Plant
LTI model

Plant, specified as an LTI model such as a state-space (`ss`) model. If `P` is a generalized state-space model with uncertain or tunable control design blocks, then `hinffc` uses the nominal or current value of those elements.

Construct `P` so that it has the partitioned form

$$\begin{aligned} dx &= Ax + B_1 w + u_1 \\ z &= C_1 x + D_{11} w + u_2 \\ y &= C_2 x + D_{21} w\,. \end{aligned}$$

Here,

- w represents the disturbance inputs
- u_1 represents the inputs that affect the state vector
- u_2 represents the inputs that affect the error
- z represents the error outputs to be kept small
- y represents the measurement outputs

Construct `P` such that the `nmeas` measurement outputs are the last outputs.

For information about conditions imposed on the plant matrices and how the software addresses them, see `hinfsyn`.

`nmeas` — Number of measurements
nonnegative integer

Number of measurement output signals in the plant, specified as a nonnegative integer. `hinffc` takes the last `nmeas` plant outputs as the measurements y. The returned gain matrix `K` has `nmeas` inputs.

`gamTry` — Target performance level
positive scalar

Target performance level, specified as a positive scalar. `hinffc` attempts to compute a gain matrix such that the H_∞ of the closed-loop system does not exceed `gamTry`. If this performance level is achievable, then the returned gain matrix has `gamma` ≤ `gamTry`. If `gamTry` is not achievable, `hinffc` returns an empty matrix.

`gamRange` — Performance range for search
`[0,Inf]` (default) | vector of form `[gmin,gmax]`

Performance range for search, specified as a vector of the form `[gmin,gmax]`. The `hinffc` command tests only performance levels within that range. It returns a gain matrix with performance:

- `gamma` ≤ `gmin`, when `gmin` is achievable.
- `gmin` < `gamma` < `gmax`, when `gmax` is achievable and but `gmin` is not.
- `gamma` = `Inf` when `gmax` is not achievable. In this case, `hinffc` returns `[]` for `K` and `CL`.

If you know a range of feasible performance levels, specifying this range can speed up computation by reducing the number of iterations performed by `hinffc` to test different performance levels.

`opts` — Options
`hinfsynOptions` object

Additional options for the computation, specified as an options object you create using `hinfsynOptions`. Available options include displaying algorithm progress at the command line, turning off automatic scaling and regularization, and specifying an optimization method. For more information, see `hinfsynOptions`.

Output Arguments

`K` — Gain matrix
matrix | `[]`

Gain matrix, returned as a matrix or `[]`. The gain-matrix dimensions are n_u-by-`nmeas`, where n_u is the number of states plus the number of error outputs of `P` (outputs not included in `nmeas`).

If you supply `gamTry` or `gamRange` and the specified performance values are not achievable, then `K = []`.

`CL` — Closed-loop transfer function
`ss` model object | `[]`

Closed-loop transfer function, returned as a state-space (`ss`) model object or `[]`. The returned performance level `gamma` is the H_∞ norm of `CL`.

If you supply `gamTry` or `gamRange` and the specified performance levels are not achievable, then `CL = []`.

`gamma` — Closed-loop performance
nonnegative scalar | `Inf`

Closed-loop performance, returned as a nonnegative scalar value or `Inf`. This value is the H_∞ norm of `CL`. If you do not provide performance levels to test using `gamTry` or `gamRange`, then `gamma` is the best achievable performance level.

If you provide `gamTry` or `gamRange`, then `gamma` is the actual performance level achieved by the gain matrix computed for the best passing performance level that the function tries. If the specified performance levels are not achievable, then `gamma = Inf`.

`info` — Synthesis data
structure | `[]`

Additional synthesis data, returned as a structure or `[]` (if the specified performance level is not achievable). `info` has the following fields.

Field	Description
`gamma`	Performance level used to compute the gain matrix `K`, returned as a nonnegative scalar. Typically, `hinffc` tests multiple target performance levels and returns a gain matrix corresponding to the best passing performance level (see the Algorithms section of `hinfsyn` for details). The value `info.gamma` is an upper limit on the actual achieved performance returned as the output argument `gamma`.
`Y`	Riccati solution Y_∞ for the performance level `info.gamma`, returned as matrix. For more information, see the Algorithms section of `hinfsyn`.
`Preg`	Regularized plant used for `hinffc` computation, returned as a state-space (`ss`) model object. By default, `hinffc` automatically adds extra disturbances and errors to the plant to ensure that it meets certain conditions (see the Algorithms section of `hinfsyn`). The field `info.Preg` contains the resulting plant model.

Algorithms

For information about the algorithms used for H_∞ synthesis, see `hinfsyn`.

References

[1] Doyle, J.C., K. Glover, P. Khargonekar, and B. Francis. "State-space solutions to standard H_2 and H_∞ control problems." *IEEE Transactions on Automatic Control*, Vol 34, Number 8, August 1989, pp. 831–847.

See Also
`hinffi` | `hinfsyn` | `hinfsynOptions`

Introduced in R2018b

hinffi

Full-information H-infinity synthesis

Syntax

```
[K,CL,gamma] = hinffi(P,ncont)
[K,CL,gamma] = hinffi(P,ncont,gamTry)
[K,CL,gamma] = hinffi(P,ncont,gamRange)
[K,CL,gamma] = hinffi( ___ ,opts)
[K,CL,gamma,info] = hinffi( ___ )
```

Description

Full-information synthesis assumes the controller has access to both the state vector x and the disturbance signal w. Synthesis with `hinffi` is the dual of the full-control problem covered by `hinffc`. For the more general output-feedback case when only output measurements are available, use `hinfsyn`.

`[K,CL,gamma] = hinffi(P,ncont)` computes the H_∞-optimal control law

$$u = K\begin{bmatrix} x \\ w \end{bmatrix}$$

for the plant `P`. The plant is described by the state-space equations:

$$\begin{aligned} dx &= Ax + B_1 w + B_2 u \\ z &= C_1 x + D_{11} w + D_{12} u\,. \end{aligned}$$

Here, w represents the disturbance inputs, and z represents the error outputs to be kept small.

`ncont` is the number of control inputs u, which must be the last inputs of `P`. The gain matrix `K` minimizes the H_∞ norm of the closed-loop transfer function `CL` from the disturbance signals w to the error signals z.

`[K,CL,gamma] = hinffi(P,ncont,gamTry)` calculates a gain matrix for the target performance level `gamTry`. Specifying `gamTry` can be useful when the optimal achievable performance is better than you need for your application. In that case, a less-than-optimal solution can have smaller gains and be more numerically well-conditioned. If `gamTry` is not achievable, `hinffi` returns `[]` for `K` and `CL`, and `Inf` for `gamma`.

`[K,CL,gamma] = hinffi(P,ncont,gamRange)` searches the range `gamRange` for the best achievable performance. Specify the range with a vector of the form `[gmin,gmax]`. Limiting the search range can speed up computation by reducing the number of iterations performed to test different performance levels.

`[K,CL,gamma] = hinffi( ___ ,opts)` specifies additional computation options. To create `opts`, use `hinfsynOptions`. Specify `opts` after all other input arguments.

`[K,CL,gamma,info] = hinffi( ___ )` returns a structure containing additional information about the H_∞ synthesis computation. You can use this argument with any of the previous syntaxes.

Input Arguments

P — Plant
LTI model

Plant, specified as an LTI model such as a state-space (`ss`) model. If `P` is a generalized state-space model with uncertain or tunable control design blocks, then `hinffi` uses the nominal or current value of those elements.

Construct `P` so that it has the partitioned form

$$\begin{aligned} dx &= Ax + B_1 w + B_2 u \\ z &= C_1 x + D_{11} w + D_{12} u\,. \end{aligned}$$

Here, w represents the disturbance inputs, and z represents the error outputs to be kept small. The `ncont` control inputs u are the last inputs.

For information about conditions imposed on the plant matrices and how the software addresses them, see `hinfsyn`.

`ncont` — Number of controls
nonnegative integer

Number of control input signals in the plant, specified as a nonnegative integer. `hinffi` takes the last `ncont` plant inputs as the control *u*. The returned gain matrix `K` has `ncont` outputs.

`gamTry` — Target performance level
positive scalar

Target performance level, specified as a positive scalar. `hinffi` attempts to compute a gain matrix such that the H_∞ of the closed-loop system does not exceed `gamTry`. If this performance level is achievable, then the returned gain matrix has `gamma` ≤ `gamTry`. If `gamTry` is not achievable, `hinffi` returns an empty matrix.

gamRange — Performance range for search
`[0,Inf]` (default) | vector of form `[gmin,gmax]`

Performance range for search, specified as a vector of the form `[gmin,gmax]`. The `hinffi` command tests only performance levels within that range. It returns a gain matrix with performance:

- `gamma` ≤ `gmin`, when `gmin` is achievable.
- `gmin` < `gamma` < `gmax`, when `gmax` is achievable and but `gmin` is not.
- `gamma` = `Inf` when `gmax` is not achievable. In this case, `hinffi` returns `[]` for `K` and `CL`.

If you know a range of feasible performance levels, specifying this range can speed up computation by reducing the number of iterations performed by `hinffi` to test different performance levels.

`opts` — Options
`hinfsynOptions` object

Additional options for the computation, specified as an options object you create using `hinfsynOptions`. Available options include displaying algorithm progress at the command line, turning off automatic scaling and regularization, and specifying an optimization method. For more information, see `hinfsynOptions`.

Output Arguments

K — Gain matrix
matrix | `[]`

Gain matrix, returned as a matrix or `[]`. The gain-matrix dimensions are `ncont`-by-n_y where n_y is the number of states plus the number disturbance inputs of `P` (inputs not included in `ncont`).

If you supply `gamTry` or `gamRange` and the specified performance values are not achievable, then `K = []`.

`CL` — Closed-loop transfer function
`ss` model object | `[]`

Closed-loop transfer function, returned as a state-space (`ss`) model object or `[]`. The returned performance level `gamma` is the H_∞ norm of `CL`.

If you supply `gamTry` or `gamRange` and the specified performance levels are not achievable, then `CL = []`.

`gamma` — Closed-loop performance
nonnegative scalar | `Inf`

Closed-loop performance, returned as a nonnegative scalar value or `Inf`. This value is the H_∞ norm of `CL`. If you do not provide performance levels to test using `gamTry` or `gamRange`, then `gamma` is the best achievable performance level.

If you provide `gamTry` or `gamRange`, then `gamma` is the actual performance level achieved by the gain matrix computed for the best passing performance level that the function tries. If the specified performance levels are not achievable, then `gamma = Inf`.

`info` — Synthesis data
structure | `[]`

Additional synthesis data, returned as a structure or `[]` (if the specified performance level is not achievable). `info` has the following fields

Field	Description
`gamma`	Performance level used to compute the gain matrix `K`, returned as a nonnegative scalar. Typically, `hinffi` tests multiple target performance levels and returns a gain matrix corresponding to the best passing performance level (see the Algorithms section of `hinfsyn` for details). The value `info.gamma` is an upper limit on the actual achieved performance returned as the output argument `gamma`.

Field	Description
`X`	Riccati solution X_∞ for the performance level `info.gamma`, returned as a matrix. For more information, see the Algorithms section of `hinfsyn`.
`Preg`	Regularized plant used for `hinffi` computation, returned as a state-space (`ss`) model object. By default, `hinffi` automatically adds extra disturbances and errors to the plant to ensure that it meets certain conditions (see the Algorithms section of `hinfsyn`). The field `info.Preg` contains the resulting plant model.

Algorithms

For information about the algorithms used for H_∞ synthesis, see `hinfsyn`.

References

[1] Doyle, J.C., K. Glover, P. Khargonekar, and B. Francis. "State-space solutions to standard H_2 and H_∞ control problems." *IEEE Transactions on Automatic Control*, Vol 34, Number 8, , August 1989, pp. 831–847.

See Also

`hinffc` | `hinfsyn` | `hinfsynOptions`

Introduced in R2018b

hinfgs

Synthesis of gain-scheduled H_∞ controllers

Syntax

```
[gopt,pdK,R,S] = hinfgs(pdP,r,gmin,tol,tolred)
```

Description

Given an affine parameter-dependent plant

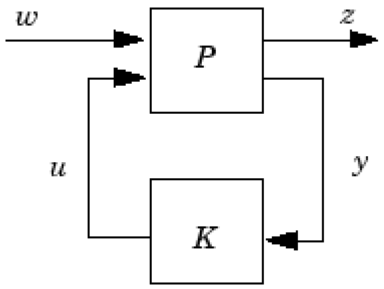
$$P\begin{cases} \dot{x} = A(p)x + B_1(p)w + B_2u \\ z = C_1(p)x + D_{11}(p)w + D_{12}u \\ y = C_2x + D_{21}w + D_{22}u \end{cases}$$

where the time-varying parameter vector $p(t)$ ranges in a box and is measured in real time, `hinfgs` seeks an affine parameter-dependent controller

$$K\begin{cases} \dot{\zeta} = A_K(p)\zeta + B_K(p)y \\ u = C_K(p)\zeta + D_K(P)y \end{cases}$$

scheduled by the measurements of $p(t)$ and such that

- K stabilizes the closed-loop system



for all admissible parameter trajectories $p(t)$

- K minimizes the closed-loop quadratic H_∞ performance from w to z.

The description `pdP` of the parameter-dependent plant P is specified with `psys` and the vector `r` gives the number of controller inputs and outputs (set `r=[p2,m2]` if $y \in R^{p2}$ and $u \in R^{m2}$). Note that `hinfgs` also accepts the polytopic model of P returned, e.g., by `aff2pol`.

`hinfgs` returns the optimal closed-loop quadratic performance `gopt` and a polytopic description of the gain-scheduled controller `pdK`. To test if a closed-loop quadratic performance γ is achievable, set the third input `gmin` to γ. The arguments `tol` and `tolred` control the required relative accuracy on `gopt` and the threshold for order reduction. Finally, `hinfgs` also returns solutions R, S of the characteristic LMI system.

Controller Implementation

The gain-scheduled controller `pdK` is parametrized by $p(t)$ and characterized by the values $K_{\Pi j}$ of $\begin{pmatrix} A_K(p) & B_K(p) \\ C_K(p) & D_K(p) \end{pmatrix}$ at the corners $^3{}_j$ of the parameter box. The command

```
Kj = psinfo(pdK,'sys',j)
```

returns the j-th vertex controller $K_{\Pi j}$ while

```
pv = psinfo(pdP,'par')
vertx = polydec(pv)
Pj = vertx(:,j)
```

gives the corresponding corner $^3{}_j$ of the parameter box (`pv` is the parameter vector description).

The controller scheduling should be performed as follows. Given the measurements $p(t)$ of the parameters at time t,

1. Express $p(t)$ as a convex combination of the $^3{}_j$:

$$p(t) = \alpha_1 3_1 + \ldots + \alpha_N 3_N,\ \alpha_j \geq 0,\ \sum_{i=1}^{N} \alpha_j = 1$$

   This convex decomposition is computed by `polydec`.

2 Compute the controller state-space matrices at time t as the convex combination of the vertex controllers $K_{\Pi j}$:

$$\begin{pmatrix} A_K(t) & B_K(t) \\ C_K(t) & D_K(t) \end{pmatrix} = \sum_{i=1}^{N} \alpha_j K_{\Pi_i} .$$

3 Use $A_K(t)$, $B_K(t)$, $C_K(t)$, $D_K(t)$ to update the controller state-space equations.

References

Apkarian, P., P. Gahinet, and G. Becker, “Self-Scheduled H_∞ Control of Linear Parameter-Varying Systems,” *Automatica*, 31 (1995), pp. 1251–1261.

Becker, G., Packard, P., “Robust Performance of Linear-Parametrically Varying Systems Using Parametrically-Dependent Linear Feedback,” *Systems and Control Letters*, 23 (1994), pp. 205–215.

Packard, A., “Gain Scheduling via Linear Fractional Transformations,” *Syst. Contr. Letters*, 22 (1994), pp. 79–92.

See Also

`pdsimul` | `polydec` | `psys` | `pvec`

Introduced before R2006a

hinfnorm

H_∞ norm of dynamic system

Syntax

```
ninf = hinfnorm(sys)
ninf = hinfnorm(sys,tol)
[ninf,fpeak] = hinfnorm( ___ )
```

Description

`ninf = hinfnorm(sys)` returns the H_∞ norm in absolute units of the dynamic system model, `sys`.

- If `sys` is a stable SISO system, then the H_∞ norm is the peak gain, the largest value of the frequency response magnitude.
- If `sys` is a stable MIMO system, then the H_∞ norm is the largest singular value across frequencies.
- If `sys` is an unstable system, then the H_∞ norm is defined as `Inf`.
- If `sys` is a model that has tunable or uncertain parameters, then `hinfnorm` evaluates the H_∞ norm at the current or nominal value of `sys`.
- If is a model array, then `hinfnorm` returns an array of the same size as `sys`, where `ninf(k) = hinfnorm(sys(:,:,k))` .

For stable systems, `hinfnorm(sys)` is the same as `getPeakGain(sys)`.

`ninf = hinfnorm(sys,tol)` returns the H_∞ norm of `sys` with relative accuracy `tol`.

`[ninf,fpeak] = hinfnorm( ___ )` also returns the frequency, `fpeak`, at which the peak gain or largest singular value occurs. You can use this syntax with any of the input arguments in previous syntaxes. If `sys` is unstable, then `fpeak = Inf`.

Examples

Norm of MIMO System

Compute the H_∞ norm of the following 2-input, 2-output dynamic system and the frequency at which the peak singular value occurs.

$$G(s) = \begin{bmatrix} 0 & \dfrac{3s}{s^2 + s + 10} \\ \dfrac{s+1}{s+5} & \dfrac{2}{s+6} \end{bmatrix}.$$

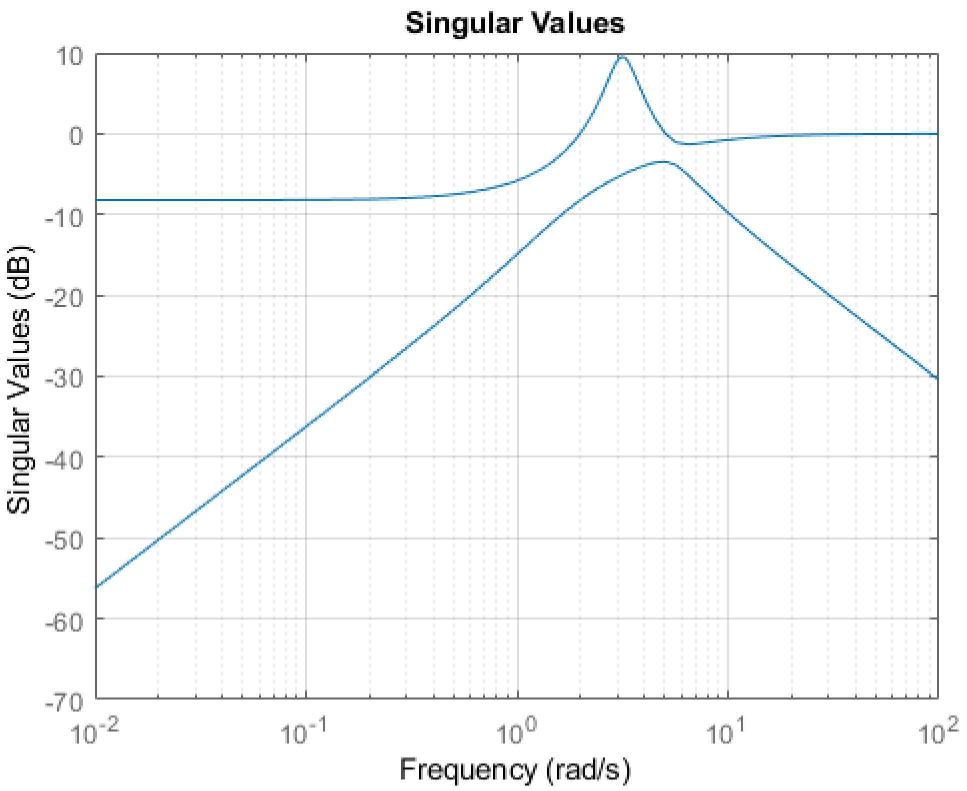
```
G = [0 tf([3 0],[1 1 10]);tf([1 1],[1 5]),tf(2,[1 6])];
[ninf,fpeak] = hinfnorm(G)
```

```
ninf = 3.0150
```

```
fpeak = 3.1623
```

The H_∞ norm of a MIMO system is its maximum singular value. Plot the singular values of `G` and compare the result from `hinfnorm`.

```
sigma(G),grid
```



The values `ninf` and `fpeak` are consistent with the singular value plot, which displays the values in dB.

Input Arguments

`sys` — Input dynamic system
dynamic system model | model array

Input dynamic system, specified as any dynamic system model or model array. `sys` can be SISO or MIMO.

`tol` — Relative accuracy
0.01 (default) | positive real scalar

Relative accuracy of the peak gain, specified as a positive real scalar value. `hinfnorm` calculates `ninf` such that the fractional difference between `ninf` and the true H_∞ norm of `sys` is no greater than `tol`.

Output Arguments

`ninf` — H_∞ norm of dynamic system
`Inf` | scalar | array

H_∞ norm of `sys`, returned as `Inf`, a scalar value, or an array.

- If `sys` is a single stable model, then `ninf` is a scalar value.
- If `sys` is a single unstable model, then `ninf` is `Inf`.
- If `sys` is a model array, then `ninf` is an array of the same size as `sys`, where `ninf(k) = hinfnorm(sys(:,:,k))`.

`fpeak` — Frequency of peak gain or largest singular value
`Inf` | nonnegative real scalar | array

Frequency at which the peak gain or largest singular value occurs, returned as `Inf`, a nonnegative real scalar value, or an array. The frequency is expressed in units of rad/`TimeUnit`, relative to the `TimeUnit` property of `sys`.

- If `sys` is a single stable model, then `fpeak` is a scalar.
- If `sys` is a single unstable model, then `fpeak` is `Inf`.
- If `sys` is a model array, then `fpeak` is an array of the same size as `sys`.In this case, `fpeak(k)` is the peak gain or largest singular value frequency of the *k*th model in the array.

See Also

`freqresp` | `getPeakGain` | `norm` | `sigma`

Introduced in R2013b

hinfstruct

H_∞ tuning of fixed-structure controllers

Syntax

```
CL = hinfstruct(CL0)
[CL,gamma,info] = hinfstruct(CL0)
[CL,gamma,info] = hinfstruct(CL0,options)
[C,gamma,info] = hinfstruct(P,C0,options)
```

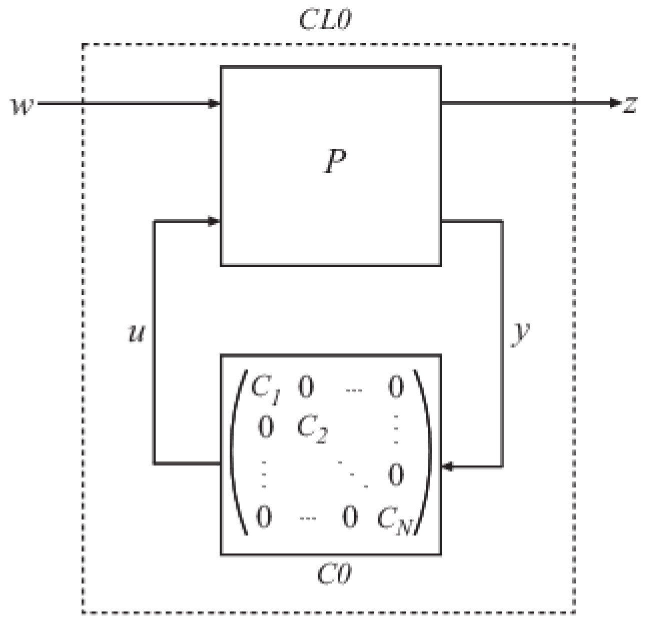
Description

`CL = hinfstruct(CL0)` tunes the free parameters of the tunable `genss` model `CL0`. This tuning minimizes the H_∞ norm of the closed-loop transfer function modeled by `CL0`. The model `CL0` represents a closed-loop control system that includes tunable components such as controllers or filters. `CL0` can also include weighting functions that capture design requirements.

`[CL,gamma,info] = hinfstruct(CL0)` returns `gamma` (the minimum H_∞ norm) and a data structure `info` with additional information about each optimization run.

`[CL,gamma,info] = hinfstruct(CL0,options)` allows you to specify additional options for the optimizer using `hinfstructOptions`.

`[C,gamma,info] = hinfstruct(P,C0,options)` tunes the parametric controller blocks `C0`. This tuning minimizes the H_∞ norm of the closed-loop system `CL0 = lft(P,C0)`. To use this syntax, express your control system and design requirements as a Standard Form model, as in the following illustration:


CL0
w
z
P
u
y
$\begin{pmatrix} C_1 & 0 & \cdots & 0 \\ 0 & C_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & C_N \end{pmatrix}$
C0

P is a numeric LTI model that includes the fixed elements of the control architecture. P can also include weighting functions that capture design requirements. C0 can be a single tunable component (for example, a Control Design Block (Control System Toolbox) or a genss model) or a cell array of multiple tunable components. C is a parametric model or array of parametric models of the same types as C0.

Input Arguments

CL0

Generalized state-space (genss) model describing the weighted closed-loop transfer function of a control system. CL0 includes both the fixed and tunable components of the control system. The tunable components of the control system are represented as control design blocks (Control System Toolbox), and are stored in the CL0.Blocks property of the genss model. hinfstruct adjusts the tunable parameters of the control design blocks in CL0 to minimize the H_∞ norm of the weighted transfer function. For more information about constructing this generalized model, see “Build Tunable Closed-Loop Model for Tuning with hinfstruct”.

`CL0` can be a continuous-time or discrete-time model. In discrete time, the sample time must be specified (`Ts` ≠ –1).

`CL0` can be an array of models, allowing you to tune a set of controller parameters for multiple models simultaneously. For example, the multiple models might represent different system configurations, different operating points, or different failure modes of a system.

`CL0` can include uncertain control design blocks as well as tunable blocks. If `CL0` has uncertain blocks, `hinfstruct` performs robust tuning.

`P`

Numeric LTI model representing the fixed elements of the control architecture to be tuned. `P` can also include weighting functions that capture design requirements. You can obtain `P` in two ways:

- In MATLAB, model the fixed elements of your control system as numeric LTI models. Then, use block-diagram building functions (such as `connect` and `feedback`) to build `P` from the modeled components. Also include any weighting functions that represent your design requirements.
- If you have a Simulink model of your control system and have Simulink Control Design™, use `linlft` to obtain a linear model of the fixed elements of your control system. The `linlft` command linearizes your Simulink model, excluding specified Simulink blocks (the blocks that represent the controller elements you want to tune). If you are using weighting functions to represent your design requirements, connect them in series with the linear model of your plant to obtain `P`.

`P` can be a continuous-time or discrete-time model. In discrete time, the sample time must be specified (`Ts` ≠ –1), and must match the sample time of `C0`.

`C0`

Single tunable component or cell array of tunable components of the control structure.

Each entry in `C0` represents one tunable element of your control architecture, such as a PID controller, a gain block, or a fixed-order transfer function. The entries of `C0` can be Control Design Blocks (Control System Toolbox) or `genss` models.

For more information and examples of creating tunable models, see “Models with Tunable Coefficients” (Control System Toolbox) in the *Control System Toolbox™ User's Guide*.

C0 can be a continuous-time or discrete-time model, as long as the sample time matches that of P.

options

Set of options for hinfstruct. Use hinfstructOptions to define options. For information about the available options, see the hinfstructOptions reference page.

Output Arguments

CL

Tuned version of the generalized state-space (genss) model CL0.

The hinfstruct command tunes the free parameters of CL0 to achieve a minimum H_∞ norm. CL.Blocks contains the same types of Control Design Blocks as CL0.Blocks, except that in CL, the parameters have tuned values.

To access the tuned parameter values, use getValue. You can also access them directly in CL.Blocks.

C

Tuned versions of the parametric models C0.

When C0 is a single parametric model, C is a parametric model of the same type, with tuned parameter values.

When C0 is a cell array of parametric models, C is also a cell array. The entries in C are parametric models of the same type as the corresponding entries in C0.

gamma

Best achieved value for the closed-loop H_∞ norm.

In some cases, hinfstruct performs more than one minimization run (when the hinfstructOptions option RandomStarts > 0). In such cases, gamma is the smallest H_∞ norm of all runs.

`info`

Data structure array containing results from each optimization run. The fields of `info` are:

- `Objective` — Minimum H_∞ norm value for each run.

  When `RandomStarts` = 0, `Objective` = `gamma`.
- `Iterations` — Number of iterations before convergence for each run.
- `TunedBlocks` — Tuned control design blocks for each run.

  `TunedBlocks` differs from `C` in that `C` contains only the result from the best run. When `RandomStarts` = 0, `TunedBlocks` = `C`.

Tips

- `hinfstruct` is related to `hinfsyn`, which also uses H_∞ techniques to design a controller for a MIMO plant. However, unlike `hinfstruct`, `hinfsyn` imposes no restriction on the structure and order of the controller. For that reason, `hinfsyn` always returns a smaller `gamma` than `hinfstruct`. You can therefore use `hinfsyn` to obtain a lower bound on the best achievable performance.
- Using `hinfstruct` requires some familiarity with H_∞ techniques. It requires expressing your design requirements as frequency-weighting functions on plant inputs and outputs, as described in “Formulating Design Requirements as H-Infinity Constraints”. For a simpler approach to fixed-structure tuning, use `systune` or `looptune`.

Algorithms

`hinfstruct` uses specialized nonsmooth optimization techniques to enforce closed-loop stability and minimize the H_∞ norm as a function of the tunable parameters. These techniques are based on the work in [1].

`hinfstruct` computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis," *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71-86.

[2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true` using `hinfstructOptions`.

See Also

`genss` | `getValue` | `hinfstructOptions` | `hinfsyn` | `looptune` | `systune`

Topics

“Build Tunable Closed-Loop Model for Tuning with hinfstruct”
“Fixed-Structure H-infinity Synthesis with HINFSTRUCT”
“What Is hinfstruct?”
“Formulating Design Requirements as H-Infinity Constraints”
“Structured H-Infinity Synthesis Workflow”
“Models with Tunable Coefficients” (Control System Toolbox)

Introduced in R2010b

hinfstructOptions

Set options for hinfstruct

Syntax

```
options = hinfstructOptions
options = hinfstructOptions(Name,Value)
```

Description

`options = hinfstructOptions` returns the default option set for the `hinfstruct` command.

`options = hinfstructOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`hinfstructOptions` takes the following `Name` arguments:

`Display`

Determines the amount of information to display during `hinfstruct` optimization runs.

`Display` takes the following values:

- `'off'` — `hinfstruct` runs in silent mode, displaying no information during or after the run.

- `'iter'` — Display optimization progress after each iteration. The display includes the value of the closed-loop H_∞ norm after each iteration. The display also includes a `Progress` value indicating the percent change in the H_∞ norm from the previous iteration.
- `'final'` — Display a one-line summary at the end of each optimization run. The display includes the minimized value of the closed-loop H_∞ norm and the number of iterations for each run.

Default: `'final'`

`MaxIter`

Maximum number of iterations in each optimization run.

Default: 300

`RandomStart`

Number of additional optimizations starting from random values of the free parameters in the controller.

If `RandomStart = 0`, `hinfstruct` performs a single optimization run starting from the initial values of the tunable parameters. Setting `RandomStart = N > 0` runs *N* additional optimizations starting from *N* randomly generated parameter values.

`hinfstruct` finds a local minimum of the gain minimization problem. To increase the likelihood of finding parameter values that meet your design requirements, set `RandomStart > 0`. You can then use the best design that results from the multiple optimization runs.

Use with `UseParallel = true` to distribute independent optimization runs among MATLAB workers (requires Parallel Computing Toolbox™ software).

Default: 0

`UseParallel`

Parallel processing flag.

Set to `true` to enable parallel processing by distributing randomized starts among workers in a parallel pool. If there is an available parallel pool, then the software performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

- If **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences (Parallel Computing Toolbox), then the software starts a parallel pool using the settings in those preferences.
- If **Automatically create a parallel pool** is not selected in your preferences, then the software performs the optimization runs successively, without parallel processing.

If **Automatically create a parallel pool** is not selected in your preferences, you can manually start a parallel pool using `parpool` before running the tuning command.

Using parallel processing requires Parallel Computing Toolbox software.

Default: `false`

`TargetGain`

Target H_∞ norm.

The `hinfstruct` optimization stops when the H_∞ norm (peak closed-loop gain) falls below the specified `TargetGain` value.

Set `TargetGain = 0` to optimize controller performance by minimizing the peak closed-loop gain. Set `TargetGain = Inf` to just stabilize the closed-loop system.

Default: 0

`TolGain`

Relative tolerance for termination. The optimization terminates when the H_∞ norm decreases by less than `TolGain` over 10 consecutive iterations. Increasing `TolGain` speeds up termination, and decreasing `TolGain` yields tighter final values.

Default: 0.001

`MaxFrequency`

Maximum closed-loop natural frequency.

Setting `MaxFrequency` constrains the closed-loop poles to satisfy `|p| < MaxFrequency`.

To let `hinfstruct` choose the closed-loop poles automatically based upon the system's open-loop dynamics, set `MaxFrequency = Inf`. To prevent unwanted fast dynamics or high-gain control, set `MaxFrequency` to a finite value.

Specify `MaxFrequency` in units of 1/`TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

Default: `Inf`

`MinDecay`

Minimum decay rate for closed-loop poles

Constrains the closed-loop poles to satisfy `Re(p) < -MinDecay`. Increase this value to improve the stability of closed-loop poles that do not affect the closed-loop gain due to pole/zero cancellations.

Specify `MinDecay` in units of 1/`TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

Default: `1e-7`

Output Arguments

`options`

Option set containing the specified options for the `hinfstruct` command.

Examples

Create Options Set for hinfstruct

Create an options set for a `hinfstruct` run using three random restarts and a stability offset of 0.001. Also, configure the `hinfstruct` run to stop as soon as the closed-loop gain is smaller than 1.

```
 options = hinfstructOptions('TargetGain',1,...
                             'RandomStart',3,'StableOffset',1e-3);
```

Alternatively, use dot notation to set the values of `options`.

```
options = hinfstructOptions;
options.TargetGain = 1;
options.RandomStart = 3;
options.StableOffset = 1e-3;
```

Configure Option Set for Parallel Optimization Runs

Configure an option set for a `hinfstruct` run using 20 random restarts. Execute these independent optimization runs concurrently on multiple workers in a parallel pool.

If you have the Parallel Computing Toolbox software installed, you can use parallel computing to speed up `hinfstruct` tuning of fixed-structure control systems. When you run multiple randomized `hinfstruct` optimization starts, parallel computing speeds up tuning by distributing the optimization runs among workers.

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences (Parallel Computing Toolbox), manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create an `hinfstructOptions` set that specifies 20 random restarts to run in parallel.

```
options = hinfstructOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the `hinfstructOptions` set when you call `hinfstruct`. For example, suppose you have already created a tunable closed loop model `CL0`. In this case, the following command uses parallel computing to tune `CL0`.

```
[CL,gamma,info] = hinfstruct(CL0,options);
```

See Also

`hinfstruct`

Introduced in R2010b

hinfsyn

Compute H-infinity optimal controller

Syntax

```
[K,CL,gamma] = hinfsyn(P,nmeas,ncont)
[K,CL,gamma] = hinfsyn(P,nmeas,ncont,gamTry)
[K,CL,gamma] = hinfsyn(P,nmeas,ncont,gamRange)
[K,CL,gamma] = hinfsyn( ___ ,opts)
[K,CL,gamma,info] = hinfsyn( ___ )
```

Description

`[K,CL,gamma] = hinfsyn(P,nmeas,ncont)` computes a stabilizing H_∞-optimal controller `K` for the plant `P`. The plant has a partitioned form

$$\begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} w \\ u \end{bmatrix},$$

where:

- w represents the disturbance inputs.
- u represents the control inputs.
- z represents the error outputs to be kept small.
- y represents the measurement outputs provided to the controller.

`nmeas` and `ncont` are the number of signals in y and u, respectively. y and u are the last outputs and inputs of `P`, respectively. `hinfsyn` returns a controller `K` that stabilizes `P` and has the same number of states. The closed-loop system `CL = lft(P,K)` achieves the performance level `gamma`, which is the H_∞ norm of `CL` (see `hinfnorm`).

`[K,CL,gamma] = hinfsyn(P,nmeas,ncont,gamTry)` calculates a controller for the target performance level `gamTry`. Specifying `gamTry` can be useful when the optimal controller performance is better than you need for your application. In that case, a less-than-optimal controller can have smaller gains and be more numerically well-conditioned. If `gamTry` is not achievable, `hinfsyn` returns `[]` for `K` and `CL`, and `Inf` for `gamma`.

`[K,CL,gamma] = hinfsyn(P,nmeas,ncont,gamRange)` searches the range `gamRange` for the best achievable performance. Specify the range with a vector of the form `[gmin,gmax]`. Limiting the search range can speed up computation by reducing the number of iterations performed by `hinfsyn` to test different performance levels.

`[K,CL,gamma] = hinfsyn(___,opts)` specifies additional computation options. To create `opts`, use `hinfsynOptions`. Specify `opts` after all other input arguments.

`[K,CL,gamma,info] = hinfsyn(___)` returns a structure containing additional information about the H_∞ synthesis computation. You can use this argument with any of the previous syntaxes.

Examples

H-Infinity Controller Synthesis

Synthesize a controller using different target performance levels. The plant in this example is based on the augmented plant model used in “Robust Control of an Active Suspension”. Load the plant.

```
load hinfsynExData P
size(P)
```

```
State-space model with 5 outputs, 4 inputs, and 9 states.
```

This plant has five outputs and four inputs, where the last two outputs are measurement signals to provide to the controller, and the last input is a control signal. Compute an H_∞-optimal controller.

```
ncont = 1;
nmeas = 2;
[K1,CL,gamma] = hinfsyn(P,nmeas,ncont);
```

The resulting two-input, one-output controller has the same number of states as P.

```
size(K1)
```

```
State-space model with 1 outputs, 2 inputs, and 9 states.
```

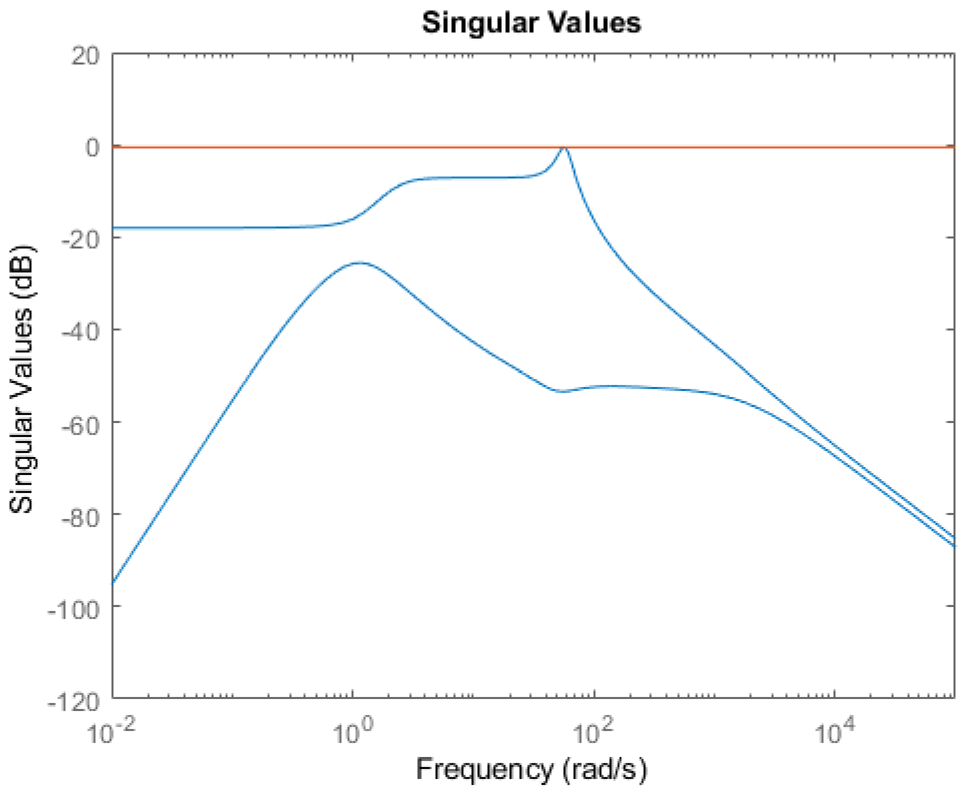
The optimal performance level achieved by this controller is returned as `gamma`. This value is the H_∞ norm of the closed-loop system `CL`.

```
gamma
```

```
gamma = 0.9405
```

You can examine the singular value plot of the closed-loop system to confirm that its largest singular value does not exceed `gamma`.

```
sigma(CL,ss(gamma))
ylim([-120,20]);
```



Controller Synthesis with Target Performance Level

For controllers that are close to optimal performance, controller gains can sometimes get large. If you know that your application does not require the optimal achievable performance level, you can limit the range of γ values that `hinfsyn` tests. Suppose you know that $\gamma \approx 1.5$ is good enough for your application. Using the same plant as in the example "H-Infinity Controller Synthesis" on page 1-211, compute a controller using a target performance range of [1.4,1.6]. Turn on the display to see the progress of the computation.

```
load hinfsynExData P
ncont = 1;
nmeas = 2;

opts = hinfsynOptions('Display','on');
gamRange = [1.4 1.6];
[K,CL,gamma,info] = hinfsyn(P,nmeas,ncont,gamRange,opts);
```

```
  Test bounds:  1.4 <=  gamma  <=  1.6

   gamma        X>=0        Y>=0       rho(XY)<1    p/f
  1.60e+00     4.9e-07     0.0e+00     1.462e-02     p
  1.50e+00     5.0e-07     0.0e+00     1.681e-02     p
  1.45e+00     5.0e-07     0.0e+00     1.803e-02     p
  1.42e+00     5.0e-07     1.7e-20     1.868e-02     p
  1.41e+00     5.0e-07     0.0e+00     1.902e-02     p

  Best performance (actual): 0.946
```

The display shows all the performance levels tested by `hinfsyn`. In this case, all tested performance levels pass the tests that `hinfsyn` applies for closed-loop stability (see "Algorithms" on page 1-222). Although the smallest tested level is 1.41, the controller returned for that value achieves an actual performance level of `gamma`, which is about 0.95. The smallest tested level is returned in the `gamma` field of the `info` structure.

```
info.gamma
```

```
ans = 1.4117
```

If you try to obtain a performance level that is not achievable with any controller, the display informs you that the target is too small, and returns an empty controller and closed-loop system. For example, suppose you try to achieve a performance level of 0.75.

```
gamTry = 0.75
gamTry = 0.7500
```

```
[K,CL,gamma] = hinfsyn(P,nmeas,ncont,gamTry,opts)
```

```
Specified upper limit GMAX=0.75 is too small, needs to be greater than 0.94.

K =

     []


CL =

     []

gamma = Inf
```

Mixed-Sensitivity Synthesis

Design a mixed-sensitivity controller for the following plant, augmented by the following loop-shaping filters (see `mixsyn`).

$$G(s) = \frac{s-1}{s+1}, W_1 = \frac{0.1(s+100)}{100s+1}, W_2 = 0.1, \text{no } W_3.$$

Define the plant, weighting filters, and augmented plant.

```
s = zpk('s');
G = (s-1)/(s+1);
W1 = 0.1*(s+100)/(100*s+1);
W2 = 0.1;
W3 = [];
P = augw(G,W1,W2,W3);
```

Synthesize the controller.

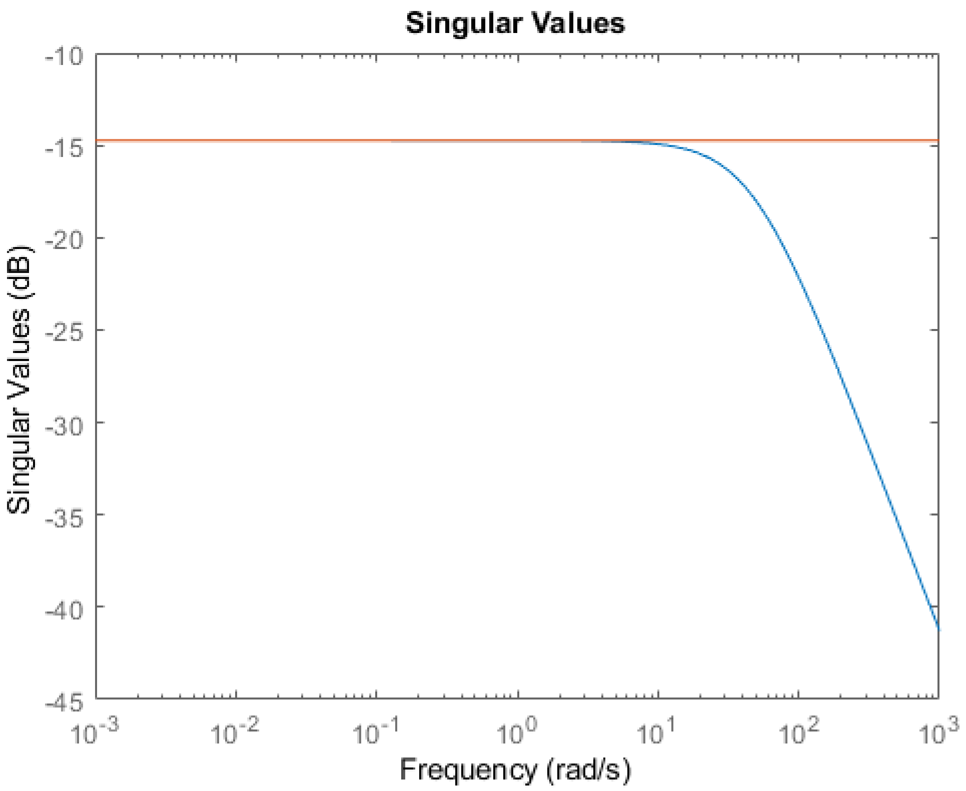
```
[K,CL,gamma] = hinfsyn(P,1,1);
gamma
```

```
gamma = 0.1831
```

For this system, `gamma` is about 0.18, or about –15 dB.

Examine the singular values of the closed-loop result.

```
sigma(CL,ss(gamma))
```



Compute a new controller for the same system with no W_1.

```
W1 = [];
P = augw(G,W1,W2,W3);
[K,CL,gamma] = hinfsyn(P,1,1);
```

In this case, the resulting controller `K` is zero, and the closed-loop transfer function `CL = K*(1+G*K)` is also zero.

Input Arguments

P — Plant
dynamic system model

Plant, specified as a dynamic system model such as a state-space (`ss`) model. `P` can be any LTI model with inputs [*w*;*u*] and outputs [*z*;*y*], where:

- *w* represents the disturbance inputs.
- *u* represents the control inputs.
- *z* represents the error outputs to be kept small.
- *y* represents the measurement outputs provided to the controller.

Construct `P` such that measurement outputs *y* are the last outputs, and the control inputs *u* are the last inputs.

The function converts `P` to a state-space model of the form:

$$\begin{aligned} dx &= Ax + B_1 w + B_2 u \\ z &= C_1 x + D_{11} w + D_{12} u \\ y &= C_2 x + D_{21} w + D_{22} u\,. \end{aligned}$$

If `P` is a generalized state-space model with uncertain or tunable control design blocks, then `hinfsyn` uses the nominal or current value of those elements.

One application of H_∞ control is direct shaping of closed-loop singular value plots of control systems. In such applications, you augment the plant inputs and outputs with weighting functions (loop-shaping filters) that represent control objectives that you want the H_∞ controller to satisfy. For a detailed example that constructs such a partitioned, augmented plant for H_∞ synthesis, see "Robust Control of an Active Suspension". For further information, see `mixsyn`.

Conditions on P

For the H_∞ synthesis problem to be solvable, (A,B_2) must be stabilizable, and (A,C_2) must be detectable. For the default Riccati method, the plant is further restricted in that P_{12} and P_{21} must have no zeros on the imaginary axis (continuous-time plants) or the unit circle (discrete-time plants). In continuous time, this restriction means that

$$\begin{bmatrix} A - j\omega & B_2 \\ C_1 & D_{12} \end{bmatrix}$$

has full column rank for all frequencies ω. By default, `hinfsyn` automatically adds extra disturbances and errors to the plant to ensure that the restriction on P_{12} and P_{21} is met. This process is called regularization. If you are certain your plant meets the conditions, you can turn off regularization using the `Regularize` option of `hinfsynOptions`.

`nmeas` — Number of measurement outputs

1 (default) | nonnegative integer

Number of measurement output signals in the plant, specified as a nonnegative integer. The function takes the last `nmeas` plant outputs as the measurements *y*. The returned controller `K` has `nmeas` inputs.

`ncont` — Number of control inputs

1 (default) | nonnegative integer

Number of control input signals in the plant, specified as a nonnegative integer. The function takes the last `ncont` plant inputs as the controls *u*. The returned controller `K` has `ncont` outputs.

`gamTry` — Target performance level

positive scalar

Target performance level, specified as a positive scalar. `hinfsyn` attempts to compute a controller such that the H_∞ of the closed-loop system does not exceed `gamTry`. If this performance level is achievable, then the returned controller has `gamma ≤ gamTry`. If `gamTry` is not achievable, `hinfsyn` returns an empty controller.

gamRange — Performance range for search

`[0,Inf]` (default) | vector of form `[gmin,gmax]`

Performance range for search, specified as a vector of the form `[gmin,gmax]`. The `hinfsyn` command tests only performance levels within that range. It returns a controller with performance:

- `gamma ≤ gmin`, when `gmin` is achievable.
- `gmin < gamma < gmax`, when `gmax` is achievable and but `gmin` is not.
- `gamma = Inf` when `gmax` is not achievable. In this case, `hinfsyn` returns `[]` for `K` and `CL`.

If you know a range of feasible performance levels, specifying this range can speed up computation by reducing the number of iterations performed by `hinfsyn` to test different performance levels.

opts — Additional options
hinfsynOptions object

Additional options for the computation, specified as an options object you create using hinfsynOptions. Available options include:

- Display algorithm progress at the command line.
- Turn off automatic scaling and regularization.
- Specify an optimization method.

For information about all options, see hinfsynOptions.

Output Arguments

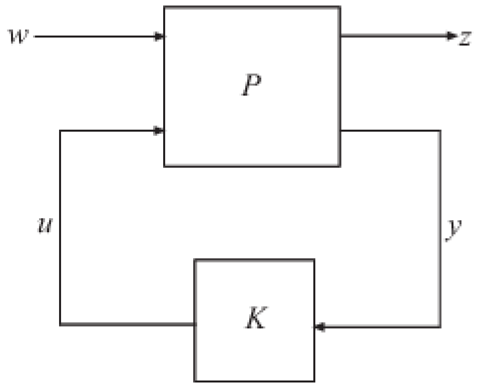
K — Controller
ss model object | []

Controller, returned as a state-space (ss) model object or []. The controller stabilizes P and has the same number of states as P. The controller has nmeas inputs and ncont outputs.

If you supply gamTry or gamRange and the specified performance values are not achievable, then K = [].

CL — Closed-loop transfer function
ss model object | []

Closed-loop transfer function, returned as a state-space (ss) model object or []. The closed-loop transfer function is given by CL = lft(P,K) as in the following diagram.



The returned performance level gamma is the H_∞ norm of CL.

If you supply gamTry or gamRange and the specified performance levels are not achievable, then CL = [].

gamma — Controller performance
nonnegative scalar | Inf

Controller performance, returned as a nonnegative scalar value or Inf. This value is the performance achieved using the returned controller K, and is the H_∞ norm of CL (see hinfnorm). If you do not provide performance levels to test using gamTry or gamRange, then gamma is the best achievable performance level.

If you provide gamTry or gamRange, then gamma is the actual performance level achieved by the controller computed for the best passing performance level that hinfsyn tries. If the specified performance levels are not achievable, then gamma = Inf.

info — Synthesis data
structure | []

Additional synthesis data, returned as a structure or [] (if the specified performance level is not achievable). For the default Riccati-based synthesis method, info has the following fields.

Field	Description
gamma	Performance level used to compute the controller K, returned as a nonnegative scalar. Typically, hinfsyn tests multiple target performance levels and returns a controller corresponding to the best passing performance level (see “Algorithms” on page 1-222). The value info.gamma is an upper limit on the actual achieved performance returned as the output argument gamma.
X,Y	Riccati solutions X_∞ and Y_∞ for the performance level info.gamma, returned as nonnegative scalars. For more information, see “Algorithms” on page 1-222 and [5].
Ku,Kw	State feedback gains of controller K expressed in observer form, returned as matrices. For more information about the observer-form controller, see “Tips” on page 1-221.
Lx,Lu	Observer gains of controller K expressed in observer form, returned as matrices. For more information about the observer-form controller, see “Tips” on page 1-221.
Preg	Regularized plant used for hinfsyn computation, returned as a state-space (ss) model object. By default, hinfsyn automatically adds extra disturbances and errors to the plant to ensure that it meets certain conditions (see the input argument P). The field info.Preg contains the resulting plant model.
AS	All-solutions controller on page 1-220 parameterization, returned as a state-space (ss) model object.

For the LMI-based synthesis method, info contains the best performance gamma and the corresponding LMI solutions R and S. (Use hinfsynOptions to change the synthesis method.)

More About

All-solutions controller

In general, the solution to the infinity-norm optimal control problem is nonunique. The controller returned by hinfsyn is only one particular solution, K. For the default Riccati-based method, info.AS contains the all-solution controller parameterization K_{AS}. All solutions with closed-loop performance of γ or less are parameterized by a free stable contraction map Q, which is constrained by $\|Q\|_\infty < \gamma$.

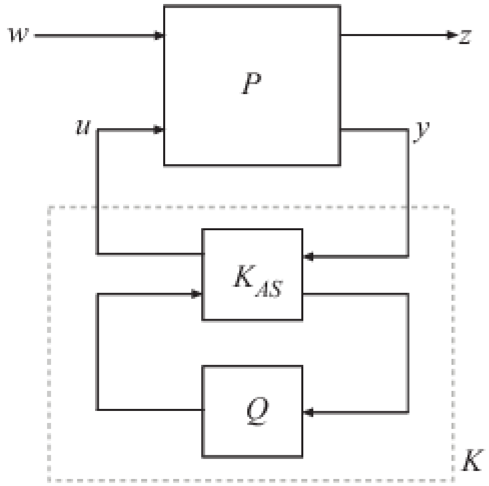
In other words, the solutions include every stabilizing controller $K(s)$ that makes

$$\|T_{y_1u_1}\|_\infty \triangleq \sup_\omega \sigma_{\max}\big(T_{y_1u_1}(j\omega)\big) < \gamma\,.$$

Here, $T_{y_1u_1}$ is the closed-loop transfer function `CL`. These controllers $K(s)$ are given by:

```
Ks = lft(info.AS,Q)
```

where `Q` is a stable LTI system satisfying `norm(Q,Inf) < info.gamma`.



Tips

- `hinfsyn` gives you state-feedback gains and observer gains that you can use to express the controller in observer form. The observer form of the controller `K` is:

$$\begin{aligned} dx_e &= Ax_e + B_1w_e + B_2u + L_xe \\ u &= K_ux_e + L_ue \\ w_e &= K_wx_e\,. \end{aligned}$$

Here, w_e is an estimate of the worst-case perturbation and the innovation term e is given by:

$$e = y - C_2 x_e - D_{21} w_e - D_{22} u\,.$$

hinfsyn returns the state-feedback gains K_u and K_w and the observer gains L_x and L_u as fields in the info output argument.

You can use this form of the controller for gain scheduling in Simulink. To do so, tabulate the plant matrices and the controller gain matrices as a function of the scheduling variables using the Matrix Interpolation block. Then, use the observer form of the controller to update the controller variables as the scheduling variables change.

Algorithms

By default, hinfsyn uses the two-Riccati formulae ([1],[2]) with loop shifting [3]. You can use hinfsynOptions to change to an LMI-based method ([4],[5],[6]). You can also specify a maximum-entropy method. In that method, hinfsyn returns the H_∞ controller that maximizes an entropy integral relating to the point S0. For continuous-time systems, this integral is:

$$\text{Entropy} = \frac{\gamma^2}{2\Pi}\int_{-\infty}^{\infty} \ln\left|\det I - \gamma^{-2} T_{y_1u_1}(j\omega)' T_{y_1u_1}(j\omega)\right|\left[\frac{s_o{}^2}{s_0{}^2 + \omega^2}\right] d\omega$$

where $T_{y_1u_1}$ is the closed-loop transfer function CL. A similar integral is used for discrete-time systems.

For all methods, the function uses a standard γ-iteration technique to determine the optimal value of the performance level γ. γ-iteration is a bisection algorithm that starts with high and low estimates of γ and iterates on γ values to approach the optimal H_∞ control design.

At each value of γ, the algorithm tests a γ value to determine whether a solution exists. In the Riccati-based method, the algorithm computes the smallest performance level for which the stabilizing Riccati solutions $X = X_\infty/\gamma$ and $Y = Y_\infty/\gamma$ exist. For any γ greater than that performance level and in the range gamRange, the algorithm evaluates the central controller formulas (K formulas) and checks the closed-loop stability of `CL = lft(P,K)`. This step is equivalent to verifying the conditions:

- `min(eig(X)) ≥ 0`
- `min(eig(Y)) ≥ 0`
- `rho(XY)` < 1, where the spectral radius `rho(XY) = max(abs(eig(XY)))`

A γ that meets these conditions passes. The stopping criterion for the bisection algorithm requires the relative difference between the last γ value that failed and the last γ value that passed be less than 0.01. (You can change this criterion using `hinfsynOptions`.) `hinfsyn` returns the controller corresponding to the smallest tested γ value that passes. For discrete-time controllers, the algorithm performs additional computations to construct the feedthrough matrix D_K.

Use the `Display` option of `hinfsynOptions` to make `hinfsyn` display values showing which of the conditions are satisfied for each γ value tested.

The algorithm works best when the following conditions are satisfied by the plant:

- D_{12} and D_{21} have full rank.
- $\begin{bmatrix} A - j\omega I & B_2 \\ C_1 & D_{12} \end{bmatrix}$ has full column rank for all $\omega \epsilon R$.
- $\begin{bmatrix} A - j\omega I & B_1 \\ C_2 & D_{21} \end{bmatrix}$ has full row rank for all $\omega \epsilon R$.

When these rank conditions do not hold, the controller may have undesirable properties. If D_{12} and D_{21} are not full rank, then the H_∞ controller `K` might have large high-frequency gain. If either of the latter two rank conditions does not hold at some frequency ω, the controller might have very lightly damped poles near that frequency.

Compatibility Considerations

Name,Value options are not recommended

Not recommended starting in R2018b

As of R2018b, using `Name,Value` syntax to specify options for `hinfsyn` is not recommended. Instead, to set a target performance range, use the `gamRange` input argument. For other options, create an options set with `hinfsynOptions`.

The following table shows how to update your calls to `hinfsyn` to use the recommended ways of specifying options.

Not Recommended	Recommended
`[K,CL,GAM] = hinfsyn(___,'GMIN',gmin,'GMAX',gmax)`	`[K,CL,GAM] = hinfsyn(___,gamRange)`
`[K,CL,GAM] = hinfsyn(___,'TOLGAM',tol)`	`opts = hinfsynOptions('RelTol',tol);` `[K,CL,GAM] = hinfsyn(___,opts);`
`[K,CL,GAM] = hinfsyn(___,'METHOD',meth)`	`opts = hinfsynOptions('Method',meth);` `[K,CL,GAM] = hinfsyn(___,opts);`
`[K,CL,GAM] = hinfsyn(___,'DISPLAY','on')`	`opts = hinfsynOptions('Display','on');` `[K,CL,GAM] = hinfsyn(___,opts);`

For more information, see `hinfsynOptions`.

`info` output argument changed

Behavior changed in R2018b

The fields of the optional output argument `info` changed in R2018b. Prior to that release, `info` was a structure with the following fields.

`AS`	All-solutions controller parameterization, scaled so that $\|Q\|_\infty < 1$
`KFI`	Full information gain matrix (constant feedback) $$u_2(t) = K_{FI}\begin{bmatrix} x(t) \\ u_1(t) \end{bmatrix}$$
`KFC`	Full control gain matrix (constant output-injection; K_{FC} is the dual of K_{FI})
`GAMFI`	H_∞ cost for full information K_{FI}
`GAMFC`	H_∞ cost for full control K_{FC}

`info.AS` is still available, but its scaling has changed. See “Scaling of info.AS” on page 1-225.

For the remaining fields, the following functions are recommended instead:

- `info.KFI`, `info.GAMFI` — Use `hinffi` for full-information synthesis.
- `info.KFC`, `info.GAMFC` — Use `hinffc` for full-control synthesis.

These fields are hidden in the `info` argument returned by `hinfsyn`. However, you can still access them using dot notation. For instance:

```
[K,CL,gamma,info] = hinfsyn(P,nmeas,ncont);
gfi = info.GAMFI;
gfc = info.GAMFC;
```

Scaling of `info.AS`

Prior to R2018b, the all-solutions controller parameterization `info.AS` was scaled so that the free stable contraction map Q was constrained by $\|Q\|_\infty < 1$. In R2018b, the scaling of `info.AS` has changed, so that the constraint on Q is $\|Q\|_\infty < \gamma$, where γ is `info.gamma`. This new constraint ensures that the all-solutions controller K_{AS} has a finite limit as `gamTry` $\rightarrow \infty$.

References

[1] Glover, K., and J.C. Doyle. "State-space formulae for all stabilizing controllers that satisfy an H_∞ norm bound and relations to risk sensitivity." *Systems & Control Letters*, Vol. 11, Number 8, 1988, pp. 167–172.

[2] Doyle, J.C., K. Glover, P. Khargonekar, and B. Francis. "State-space solutions to standard H_2 and H_∞ control problems." *IEEE Transactions on Automatic Control*, Vol 34, Number 8, August 1989, pp. 831–847.

[3] Safonov, M.G., D.J.N. Limebeer, and R.Y. Chiang. "Simplifying the H_∞ Theory via Loop Shifting, Matrix Pencil and Descriptor Concepts." *Int. J. Contr.*, Vol. 50, Number 6, 1989, pp. 2467-2488.

[4] Packard, A., K. Zhou, P. Pandey, J. Leonhardson, and G. Balas. "Optimal, constant I/O similarity scaling for full-information and state-feedback problems." *Systems & Control Letters*, Vol. 19, Number 4, 1992, pp. 271–280.

[5] Gahinet, P., and P. Apkarian. "A linear matrix inequality approach to H_∞-control." *Int. J. Robust and Nonlinear Control*, Vol. 4, Number. 4, 1994, pp. 421–448.

[6] Iwasaki, T., and R.E. Skelton. "All controllers for the general H_∞-control problem: LMI existence conditions and state space formulas." *Automatica*, Vol. 30, Number 8, 1994, pp. 1307–1317.

See Also

`augw` | `h2syn` | `hinffc` | `hinffi` | `hinfsynOptions` | `loopsyn` | `mixsyn` | `ncfsyn`

Topics

"Robust Control of an Active Suspension"

Introduced before R2006a

hinfsynOptions

Option set for `hinfsyn` and `mixsyn`

Syntax

```
opts = hinfsynOptions
opts = hinfsynOptions(Name,Value)
```

Description

`opts = hinfsynOptions` creates the default option set for the `hinfsyn` and `mixsyn`commands.

`opts = hinfsynOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Examples

Specify Algorithm and Display for H-Infinity Synthesis

Use the LMI-based algorithm to compute an H_∞-optimal controller for a plant with one control signal and two measurement signals. Turn on the display that shows the progress of the computation.

Load the plant, and specify the numbers of measurements and controls.

```
load hinfsynExData P
ncont = 1;
nmeas = 2;
```

Create an options set for `hinfsyn` that specifies the LMI-based algorithm and turns on the display.

```
opts = hinfsynOptions('Method','LMI','Display','on');
```

Alternatively, start with the default options set, and use dot notation to change option values.

```
opts = hinfsynOptions;
opts.Method = 'LMI';
opts.Display = 'on';
```

Compute the controller.

```
[K,CL,gamma] = hinfsyn(P,nmeas,ncont,opts);
```

```
 Minimization of gamma:

 Solver for linear objective minimization under LMI constraints

 Iterations   :    Best objective value so far

     1
     2                  223.728733
     3                  138.078240
     4                  138.078240
     5                   74.644885
     6                   48.270221
     7                   48.270221
     8                   48.270221
     9                   19.665676
    10                   19.665676
    11                   11.607238
    12                   11.607238
    13                   11.607238
    14                    4.067958
    15                    4.067958
    16                    4.067958
    17                    2.154349
    18                    2.154349
    19                    2.154349
    20                    1.579564
    21                    1.579564
    22                    1.579564
    23                    1.236727
    24                    1.236727
    25                    1.236727
    26                    0.993344
    27                    0.993344
    28                    0.949319
    29                    0.949319
    30                    0.949319
    31                    0.945762
    32                    0.944063
    33                    0.941246
    34                    0.941246
    35                    0.940604
***                  new lower bound:     0.931668

 Result:  feasible solution of required accuracy
          best objective value:     0.940604
          guaranteed absolute accuracy:  8.94e-03
          f-radius saturation:  0.405% of R =  1.00e+08

 Optimal Hinf performance:  9.397e-01
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Display','on','RelTol',0.05`

General Options

`Display` — Display progress and generate report
`'off'` (default) | `'on'`

Display optimization progress and generate report in the command window, specified as the comma-separated pair consisting of `'Display'` and `'on'` or `'off'`. The contents of the display depend on the value of the `'Method'` option.

For `'Method' = 'RIC'`, the display shows the range of performance targets (`gamma` values) tested. For each `gamma`, the display shows:

- The smallest eigenvalues of the normalized Riccati solutions $X = X_\infty/\gamma$ and $Y = Y_\infty/\gamma$

- The spectral radius `rho(XY) = max(abs(eig(XY)))`
- A pass/fail (`p/f`) flag indicating whether that `gamma` value satisfies the conditions $X \geq 0$, $Y \geq 0$, and `rho(XY)` < 1
- The best achieved `gamma` performance value

For more information about the displayed information, see the Algorithms section of `hinfsyn`.

For `'Method' = 'LMI'`, the display shows the best achieved `gamma` value for each iteration of the optimization problem. It also displays a report of the best achieved value and other parameters of the computation.

Example: `opts = hinfsynOptions('Display','on')` creates an option set that turns the progress display on.

Method — Optimization algorithm
`'RIC'` (default) | `'LMI'`

Optimization algorithm that `hinfsyn` or `mixsyn` uses to optimize closed-loop performance, specified as the comma-separated pair consisting of `'Method'` and one of the following:

- `'RIC'` — Riccati-based algorithm. The Riccati method is fastest, but cannot handle singular problems without first adding extra disturbances and errors. This process is called *regularization*, and is performed automatically by `hinfsyn` and `mixsyn` unless you set the `'Regularize'` option to `'off'`. With regularization, this method works well for most problems.

  When `'Method' = 'RIC'`, the additional options listed under “Riccati Method Options” on page 1-0 are available.
- `'LMI'` — LMI-based algorithm. This method requires no regularization, but is computationally more intensive than the Riccati method.

  When `'Method' = 'LMI'`, the additional options listed under “LMI Method Options” on page 1-0 are available.
- `'MAXE'` — Maximum-entropy algorithm.

  When `'Method' = 'MAXE'`, the additional options listed under “Maximum-Entropy Method Options” on page 1-0 are available.

For more information about how these algorithms work, see the Algorithms section of `hinfsyn`.

Example: `opts = hinfsynOptions('Mathod','LMI')` creates an option set that specifies the LMI-based optimization algorithm.

`RelTol` — Relative accuracy on optimal H_∞ performance

0.01 (default) | positive scalar

Relative accuracy on the optimal H_∞ performance, specified as the comma-separated pair consisting of `'RelTol'` and a positive scalar value. The algorithm stops testing γ values when the relative difference between the last failing value and last passing value is less than `RelTol`.

Example: `opts = hinfsynOptions('RelTol',0.05)` creates an option set that sets the relative accuracy to 0.05.

Riccati Method Options

`AbsTol` — Absolute accuracy on optimal H_∞ performance

10^{-6} (default) | positive scalar

Absolute accuracy on the optimal H_∞ performance, specified as the comma-separated pair consisting of `'AbsTol'` and a positive scalar value.

Example: `opts = hinfsynOptions('AbsTol',1e-4)` creates an option set that sets the absolute accuracy to 0.0001.

`AutoScale` — Automatic plant scaling

`'on'` (default) | `'off'`

Automatic plant scaling, specified as the comma-separated pair consisting of `'AutoScale'` and one of the following:

- `'on'` — Automatically scales the plant states, controls, and measurements to improve numerical accuracy. `hinfsyn` always returns the controller `K` in the original unscaled coordinates.
- `'off'` — Does not change the plant scaling. Turning off scaling when you know your plant is well scaled can speed up the computation.

Example: `opts = hinfsynOptions('AutoScale','off')` creates an option set that turns off automatic scaling.

`Regularize` — Automatic regularization
`'on'` (default) | `'off'`

Automatic regularization of the plant, specified as the comma-separated pair consisting of `'Regularize'` and one of:

- `'on'` — Automatically regularizes the plant to enforce requirements on P_{12} and P_{21} (see `hinfsyn`). Regularization is a process of adding extra disturbances and errors to handle singular problems.
- `'off'` — Does not regularize the plant. Turning off regularization can speed up the computation when you know your problem is far enough from singular.

Example: `opts = hinfsynOptions('Regularize','off')` creates an option set that turns off regularization.

`LimitGain` — Limit on controller gains
`'on'` (default) | `'off'`

Limit on controller gains, specified as the comma-separated pair consisting of `'LimitGain'` and either `'on'` or `'off'`. For continuous-time plants, regularization of plant feedthrough matrices D_{12} or D_{21} (see `hinfsyn`) can result in controllers with large coefficients and fast dynamics. Use this option to automatically seek a controller with the same performance but lower gains and better conditioning.

LMI Method Options

`LimitRS` — Limit on norm of LMI solutions
0 (default) | scalar in [0,1]

Limit on norm of LMI solutions, specified as the comma-separated pair consisting of `'LimitRS'` and a scalar factor in the range [0,1]. Increase this value to slow the controller dynamics by penalizing large-norm LMI solutions. See [1].

`TolRS` — Reduced-order synthesis tolerance
0.001 (default) | positive scalar

Reduced-order synthesis tolerance, specified as the comma-separated pair consisting of `'TolRS'` and a positive scalar value. `hinfsyn` computes a reduced-order controller when `1 <= rho(R*S) <= TolRs`, where `rho(A)` is the spectral radius, `max(abs(eig(A)))`.

Maximum-Entropy Method Options

`S0` — Frequency at which to evaluate entropy

`Inf` (default) | real scalar

Frequency at which to evaluate entropy, specified as a real scalar value. For more information, see the Algorithms section of `hinfsyn`.

Output Arguments

`opts` — Options for `hinfsyn` and `mixsyn`

`hinfsyn` options object

Options for the `hinfsyn` or `mixsyn` computation, returned as an `hinfsyn` options object. Use the object as an input argument to `hinfsyn` or `mixsyn`. For example:

```
[K,CL,gamma,info] = hinfsyn(P,nmeas,ncont,opts);
```

References

[1] Gahinet, P., and P. Apkarian. "A linear matrix inequality approach to H_∞-control." *Int J. Robust and Nonlinear Control*, Vol. 4, No. 4, 1994, pp. 421–448.

See Also

`hinfsyn` | `mixsyn`

Introduced in R2018b

icomplexify

Helper function for complexify

Syntax

```
DeltaR = icomplexify(DeltaCR)
```

Description

`icomplexify` works on structures to extract a real value from a pair of related fields.

`DeltaR = icomplexify(DeltaCR)` affects field pairs of `DeltaCR` named `'foo'` and `'foo_cmpxfy'` where `'foo'` can be any field name. `DeltaR` is the same as `DeltaCR` except that the fields `'foo_cmpxfy'` are removed. `complexify`, by default, complexifies the real uncertainty with ucomplex atoms, though optionally `ultidyn` atoms can be used. If a `ucomplex` uncertainty was used to complexify the uncertain system, the real parts of `'foo_cmpxfy'` are added to the real parts of `'foo'`. If a `ultidyn` uncertainty was used to complexify the uncertain system, only the real parts of `'foo'` are returned.

See Also

`complexify` | `robstab`

Introduced in R2007a

iconnect

Create empty `iconnect` (interconnection) objects

Syntax

```
H = iconnect
```

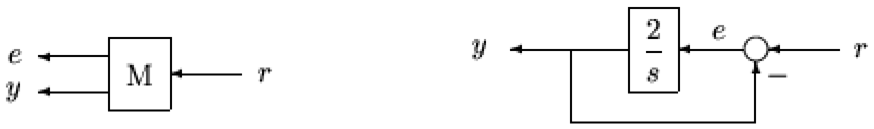
Description

Interconnection objects (class `iconnect`) are an alternative to `sysic`, and are used to build complex interconnections of uncertain matrices and systems.

An `iconnect` object has 3 fields to be set by the user, `Input`, `Output` and `Equation`. `Input` and `Output` are `icsignal` objects, while `Equation`.is a cell-array of equality constraints (using `equate`) on `icsignal` objects. Once these are specified, then the `System` property is the input/output model, implied by the constraints in `Equation`. relating the variables defined in `Input` and `Output`.

Examples

`iconnect` can be used to create the transfer matrix `M` as described in the following figure.



Create three scalar `icsignal: r, e` and `y`. Create an empty `iconnect` object, `M`. Define the output of the interconnection to be `[e; y]`, and the input to be `r`. Define two constraints among the variables: `e = r-y`, and `y = (2/s) e`. Get the transfer function representation of the relationship between the input (`r`) and the output `[e; y]`.

```
r = icsignal(1);
e = icsignal(1);
```

```
y = icsignal(1);
M = iconnect;
M.Input = r;
M.Output = [e;y];
M.Equation{1} = equate(e,r-y);
M.Equation{2} = equate(y,tf(2,[1 0])*e);
tf(M.System)
```

The transfer functions from input to outputs are

```
         s
 #1:  -----
      s + 2

         2
 #2:  -----
      s + 2
```

By not explicitly introducing `e`, this can be done more concisely with only one equality constraint.

```
r = icsignal(1);
y = icsignal(1);
N = iconnect;
N.Input = r;
N.Output = [r-y;y];
N.Equation{1} = equate(y,tf(2,[1 0])*(r-y));
tf(N.System)
```

You have created the same transfer functions from input to outputs.

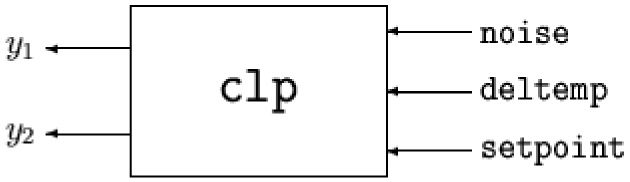
```
         s
 #1:  -----
      s + 2

         2
 #2:  -----
      s + 2
```

You can also specify uncertain, multivariable interconnections using `iconnect`. Consider two uncertain motor/generator constraints among 4 variables `[V;I;T;W]`, `V-R*I-K*W=0`, and `T=K*I`. Find the uncertain `2x2` matrix `B` so that `[V;T] = B*[W;I]`.

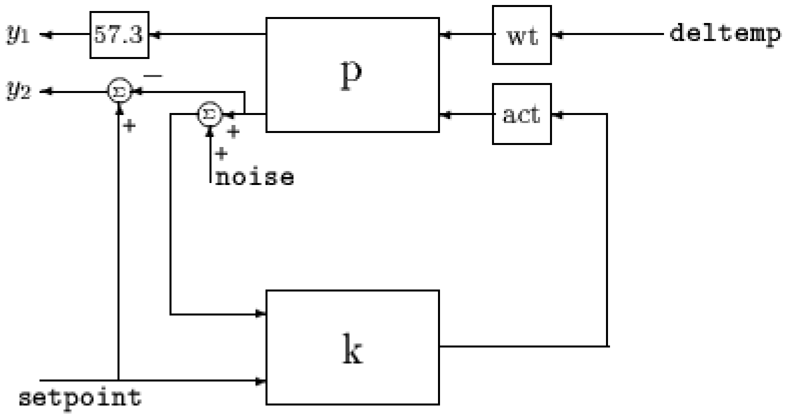
```
R = ureal('R',1,'Percentage',[-10 40]);
K = ureal('K',2e-3,'Percentage',[-30 30]);
V = icsignal(1);
```

```
I = icsignal(1);
T = icsignal(1);
W = icsignal(1);
M = iconnect;
M.Input = [W;I];
M.Output = [V;T];
M.Equation{1} = equate(V-R*I-K*W,iczero(1));
M.Equation{2} = equate(T,K*I);
B = M.System
UMAT: 2 Rows, 2 Columns
  K: real, nominal = 0.002, variability = [-30  30]%, 2 occurrences
  R: real, nominal = 1, variability = [-10  40]%, 1 occurrence
B.NominalValue
ans =
    0.0020    1.0000
         0    0.0020
```

A simple system interconnection, identical to the system illustrated in the `sysic` reference pages. Consider a three-input, two-output state-space matrix *T*,



which has internal structure



```
P = rss(3,2,2);
K = rss(1,1,2);
A = rss(1,1,1);
```

```
W = rss(1,1,1);
M = iconnect;
noise = icsignal(1);
deltemp = icsignal(1);
setpoint = icsignal(1);
yp = icsignal(2);
rad2deg = 57.3
rad2deg =
   57.3000
M.Equation{1} = equate(yp,P*[W*deltemp;A*K*[noise+yp(2);setpoint]]);
M.Input = [noise;deltemp;setpoint];
M.Output = [rad2deg*yp(1);setpoint-yp(2)];
T = M.System;
size(T)
State-space model with 2 outputs, 3 inputs, and 6 states.
```

Limitations

The syntax for `iconnect` objects and `icsignals` is very flexible. Without care, you can build inefficient (i.e., nonminimal) representations where the state dimension of the interconnection is greater than the sum of the state dimensions of the components. This is in contrast to `sysic`. In `sysic`, the syntax used to specify inputs to systems (the `input_to_ListedSubSystemName` variable) forces you to include each subsystem of the interconnection only once in the equations. Hence, interconnections formed with `sysic` are componentwise minimal. That is, the state dimension of the interconnection equals the sum of the state dimensions of the components.

Algorithms

Each equation represents an equality constraint among the variables. You choose the input and output variables, and the `imp2exp` function makes the implicit relationship between them explicit.

See Also

`icsignal` | `sysic`

Introduced before R2006a

icsignal

Create `icsignal` object of specified dimension

Syntax

```
v = icsignal(n);
```

```
v = icsignal(n,name)
```

Description

`icsignal` creates an `icsignal` object, which is a symbolic column vector. The `icsignal` object is used with `iconnect` objects to specify signal constraints described by the interconnection of components.

`v = icsignal(n)` creates an `icsignal` object of vector length `n`. The value of `n` must be a nonnegative integer. `icsignal` objects are symbolic column vectors, used in conjunction with `iconnect` (interconnection) objects to specify the signal constraints described by an interconnection of components.

`v = icsignal(n,name)` creates an `icsignal` object of dimension `n`, with internal name identifier given by the character vector `name`.

See Also

`iconnect` | `sysic`

Introduced before R2006a

imp2ss

System realization via Hankel singular value decomposition

Syntax

```
[a,b,c,d,totbnd,hsv] = imp2ss(y)
```

```
[a,b,c,d,totbnd,hsv] = imp2ss(y,ts,nu,ny,tol)
```

```
[ss,totbnd,hsv] = imp2ss(imp)
```

```
[ss,totbnd,hsv] = imp2ss(imp,tol)
```

Description

The function `imp2ss` produces an approximate state-space realization of a given impulse response

```
 imp=mksys(y,t,nu,ny,'imp');
```

using the Hankel SVD method proposed by S. Kung [2]. A continuous-time realization is computed via the inverse Tustin transform (using `bilin`) if *t* is positive; otherwise a discrete-time realization is returned. In the SISO case the variable *y* is the impulse response vector; in the MIMO case *y* is an *N*+1-column matrix containing *N* + 1 time samples of the matrix-valued impulse response H_0, ..., H_N of an `nu`-input, `ny`-output system stored row-wise:

$$y \quad = \quad [H_0(:)';H_2(:)'; \quad H_3(:)'; \quad \ldots \quad ;H_N(:)'$$

The variable *tol* bounds the H_∞ norm of the error between the approximate realization (*a, b, c, d*) and an exact realization of *y*; the order, say *n*, of the realization (*a, b, c, d*) is determined by the infinity norm error bound specified by the input variable `tol`. The inputs `ts`, `nu`, `ny`, and `tol` are optional. If omitted, they default to the values `ts = 0`, `nu = 1`, `ny =` (number of rows of *y*)/`nu`, `tol` = $0.01\bar{\sigma}_1$. The output *hsv* = $[\bar{\sigma}_1, \bar{\sigma}_2, \ldots]'$returns the singular values (arranged in descending order of magnitude) of the Hankel matrix:

$$\Gamma = \begin{bmatrix} H_1 & H_2 & H_3 & \dots & H_N \\ H_2 & H_3 & H_4 & \dots & 0 \\ H_3 & H_4 & H_5 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ H_N & 0 & \dots & \dots & 0s \end{bmatrix}$$

Denoting by G_N a high-order exact realization of y, the low-order approximate model G enjoys the H_∞ norm bound

$$\|G - G_N\|_\infty \le totbnd$$

where

$$totbnd = 2\sum_{i = n+1}^{N} \bar{\sigma}_i \,.$$

Algorithms

The realization (*a, b, c, d*) is computed using the Hankel SVD procedure proposed by Kung [2] as a method for approximately implementing the classical Hankel factorization realization algorithm. Kung's SVD realization procedure was subsequently shown to be equivalent to doing balanced truncation (`balmr`) on an exact state-space realization of the finite impulse response {*y*(1),....*y*(*N*)} [3]. The infinity norm error bound for discrete balanced truncation was later derived by Al-Saggaf and Franklin [1]. The algorithm is as follows:

1 Form the Hankel matrix Γ from the data *y*.

2 Perform SVD on the Hankel matrix

$$\Gamma = U\sum V* = [U_1 U_2]\begin{bmatrix} \sum_1 & 0 \\ 0 & \sum_2 \end{bmatrix}\begin{bmatrix} V*_1 \\ V*_2 \end{bmatrix} = U_1 \sum_1 V*_1$$

where Σ_1 has dimension $n \times n$ and the entries of Σ_2 are nearly zero. U_1 and V_1 have *ny* and *nu* columns, respectively.

3 Partition the matrices U_1 and V_1 into three matrix blocks:

$$U1 = \begin{bmatrix} U_{11} \\ U_{12} \\ U_{13} \end{bmatrix} \begin{bmatrix} V_{11} \\ V_{12} \\ V_{13} \end{bmatrix}$$

where $U_{11}, U_{13} \in C^{ny \times n}$ and $V_{11}, V_{13} \in C^{nu \times n}$.

4 A discrete state-space realization is computed as

$$A = \textstyle\sum_1^{-\frac{1}{2}} \bar{U} \textstyle\sum_1^{-\frac{1}{2}}$$

$$B = \textstyle\sum_1^{-\frac{1}{2}} V*_{11}$$

$$C = U_{11} \textstyle\sum_1^{-\frac{1}{2}}$$

$$D = H_0$$

where

$$\bar{U} = \begin{bmatrix} U_{11} \\ U_{12} \end{bmatrix}' \begin{bmatrix} U_{12} \\ U_{13} \end{bmatrix}$$

5 If the sample time t is greater than zero, then the realization is converted to continuous time via the inverse of the Tustin transform

$$s = \frac{2}{t}\frac{z-1}{z+1} \ .$$

Otherwise, this step is omitted and the discrete-time realization calculated in Step 4 is returned.

References

[1] Al-Saggaf, U.M., and G.F. Franklin, “An Error Bound for a Discrete Reduced Order Model of a Linear Multivariable System,” *IEEE Trans. on Autom. Contr.*, AC-32, 1987, p. 815-819.

[2] Kung, S.Y., "A New Identification and Model Reduction Algorithm via Singular Value Decompositions," *Proc. Twelfth Asilomar Conf. on Circuits, Systems and Computers*, November 6-8, 1978, p. 705-714.

[3] Silverman, L.M., and M. Bettayeb, "Optimal Approximation of Linear Systems," *Proc. American Control Conf.*, San Francisco, CA, 1980.

Introduced before R2006a

ispsys

True for parameter-dependent systems

Syntax

```
bool = ispsys(sys)
```

Description

`bool = ispsys(sys)` returns `1` if `sys` is a polytopic or parameter-dependent system.

See Also

`psinfo` | `psys`

Introduced before R2006a

isuncertain

Check whether argument is uncertain class type

Syntax

```
B = isuncertain(A)
```

Description

Returns `true` if input argument is uncertain, `false` otherwise. Uncertain classes are `umat`, `ufrd`, `uss`, `ureal`, `ultidyn`, `ucomplex`, `ucomplexm`, and `udyn`.

Examples

In this example, you verify the correct operation of `isuncertain` on `double`, `ureal`, `ss`, and `uss` `objects`.

```
isuncertain(rand(3,4))
ans =
     0
isuncertain(ureal('p',4))
ans =
     1
isuncertain(rss(4,3,2))
ans =
     0
isuncertain(rss(4,3,2)*[ureal('p1',4) 6;0 1])
ans =
     1
```

Limitations

`isuncertain` only checks the class of the input argument, and does not actually verify that the input argument is truly uncertain. Create a `umat` by *lifting* a constant (i.e., not-uncertain) matrix to the `umat` class.

```
A = umat([2 3;4 5;6 7]);
```

Note that although `A` is in class `umat`, it is not actually uncertain. Nevertheless, based on class, the result of `isuncertain(A)` is `true`.

```
isuncertain(A)
ans =
     1
```

The result of `simplify(A)` is a `double`, and hence not uncertain.

```
isuncertain(simplify(A))
ans =
     0
```

Introduced before R2006a

lftdata

Decompose uncertain objects into fixed certain and normalized uncertain parts

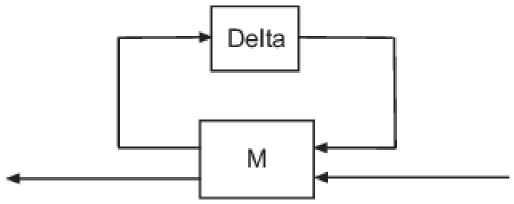
Syntax

```
[M,Delta] = lftdata(A);

[M,Delta] = lftdata(A,List);

[M,Delta,Blkstruct] = lftdata(A);

[M,Delta,Blkstruct,Normunc] = lftdata(A);
```

Description

`lftdata` decomposes an uncertain object into a fixed certain part and a normalized uncertain part. `lftdata` can also partially decompose an uncertain object into an uncertain part and a normalized uncertain part. Uncertain objects (`umat, ufrd, uss`) are represented as certain (i.e., not-uncertain) objects in feedback with block-diagonal concatenations of uncertain elements.

`[M,Delta] = lftdata(A)` separates the uncertain object `A` into a certain object `M` and a normalized uncertain matrix `Delta` such that `A` is equal to `lft(Delta,M)`, as shown below.



If `A` is a `umat`, then `M` will be `double`; if `A` is a `uss`, then `M` will be `ss`; if `A` is a `ufrd`, then `M` will be `frd`. In all cases, `Delta` is a `umat`.

`[M,Delta] = lftdata(A,List)` separates the uncertain object `A` into an uncertain object `M`, in feedback with a normalized uncertain matrix `Delta`. `List` is a cell (or char) array of names of uncertain elements of A that make up Delta. All other uncertainty in A remains in M.

`lftdata(A,fieldnames(A.Uncertainty))` is the same as `lftdata(A)`.

`[M,DELTA,BLKSTRUCT] = lftdata(A)` returns an *N*-by-1 structure array BLKSTRUCT, where BLKSTRUCT(i) describes the i-th normalized uncertain element. This uncertainty description can be passed directly to the low-level structured singular value analysis function mussv.

`[M,DELTA,BLKSTRUCT,NORMUNC] = lftdata(A)` returns the cell array NORMUNC of normalized uncertain elements. Each normalized element has 'Normalized' appended to its original name to avoid confusion. Note that `lft(blkdiag(NORMUNC{:}),M)` is equivalent to A.

Examples

Create an uncertain matrix A with 3 uncertain parameters p1, p2 and p3. You can decompose A into its certain, M, and normalized uncertain parts, Delta.

```
p1 = ureal('p1',-3,'perc',40);
p2 = ucomplex('p2',2);
A = [p1 p1+p2;1 p2];
[M,Delta] = lftdata(A);
```

You can inspect the difference between the original uncertain matrix, A, and the result formed by combining the two results from the decomposition.

```
simplify(A-lft(Delta,M))
ans =
     0     0
     0     0
M
M =
         0         0    1.0954    1.0954
         0         0         0    1.0000
    1.0954    1.0000   -3.0000   -1.0000
         0    1.0000    1.0000    2.0000
```

You can check the worst-case norm of the uncertain part using wcnorm. Compare samples of the uncertain part A with the uncertain matrix A.

```
wcn = wcnorm(Delta)
wcn =
    lbound: 1.0000
    ubound: 1.0001
usample(Delta,5)
ans(:,:,1) =
   0.8012                  0
        0           0.2499 + 0.6946i
ans(:,:,2) =
   0.4919                  0
        0           0.2863 + 0.6033i
ans(:,:,3) =
  -0.1040                  0
        0           0.7322 - 0.3752i
ans(:,:,4) =
   0.8296                  0
        0           0.6831 + 0.1124i
ans(:,:,5) =
   0.6886                  0
        0           0.0838 + 0.3562i
```

Uncertain Systems

Create an uncertain matrix A with 2 uncertain real parameters v1 and v2 and create an uncertain system G using A as the dynamic matrix and simple matrices for the input and output.

```
A = [ureal('p1',-3,'perc',40) 1;1 ureal('p2',-2)];
sys = ss(A,[1;0],[0 1],0);
sys.InputGroup.ActualIn = 1;
sys.OutputGroup.ActualOut = 1;
```

You can decompose G into a certain system, Msys, and a normalized uncertain matrix, Delta. You can see from Msys that it is certain and that the input and output groups have been adjusted.

```
[Msys,Delta] = lftdata(sys);
Msys

a =
       x1  x2
   x1  -3   1
   x2   1  -2
```

```
b =
          u1     u2     u3
   x1  1.095      0      1
   x2      0      1      0


c =
          x1     x2
   y1  1.095      0
   y2      0      1
   y3      0      1


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

Input groups:
      Name      Channels
    ActualIn       3
     p1_NC         1
     p2_NC         2

Output groups:
      Name       Channels
    ActualOut       3
      p1_NC         1
      p2_NC         2

Continuous-time model.
```

You can compute the norm on samples of the difference between the original uncertain matrix and the result formed by combining `Msys` and `Delta`.

```
norm(usample(sys-lft(Delta,Msys),'p1',4,'p2',3),'inf')
ans =
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

Partial Decomposition

Create an uncertain matrix A and derive an uncertain matrix B using an implicit-to-explicit conversion, imp2exp. Note that B has 2 uncertain parameters R and K. You can decompose B into certain, M, and normalized uncertain parts, Delta.

```
R = ureal('R',1,'Percentage',[-10 40]);
K = ureal('K',2e-3,'Percentage',[-30 30]);
A = [1 -R 0 -K;0 -K 1 0];
Yidx = [1 3];
Uidx = [4 2];
B = imp2exp(A,Yidx,Uidx);
[M,Delta] = lftdata(B);
```

The same operation can be performed by defining the uncertain parameters, K and R, to be extracted.

```
[MK,DeltaR] = lftdata(B,'R');
MK
UMAT: 3 Rows, 3 Columns
  K: real, nominal = 0.002, variability = [-30  30]%, 2 occurrences
[MR,DeltaK] = lftdata(B,'K');
MR
UMAT: 4 Rows, 4 Columns
  R: real, nominal = 1, variability = [-10  40]%, 1 occurrence

simplify(B-lft(Delta,M))
ans =
     0     0
     0     0
simplify(B-lft(DeltaR,MK))
ans =
     0     0
     0     0
simplify(B-lft(DeltaK,MR))
ans =
     0     0
     0     0
```

Sample and inspect the uncertain part as well as the difference between the original uncertain matrix and the sampled matrix. You can see the result formed by combining the two results from the decomposition.

```
[Mall,Deltaall] = lftdata(B,{'K';'R'});
simplify(Mall)-M
```

```
ans =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
```

See Also

`lft` | `ssdata`

Introduced before R2006a

lmiedit

Specify or display systems of LMIs as MATLAB expressions

Syntax

```
lmiedit
```

Description

`lmiedit` is a graphical user interface for the symbolic specification of LMI problems. Typing `lmiedit` calls up a window with two editable text areas and various buttons. To specify an LMI system,

1. Give it a name (top of the window).
2. Declare each matrix variable (name and structure) in the upper half of the window. The structure is characterized by its type (`S` for symmetric block diagonal, `R` for unstructured, and `G` for other structures) and by an additional structure matrix similar to the second input argument of `lmivar`. Please use one line per matrix variable in the text editing areas.
3. Specify the LMIs as MATLAB expressions in the lower half of the window. An LMI can stretch over several lines. However, do not specify more than one LMI per line.

Once the LMI system is fully specified, you can perform the following operations by pressing the corresponding button:

- Visualize the sequence of `lmivar`/`lmiterm` commands needed to describe this LMI system (`view commands` buttons)
- Conversely, display the symbolic expression of the LMI system produced by a particular sequence of `lmivar`/`lmiterm` commands (click the `describe...` buttons)
- Save the symbolic description of the LMI system (`save` button). This description can be reloaded later on by pressing the `load` button
- Read a sequence of `lmivar`/`lmiterm` commands from a file (`read` button). The matrix expression of the LMI system specified by these commands is then displayed by clicking on `describe the LMIs...`

- Write in a file the sequence of `lmivar`/`lmiterm` commands needed to specify a particular LMI system (`write` button)
- Generate the internal representation of the LMI system by pressing `create`. The result is written in a MATLAB variable with the same name as the LMI system

Tips

Editable text areas have built-in scrolling capabilities. To activate the scroll mode, click in the text area, maintain the mouse button down, and move the mouse up or down. The scroll mode is only active when all visible lines have been used.

See Also

`lmiinfo` | `lmiterm` | `lmivar` | `newlmi`

Introduced before R2006a

lmiinfo

Information about variables and term content of LMIs

Syntax

```
lmiinfo
```

Description

`lmiinfo` provides qualitative information about the system of LMIs `lmisys`. This includes the type and structure of the matrix variables, the number of diagonal blocks in the inner factors, and the term content of each block.

`lmiinfo` is an interactive facility where the user seeks specific pieces of information. General LMIs are displayed as

```
N' * L(x) * N < M' * R(x) * M
```

where `N,M` denote the outer factors and `L,R` the left and right inner factors. If the outer factors are missing, the LMI is simply written as

```
L(x) < R(x)
```

If its right side is zero, it is displayed as

```
N' * L(x) * N < 0
```

Information on the block structure and term content of `L(x)` and `R(x)` is also available. The term content of a block is symbolically displayed as

```
C1 + A1*X2*B1 + B1'*X2*A1' + a2*X1 + x3*Q1
```

with the following conventions:

- `X1, X2, x3` denote the problem variables. Upper-case `X` indicates matrix variables while lower-case `x` indicates scalar variables. The labels 1,2,3 refer to the first, second, and third matrix variable in the order of declaration.

- `Cj` refers to constant terms. Special cases are `I` and `–I` (`I` = identity matrix).
- `Aj, Bj` denote the left and right coefficients of variable terms. Lower-case letters such as `a2` indicate a scalar coefficient.
- `Qj` is used exclusively with scalar variables as in `x3*Q1`.

The index `j` in `Aj, Bj, Cj, Qj` is a dummy label. Hence `C1` may appear in several blocks or several LMIs without implying any connection between the corresponding constant terms. Exceptions to this rule are the notations `A1*X2*A1'` and `A1*X2*B1 + B1'*X2'*A1'` which indicate symmetric terms and symmetric pairs in diagonal blocks.

Examples

Consider the LMI

$$0\begin{pmatrix} -2X + A^TYB + B^TY^TA + I & XC \\ C^TX & -zI \end{pmatrix}$$

where the matrix variables are X of Type 1, Y of Type 2, and z scalar. If this LMI is described in `lmis`, information about X and the LMI block structure can be obtained as follows:

```
lmiinfo(lmis)

                  LMI ORACLE
                ------- 

This is a system of 1 LMI with 3 variable matrices

Do you want information on
    (v) matrix variables     (l) LMIs     (q) quit

?> v

Which variable matrix (enter its index k between 1 and 3) ? 1
    X1 is a 2x2 symmetric block diagonal matrix
      its (1,1)-block is a full block of size 2

                      -------

This is a system of 1 LMI with 3 variable matrices
```

```
Do you want information on
    (v) matrix variables     (l) LMIs     (q) quit

?> l

Which LMI (enter its number k between 1 and 1) ? 1

    This LMI is of the form
            0 < R(x)
where the inner factor(s) has 2 diagonal block(s)


Do you want info on the right inner factor ?

    (w) whole factor     (b) only one block
    (o) other LMI        (t) back to top level

?> w

Info about the right inner factor

    block (1,1) : I + a1*X1 + A2*X2*B2 + B2'*X2'*A2'

    block (2,1) : A3*X1

    block (2,2) : x3*A4

    (w) whole factor     (b) only one block
    (o) other LMI        (t) back to top level

                    -------

This is a system of 1 LMI with 3 variable matrices

Do you want information on
    (v) matrix variables     (l) LMIs     (q) quit

?> q

It has been a pleasure serving you!
```

Note that the prompt symbol is ?> and that answers are either indices or letters. All blocks can be displayed at once with option `(w)`, or you can prompt for specific blocks with option `(b)`.

Tips

`lmiinfo` does not provide access to the numerical value of LMI coefficients.

See Also

`decinfo` | `decnbr` | `lminbr` | `matnbr`

Introduced before R2006a

lminbr

Return number of LMIs in LMI system

Syntax

```
k = lminbr(lmisys)
```

Description

`lminbr` returns the number k of linear matrix inequalities in the LMI problem described in `lmisys`.

See Also

`lmiinfo` | `matnbr`

Introduced before R2006a

lmireg

Specify LMI regions for pole placement

Syntax

```
region = lmireg

region = lmireg(reg1,reg2,...)
```

Description

`lmireg` is an interactive facility to specify the LMI regions involved in multi-objective H_∞ synthesis with pole placement constraints (see `msfsyn` and `h2hinfsyn`). An LMI region is any convex subset D of the complex plane that can be characterized by an LMI in z and z^{-}, i.e.,

$$D = \left\{ z \in C : L + Mz + M^T \bar{z} < 0 \right\}$$

for some fixed real matrices M and $L = L^T$. This class of regions encompasses half planes, strips, conic sectors, disks, ellipses, and any intersection of the above.

Calling `lmireg` without argument starts an interactive query/answer session where you can specify the region of your choice. The matrix `region` = [*L, M*] is returned upon termination. This matrix description of the LMI region can be passed directly to `msfsyn` for synthesis purposes.

The function `lmireg` can also be used to intersect previously defined LMI regions `reg1, reg2,....` The output region is then the [*L, M*] description of the intersection of these regions.

Examples

Define LMI Region for Pole Placement

For LMI controller synthesis with functions like `msfsyn` and `h2hinfsyn`, you can restrict the eigenvalues of the closed-loop system to an LMI region. The region is specified as a matrix of the form `[L M]`. In this example, use `lmireg` interactively to generate a matrix you can use to restrict the poles of the closed-loop system to Re(z) < –1.

Start the interactive process.

```
region = lmireg
```

```
Select a region among the following:

h)   Half-plane
d)   Disk
c)   Conic sector
e)   Ellipsoid
p)   Parabola
s)   Horizontal strip
m)   Matrix description of the LMI region
q)   Quit
choice:
```

The software prompts you to select the geometry of the region. For this example, enter `h` to specify a half-plane region. The software now prompts you to specify left half-plane (Re(z) less than some value) or right half-plane (Re(z) greater than some value).

```
Orientation (x < x0 -> l , x > x0 -> r):
```

Enter `l` to specify a left half-plane. The software prompts you to specify a value for `x0`.

```
Specify x0:
```

Enter `-1`. You have now completely specified the restriction Re(z) < –1. If you want to specify additional regional constraints on the pole locations, you can select another geometry now and follow the prompts. For this example, enter `q` to generate the LMI region matrix corresponding to Re(z) < –1.

```
region =

   2.0000 + 1.0000i   1.0000 + 0.0000i
```

You can now use `region` with `msfsyn` or `h2hinfsyn`.

See Also

`h2hinfsyn` | `msfsyn`

Introduced before R2006a

lmiterm

Specify term content of LMIs

Syntax

```
lmiterm(termID,A,B,flag)
```

Description

`lmiterm` specifies the term content of an LMI one term at a time. Recall that *LMI term* refers to the elementary additive terms involved in the block-matrix expression of the LMI. Before using `lmiterm`, the LMI description must be initialized with `setlmis` and the matrix variables must be declared with `lmivar`. Each `lmiterm` command adds one extra term to the LMI system currently described.

LMI terms are one of the following entities:

- outer factors
- constant terms (fixed matrices)
- variable terms AXB or AX^TB where X is a matrix variable and A and B are given matrices called the term coefficients.

When describing an LMI with several blocks, remember to specify **only the terms in the blocks on or below the diagonal** (or equivalently, only the terms in blocks on or above the diagonal). For instance, specify the blocks (1,1), (2,1), and (2,2) in a two-block LMI.

In the calling of `lmiterm`, `termID` is a four-entry vector of integers specifying the term location and the matrix variable involved.

$$\text{termID (1)} = \begin{cases} +\text{p} \\ -\text{p} \end{cases}$$

where positive p is for terms on the *left-side* of the *p*-th LMI and negative p is for terms on the *right-side* of the *p*-th LMI.

Recall that, by convention, the left side always refers to the smaller side of the LMI. The index `p` is relative to the order of declaration and corresponds to the identifier returned by `newlmi`.

$$\text{termID}(2\!:\!3) = \begin{cases} [0,0] \text{ for outer factors} \\ [i, j] \text{ for terms in the } (i, j)\text{-th block of the left or right inner factor} \end{cases}$$

$$\text{termID}(4) = \begin{cases} 0 \text{ for outer factors} \\ x \text{ for variable terms } AXB \\ -x \text{ for variable terms } AX^TB \end{cases}$$

where `x` is the identifier of the matrix variable `X` as returned by `lmivar`.

The arguments `A` and `B` contain the numerical data and are set according to:

Type of Term	**A**	**B**
outer factor N	matrix value of N	omit
constant term C	matrix value of C	omit
variable term AXB or AX^TB	matrix value of A (1 if A is absent)	matrix value of B (1 if B is absent)

Note that identity outer factors and zero constant terms need not be specified.

The extra argument `flag` is optional and concerns only conjugated expressions of the form

$$(AXB) + (AXB)^T = AXB + B^TX^TA^T$$

in *diagonal blocks*. Setting `flag = 's'` allows you to specify such expressions with a single `lmiterm` command. For instance,

```
lmiterm([1 1 1 X],A,1,'s')
```

adds the symmetrized expression $AX + X^TA^T$ to the (1,1) block of the first LMI and summarizes the two commands

```
lmiterm([1 1 1 X],A,1)
lmiterm([1 1 1 -X],1,A')
```

Aside from being convenient, this shortcut also results in a more efficient representation of the LMI.

Examples

Specify LMI Terms

Consider the LMI

$$\begin{pmatrix} 2AX_2A^T - x_3E + DD^T & B^TX_1 \\ X_1^TB & -I \end{pmatrix} < M^T \begin{pmatrix} CX_1C^T + CX_1^TC^T & 0 \\ 0 & -fX_2 \end{pmatrix} M$$

where X_1, X_2 are matrix variables of Types 2 and 1, respectively, and x_3 is a scalar variable (Type 1).

After you initialize the LMI description using `setlmis` and declare the matrix variables using `lmivar`, specify the terms on the left side of this LMI.

```
lmiterm([1 1 1 X2],2*A,A')  % 2*A*X2*A'
lmiterm([1 1 1 x3],-1,E)    % -x3*E
lmiterm([1 1 1 0],D*D')     % D*D'
lmiterm([1 2 1 -X1],1,B)    % X1'*B
lmiterm([1 2 2 0],-1)       % -I
```

Here `X1`, `X2`, and `x3` are the variable identifiers returned by `lmivar` when you declare the variables.

Similarly, specify the term content of the right side.

```
lmiterm([-1 0 0 0],M)         % outer factor M
lmiterm([-1 1 1 X1],C,C','s') % C*X1*C'+C*X1'*C'
lmiterm([-1 2 2 X2],-f,1)     % -f*X2
```

Note that $CX_1C^T + CX_1{}^TC^T$ is specified by a single `lmiterm` command with the flag `'s'` to ensure proper symmetrization.

See Also

`getlmis` | `lmiedit` | `lmivar` | `newlmi` | `setlmis`

Introduced before R2006a

lmivar

Specify matrix variables in LMI problem

Syntax

```
X = lmivar(type,struct)
```

```
[X,n,sX] = lmivar(type,struct)
```

Description

`lmivar` defines a new matrix variable X in the LMI system currently described. The optional output `X` is an identifier that can be used for subsequent reference to this new variable.

The first argument `type` selects among available types of variables and the second argument `struct` gives further information on the structure of X depending on its type. Available variable types include:

type=1: Symmetric matrices with a block-diagonal structure. Each diagonal block is either full (arbitrary symmetric matrix), scalar (a multiple of the identity matrix), or identically zero.

If X has R diagonal blocks, `struct` is an R-by-2 matrix where

- `struct(r,1)` is the size of the r-th block
- `struct(r,2)` is the type of the r-th block (1 for full, 0 for scalar, –1 for zero block).

type=2: Full m-by-n rectangular matrix. Set `struct = [m,n]` in this case.

type=3: Other structures. With Type 3, each entry of X is specified as zero or $\pm x$ where x_n is the n-th decision variable.

Accordingly, `struct` is a matrix of the same dimensions as X such that

- `struct(i,j)=0` if $X(i, j)$ is a hard zero

- `struct(i,j)=n` if $X(i, j) = x_n$
- `struct(i,j)=–n` if $X(i, j) = -x_n$

Sophisticated matrix variable structures can be defined with Type 3. To specify a variable X of Type 3, first identify how many *free independent entries* are involved in X. These constitute the set of decision variables associated with X. If the problem already involves n decision variables, label the new free variables as $x_{n+1}, \ldots, x_{n+p}$. The structure of X is then defined in terms of $x_{n+1}, \ldots, x_{n+p}$ as indicated above. To help specify matrix variables of Type 3, `lmivar` optionally returns two extra outputs: (1) the total number n of scalar decision variables used so far and (2) a matrix `sX` showing the entry-wise dependence of X on the decision variables $x_1, \ldots, x_n$.

Examples

Type 1 and Type 2 Matrix Variables

Consider an LMI system with three matrix variables X_1, X_2, and X_3 such that

- X_1 is a 3-by-3 symmetric matrix (unstructured),
- X_2 is a 2-by-4 rectangular matrix (unstructured),
- $X_3 =$

$$\begin{pmatrix} \Delta & 0 & 0 \\ 0 & \delta_1 & 0 \\ 0 & 0 & \delta_2 I_2 \end{pmatrix},$$

where Δ is an arbitrary 5-by-5 symmetric matrix, δ_1 and δ_2 are scalars, and I_2 denotes the identity matrix of size 2.

Define these three variables using `lmivar`.

```
setlmis([])
X1 = lmivar(1,[3 1]);          % Type 1
X2 = lmivar(2,[2 4]);         % Type 2 of dimension 2-by-4
X3 = lmivar(1,[5 1;1 0;2 0]);  % Type 1
```

The last command defines X_3 as a variable of Type 1 with one full block of size 5 and two scalar blocks of sizes 1 and 2, respectively.

Type 3 Matrix Variables

Combined with the extra outputs `n` and `sX` of `lmivar`, Type 3 allows you to specify fairly complex matrix variable structures. For instance, consider a matrix variable X with structure given by:

$$X = \begin{pmatrix} X_1 & 0 \\ 0 & X_2 \end{pmatrix}$$

where X_1 and X_2 are 2-by-3 and 3-by-2 rectangular matrices, respectively. Specify this structure as follows.

Define the rectangular variables X_1 and X_2.

```
setlmis([])
[X1,n,sX1] = lmivar(2,[2 3]);
[X2,n,sX2] = lmivar(2,[3 2]);
```

The outputs `sX1` and `sX2` give the decision variable content of X_1 and X_2.

```
sX1
```

```
sX1 = 2×3

     1     2     3
     4     5     6
```

```
sX2
```

```
sX2 = 3×2

     7     8
     9    10
    11    12
```

For instance, `sX2(1,1) = 7` means that the (1,1) entry of X_2 is the seventh decision variable.

Next, use Type 3 to specify the matrix variable X, and define its structure in terms of the structures of X_1 and X_2.

```
[X,n,sX] = lmivar(3,[sX1,zeros(2);zeros(3),sX2]);
```

Confirm that the resulting X has the desired structure.

```
sX
```

```
sX = 5×5

     1     2     3     0     0
     4     5     6     0     0
     0     0     0     7     8
     0     0     0     9    10
     0     0     0    11    12
```

See Also

delmvar | getlmis | lmiedit | lmiterm | setlmis | setmvar | skewdec

Introduced before R2006a

lncf

Left normalized coprime factorization

Syntax

```
fact = lncf(sys)
[fact,Ml,Nl] = lncf(sys)
```

Description

`fact = lncf(sys)` computes the left normalized coprime factorization of the dynamic system model `sys`. The factorization is given by:

$$sys = M_l^{-1} N_l, \quad M_l M_l^* + N_l N_l^* = I\,.$$

Here, M_l^* denotes the conjugate of M_l (see `ctranspose`). . The returned model `fact` is a minimal state-space realization of the stable system $[M_l,N_l]$. This factorization is used in other normalized coprime factor computations such as model reduction (`ncfmr`) and controller synthesis (`ncfsyn`).

`[fact,Ml,Nl] = lncf(sys)` also returns the coprime factors M_l and N_l.

Examples

Left Normalized Coprime Factorization of SISO System

Compute the left normalized coprime factorization of a SISO system.

```
sys = zpk([1 -1+2i -1-2i],[-1 2+1i 2-1i],1);
[fact,Ml,Nl] = lncf(sys);
```

Examine the original system and its factors.

```
sys
```

```
sys =

  (s-1) (s^2 + 2s + 5)
  --------------------
  (s+1) (s^2 - 4s + 5)

Continuous-time zero/pole/gain model.
```

```
zpk(Ml)
```

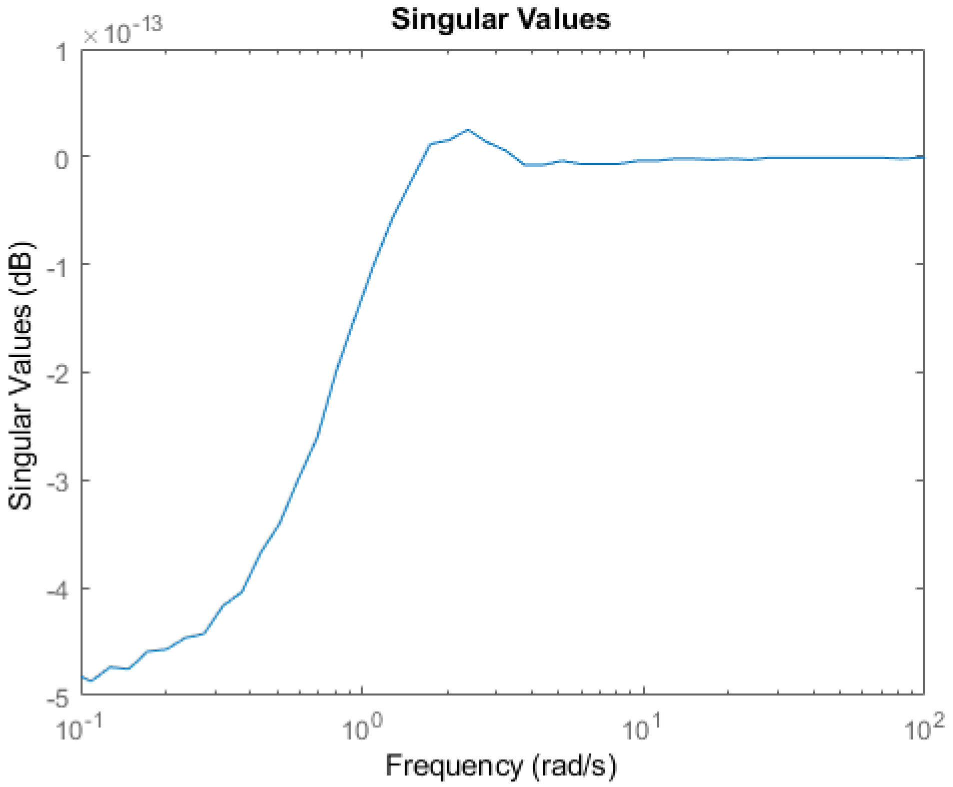
```
ans =

  0.70711 (s+1) (s^2 - 4s + 5)
  ----------------------------
    (s+1) (s^2 + 3.162s + 5)

Continuous-time zero/pole/gain model.
```

```
zpk(Nl)
```

```
ans =

  0.70711 (s-1) (s^2 + 2s + 5)
  ----------------------------
    (s+1) (s^2 + 3.162s + 5)

Continuous-time zero/pole/gain model.
```

The numerators of the factors `Ml` and `Nl` are the denominator and numerator of `sys`, respectively. Thus, `sys = Ml\Nl`. `lncf` chooses the denominators of the factors such that the system $[M_l(j\omega), N_l(j\omega)]$ is a unit vector at all frequencies. To confirm that property of the factorization, examine the singular values of `fact`, which is a stable minimal realization of $[M_l(j\omega), N_l(j\omega)]$.

```
sigma(fact)
```



Within a small numerical error, the singular value of `fact` is 1 (0 dB) at all frequencies.

Left Normalized Coprime Factorization of MIMO System

Compute the left normalized coprime factorization of a state-space model that has two outputs, two inputs, and three states.

```
rng(0); % for reproducibility
sys = rss(3,2,2);
[fact,Ml,Nl] = lncf(sys);
```

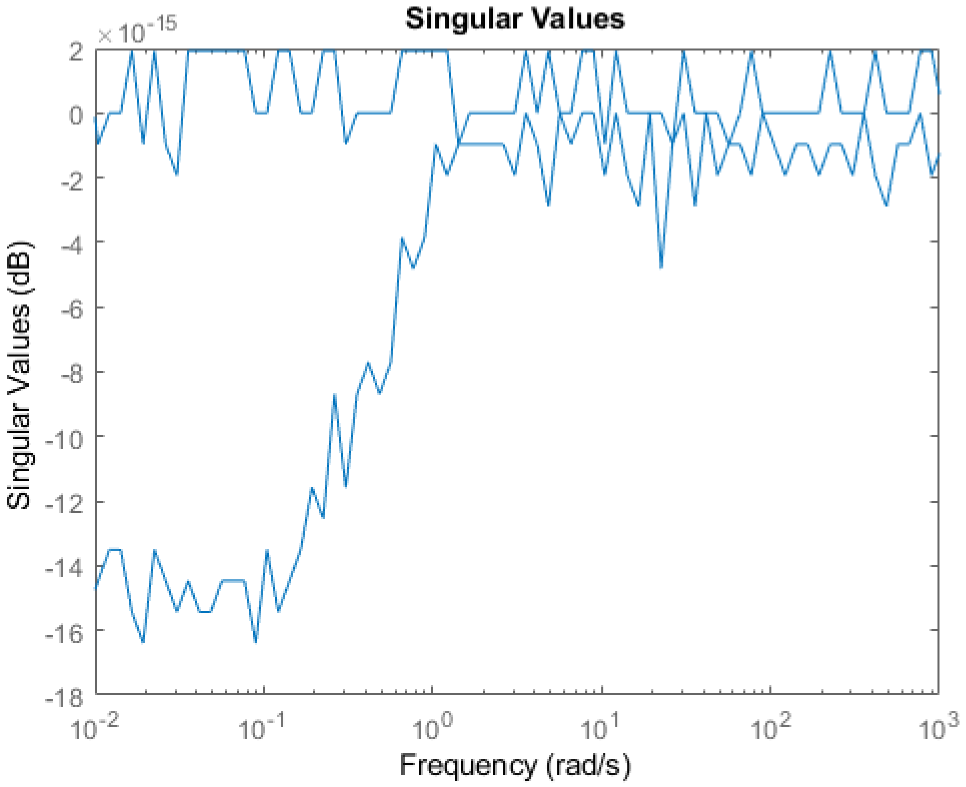
`fact` is a stable minimal realization of the factorization given by `[Ml,Nl]`.

```
isstable(fact)
```

```
ans = logical
   1
```

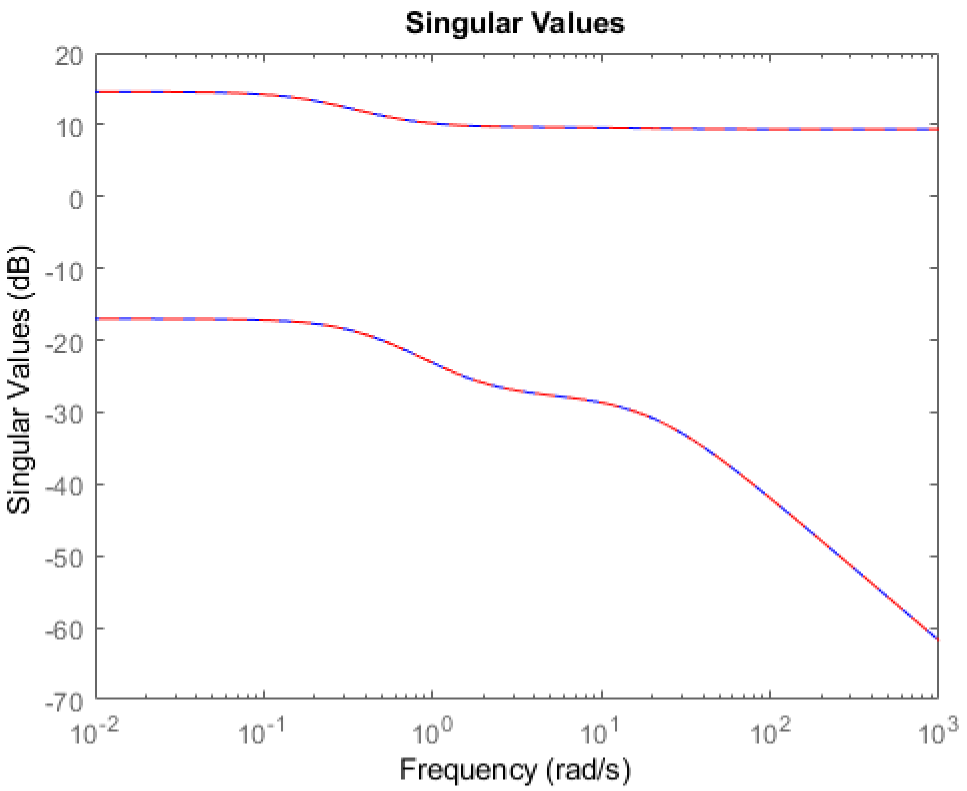
Another property of `fact` is that its frequency response $F(j\omega)$ is an orthogonal matrix at all frequencies ($F(j\omega)'F(j\omega) = I$). Confirm this property by examining the singular values of `fact`. Within a small numerical error, the singular values are 1 (0 dB) at all frequencies.

```
sigma(fact)
```



Confirm that the factors satisfy `sys = Ml\Nl` by examining the singular values of both.

```
sigma(sys,'b-',Ml\Nl,'r--')
```



Input Arguments

sys — Input system
dynamic system model

Input system to factorize, specified as a dynamic system model such as a state-space (`ss`) model. If `sys` is a generalized state-space model with uncertain or tunable control design blocks, then the function uses the nominal or current value of those elements. `sys` cannot be an `frd` model or a model with time delays.

Output Arguments

`fact` — Minimal realization of `[Ml,Nl]`
`ss` model

Minimal realization of `[Ml,Nl]`, returned as a state-space model. `fact` is stable and its frequency response is an orthogonal matrix at all frequencies. If `sys` has `p` outputs and `m` inputs, then `fact` has `p` outputs and `m+p` inputs. `fact` has the same number of states as `sys`.

`Ml,Nl` — Left coprime factors
`ss` models

Left coprime factors of `sys`, returned as state-space models. If `sys` has `p` outputs and `m` inputs, then:

- `Ml` has `p` outputs and `p` inputs.
- `Nl` has `p` outputs and `m` inputs.

Both factors have the same number of states as `sys` and the same `A` and `C` matrices as `fact`.

Tips

- `fact` is a minimal realization of `[Ml,Nl]`. If you need to use `[Ml,Nl]` or `[Ml,Nl]'` in a computation, it is better to use `fact` than to concatenate the factors yourself. Such manual concatenation results in extra (nonminimal) states, which can lead to decreased numerical accuracy.

See Also

`ncfmr` | `ncfsyn` | `rncf`

Introduced in R2019a

loopmargin

(Not recommended) Stability margin analysis of LTI and Simulink feedback loops

Note `loopmargin` is not recommended. Use `diskmargin` or `allmargin` instead. For more information, see "Compatibility Considerations".

Syntax

```
[cm,dm,mm] = loopmargin(L)
[m1,...,mn] = loopmargin(L,MFLAG)
[cmi,dmi,mmi,cmo,dmo,mmo,mmio] = loopmargin(P,C)
[m1,...,mn] = loopmargin(P,C,MFLAG)
[cm,dm,mm] = loopmargin(Model,Blocks,Ports)
[cm,dm,mm,info] = loopmargin(Model,Blocks,Ports,OP)
[m1,...,mn,info] = loopmargin(Model,Blocks,Ports,MFLAG)
[m1,...,mn,info] = loopmargin(Model,Blocks,Ports,OP,MFLAG)
```

Description

`[cm,dm,mm] = loopmargin(L)` analyzes the multivariable feedback loop consisting of the loop transfer matrix `L` (size *N*-by-*N*) in negative feedback with an *N*-by-*N* identity matrix.

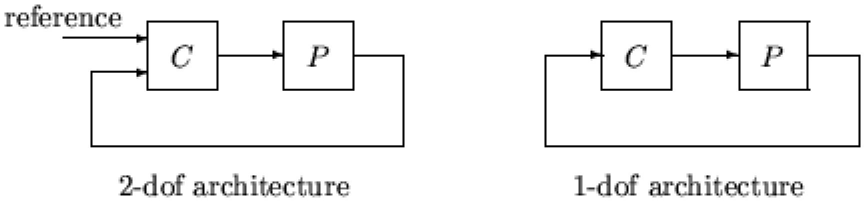
`cm`, or classical gain and phase margins, is an *N*-by-1 structure corresponding to loop-at-a-time gain and phase margins for each channel. `L` is an LTI model. Use `-L` to specify positive feedback.

`dm` is an *N*-by-1 structure corresponding to loop-at-a-time disk gain and phase margins for each channel. The disk margin for the i-th feedback channel defines a circular region centered on the negative real axis at the average GainMargin (`GM`), e.g. , ($\mathtt{GM}_{low}$+$\mathtt{GM}_{high}$)/2, such that `L(i,i)` does not enter that region. Gain and phase disk margin bounds are derived from the radius of the circle, calculated based on the balanced sensitivity function.

`mm`, the multiloop disk margin, is a structure. `mm` describes how much independent and concurrent gain and phase variation can occur independently in each feedback channel while maintaining stability of the closed-loop system. Note that mm is a single structure, independent of the number of channels. This is because variations in all channels are considered simultaneously. As in the case for disk margin, the guaranteed bounds are calculated based on a balanced sensitivity function.

[m1,...,mn] = loopmargin(L,MFLAG) returns a subset of the margins, specified by the character vector MFLAG. This optional argument may be any combination, in any order, of the 3 characters 'c', 'd' and 'm'. For example, [m1,m2] = loopmargin(L,'m,c') returns the multi-loop disk margin ('m') in m1, and the classical margins ('c') in m2. Use 'd' to specify the disk margin.

[cmi,dmi,mmi,cmo,dmo,mmo,mmio] = loopmargin(P,C) analyzes the multivariable feedback loop consisting of the controller C in negative feedback with the plant, P. C should only be the compensator in the feedback path, without reference channels, if it is a 2-dof architecture. That is, if the closed-loop system has a 2-dof architecture the reference channel of the controller should be eliminated, resulting in a 1-dof architecture, as shown.



cmi,dmi and mmi structures correspond to the classical loop-at-a-time gain and phase margins, disk margins and multiloop channel margins at the plant input respectively. The structures cmo, dmo and mmo have the same fields as described for cmi, dmi and mmi though they correspond to the plant outputs. mmio, or multi-input/multi-output margins, is a structure corresponding to simultaneous, independent, variations in all the individual input and output channels of the feedback loops. mmio has the same fields as mmi and mmo.

[m1,...,mn] = loopmargin(P,C,MFLAG) returns a subset of the margins, specified by MFLAG. This optional argument may be any combination, in any order, of the 7 character pairs 'ci', 'di', 'mi', 'co', 'do, 'mo', and 'mm'. For example,[m1,m2,m3] = loopmargin(P,C,'mo,ci,mm') returns the multi-loop disk margin at the plant output ('mo') in m1, the classical margins at the plant input ('ci') in m2, and the multi-loop disk margins for simultaneous, independent variations in all input and output channels ('mm') in m3.

Usage with Simulink

`[cm,dm,mm] = loopmargin(Model,Blocks,Ports)` does a multi-loop stability margin analysis using Simulink Control Design software. `Model` specifies the name of the Simulink diagram for analysis. The margin analysis points are defined at the output ports (`Ports`) of blocks (`Blocks`) within the model. `Blocks` is a cell array of full block path names and `Ports` is a vector of the same dimension as `Blocks`. If all `Blocks` have a single output port, then `Ports` would be a vector of ones with the same length as `Blocks`.

Three types of stability margins are computed: loop-at-a-time classical gain and phase margins (`cm`), loop-at-a-time disk margins (`dm`) and a multi-loop disk margin (`mm`).

`[cm,dm,mm] = loopmargin(Model,Blocks,Ports,OP)` uses the operating point object `OP` to create linearized systems from the Simulink `Model`.

`[cm,dm,mm,info] = loopmargin(Model,Blocks,Ports,OP)` returns `info` in addition to the margins. `info` is a structure with fields `OperatingPoint`, `LinearizationIO` and `SignalNames` corresponding to the analysis.

`[m1,...,mn,info] = loopmargin(Model,Blocks,Ports,MFLAG)` and `[m1,...,mn,info] = loopmargin(Model,Blocks,Ports,OP,MFLAG)` return a subset of the margins, specified by the character vector `MFLAG`. This optional argument may be any combination, in any order, of the 3 characters `'c'`, `'d'` and `'m'`. For example, `[m1,m2] = loopmargin(Model,Blocks,Ports,'m,c')` returns the multi-loop disk margin (`'m'`) in `m1`, and the classical margins (`'c'`) in `m2`. Use `'d'` to specify the disk margin.

Basic Syntax

`[cm,dm,mm] = loopmargin(L)` `cm` is calculated using the `allmargin` command and has the same fields as `allmargin`. The output `cm` is an N-by-1 structure of classical gain and phase margins for each feedback channel with all other loops closed. `cm` has the following fields:

Field	Description
`GMFrequency`	All –180 deg crossover frequencies (in radians-per-second)
`GainMargin`	Corresponding gain margins (`GM = 1/L` where `L` is the gain at crossover)

Field	Description
`PhaseMargin`	Corresponding phase margins (in degrees)
`PMFrequency`	All 0 dB crossover frequencies (in radians-per-second)
`DelayMargin`	Delay margins (in seconds for continuous-time systems, and multiples of the sample time for discrete-time systems)
`Stable`	1 if nominal closed loop is stable, 0 otherwise. If `L` is a `frd` or `ufrd` object, the `Stable` flag is set to `NaN`.

`dm`, or Disk Margin, is an N-by-1 structure of disk margins for each feedback channel with all other loops closed. `dm` has the following fields:

Field	Description
`GainMargin`	Smallest gain variation (`GM`) such that a disk centered at the point –(`GM(1) + GM(2)`)/2 just touches the Nyquist plot of the loop transfer function.
`PhaseMargin`	Smallest phase variation, in degrees, corresponding to the disk described in the `GainMargin` field.
`Frequency`	Frequency with the weakest disk margin, in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `L`. For `frd` models, `loopmargin` computes margins at all the frequency points in the model, and returns the frequency with the weakest margin of these values.

`mm` is a structure with the following fields.

Field	Description
`GainMargin`	Guaranteed bound on simultaneous, independent, gain variations allowed in all plant channels.
`PhaseMargin`	Guaranteed bound on simultaneous, independent, phase variations allowed in all plant channels (degrees).

Field	Description
`Frequency`	Frequency with the weakest disk margin, in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `L`. For `frd` models, `loopmargin` computes margins at all the frequency points in the model, and returns the frequency with the weakest margin of these values.

Relationship Between Disk Margin and Gain and Phase Margins

The disk margin is based on a multiplicative uncertainty model in which the loop gain *L* of each loop channel becomes

$$L \to L\frac{1 + \Delta/2}{1 - \Delta/2}, \quad |\Delta| < \alpha\,.$$

where Δ is complex. The uncertainty size α is the disk margin. The uncertain quantity (1 + Δ)/(1 – Δ) has a gain component and a phase component. Thus, enforcing a disk margin α also enforces minimum gain and phase margins given by

$$GM = \frac{1 + \alpha}{1 - \alpha},$$
$$PM = 2\arctan(\alpha),$$

with *GM* in absolute units and *PM* in degrees. The gain and phase margins are therefore related by

$$GM = \frac{1 + \tan(PM/2)}{1 - \tan(PM/2)}\,.$$

When you specify independent gain and phase margins for tuning, the software chooses the smallest α that enforces both values, which is

$$\alpha = \max\left[\frac{GM - 1}{GM + 1},\ \tan(PM/2)\right].$$

Note that *GM* and *PM* are not the same as the classical gain and phase margins. Rather, they provide stronger guarantees of stability, because both of the following can occur at the same time without loss of stability:

- The loop gain can increase or decrease by a factor of *GM*, and
- The loop phase can increase or decrease by *PM* degrees.

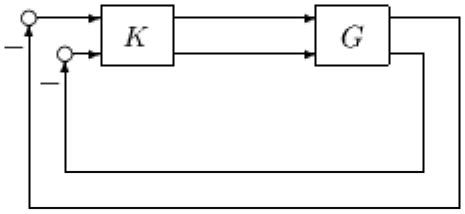
By contrast, the classical gain and phase margins consider only gain variations or phase variations at a single frequency, the crossover frequency.

Examples

MIMO Loop-at-a-Time Margins

This example shows how to compute loop-at-a-time margins (gain, phase, and/or distance to –1), and also illustrates that such margins can be inaccurate measures of multivariable robustness margins. Margins of individual loops can be very sensitive to small perturbations within other loops.

Consider the nominal closed-loop system of the following illustration.



G and K are 2-by-2 (MIMO) systems, given by:

$$G = \frac{1}{s^2+\alpha^2}\begin{bmatrix} s^2-\alpha^2 & \alpha(s+1) \\ -\alpha(s+1) & s^2-\alpha^2 \end{bmatrix},\ K = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}.$$

Set $\alpha = 10$, construct `G` in state-space form, and compute the loop margins.

```
a = [0 10;-10 0];
b = eye(2);
c = [1 8;-10 1];
d = zeros(2,2);
G = ss(a,b,c,d);
K = [1 -2;0 1];
[cmi,dmi,mmi,cmo,dmo,mmo,mmio] = loopmargin(G,K);
```

First consider the margins at the input to the plant. The first input channel has infinite gain margin and 90 degrees of phase margin based on the results from the `loopmargin` command, `cmi(1)`.

```
cmi(1)
```

```
ans = struct with fields:
     GainMargin: [1x0 double]
    GMFrequency: [1x0 double]
    PhaseMargin: 90
    PMFrequency: 21
    DelayMargin: 0.0748
    DMFrequency: 21
         Stable: 1
```

The disk margin analysis, `dmi`, of the first channel provides similar results.

```
dmi(1)
```

```
ans = struct with fields:
     GainMargin: [0 Inf]
    PhaseMargin: [-90 90]
      Frequency: 0
```

The second input channel has a gain margin of 2.105 and infinite phase margin based on the single-loop analysis, `cmi(2)`.

```
cmi(2)
```

```
ans = struct with fields:
     GainMargin: 2.1053
    GMFrequency: 0
    PhaseMargin: [1x0 double]
    PMFrequency: [1x0 double]
    DelayMargin: [1x0 double]
    DMFrequency: [1x0 double]
         Stable: 1
```

The disk margin analysis, `dmi(2)`, which allows for simultaneous gain and phase variations a loop-at-a-time results in maximum gain margin variations of 0.475 and 2.105 and phase margin variations of +/- 39.18 degs.

```
dmi(2)
```

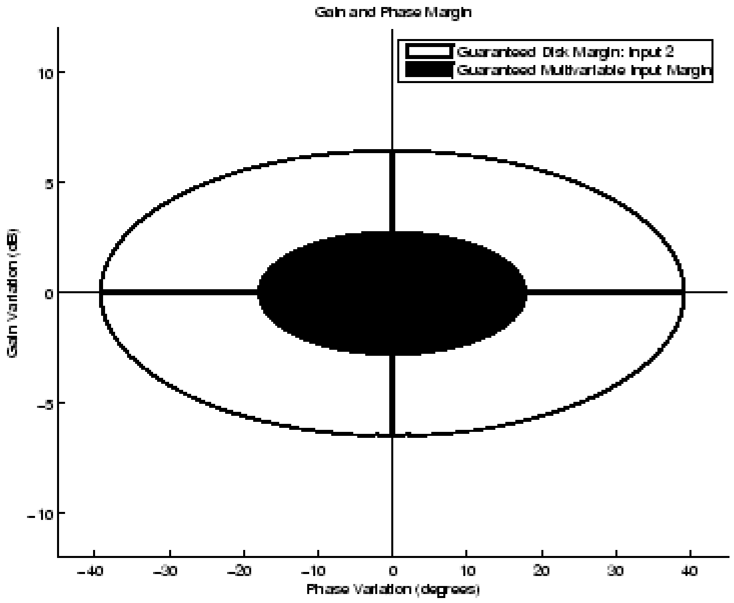
```
ans = struct with fields:
     GainMargin: [0.4750 2.1053]
    PhaseMargin: [-39.1846 39.1846]
      Frequency: 0
```

The multiple margin analysis of the plant inputs corresponds to allowing simultaneous, independent gain and phase margin variations in each channel. Allowing independent variation of the input channels further reduces the tolerance of the closed-loop system to variations at the input to the plant. The multivariable margin analysis, `mmi`, leads to a maximum allowable gain margin variation of 0.728 and 1.373 and phase margin variations of +/- 17.87 deg. Hence even though the first channel had infinite gain margin and 90 degrees of phase margin, allowing variation in both input channels leads to a factor of two reduction in the gain and phase margin.

```
mmi
```

```
mmi = struct with fields:
     GainMargin: [0.7288 1.3721]
    PhaseMargin: [-17.8304 17.8304]
      Frequency: 0
```

The guaranteed region of phase and gain variations for the closed-loop system can be illustrated graphically. The disk margin analysis, `dmi(2)`, indicates the closed-loop system will remain stable for simultaneous gain variations of 0.475 and 2.105 (± 6.465 dB) and phase margin variations of ± 39.18 deg in the second input channel. This is denoted by the region associated with the large ellipse in the following figure. The multivariable margin analysis at the input to the plant, `mmi`, indicates that the closed-loop system will be stable for independent, simultaneous, gain margin variation up to 0.728 and 1.373 (±2.753 dB) and phase margin variations up to ± 17.87 deg (the dark ellipse region) in both input channels.



The output channels have single-loop margins of infinite gain and 90 deg phase variation. The output multivariable margin analysis, `mmo`, leads to a maximum allowable gain margin variation of 0.607 and 1.649 and phase margin variations of +/- 27.53 degs. Hence even though both output channels had infinite gain margin and 90 degrees of phase margin, simultaneous variations in both channels significantly reduce the margins at the plant outputs.

```
mmo
```

```
mmo = struct with fields:
     GainMargin: [0.6070 1.6474]
    PhaseMargin: [-27.4826 27.4826]
      Frequency: 0.2663
```

The margins when all the input and output channels are allowed to vary independently are in the output `mmio`. For this system, this output shows that the allowable gain margin variations are 0.827 and 1.210 and allowable phase margin variations are +/- 10.84 deg.

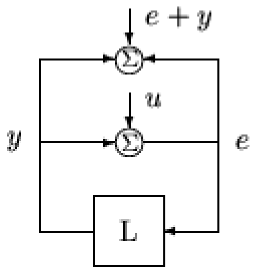
```
mmio
```

```
mmio = struct with fields:
     GainMargin: [0.8270 1.2092]
```

```
    PhaseMargin: [-10.8190 10.8190]
      Frequency: 0
```

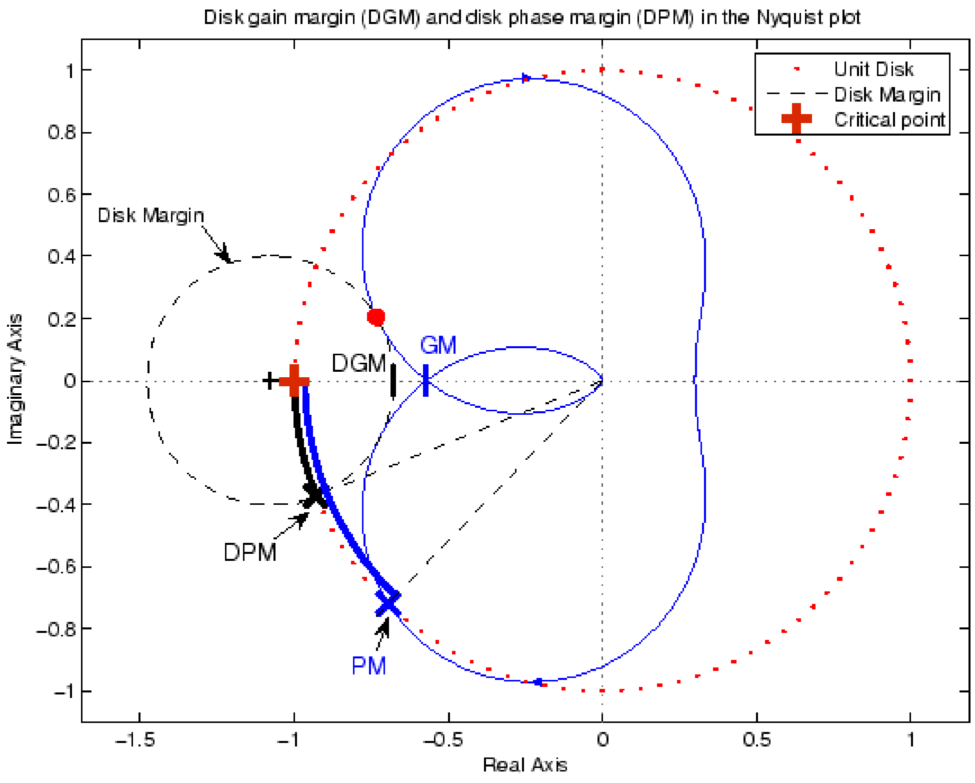
Algorithms

Two well-known loop robustness measures are based on the sensitivity function $S=(I–L)^{-1}$ and the complementary sensitivity function $T=L(I–L)^{-1}$ where L is the loop gain matrix associated with the input or output loops broken simultaneously. In the following figure, S is the transfer matrix from summing junction input u to summing junction output e. T is the transfer matrix from u to y. If signals e and y are summed, the transfer matrix from u to $e+y$ is given by $(I+L)\cdot (I–L)^{-1}$, the balanced sensitivity function. It can be shown (Dailey, 1991, Blight, Daily and Gangass, 1994) that each broken-loop gain can be perturbed by the complex gain $(1+\Delta)(1–\Delta)$ where $|\Delta|<1/\mu(S+T)$ or $|\Delta|<1/\sigma_{max}(S+T)$ at each frequency without causing instability at that frequency. The peak value of $\mu(S+T)$ or $\sigma_{max}(S+T)$ gives a robustness guarantee for all frequencies, and for $\mu(S+T)$ the guarantee is nonconservative (Blight, Daily and Gangass, 1994).



$$\begin{aligned} e &= (I-L)^{-1}u &&= Su \\ y &= L(I-L)^{-1}u &&= Tu \\ e+y &= (I+L)\cdot(I-L)^{-1}u &&= (S+T)u \end{aligned}$$

This figure shows a comparison of a disk margin analysis with the classical notations of gain and phase margins.



The Nyquist plot is of the loop transfer function $L(s)$

$$L(s) = \frac{\frac{s}{30} + 1}{(s + 1)(s^2 + 1.6s + 16)}$$

- The Nyquist plot of L corresponds to the blue line.
- The unit disk corresponds to the dotted red line.
- GM and PM indicate the location of the classical gain and phase margins for the system L.
- DGM and DPM correspond to the disk gain and phase margins. The disk margins provide a lower bound on classical gain and phase margins.
- The disk margin circle corresponds to the dashed black line. The disk margin corresponds to the largest disk centered at (GMD + 1/GMD)/2 that just touches the loop transfer function L. This location is indicated by the red dot.

The disk margin and multiple channel margins calculation involve the balanced sensitivity function *S*+*T*. For a given peak value of μ(*S*+*T*), any simultaneous phase and gain variations applied to each loop independently will not destabilize the system if the perturbations remain inside the corresponding circle or disk. This corresponds to the disk margin calculation to find `dmi` and `dmo`.

Similarly, the multiple channel margins calculation involves the balanced sensitivity function *S*+*T*. Instead of calculating μ(*S*+*T*) a single loop at a time, all the channels are included in the analysis. A μ-analysis problem is formulated with each channel perturbed by an independent, complex perturbation. The peak μ(*S*+*T*) value guarantees that any simultaneous, independent phase and gain variations applied to each loop simultaneously will not destabilize the system if they remain inside the corresponding circle or disk of size μ(*S*+*T*).

For frequency-response data (`frd`) models, `loopmargin` uses the techniques of *μ*-analysis to compute the disk margin at each frequency point in the model, and returns the weakest margin of these values. For all other models, the *μ*-analysis computation identifies the frequency with the weakest margin.

Compatibility Considerations

`loopmargin` is not recommended

Not recommended starting in R2018b

To compute disk-based stability margins of SISO and MIMO systems, use `diskmargin`. For loop-at-a-time classical gain margins, use `allmargin`. For stability margin analysis of feedback loops modeled in Simulink, first linearize the model and then use `diskmargin`.

`diskmargin`, introduced in R2018b, has improved numeric stability and more reliable results relative to `loopmargin`. The new command also includes an option for varying the eccentricity of the disk for better margin estimates. For more information, see `diskmargin`.

Update Code

To update your code to use `diskmargin` or `allmargin`:

Old code	New code
`[CM,DM,MM] = loopmargin(L)`	`[DM,MM] = diskmargin(L)` returns the disk margins of each feedback channel with all other loops closed in the structure `DM`, and the multiloop disk margin in the structure `MM`. `S = allmargin(L)` returns the classical loop-at-a-time gain and phase margins returned by `loopmargin` as `CM`. The margins are organized differently in the structure `S`. For more information, enter `help allmargin` at the MATLAB command prompt.
`[CMI,DMI,MMI,CMO,DMO,MMO,MMIO] = loopmargin(P,C)`	`MMIO = diskmargin(L)` All the multiloop disk margins returned by `loopmargin` are in the structure `MMIO`.
`[cm,dm,mm] = loopmargin(Model,Blocks,Ports)`	First linearize the Simulink model and then use `diskmargin` or `allmargin`.

References

Barrett, M.F., Conservatism with robustness tests for linear feedback control systems, Ph.D. Thesis, Control Science and Dynamical Systems, University of Minnesota, 1980.

Blight, J.D., R.L. Dailey, and D. Gangsass, “Practical control law design for aircraft using multivariable techniques,” *International Journal of Control*, Vol. 59, No. 1, 1994, pp. 93-137.

Bates, D., and I. Postlethwaite, “Robust Multivariable Control of Aerospace Systems,” *Delft University Press,* Delft, The Netherlands, ISBN: 90-407-2317-6, 2002.

See Also

`allmargin` | `diskmargin` | `loopsens` | `robstab` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

loopsens

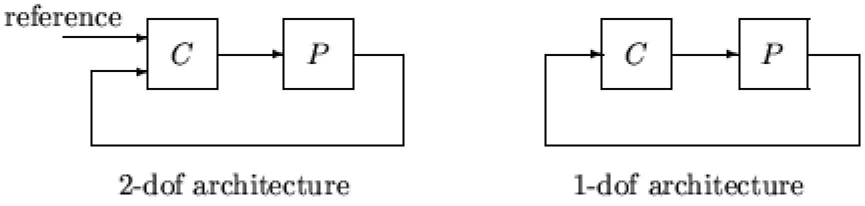
Sensitivity functions of plant-controller feedback loop

Syntax

```
loops = loopsens(P,C)
```

Description

`loops = loopsens(P,C)` creates a `struct`, `loops`, whose fields contain the multivariable sensitivity, complementary and open-loop transfer functions. The closed-loop system consists of the controller `C` in negative feedback with the plant `P`. `C` should only be the compensator in the feedback path, not any reference channels, if it is a 2-Dof controller as seen in the figure below. The plant and compensator `P` and `C` can be constant matrices, `double, lti` objects, `frd/ss/tf/zpk`, or uncertain objects `umat/ufrd/uss`.

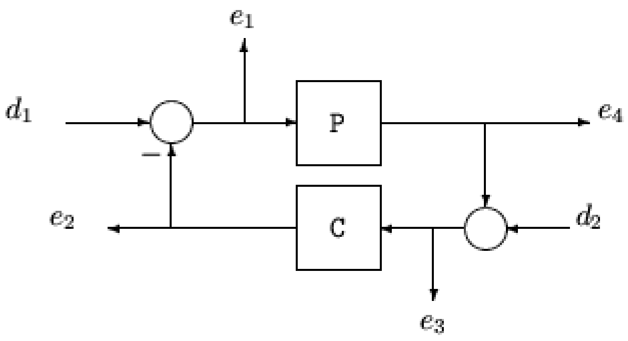


The `loops` returned variable is a structure with fields:

Field	Description
`Poles`	Closed-loop poles. `NaN` for `frd/ufrd` objects
`Stable`	1 if nominal closed loop is stable, 0 otherwise. `NaN` for `frd/ufrd` objects
`Si`	Input-to-plant sensitivity function
`Ti`	Input-to-plant complementary sensitivity function
`Li`	Input-to-plant loop transfer function

Field	Description
`So`	Output-to-plant sensitivity function
`To`	Output-to-plant complementary sensitivity function
`Lo`	Output-to-plant loop transfer function
`PSi`	Plant times input-to-plant sensitivity function
`CSo`	Compensator times output-to-plant sensitivity function

The multivariable closed-loop interconnection structure, shown below, defines the input/output sensitivity, complementary sensitivity, and loop transfer functions.



The following table gives the values of the input and output sensitivity functions for this control structure.

Description	Equation
Input sensitivity S_i (closed-loop transfer function from d_1 to e_1)	$S_i = (I + CP)^{-1}$
Input complementary sensitivity T_i (closed-loop transfer function from d_1 to e_2)	$T_i = CP(I + CP)^{-1}$
Output sensitivity S_o (closed-loop transfer function from d_2 to e_2)	$S_o = (I + PC)^{-1}$
Output complementary sensitivity T_o (closed-loop transfer function from d_2 to e_4)	$T_o = PC(I + PC)^{-1}$
Input loop transfer function L_i	$L_i = CP$
Output loop transfer function L_o	$L_o = PC$

Examples

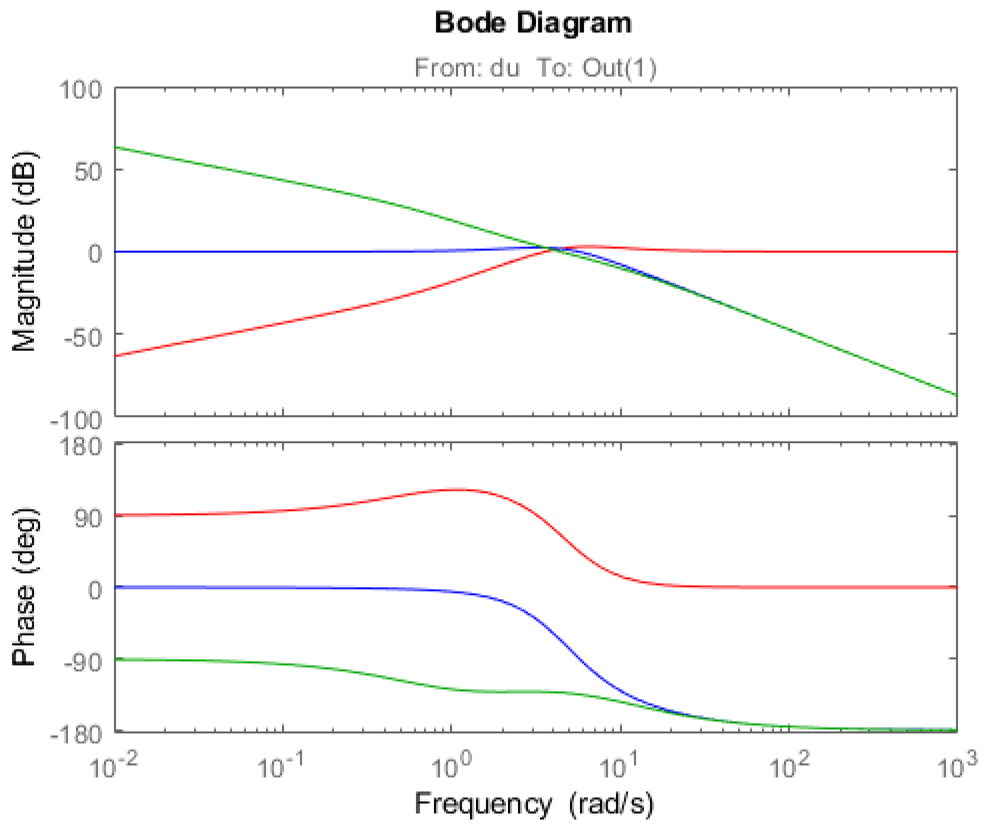
Single Input, Single Output (SISO) Loop Sensitivities

Consider PI controller for a dominantly 1st-order plant, with the closed-loop bandwidth of 2.5 rads/sec. Since the problem is SISO, all gains are the same at input and output.

```
gamma = 2; tau = 1.5; taufast = 0.1;
P = tf(gamma,[tau 1])*tf(1,[taufast 1]);
tauclp = 0.4;
xiclp = 0.8;
wnclp = 1/(tauclp*xiclp);
KP = (2*xiclp*wnclp*tau - 1)/gamma;
KI = wnclp^2*tau/gamma;
C = tf([KP KI],[1 0]);
```

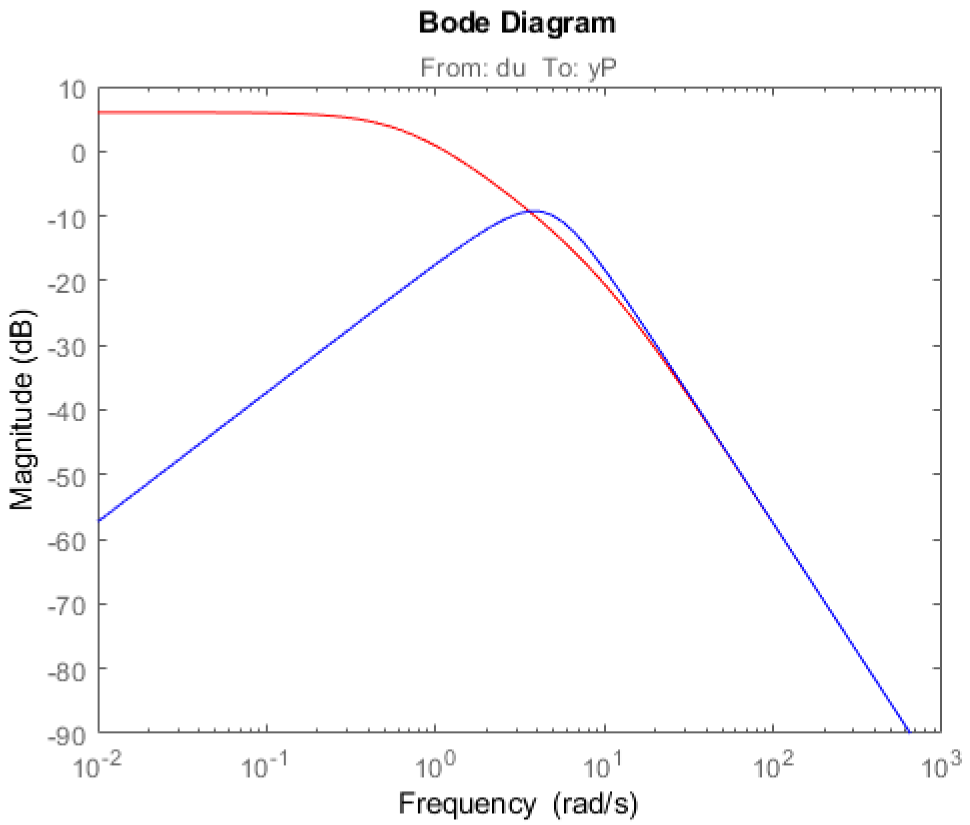
Form the closed-loop (and open-loop) systems with `loopsens`, and plot Bode plots using the gains at the plant input.

```
loops = loopsens(P,C);
bode(loops.Si,'r',loops.Ti,'b',loops.Li,'g')
```



Finally, compare the open-loop plant gain to the closed-loop value of `PSi`.

```
bodemag(P,'r',loops.PSi,'b')
```

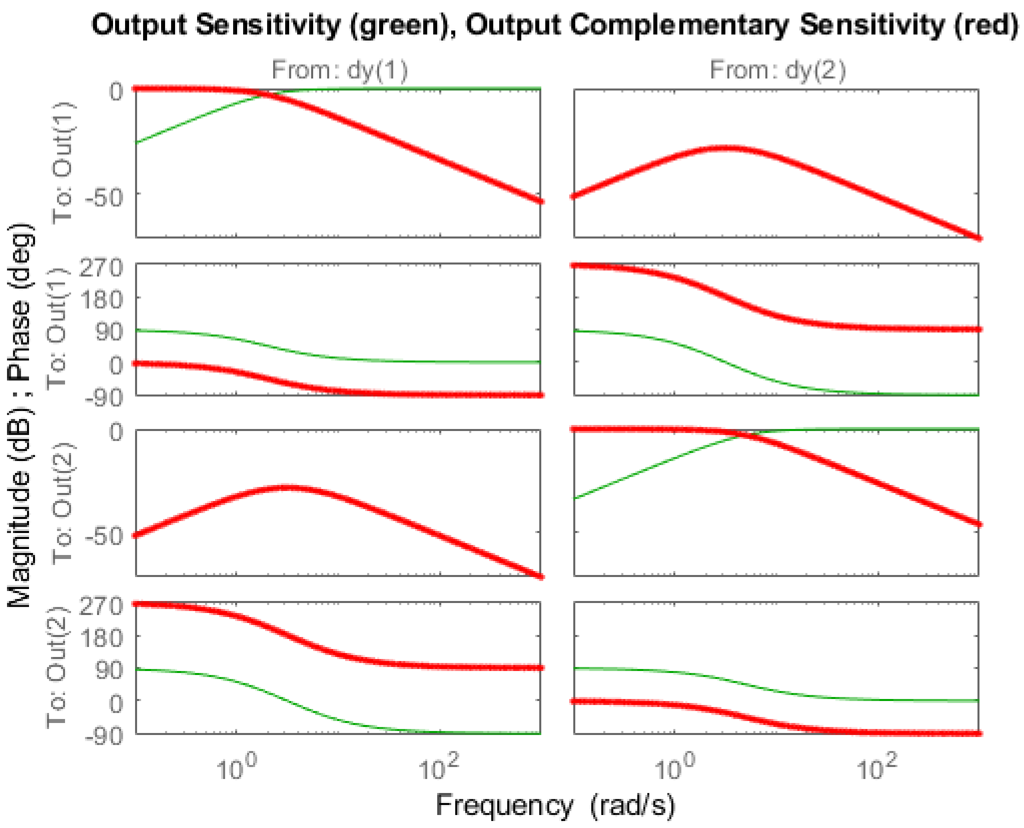


Multi Input, Multi Output (MIMO) Loop Sensitivities

Consider an integral controller for a constant-gain, 2-input, 2-output plant. For purposes of illustration, the controller is designed via inversion, with different bandwidths in each rotated channel.

```
P = ss([2 3;-1 1]);
BW = diag([2 5]);
[U,S,V] = svd(P.d);                % get SVD of Plant Gain
Csvd = V*inv(S)*BW*tf(1,[1 0])*U'; % inversion based on SVD
loops = loopsens(P,Csvd);
```

```
bode(loops.So,'g',loops.To,'r.',logspace(-1,3,120))
title('Output Sensitivity (green), Output Complementary Sensitivity (red)');
```



See Also

diskmargin | robstab | wcdiskmargin | wcgain

Introduced before R2006a

loopsyn

H_∞ optimal controller synthesis for LTI plant

Syntax

```
[K,CL,GAM,INFO]=loopsyn(G,Gd)
[K,CL,GAM,INFO]=loopsyn(G,Gd,RANGE)
```

Description

`loopsyn` is an H_∞ optimal method for loopshaping control synthesis. It computes a stabilizing H_∞controller K for plant G to shape the `sigma` plot of the loop transfer function GK to have desired loop shape G_d with accuracy γ = `GAM` in the sense that if ω_0 is the 0 db crossover frequency of the `sigma` plot of $G_d(j\omega)$, then, roughly,

$$\underline{\sigma}(G(j\omega)K(j\omega)) \geq \frac{1}{\gamma}\,\underline{\sigma}(G_d(j\omega)) \text{ for all } \omega > \omega_0 \tag{1-11}$$

$$\underline{\sigma}(G(j\omega)K(j\omega)) \leq \gamma\,\underline{\sigma}(G_d(j\omega)) \text{ for all } \omega > \omega_0 \tag{1-12}$$

The STRUCT array `INFO` returns additional design information, including a MIMO stable min-phase shaping pre-filter W, the shaped plant $G_s = GW$, the controller for the shaped plant $K_s = WK$, as well as the frequency range $\{\omega_{\min},\omega_{\max}\}$ over which the loop shaping is achieved

Input Argument	Description
`G`	LTI plant
`Gd`	Desired loop-shape (LTI model)
`RANGE`	(optional, default `{0,Inf}`) Desired frequency range for loop-shaping, a 1-by-2 cell array $\{\omega_{\min},\omega_{\max}\}$; $\omega_{\max}$ should be at least ten times $\omega_{\min}$

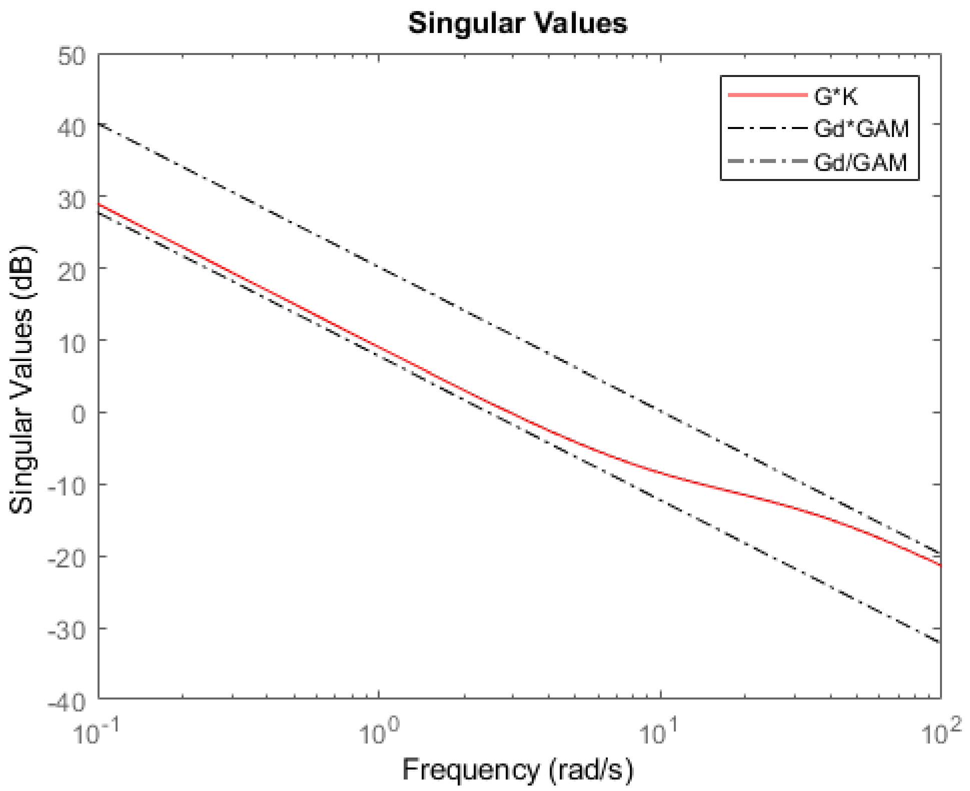
Output Argument	Description
K	LTI controller
CL= G*K/(I +GK)	LTI closed-loop system
GAM	Loop-shaping accuracy (GAM ≥ 1, with GAM=1 being perfect fit
INFO	Additional output information
INFO.W	LTI pre-filter W satisfying $\sigma(G_d) = \sigma\ (GW)$ for all ω; W is always minimum-phase.
INFO.Gs	LTI shaped plant: $G_s = GW$.
INFO.Ks	LTI controller for the shaped plant: $K_s = WK$.
INFO.range	$\{\omega_{\text{min}},\omega_{\text{max}}\}$ cell-array containing the approximate frequency range over which loop-shaping could be accurately achieved to with accuracy G. The output INFO.range is either the same as or a subset of the input range.

Examples

Optimal loopsyn Loop-Shaping Control

Calculate the optimal loopsyn loop shaping control for a 5-state, 4-output, 5-input plant with a full-rank nonmininum-phase zero at $s = 10$.

```
rng(0,'twister');
s = tf('s');
w0 = 5;
Gd = 5/s;                              % desired bandwidth w0=5
G =((s-10)/(s+100))*rss(3,4,5);      % 4-by-5 non-min-phase plant
[K,CL,GAM,INFO] = loopsyn(G,Gd);
sigma(G*K,'r',Gd*GAM,'k-.',Gd/GAM,'k-.',{.1,100})  % plot result
legend('G*K','Gd*GAM','Gd/GAM')
```



This plot shows that the controller `K` optimally fits `sigma(G*K)`. The controller falls between `sigma(Gd)+ GAM` and `sigma(Gd)- GAM` (expressed in dB). In this example, `GAM` = 2.0423 = 6.2026 dB.

Limitations

The plant G must be stabilizable and detectable, must have at least as many inputs as outputs, and must be full rank; i.e,

- `size(G,2) ≥ size(G,1)`

- `rank(freqresp(G,w)) = size(G,1)` for some frequency w.

The order of the controller K can be large. Generically, when G_d is given as a SISO LTI, then the order N_K of the controller K satisfies

$$\begin{aligned} N_K &= N_{Gs} + N_W \\ &= N_y N_{Gd} + N_{RHP} + N_W \\ &= N_y N_{Gd} + N_{RHP} + N_G \end{aligned}$$

where

- N_y denotes the number of outputs of the plant G.
- N_{RHP} denotes the total number of nonstable poles and nonminimum-phase zeros of the plant G, including those on the stability boundary and at infinity.
- N_G, N_{Gs}, N_{Gd} and N_W denote the respective orders of G, G_s, G_d and W.

Model reduction can help reduce the order of K — see `reduce` and `ncfmr`.

Algorithms

Using the GCD formula of Le and Safonov [1], `loopsyn` first computes a stable-minimum-phase loop-shaping, squaring-down prefilter W such that the shaped plant $G_s = GW$ is square, and the desired shape G_d is achieved with good accuracy in the frequency range $\{\omega_{\min},\omega_{\max}\}$ by the shaped plant; i.e.,

$$\sigma(G_d) \approx \sigma(G_s) \quad \text{for} \quad \text{all} \quad \omega \quad \epsilon \quad \{\omega_{\min},\omega_{\max}\}.$$

Then, `loopsyn` uses the Glover-McFarlane [2] normalized-coprime-factor control synthesis theory to compute an optimal "loop-shaping" controller for the shaped plant via `Ks=ncfsyn(Gs), and` returns `K=W*Ks`.

If the plant G is a continuous time LTI and

1. G has a full-rank D-matrix, and
2. no finite zeros on the $j\omega$-axis, and
3. $\{\omega_{\min},\omega_{\max}\}=[0,\infty]$,

then *GW* theoretically achieves a perfect accuracy fit $\sigma(G_d) = \sigma(GW)$ for all frequency ω. Otherwise, `loopsyn` uses a bilinear pole-shifting bilinear transform [3] of the form

```
Gshifted=bilin(G,-1,'S_Tust',[ωmin,ωmax]),
```

which results in a perfect fit for transformed `Gshifted` and an approximate fit over the smaller frequency range $[\omega_{\min},\omega_{\max}]$ for the original unshifted *G* provided that $\omega_{\max} >> \omega_{\min}$. For best results, you should choose $\omega_{\max}$ to be at least 100 times greater than $\omega_{\min}$. In some cases, the computation of the optimal *W* for `Gshifted` may be singular or ill-conditioned for the range $[\omega_{\min},\omega_{\max}]$, as when `Gshifted` has undamped zeros or, in the continuous-time case only, `Gshifted` has a *D*-matrix that is rank-deficient); in such cases, `loopsyn` automatically reduces the frequency range further, and returns the reduced range $[\omega_{\min},\omega_{\max}]$ as a cell array in the output `INFO.range=`$\{\omega_{\min},\omega_{\max}\}$

References

[1] Le, V.X., and M.G. Safonov. Rational matrix GCD's and the design of squaring-down compensators—a state space theory. *IEEE Trans. Autom.Control*, AC-36(3):384–392, March 1992.

[2] Glover, K., and D. McFarlane. Robust stabilization of normalized coprime factor plant descriptions with H_∞-bounded uncertainty. *IEEE Trans. Autom. Control*, AC-34(8):821–830, August 1992.

[3] Chiang, R.Y., and M.G. Safonov. H_∞ synthesis using a bilinear pole-shifting transform. *AIAA J. Guidance, Control and Dynamics*, 15(5):1111–1115, September–October 1992.

See Also

`mixsyn` | `ncfsyn`

Topics

"Loop Shaping of HIMAT Pitch Axis Controller"

Introduced before R2006a

ltiarray2uss

Compute uncertain system bounding given LTI `ss` array

Compatibility

Note `ltiarray2uss` will be removed in a future release. Use `ucover` instead.

Syntax

```
usys = ltiarray2uss(P,Parray,ord)
```

```
[usys,wt] = ltiarray2uss(P,Parray,ord)
```

```
[usys,wt,diffdata] = ltiarray2uss(P,Parray,ord)
```

```
[usys,wt,diffdata] = ltiarray2uss(P,Parray,ord,'InputMult')
```

```
[usys,wt,diffdata] = ltiarray2uss(P,Parray,ord,'OutputMult')
```

```
[usys,wt,diffdata] = ltiarray2uss(P,Parray,ord,'Additive')
```

Description

The command `ltiarray2uss`, calculates an uncertain system `usys` with nominal value `P`, and whose range of behavior includes the given array of systems, `Parray`.

`usys = ltiarray2uss(P,Parray,ord), usys` is formulated as an input multiplicative uncertainty model,

`usys = P*(I + wt*ultidyn('IMult',[size(P,2) size(P,2)]))`, where `wt` is a stable scalar system, whose magnitude overbounds the relative difference, `(P - Parray)/P`. The state order of the weighting function used to bound the multiplicative difference between `P` and `Parray` is `ord`. Both `P` and `Parray` must be in the classes `ss/tf/zpk/frd`. If `P` is an `frd` then `usys` will be a `ufrd` object, otherwise `usys` will be a uss object. The ultidyn atom is named based on the variable name of Parray in the calling workspace.

[usys,wt] = ltiarray2uss(P,Parray,ord), returns the weight wt used to bound the infinity norm of ((P - Parray)/P).

[usys,wt] = ltiarray2uss(P,Parray,ord,'OutputMult'), uses multiplicative uncertainty at the plant output (as opposed to input multiplicative uncertainty). The formula for usys is

usys = (I + wt*ultidyn('Name',[size(P,1) size(P,1)])*P).

[usys,wt] = ltiarray2uss(P,Parray,ord,'Additive'), uses additive uncertainty.

usys = P + wt*ultidyn('Name',[size(P,1) size(P,2)]). wt is a frequency domain overbound of the infinity norm of (Parray - P).

[usys,wt] = ltiarray2uss(P,Parray,ord,'InputMult'), uses multiplicative uncertainty at the plant input (this is the default). The formula for usys is usys = P*(I + wt*ultidyn('Name',[size(P,2) size(P,2)])) .

[usys,wt,diffdata] = ltiarray2uss(P,Parray,ord,type) returns the norm of the difference (absolute difference for additive, and relative difference for multiplicative uncertainty) between the nominal model P and Parray. wt satisfies diffdata(w_i) < |wt(w_i)| at all frequency points.

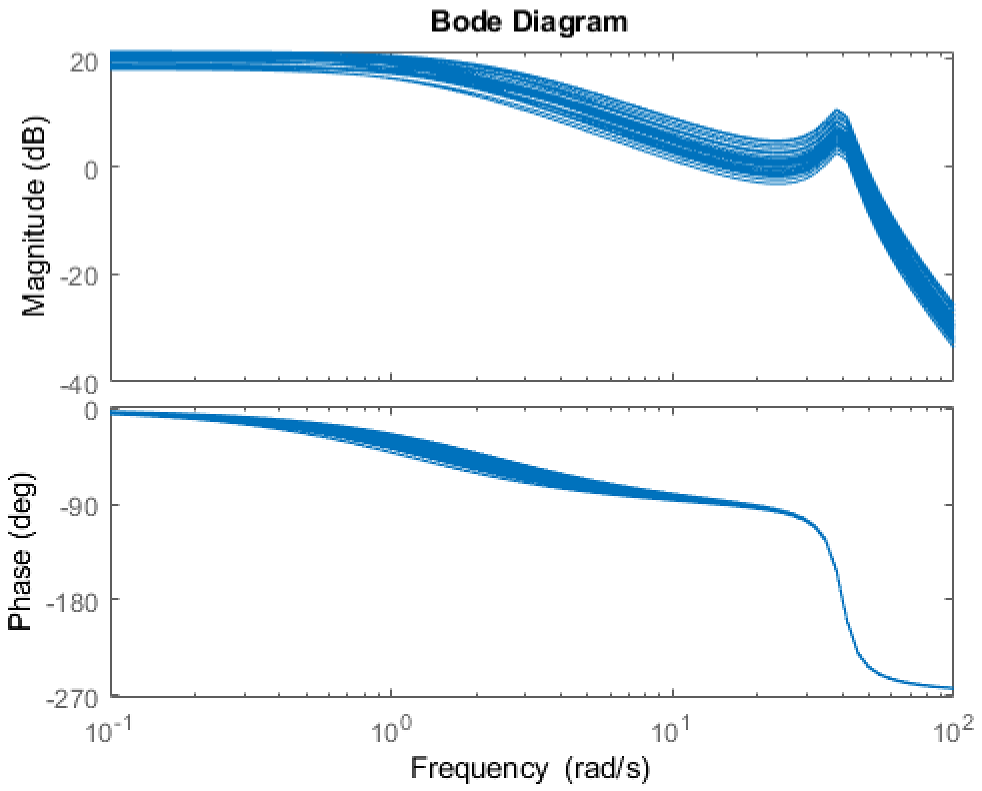
Examples

Uncertain System Bounding an LTI Array

Consider a third order transfer function with an uncertain gain, filter time constant and a lightly damped flexible mode. This model is used to represent a physical system from which frequency response data is acquired.

```
gain = ureal('gain',10,'Perc',20);
tau = ureal('tau',.6,'Range',[.42 .9]);
wn = 40;
zeta = 0.1;
usys = tf(gain,[tau 1])*tf(wn^2,[1 2*zeta*wn wn^2]);
```

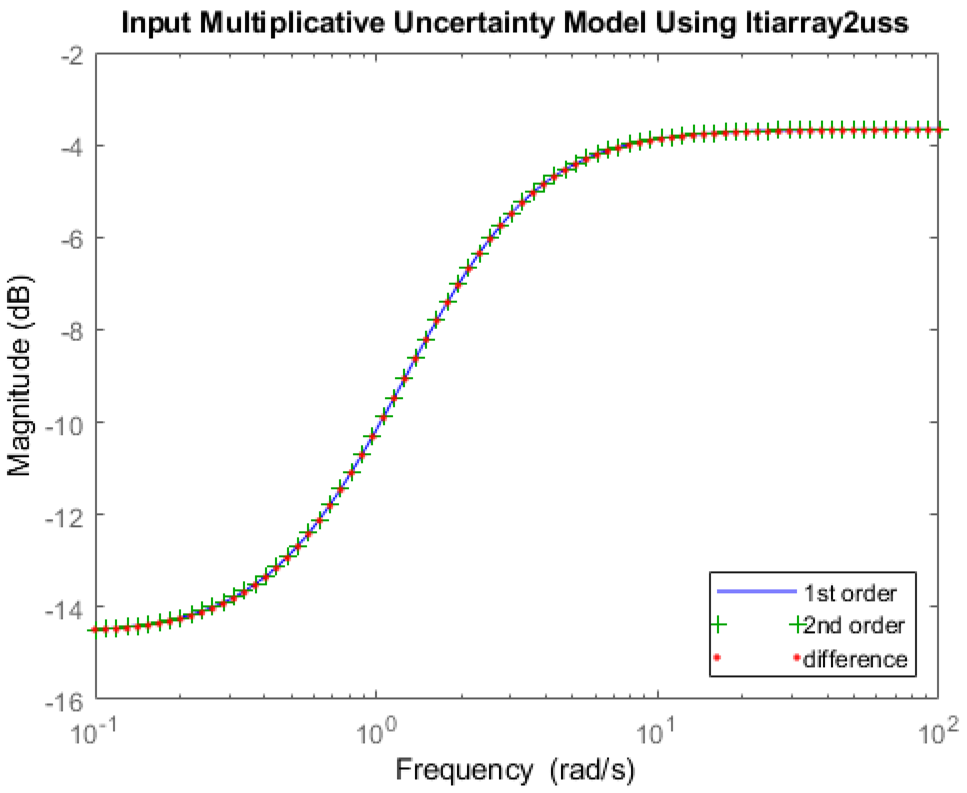
```
sysnom = usys.NominalValue;
parray = usample(usys,30);
om = logspace(-1,2,80);
parrayg = frd(parray,om);
bode(parrayg)
```



The frequency response data in `parray` represents 30 experiments performed on the system. The command `ltiarray2uss` is used to generate an uncertain model, `umod`, based on the frequency response data. Initially an input multiplicative uncertain model is used to characterize the collection of 30 frequency responses. First and second order input multiplicative uncertainty weight are calculated from the data.

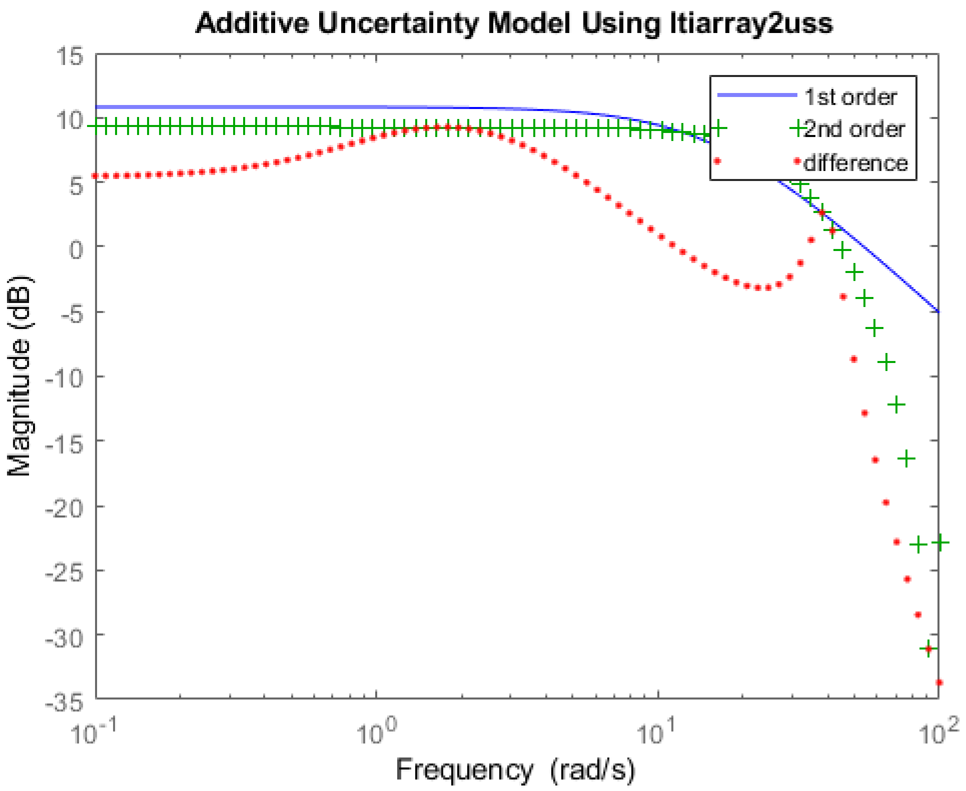
```
[umodIn1,wtIn1,diffdataIn] = ltiarray2uss(sysnom,parrayg,1);
[umodIn2,wtIn2,diffdataIn] = ltiarray2uss(sysnom,parrayg,2);
```

```
bodemag(wtIn1,'b-',wtIn2,'g+',diffdataIn,'r.',om)
title('Input Multiplicative Uncertainty Model Using ltiarray2uss')
legend('1st order','2nd order','difference','Location','SouthEast')
```



Alternatively, an additive uncertain model is used to characterize the collection of 30 frequency responses.

```
[umodAdd1,wtAdd1,diffdataAdd] = ltiarray2uss(sysnom,parrayg,1,'Additive');
[umodAdd2,wtAdd2,diffdataAdd] = ltiarray2uss(sysnom,parrayg,2,'Additive');
bodemag(wtAdd1,'b-',wtAdd2,'g+',diffdataAdd,'r.',om)
title('Additive Uncertainty Model Using ltiarray2uss')
legend('1st order','2nd order','difference')
```



See Also

`fitmagfrd` | `ultidyn` | `uss`

Topics

"First-Cut Robust Design"

Introduced in R2006a

ltrsyn

LQG loop transfer-function recovery (LTR) control synthesis

Syntax

```
[K,SVL,W1] = ltrsyn(G,F,XI,THETA,RHO)
```

```
[K,SVL,W1] = ltrsyn(G,F,XI,THETA,RHO,W)
```

```
[K,SVL,W1] = ltrsyn(G,F,XI,THETA,RHO,OPT)
```

```
[K,SVL,W1] = ltrsyn(G,F,XI,THETA,RHO,W,OPT)
```

Description

`[K,SVL,W1] = ltrsyn(G,F,XI,TH,RHO)` computes a reconstructed-state output-feedback controller K for LTI plant `G` so that `K*G` asymptotically recovers plant-input full-state feedback loop transfer function $L(s) = F(Is–A)^{-1}B+D;$ that is, at any frequency `w>0`, `max(sigma(K*G-L, w))`→0 as ρ→ ∞, where `L= ss(A,B,F,D)` is the LTI full-state feedback loop transfer function.

`[K,SVL,W1] = ltrsyn(G,F1,Q,R,RHO,'OUTPUT')` computes the solution to the `dual' problem of filter loop recovery for LTI plant `G` where `F` is a Kalman filter gain matrix. In this case, the recovery is at the plant output, and `max(sigma(G*K-L, w))`→0 as ρ→∞, where `L1` denotes the LTI filter loop feedback loop transfer function `L1= ss(A,F,C,D)`.

Only the LTI controller `K` for the final value `RHO(end)`is returned.

Inputs	
`G`	LTI plant
`F`	LQ full-state-feedback gain matrix
`XI`	plant noise intensity, or, if `OPT='`*`OUTPUT`*`'` state-cost matrix `XI=Q`,

Inputs	
THETA	sensor noise intensity or, if OPT='*OUTPUT*' control-cost matrix THETA=R,
RHO	vector containing a set of recovery gains
W	(optional) vector of frequencies (to be used for plots); if input W is not supplied, then a reasonable default is used

Outputs	
K	*K*(*s*) — LTI LTR (loop-transfer-recovery) output-feedback, for the last element of RHO (i.e., RHO(end))
SVL	sigma plot data for the recovered loop transfer function if G is MIMO or, for SISO G only, Nyquist loci SVL = [re(1:nr) im(1:nr)]
W1	frequencies for SVL plots, same as W when present

Examples

```
s=tf('s');G=ss(1e4/((s+1)*(s+10)*(s+100)));[A,B,C,D]=ssdata(G);
F=lqr(A,B,C'*C,eye(size(B,2)));
L=ss(A,B,F,0*F*B);
XI=100*C'*C; THETA=eye(size(C,1));
RHO=[1e3,1e6,1e9,1e12];W=logspace(-2,2);
nyquist(L,'k-.');hold;
[K,SVL,W1]=ltrsyn(G,F,XI,THETA,RHO,W);
```

See also ltrdemo

Limitations

The ltrsyn procedure may fail for non-minimum phase plants. For full-state LTR (default OPT='*INPUT*'), the plant should not have fewer outputs than inputs. Conversely for filter LTR (when OPT='*OUTPUT*'), the plant should not have fewer inputs than outputs. The plant must be strictly proper, i.e., the *D*-matrix of the plant should be all zeros. ltrsyn is only for continuous time plants (Ts==0)

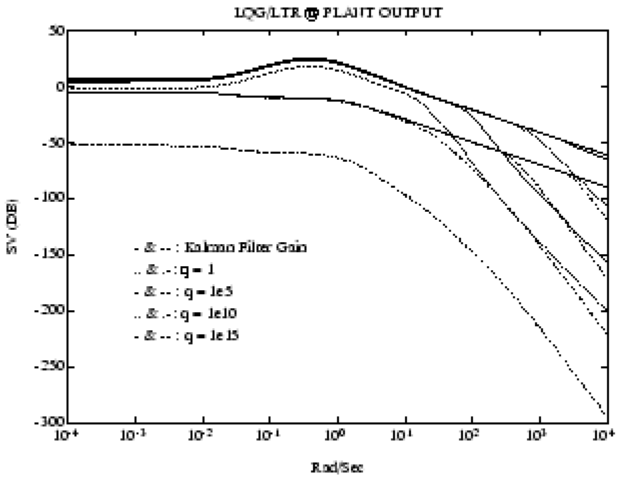
Algorithms

For each value in the vector `RHO`, `[K,SVL,W1] = ltrsyn(G,F,XI,THETA,RHO)` computes the full-state-feedback (default `OPT='`*`INPUT`*`'`) LTR controller

$$K(s) = \left[K_c(Is - A + BK_c + K_fC - K_fDK_c)^{-1}K_f\right]$$

where K_c = `F` and K_f = `lqr(A',C',XI+RHO(i)*B*B',THETA)`. The “fictitious noise” term `RHO(i)*B*B'` results in loop-transfer recovery as `RHO(i)` → ∞. The Kalman filter gain is $K_f = \sum C^T\Theta^{-1}$ where Σ satisfies the Kalman filter Riccati equation $0 = \sum A^T + A\sum - \sum C^T\Theta^{-1}C\sum + \Xi + \rho BB^T$. See [1] for further details.

Similarly for the 'dual' problem of filter loop recovery case, `[K,SVL,W1] = ltrsyn(G,F,Q,R,RHO,'OUTPUT')` computes a filter loop recovery controller of the same form, but with K_f = `F` is being the input filter gain matrix and the control gain matrix K_c computed as K_c = `lqr(A,B,Q+RHO(i)*C'*C,R)`.



Example of LQG/LTR at Plant Output.

References

[1] Doyle, J., and G. Stein, “Multivariable Feedback Design: Concepts for a Classical/Modern Synthesis,” *IEEE Trans. on Automat. Contr.*, AC-26, pp. 4-16, 1981.

See Also

`h2syn` | `hinfsyn` | `loopsyn` | `lqg` | `ncfsyn`

Introduced before R2006a

makeweight

Weighting function with monotonic gain profile

Syntax

```
W = makeweight(dcgain,[freq,mag],hfgain)
W = makeweight(dcgain,[freq,mag],hfgain,Ts)
W = makeweight(dcgain,[freq,mag],hfgain,Ts,N)
W = makeweight(dcgain,wc,hfgain,___)
```

Description

`makeweight` is a convenient way to specify loop shapes, target gain profiles, or weighting functions for applications such as controller synthesis and control system tuning.

`W = makeweight(dcgain,[freq,mag],hfgain)` creates a first-order, continuous-time weight $W(s)$ satisfying these constraints:

$$\begin{aligned} W(0) &= \text{dcgain} \\ W(\text{Inf}) &= \text{hfgain} \\ |W(j\cdot\text{freq})| &= \text{mag}. \end{aligned}$$

In other words, the gain of `W` passes through `mag` at the finite frequency `freq`.

`W = makeweight(dcgain,[freq,mag],hfgain,Ts)` creates a first-order, discrete-time weight $W(z)$ satisfying these constraints:

$$\begin{aligned} W(1) &= \text{dcgain} \\ W(-1) &= \text{hfgain} \\ \left|W\left(e^{j\cdot\text{freq}\cdot\text{Ts}}\right)\right| &= \text{mag}. \end{aligned}$$

In other words, the gain of `W` passes through `mag` at the frequency `freq`. The frequency `freq` must satisfy 0 < `freq` < π/`Ts`.

`W = makeweight(dcgain,[freq,mag],hfgain,Ts,N)` uses an `N`th-order transfer function with poles and zeros in a Butterworth pattern to meet the constraints. The higher the order N, the steeper the transition from low to high gain. To create a continuous-time higher order weighting function, use Ts = 0.

W = makeweight(dcgain,wc,hfgain,___) specifies the gain crossover frequency wc. This syntax is equivalent to setting [freq,mag] to [wc,1]. You can use this syntax with any of the previous input-argument combinations to create a continuous-time, discrete-time, or Butterworth weighting function.

Examples

Continuous-Time Weighting Functions

Create continuous-time weighting functions by specifying the low-frequency gain, high-frequency gain, and magnitude of the gain at some intermediate frequency.

For instance, create a weighting function with a gain of 40 dB at low frequency, rolling off to –20 dB at high frequency. Specify further that the gain is about 10 dB at 1 rad/s by putting these values in a vector [freq,mag]. Specify all the gains in absolute units.

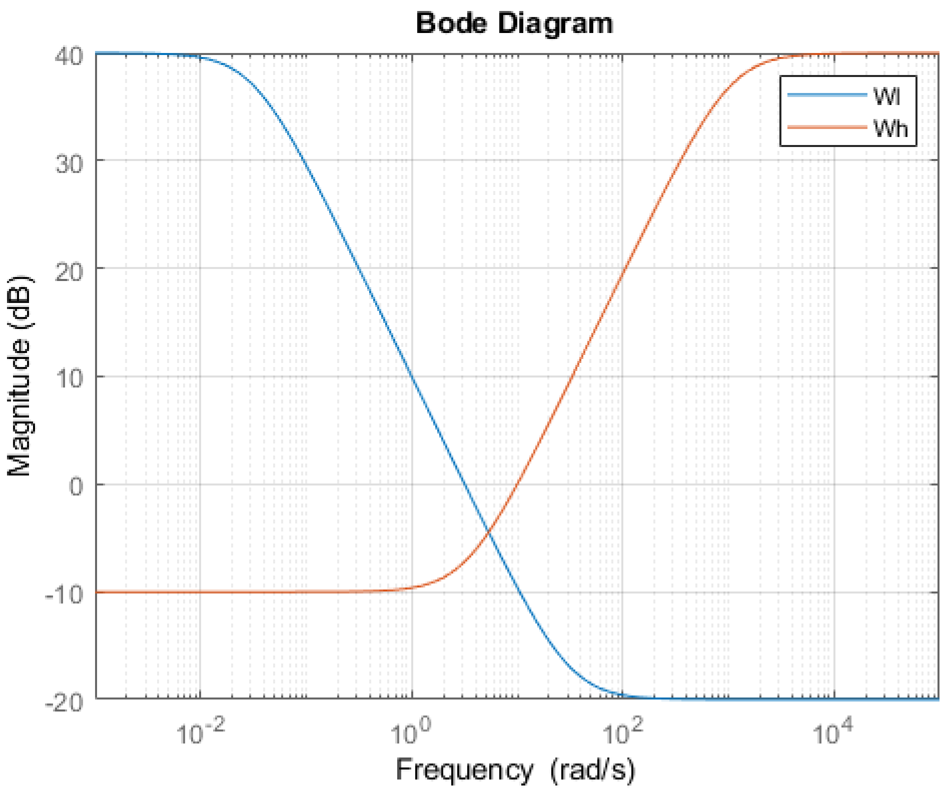
```
Wl = makeweight(100,[1,3.16],0.1);
```

Create a weighting function with a gain of –10 dB at low frequency, rising to 40 dB at high frequency. Specify a 0 dB crossover frequency of 10 rad/s. To specify a 0 dB crossover frequency, you can use the crossover frequency as the second input argument instead of the vector [freq,mag].

```
Wh = makeweight(0.316,10,100);
```

Plot the magnitudes of the weighting functions to confirm that they meet the response specifications.

```
bodemag(Wl,Wh)
legend
grid on
```



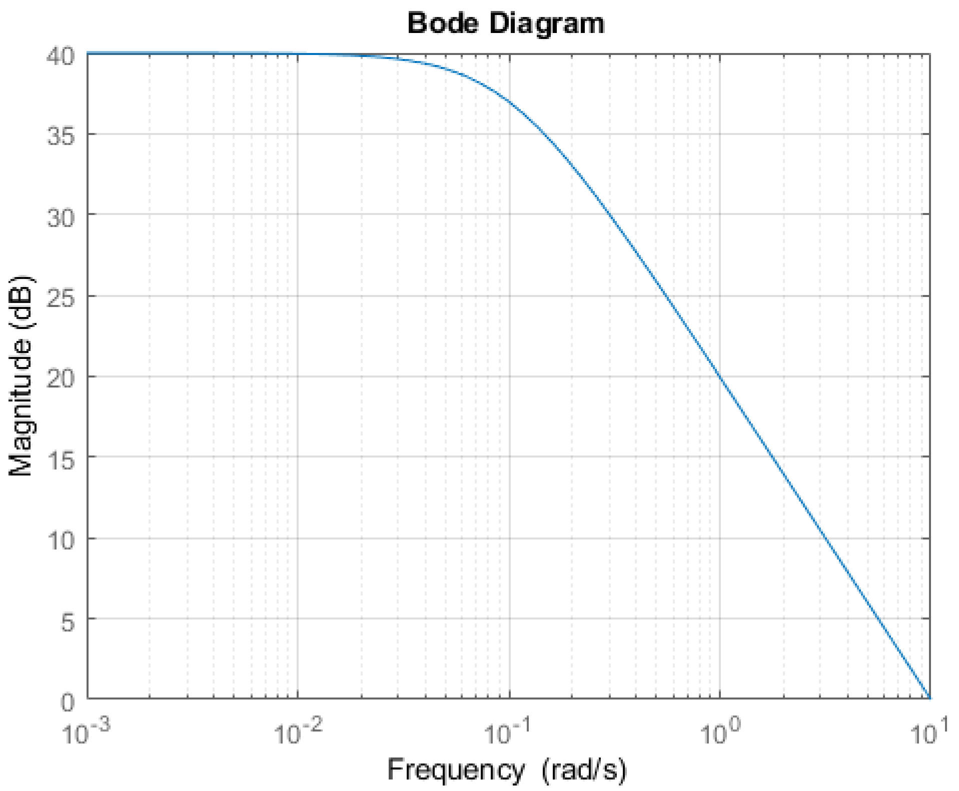
Weighting Functions with Roll-Off

Create a gain profile that rolls off at high frequency without flattening. Specify a gain of 40 dB at low frequency and a crossover frequency of 10 rad/s.

```
W = makeweight(100,[10 1],0);
```

Specifying a high-frequency gain of 0 ensures that the frequency response rolls off at high frequencies without leveling off. Plot the gain profile to confirm this shape.

```
bodemag(W)
grid on
```



Discrete-Time Weighting Functions

Create discrete-time weighting functions by specifying the low-frequency gain, high-frequency gain, magnitude of the gain at some intermediate frequency, and sample time.

Create a weighting function with a sample time of 0.1 s. Specify a gain of 40 dB at low frequency, rolling off to –20 dB at high frequency. Specify further that the gain is about 10 dB at 0.01 rad/s. Provide all gains in absolute units.

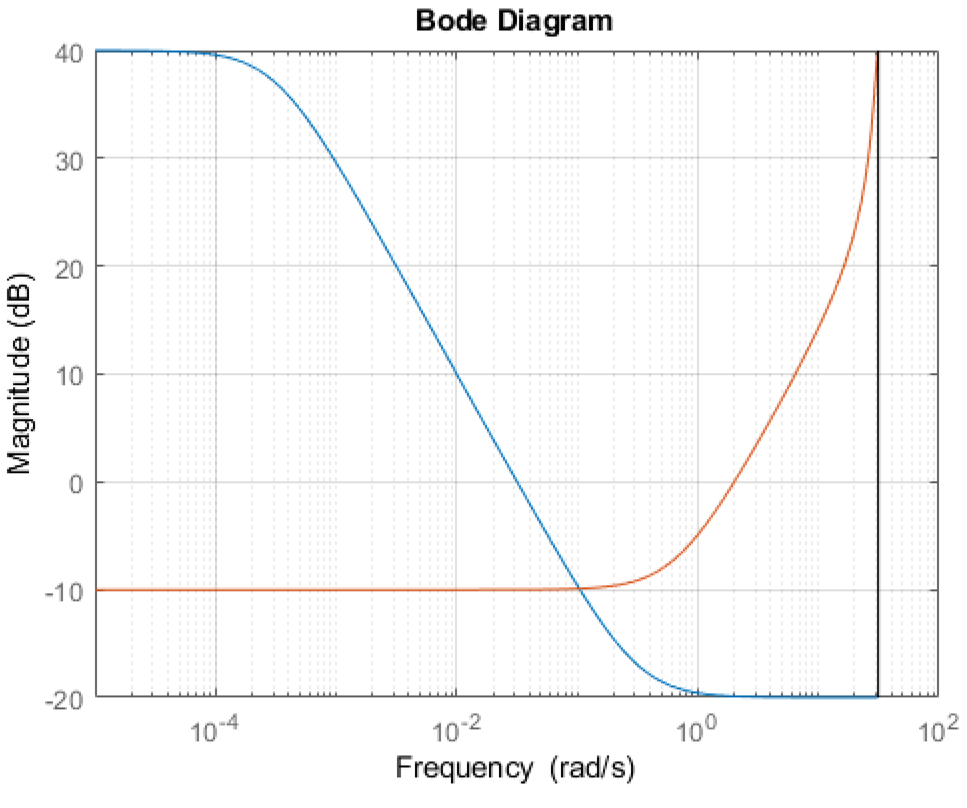
```
Wl = makeweight(100,[0.01,3.16],0.1,0.1);
```

Create a weighting function with a gain of –10 dB at low frequency, rising to 40 dB at high frequency. Specify a 0 dB crossover frequency of 2 rad/s and a sample time of 0.1 s. To specify a 0 dB crossover frequency, you can use the crossover frequency as the second input argument instead of the vector `[freq,mag]`.

```
Wh = makeweight(0.316,2,100,0.1);
```

Plot the magnitudes of the weighting functions to confirm that they meet the response specifications.

```
bodemag(Wl,Wh)
grid on
```

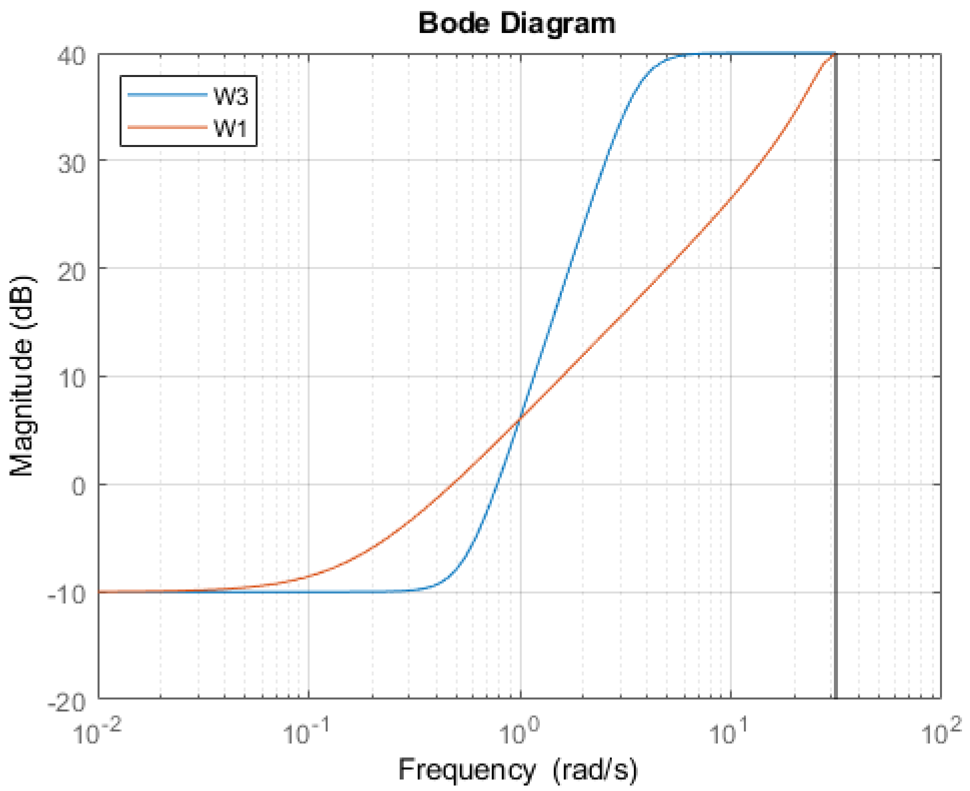


The high-frequency leveling of Wh is distorted due to the proximity of its crossover frequency to the Nyquist frequency.

Higher Order Weighting Functions

By default, makeweight creates first-order weighting functions. If you want a sharper transition between the low-frequency and high-frequency gains, you can specify the order with the last input argument. For instance, suppose you want to create a weighting function with a sample time of 0.1 s. The function has a gain of –10 dB at low frequency, rising to 40 dB at high frequency. Additionally, the gain passes through 6 dB at 1 rad/s. For comparison, create both a third-order and a first-order function with these specifications.

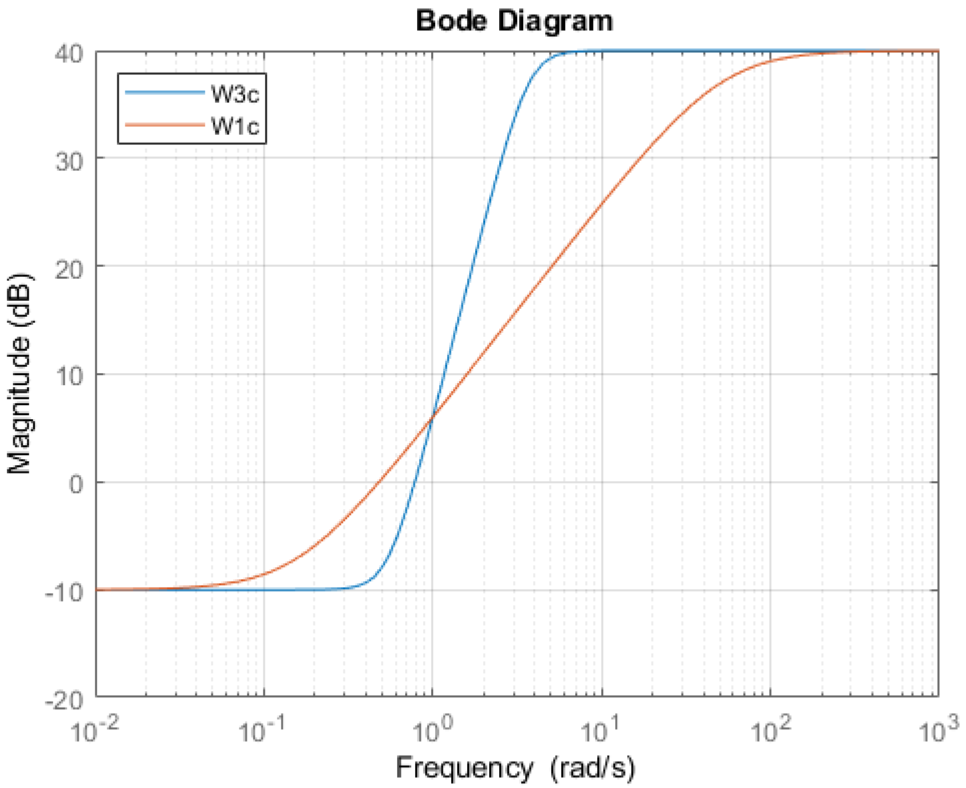
```
W3 = makeweight(0.316,[1 2],100,0.1,3);
W1 = makeweight(0.316,[1 2],100,0.1);
bodemag(W3,W1)
legend('location','northwest')
grid on
```



For the first-order function, the high-frequency leveling is distorted due to the proximity of its crossover frequency to the Nyquist frequency. Using a sharper, higher-order transition ensures that the function has leveled out before reaching the Nyquist frequency.

To create continuous-time weighting functions of higher order, set `Ts = 0`. For instance, create continuous-time weighting functions with the same gain specifications as `W1` and `W3`.

```
W3c = makeweight(0.316,[1 2],100,0,3);
W1c = makeweight(0.316,[1 2],100);
bodemag(W3c,W1c)
legend('location','northwest')
grid on
```



Input Arguments

`dcgain` — Low-frequency gain
real scalar

Low-frequency gain of the weighting function, specified as a real scalar value. Express the gain in absolute units. For example, to specify a low-frequency gain of 20 dB, set `dcgain` = 10.

The low-frequency gain, high-frequency gain, and magnitude must satisfy:

- |`dcgain`| > `mag` > |`hfgain`| for a low-pass weight
- |`dcgain`| < `mag` < |`hfgain`| for a high-pass weight

`[freq,mag]` — Target magnitude and corresponding frequency

two-element vector

Target magnitude and corresponding frequency, specified as a two-element vector. You specify where the gain of `W` transitions between the low-frequency and high-frequency values by specifying a target magnitude at a particular frequency. For instance, if you set `[freq,mag] = [10,0.1]`, then the magnitude of `W` passes through 0.1 (–10 dB) at a frequency of 10 rad/s. Similarly, setting `[freq,mag] = [5,1]` specifies a 0 dB (unit gain) crossover frequency of 5 rad/s.

The low-frequency gain, high-frequency gain, and magnitude must satisfy:

- |`dcgain`| > `mag` > |`hfgain`| for a low-pass weight
- |`dcgain`| < `mag` < |`hfgain`| for a high-pass weight

`hfgain` — High-frequency gain

real scalar

High-frequency gain of the weighting function, specified as a real scalar value. Express the gain in absolute units. For example, to specify a high-frequency gain of –20 dB, set `dcgain` = 0.1.

The low-frequency gain, high-frequency gain, and magnitude must satisfy:

- |`dcgain`| > `mag` > |`hfgain`| for a low-pass weight
- |`dcgain`| < `mag` < |`hfgain`| for a high-pass weight

`Ts` — Sample time

nonnegative scalar | –1

Sample time of discrete-time weighting function, specified as a nonnegative scalar value or –1. A positive value sets the sample time in seconds. The special value –1 creates a discrete-time state-space model with an unspecified sample time.

Setting `Ts` = 0 creates a continuous-time weighting function. This value is useful when you want to create higher order continuous-time transfer functions using the `N` input argument. For an example, see “Higher Order Weighting Functions” on page 1-316.

`N` — Order of weighting function
1 (default) | positive integer

Order of weighting function, specified as a positive integer. `makeweight` uses an `N`th-order transfer function with poles and zeros in a Butterworth pattern to meet the specified gain constraints. The higher the order `N`, the steeper the transition from low to high gain.

`wc` — Crossover frequency
positive scalar

Crossover frequency of the weighting function in radians/second, specified as a positive scalar value. Using the input argument `wc` is equivalent to using `[freq,mag] = [wc,1]`.

For discrete-time weighting functions, the crossover frequency must satisfy `wc*Ts` < π.

Output Arguments

`W` — Weighting function
state-space model

Weighting function, returned as a state-space (`ss`) model. For continuous-time weighting functions, the response of `W` satisfies the following:

$$\begin{aligned} W(0) &= \text{dcgain} \\ W(\text{Inf}) &= \text{hfgain} \\ |W(j \cdot \text{freq})| &= \text{mag}. \end{aligned}$$

For discrete-time weighting functions, the response of `W` satisfies the following:

$$\begin{aligned} W(1) &= \text{dcgain} \\ W(-1) &= \text{hfgain} \\ \left|W\left(e^{j \cdot \text{freq} \cdot \text{Ts}}\right)\right| &= \text{mag}. \end{aligned}$$

See Also

`TuningGoal.LoopShape` | `augw` | `hinfstruct` | `hinfsyn` | `mixsyn` | `mkfilter` | `musyn`

Topics

"Mixed-Sensitivity Loop Shaping"

Introduced before R2006a

matnbr

Number of matrix variables in system of LMIs

Syntax

```
K = matnbr(lmisys)
```

Description

`matnbr` returns the number `K` of matrix variables in the LMI problem described by `lmisys`.

See Also

`decinfo` | `decnbr` | `lmiinfo`

Introduced before R2006a

mat2dec

Extract vector of decision variables from matrix variable values

Syntax

```
decvec = mat2dec(lmisys,X1,X2,X3,...)
```

Description

Given an LMI system `lmisys` with matrix variables $X_1, \ldots, X_K$ and given values `X1,...,Xk` of $X_1, \ldots, X_K$, `mat2dec` returns the corresponding value `decvec` of the vector of decision variables. Recall that the decision variables are the independent entries of the matrices $X_1, \ldots, X_K$ and constitute the free scalar variables in the LMI problem.

This function is useful, for example, to initialize the LMI solvers `mincx` or `gevp`. Given an initial guess for $X_1, \ldots, X_K$, `mat2dec` forms the corresponding vector of decision variables `xinit`.

An error occurs if the dimensions and structure of `X1,...,Xk` are inconsistent with the description of $X_1, \ldots, X_K$ in `lmisys`.

Examples

Consider an LMI system with two matrix variables X and Y such that

- X is a symmetric block diagonal with one 2-by-2 full block and one 2-by-2 scalar block.
- Y is a 2-by-3 rectangular matrix.

Particular instances of X and Y are

$$X_0 = \begin{pmatrix} 1 & 3 & 0 & 0 \\ 3 & -1 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix}, \quad Y_0 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

and the corresponding vector of decision variables is given by

```
decv = mat2dec(lmisys,X0,Y0)

decv'

ans =
        1     3     -1     5     1     2     3     4     5     6
```

Note that `decv` is of length 10 since *Y* has 6 free entries while *X* has 4 independent entries due to its structure. Use `decinfo` to obtain more information about the decision variable distribution in *X* and *Y*.

See Also

`dec2mat` | `decinfo` | `decnbr`

Introduced before R2006a

mincx

Minimize linear objective under LMI constraints

Syntax

```
[copt,xopt] = mincx(lmisys,c,options,xinit,target)
```

Description

The function `mincx` solves the convex program

minimize $c^T x$ subject to $N^T L(x) N \leq M^T R(x) M$ (1-13)

where x denotes the vector of scalar decision variables.

The system of LMIs is described by `lmisys`. The vector `c` must be of the same length as x. This length corresponds to the number of decision variables returned by the function `decnbr`. For linear objectives expressed in terms of the matrix variables, the adequate `c` vector is easily derived with `defcx`.

The function `mincx` returns the global minimum `copt` for the objective $c^T x$, as well as the minimizing value `xopt` of the vector of decision variables. The corresponding values of the matrix variables is derived from `xopt` with `dec2mat`.

The remaining arguments are optional. The vector `xinit` is an initial guess of the minimizer `xopt`. It is ignored when infeasible, but may speed up computations otherwise. Note that `xinit` should be of the same length as `c`. As for `target`, it sets some target for the objective value. The code terminates as soon as this target is achieved, that is, as soon as some feasible x such that $c^T x \leq$ `target` is found. Set `options` to `[]` to use `xinit` and `target` with the default options.

Control Parameters

The optional argument `options` gives access to certain control parameters of the optimization code. In `mincx`, this is a five-entry vector organized as follows:

- `options(1)` sets the desired relative accuracy on the optimal value `lopt` (default = 10–2).
- `options(2)` sets the maximum number of iterations allowed to be performed by the optimization procedure (100 by default).
- `options(3)` sets the feasibility radius. Its purpose and usage are as for `feasp`.
- `options(4)` helps speed up termination. If set to an integer value $J > 0$, the code terminates when the objective $c^T x$ has not decreased by more than the desired relative accuracy during the last J iterations.
- `options(5) = 1` turns off the trace of execution of the optimization procedure. Resetting `options(5)` to zero (default value) turns it back on.

Setting `option(i)` to zero is equivalent to setting the corresponding control parameter to its default value. See `feasp` for more detail.

Tip for Speed-Up

In LMI optimization, the computational overhead per iteration mostly comes from solving a least-squares problem of the form

$$\min_x |Ax - b|$$

where x is the vector of decision variables. Two methods are used to solve this problem: Cholesky factorization of $A^T A$ (default), and QR factorization of A when the normal equation becomes ill conditioned (when close to the solution typically). The message

```
* switching to QR
```

is displayed when the solver has to switch to the QR mode.

Since QR factorization is incrementally more expensive in most problems, it is sometimes desirable to prevent switching to QR. This is done by setting `options(4) = 1`. While not guaranteed to produce the optimal value, this generally achieves a good trade-off between speed and accuracy.

Memory Problems

QR-based linear algebra (see above) is not only expensive in terms of computational overhead, but also in terms of memory requirement. As a result, the amount of memory required by QR may exceed your swap space for large problems with numerous LMI constraints. In such case, MATLAB issues the error

```
??? Error using ==> pds
Out of memory. Type HELP MEMORY for your options.
```

You should then ask your system manager to increase your swap space or, if no additional swap space is available, set `options(4) = 1`. This will prevent switching to QR and `mincx` will terminate when Cholesky fails due to numerical instabilities.

References

The solver `mincx` implements Nesterov and Nemirovski's Projective Method as described in

Nesterov, Yu, and A. Nemirovski, *Interior Point Polynomial Methods in Convex Programming: Theory and Applications*, SIAM, Philadelphia, 1994.

Nemirovski, A., and P. Gahinet, “The Projective Method for Solving Linear Matrix Inequalities,” *Proc. Amer. Contr. Conf.*, 1994, Baltimore, Maryland, pp. 840-844.

The optimization is performed by the C-MEX file `pds.mex`.

See Also

`dec2mat` | `decnbr` | `defcx` | `feasp` | `gevp` | `mincx`

Introduced before R2006a

mixsyn

Mixed-sensitivity H_∞ synthesis method for robust control loop-shaping design

Syntax

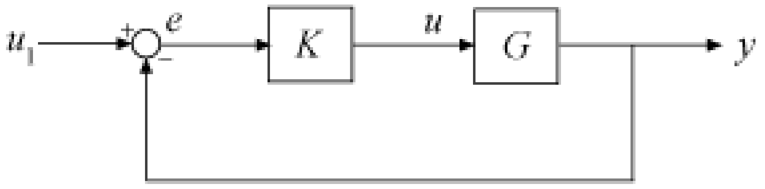
```
[K,CL,gamma,info] = mixsyn(G,W1,W2,W3)
[K,CL,gamma] = mixsyn(G,W1,W2,W3,gamTry)
[K,CL,gamma] = mixsyn(G,W1,W2,W3,gamRange)
[K,CL,gamma] = mixsyn( ___ ,opts)
[K,CL,gamma,info] = mixsyn( ___ )
```

Description

`[K,CL,gamma,info] = mixsyn(G,W1,W2,W3)` computes a controller that minimizes the H_∞ norm of the weighted closed-loop transfer function

$$M(s) = \begin{bmatrix} W_1 S \\ W_2 KS \\ W_3 T \end{bmatrix},$$

where $S = (I + GK)^{-1}$ and $T = (I - S)$ is the complementary sensitivity of the following control system.



You choose the weighting functions `W1,W2,W3` to shape the frequency responses for tracking and disturbance rejection, controller effort, and noise reduction and robustness, respectively. For details about how to choose weighting functions, see “Mixed-Sensitivity Loop Shaping”.

`mixsyn` computes the controller `K` that yields the minimum $||M(s)||_\infty$, which is returned as `gamma`. For the returned controller K,

$$\|S\|_\infty \leq \gamma\left|W_1^{-1}\right|$$

$$\|KS\|_\infty \leq \gamma\left|W_2^{-1}\right|$$

$$\|T\|_\infty \leq \gamma\left|W_3^{-1}\right|.$$

`[K,CL,gamma] = mixsyn(G,W1,W2,W3,gamTry)` calculates a controller for the target performance level `gamTry`. Specifying `gamTry` can be useful when the optimal controller performance is better than you need for your application. In that case, a less-than-optimal controller can have smaller gains and be better conditioned numerically. When `W1,W2,W3` capture the desired limits on the gains of S, KS, and T, use `gamtry` = 1 to just enforce those limits.

If `gamTry` is not achievable, `mixsyn` returns `[]` for `K` and `CL`, and `Inf` for `gamma`.

`[K,CL,gamma] = mixsyn(G,W1,W2,W3,gamRange)` searches the range `gamRange` for the best achievable performance. Specify the range with a vector of the form `[gmin,gmax]`. Limiting the search range can speed up computation by reducing the number of iterations performed by `mixsyn` to test different performance levels.

`[K,CL,gamma] = mixsyn( ___ ,opts)` specifies additional computation options. To create `opts`, use `hinfsynOptions`. Specify `opts` after all other input arguments.

`[K,CL,gamma,info] = mixsyn( ___ )` returns a structure containing additional information about the H_∞ synthesis computation. You can use this argument with any of the previous syntaxes.

Examples

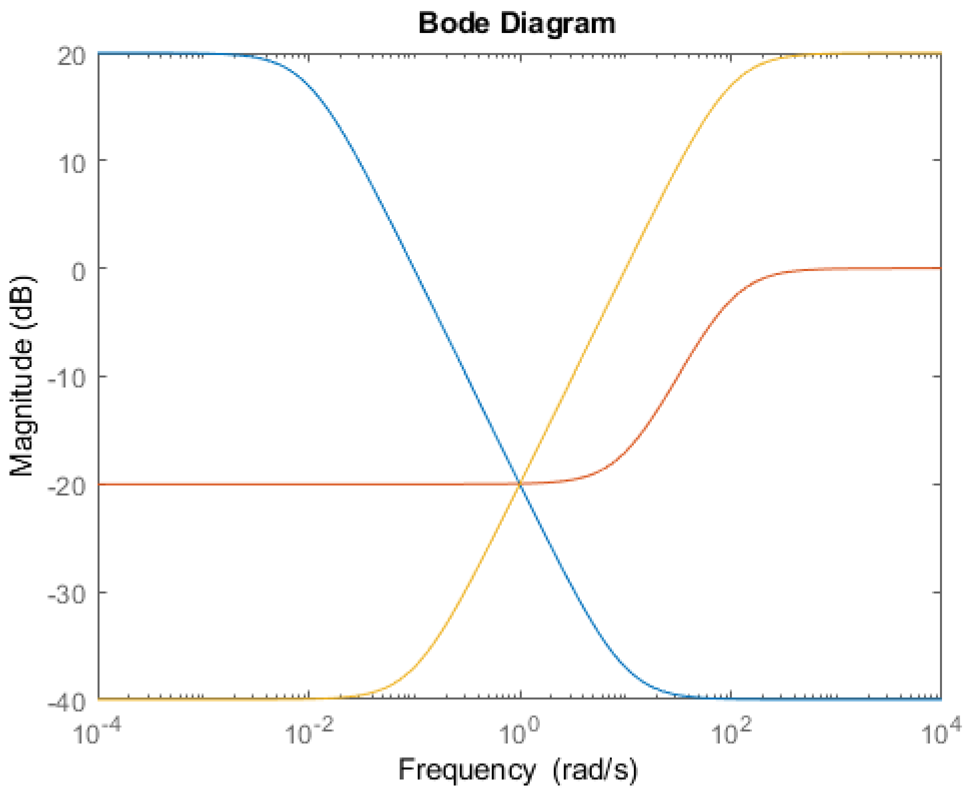
Loop Shaping with mixsyn

Use `mixsyn` for sensitivity and complementary sensitivity loop shaping. Create a plant model and weighting functions that:

- Shape the sensitivity function for reference tracking and disturbance rejection (`W1 = 1/S` large inside the control bandwidth).
- Shape the complementary sensitivity for robustness and noise attenuation (`W3 = 1/T` large outside the control bandwidth).
- Limit the control effort (`W2 = 1/KS` large inside the control bandwidth).

(For more information about choosing weighting functions, see “Mixed-Sensitivity Loop Shaping”.)

```
s = zpk('s');
G = (s-1)/(s+1)^2;

W1 = makeweight(10,[1 0.1],0.01);
W2 = makeweight(0.1,[32 0.32],1);
W3 = makeweight(0.01,[1 0.1],10);

bodemag(W1,W2,W3)
```

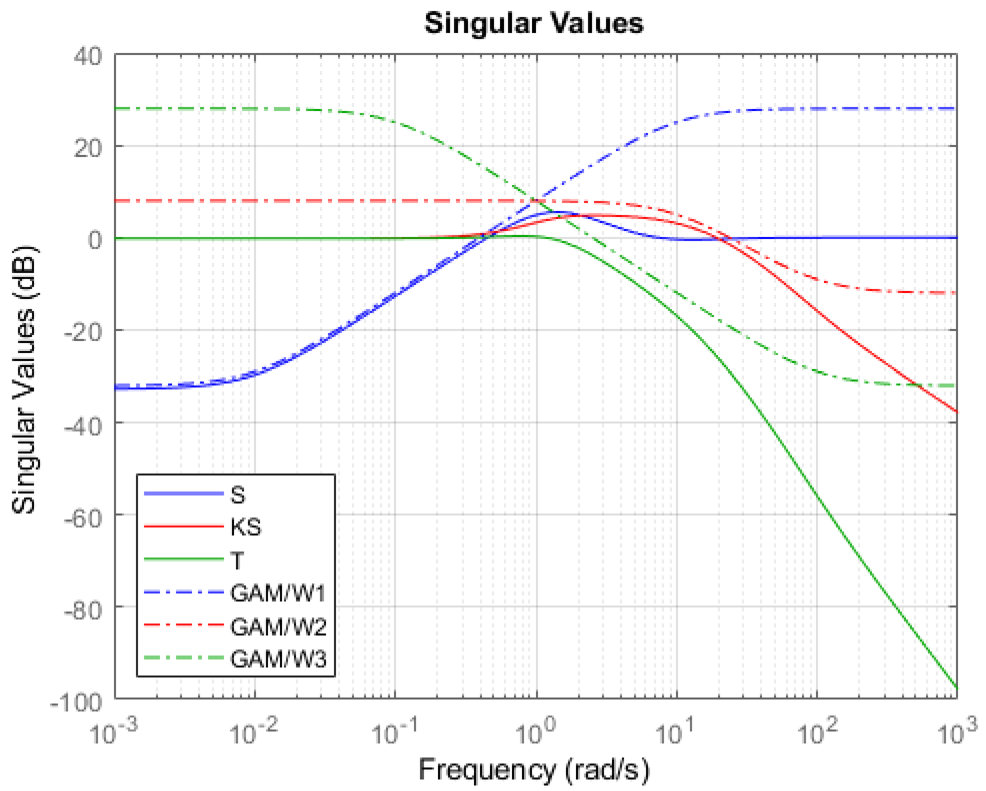


Design the controller.

```
[K,CL,gamma] = mixsyn(G,W1,W2,W3);
```

`mixsyn` shapes the singular values of the sensitivity function `S`, the complementary sensitivity function `T`, and the control effort `R = K*S`. Examine the results of the synthesis and the shapes of these transfer functions.

```
S = feedback(1,G*K);
KS = K*S;
T = 1-S;
sigma(S,'b',KS,'r',T,'g',gamma/W1,'b-.',ss(gamma/W2),'r-.',gamma/W3,'g-.',{1e-3,1e3})
legend('S','KS','T','GAM/W1','GAM/W2','GAM/W3','Location','SouthWest')
grid
```

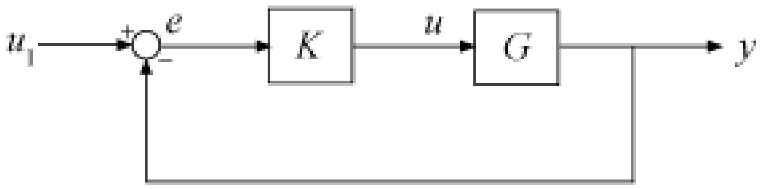


Input Arguments

`G` — Plant

dynamic system model

Plant, specified as a dynamic system model such as a state-space (`ss`) model. `G` can be any LTI model. `mixsyn` assumes the following control structure.



If `G` is a generalized state-space model with uncertain or tunable control design blocks, then `mixsyn` uses the nominal or current value of those elements.

`W1,W2,W3` — Weighting functions
dynamic system model | `[]`

Weighting functions, specified as dynamic system models. Choose the weighting functions `W1,W2,W3` to shape the frequency responses for tracking and disturbance rejection, controller effort, and noise reduction and robustness. Typically:

- For good reference-tracking and disturbance-rejection performance, choose `W1` large inside the control bandwidth to obtain small S.
- For robustness and noise attenuation, choose `W3` large outside the control bandwidth to obtain small T.
- To limit control effort in a particular frequency band, increase the magnitude of W_2 in this frequency band to obtain small KS.

If one of the weights is not needed, set it to `[]`. For instance, if you do not want to restrict control effort, use `W2 = []`.

Use `makeweight` to create weighting functions with the desired gain profiles. For details about choosing weighting functions, see “Mixed-Sensitivity Loop Shaping”.

If `G` has N_U inputs and N_Y outputs, then `W1,W2,W3` must be either SISO or square systems of size N_Y, N_U, and N_Y, respectively.

Because $S + T = I$, `mixsyn` cannot make both S and T small (less than 0 dB) in the same frequency range. Therefore, when you specify weights for loop shaping, there must be a frequency band in which both `W1` and `W3` are below 0 dB.

`gamTry` — Target performance level
positive scalar

Target performance level, specified as a positive scalar. `mixsyn` attempts to compute a controller such that the H_∞ of the weighted closed-loop system $M(s)$ does not exceed `gamTry`. If this performance level is achievable, then the returned controller has `gamma` ≤ `gamTry`. If `gamTry` is not achievable, `mixsyn` returns an empty controller.

gamRange — Performance range for search
`[0,Inf]` (default) | vector of form `[gmin,gmax]`

Performance range for search, specified as a vector of the form `[gmin,gmax]`. The `mixsyn` command tests only performance levels within that range. It returns a controller with performance:

- `gamma` ≤ `gmin`, when `gmin` is achievable.
- `gmin` < `gamma` < `gmax`, when `gmax` is achievable and but `gmin` is not.
- `gamma` = `Inf` when `gmax` is not achievable. In this case, `mixsyn` returns `[]` for `K` and `CL`.

If you know a range of feasible performance levels, specifying this range can speed up computation by reducing the number of iterations performed by `mixsyn` to test different performance levels.

`opts` — Options
`hinfsynOptions` object

Additional options for the computation, specified as an options object you create using `hinfsynOptions`.

Use `opts` to specify options for the underlying `hinfsyn` computation (see “Algorithms” on page 1-336). Available options include:

- Display algorithm progress at the command line.
- Turn off automatic scaling and regularization.
- Specify an optimization method.

For information about all options, see `hinfsynOptions`.

Output Arguments

K — Controller
`ss` model object | `[]`

Controller, returned as a state-space (`ss`) model object or `[]`.

If you supply `gamTry` or `gamRange` and the specified performance values are not achievable, then `K = []`.

CL — Augmented closed-loop transfer function
`ss` model object | `[]`

Augmented closed-loop transfer function, returned as a state-space (`ss`) model object or `[]`. The augmented closed-loop transfer function is given by

$$M(s) = \begin{bmatrix} W_1 S \\ W_2 K S \\ W_3 T \end{bmatrix},$$

where $S = (I + GK)^{-1}$ and $T = (I - S)$ is the complementary sensitivity of the unweighted control system. See “Mixed-Sensitivity Loop Shaping”.

The returned performance level `gamma` is the H_∞ norm of `CL`.

If you supply `gamTry` or `gamRange` and the specified performance levels are not achievable, then `CL = []`.

gamma — Controller performance
nonnegative scalar | `Inf`

Controller performance, returned as a nonnegative scalar value or `Inf`. This value is the performance achieved using the returned controller `K`, and is the H_∞ norm of `CL` (see `hinfnorm`). If you do not provide performance levels to test using `gamTry` or `gamRange`, then `gamma` is the best achievable performance level.

If you provide gamTry or gamRange, then gamma is the actual performance level achieved by the controller computed for the best passing performance level that hinfsyn tries. If the specified performance levels are not achievable, then gamma = Inf.

info — Synthesis data
structure | []

Additional synthesis data, returned as a structure or [] (if the specified performance level is not achievable). info contains data about the underlying hinfsyn computation used by mixsyn to minimize the H_∞ norm of the closed-loop transfer function $M(s)$. For details about the meaning of the fields of info, see the info output argument of hinfsyn.

Algorithms

mixsyn uses your weighting functions to generate an augmented plant P = augw(G,W1,W2,W3). It then invokes hinfsyn to find a controller that minimizes the H_∞ norm of the closed-loop transfer function $M(s) = LFT(P,K)$. For details, see “Mixed-Sensitivity Loop Shaping”.

Compatibility Considerations

Name,Value options are not recommended

Not recommended starting in R2019b

As of R2019b, using Name,Value syntax to specify options for mixsyn is not recommended. Instead, to set a target performance range, use the gamRange input argument. For other options, create an options set with hinfsynOptions.

The following table shows how to update your calls to mixsyn to use the recommended ways of specifying options.

Not Recommended	Recommended
`[K,CL,GAM] = mixsyn(___,'GMIN',gmin,'GMAX',gmax)`	`gamRange = [gmin gmax];` `[K,CL,GAM] = mixsyn(___,gamRange)`

Not Recommended	Recommended
`[K,CL,GAM] = mixsyn(___,'TOLGAM',tol)`	`opts = hinfsynOptions('RelTol',tol);` `[K,CL,GAM] = mixsyn(___,opts);`
`[K,CL,GAM] = mixsyn(___,'METHOD',meth)`	`opts = hinfsynOptions('Method',meth);` `[K,CL,GAM] = mixsyn(___,opts);`
`[K,CL,GAM] = mixsyn(___,'DISPLAY','on')`	`opts = hinfsynOptions('Display','on');` `[K,CL,GAM] = mixsyn(___,opts);`

For more information about these and additional options available for `mixsyn` computations, see `hinfsynOptions`.

`info` output argument changed

Behavior changed in R2019b

The fields of the optional output argument `info` changed in R2019b. The new fields of `info` are the same as those of `hinfsyn`. For more information about the change, see "info output argument changed" on page 1-224 on the `hinfsyn` reference page, which describes the same change for `hinfsyn` in R2018b.

See Also

`augw` | `hinfsyn` | `hinfsynOptions` | `makeweight`

Topics

"Mixed-Sensitivity Loop Shaping"

Introduced before R2006a

mkfilter

Generate Bessel, Butterworth, Chebyshev, or RC filter

Syntax

```
sys = mkfilter(fc,ord,type)
```

```
sys = mkfilter(fc,ord,type,psbndr)
```

Description

`sys = mkfilter(fc,ord,type)` returns a single-input, single-output analog low pass filter `sys` as an `ss` object. The cutoff frequency (Hertz) is `fc` and the filter order is `ord`, a positive integer. The argument `type` specifies the type of filter and can be one of the following:

type value	Description
`'butterw'`	Butterworth filter
`'cheby'`	Chebyshev filter
`'bessel'`	Bessel filter
`'rc'`	Series of resistor/capacitor filters

The dc gain of each filter (except even-order Chebyshev) is set to unity.

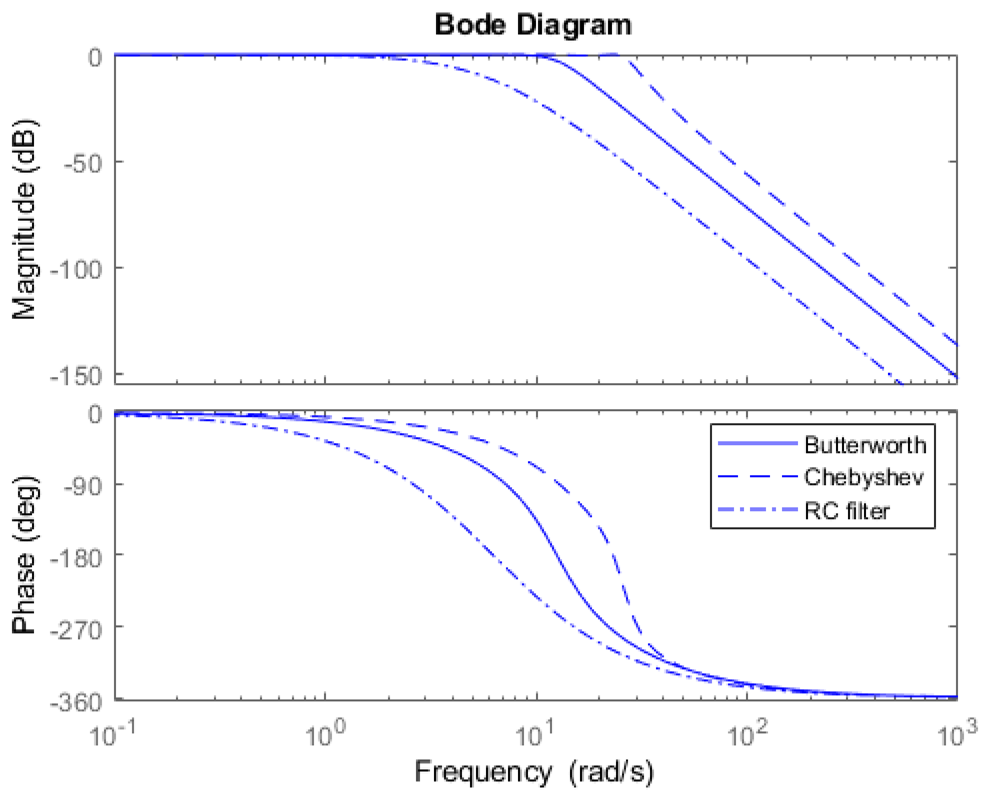
`sys = mkfilter(fc,ord,type,psbndr)` contains the input argument `psbndr` that specifies the Chebyshev passband ripple (in dB). At the cutoff frequency, the magnitude is -`psbndr` dB. For even-order Chebyshev filters the DC gain is also `-psbndr` dB.

Examples

Generate Filters

Generate several different types of filters and compare their frequency responses.

```
butw = mkfilter(2,4,'butterw');
cheb = mkfilter(4,4,'cheby',0.5);
rc = mkfilter(1,4,'rc');
bode(butw,'-',cheb,'--',rc,'-.')
legend('Butterworth','Chebyshev','RC filter')
```



Limitations

The Bessel filters are calculated using the recursive polynomial formula. This is poorly conditioned for high order filters (order > 8).

See Also

`augw`

Introduced before R2006a

mktito

Partition LTI system into two-input/two-output system

Syntax

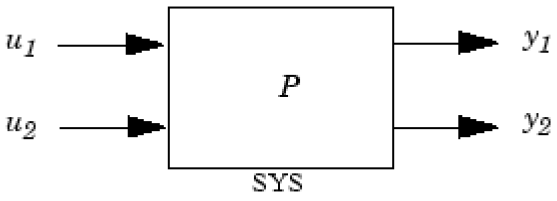
```
SYS=mktito(SYS,NMEAS,NCONT)
```

Description

`SYS=mktito(SYS,NMEAS,NCONT)` adds TITO (two-input/two-output) partitioning to LTI system `SYS`, assigning OutputGroup and InputGroup properties such that

$$\text{NMEAS} = \dim(y_2)$$

$$\text{NCONT} = \dim(u_2)$$



Any preexisting OutputGroup or InputGroup properties of `SYS` are overwritten. TITO partitioning simplifies syntax for control synthesis functions like `hinfsyn` and `h2syn`.

Examples

You can type

```
P=rss(2,4,5); P=mktito(P,2,2);
disp(P.OutputGroup); disp(P.InputGroup);
```

to create a 4-by-5 LTI system `P` with OutputGroup and InputGroup properties

```
    U1: [1 2 3]
    U2: [4 5]
    Y1: [1 2]
    Y2: [3 4]
```

Algorithms

```
[r,c]=size(SYS);
set(SYS,'InputGroup', struct('U1',1:c-NCONT,'U2',c-NCONT+1:c));
set(SYS,'OutputGroup',struct('Y1',1:r-NMEAS,'Y2',r-NMEAS+1:r));
```

See Also

`augw` | `h2syn` | `hinfsyn` | `sdhinfsyn`

Introduced before R2006a

modreal

Modal form realization and projection

Syntax

```
[G1,G2] = modreal(G,cut)
```

Description

`[G1,G2] = modreal(G,cut)` returns a set of state-space LTI objects `G1` and `G2` in modal form given a state-space `G` and the model size of `G1`, `cut`.

The modal form realization has its A matrix in block diagonal form with either 1x1 or 2x2 blocks. The real eigenvalues will be put in 1x1 blocks and complex eigenvalues will be put in 2x2 blocks. These diagonal blocks are ordered in ascending order based on eigenvalue magnitudes.

The complex eigenvalue a+bj is appearing as 2x2 block

$$\begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

This table describes input arguments for `modreal`.

Argument	Description
`G`	LTI model to be reduced.
`cut`	(Optional) an integer to split the realization. Without it, a complete modal form realization is returned

This table lists output arguments.

Argument	Description
`G1,G2`	LTI models in modal form

G can be stable or unstable. $G_1 = (A_1, B_1, C_1, D_1)$, $G_2 = (A_2, B_2, C_2, D_2)$ and $D_1 = D + C_2(-A_2)^{-1}B_2$ is calculated such that the system DC gain is preserved.

Examples

Given a continuous stable or unstable system, `G`, the following commands can get a set of modal form realizations depending on the split index -- `cut`:

```
rng(1234,'twister');
G = rss(50,2,2);
[G1,G2] = modreal(G,2); % cut = 2 for two rigid body modes
G1.D = zeros(2,2); % remove the DC gain of the system from G1
sigma(G,G1,G2)
```

Algorithms

Using a real eigen structure decomposition `reig` and ordering the eigenvectors in ascending order according to their eigenvalue magnitudes, we can form a similarity transformation out of these ordered real eigenvectors such that he resulting systems `G1` and/or `G2` are in block diagonal modal form.

Note This routine is extremely useful when model has jω-axis singularities, e.g., rigid body dynamics. It has been incorporated inside Hankel based model reduction routines - `hankelmr`, `balancmr`, `bstmr`, and `schurmr` to isolate those jω-axis poles from the actual model reduction process.

See Also

`balancmr` | `bstmr` | `hankelmr` | `hankelsv` | `ncfmr` | `reduce` | `schurmr`

Introduced before R2006a

msfsyn

Multi-model/multi-objective state-feedback synthesis

Syntax

```
[gopt,h2opt,K,Pcl,X] = msfsyn(P,r,obj,region,tol)
```

Description

Given an LTI plant `P` with state-space equations

$$\begin{cases} \dot{x} = Ax + B_1 w + B_2 u \\ z_\infty = C_1 x + D_{11} w + D_{12} u \\ z_2 = C_2 x + D_{22} u \end{cases}$$

`msfsyn` computes a state-feedback control $u = Kx$ that

- Maintains the RMS gain (H_∞ norm) of the closed-loop transfer function T_∞ from w to z_∞ below some prescribed value $\gamma_0 > 0$
- Maintains the H_2 norm of the closed-loop transfer function T_2 from w to z_2 below some prescribed value $\upsilon_0 > 0$
- Minimizes an H_2/H_∞ trade-off criterion of the form

$$\alpha \|T_\infty\|_\infty^2 + \beta \|T_2\|_2^2$$

- Places the closed-loop poles inside the LMI region specified by `region` (see `lmireg` for the specification of such regions). The default is the open left-half plane.

Set `r = size(d22)` and `obj =` [γ_0, ν_0, α, β] to specify the problem dimensions and the design parameters γ_0, ν_0, α, and β. You can perform pure pole placement by setting `obj = [0 0 0 0]`. Note also that z_∞ or z_2 can be empty.

On output, `gopt` and `h2opt` are the guaranteed H_∞ and H_2 performances, `K` is the optimal state-feedback gain, `Pcl` the closed-loop transfer function from w to $\begin{pmatrix} z_\infty \\ z_2 \end{pmatrix}$, and `X` the corresponding Lyapunov matrix.

The function `msfsyn` is also applicable to multi-model problems where `P` is a polytopic model of the plant:

$$\begin{cases} \dot{x} = A(t)x + B_1(t)w + B_2(t)u \\ z_\infty = C_1(t)x + D_{11}(t)w + D_{12}(t)u \\ z_2 = C_2(t)x + D_{22}(t)u \end{cases}$$

with time-varying state-space matrices ranging in the polytope

$$\begin{pmatrix} A(t) & B_1(t) & B_2(t) \\ C_1(t) & D_{11}(t) & D_{12}(t) \\ C_2(t) & 0 & D_{22}(t) \end{pmatrix} \in \text{Co}\left\{ \begin{pmatrix} A_k & B_k & C_k \\ C_{1k} & D_{11k} & D_{12k} \\ C_{2k} & 0 & D_{22k} \end{pmatrix} : k = 1, \ldots, K \right\}$$

In this context, `msfsyn` seeks a state-feedback gain that robustly enforces the specifications over the entire polytope of plants. Note that polytopic plants should be defined with `psys` and that the closed-loop system `Pcl` is itself polytopic in such problems. Affine parameter-dependent plants are also accepted and automatically converted to polytopic models.

See Also

`lmireg` | `psys`

Introduced before R2006a

mussv

Compute bounds on structured singular value (µ)

Syntax

```
bounds = mussv(M,BlockStructure)
```

```
[bounds,muinfo] = mussv(M,BlockStructure)
```

```
[bounds,muinfo] = mussv(M,BlockStructure,Options)
```

```
[ubound,q] = mussv(M,F,BlockStructure)
```

```
[ubound,q] = mussv(M,F,BlockStructure,'s')
```

Description

`bounds = mussv(M,BlockStructure)` calculates upper and lower bounds on the structured singular value, or µ, for a given block structure. `M` is a `double` array, an `frd` model, or a state-space (`ss`) model.

- If `M` is an N-D array (with `N` ≥ 3), then the computation is performed pointwise along the third and higher array dimensions.
- If `M` is a `frd` model, then the computations are performed pointwise in frequency (as well as any array dimensions).
- If `M` is a `ss` model, the computations are performed using state-space algorithms. Frequencies are adaptively selected, and upper bounds are guaranteed to hold over each interval between frequencies. `M` must be a single system, without array dimensions.

`BlockStructure` is a matrix specifying the perturbation block structure. `BlockStructure` has 2 columns, and as many rows as uncertainty blocks in the perturbation structure. The *i*-th row of `BlockStructure` defines the dimensions of the i'th perturbation block.

- If `BlockStructure(i,:) = [-r 0]`, then the *i*-th block is an `r`-by-`r` repeated, diagonal real scalar perturbation;

- if `BlockStructure(i,:) = [r 0]`, then the *i*-th block is an `r`-by-`r` repeated, diagonal complex scalar perturbation;
- if `BlockStructure(i,:) = [r c]`, then the *i*-th block is an `r`-by-`c` complex full-block perturbation.
- If `BlockStructure` is omitted, its default is `ones(size(M,1),2)`, which implies a perturbation structure of all 1-by-1 complex blocks. In this case, if `size(M,1)` does not equal `size(M,2)`, an error results.

If `M` is a two-dimensional matrix, then `bounds` is a `1-by-2` array containing an upper (first column) and lower (second column) bound of the structured singular value of `M`. For all matrices `Delta` with block-diagonal structure defined by `BlockStructure` and with norm less than `1/bounds(1)` (upper bound), the matrix `I - M*Delta` is not singular. Moreover, there is a matrix `DeltaS` with block-diagonal structure defined by `BlockStructure` and with norm equal to `1/bounds(2)` (lower bound), for which the matrix `I - M*DeltaS` is singular.

The format used in the 3rd output argument from `lftdata` is also acceptable for describing the block structure.

If `M` is an `frd`, the computations are always performed pointwise in frequency. The output argument `bounds` is a `1-by-2 frd` of upper and lower bounds at each frequency. Note that `bounds.Frequency` equals `M.Frequency`.

If `M` is an N-D array (either `double` or `frd`), the upper and lower bounds are computed pointwise along the 3rd and higher array dimensions (as well as pointwise in frequency, for `frd`). For example, suppose that `size(M)` is $r \times c \times d_1 \times \ldots \times d_F$. Then `size(bounds)` is $1 \times 2 \times d_1 \times \ldots \times d_F$. Using single index notation, `bounds(1,1,i)` is the upper bound for the structured singular value of `M(:,:,i)`, and `bounds(1,2,i)` is the lower bound for the structured singular value of `M(:,:,i)`. Here, any `i` between 1 and $d_1 \cdot d_2 \ldots d_F$ (the product of the d_k) would be valid.

If `M` is a `ss` model, `bounds` is returned as an `frd` model.

`bounds = mussv(M,BlockStructure,Options)` specifies computation options. `Options` is a character vector, containing any combination of the following characters:

Option	Meaning
`'a'`	Upper bound to greatest accuracy, using LMI solver. This is the default behavior when the number of decision variables within the D/G scalings is less than 45.

Option	Meaning
`'f'`	Force fast upper bound (typically not as tight as the default)
`'G'`	Force upper bound to use gradient method. This is the default behavior when the number of decision variables within the D/G scalings is greater than or equal to 45.
`'U'`	Upper-bound "only" (lower bound uses a fast/cheap algorithm).
`'g`*`N`*`'`	Use gain-based lower bound method multiple times. The value of *`N`* sets the number of times, according to `10+`*`N`*`*10`. For example, `'g6'` uses gain-based lower bound 70 times. Larger numbers typically give better lower bounds. If all uncertainty blocks described by `blk` are real, then the default is `'g1'`. If at least one uncertainty block is complex, then `mussv` uses power iteration lower bound by default.
`'i'`	Reinitialize lower bound computation at each new matrix (only relevant if `M` is `ND` array or `frd`).
`'m`*`N`*`'`	Randomly reinitialize lower bound iteration multiple times. *`N`* is an integer between 1 and 9. For example, `'m7'` randomly reinitializes the lower bound iteration 7 times. Larger numbers are typically more computationally expensive, but often give better lower bounds.
`'p'`	Use power iteration method to compute lower bound. When at least one of the uncertainty blocks described by `BlockStructure` is complex, then `'p'` is the default lower bound method.
`'s'`	Suppress progress information (silent).
`'d'`	Display warnings.
`'x'`	Decrease iterations in lower bound computation (faster but not as tight as default). Use `'U'` for an even faster lower bound.
`'an'`	Same as `'a'`, but without automatic prescaling.
`'o'`	Run "old" algorithms, from version 3.1.1 and before. Included to allow exact replication of earlier calculations.

`[bounds,muinfo] = mussv(M,BlockStructure)` returns `muinfo`, a structure containing more detailed information. The information within `muinfo` must be extracted using `mussvextract`.

Generalized Structured Singular Value

`ubound = mussv(M,F,BlockStructure)` calculates an upper bound on the generalized structured singular value (generalized μ) for a given block structure. `M` is a `double` or `frd` object. `M` and `BlockStructure` are as before. `F` is an additional (`double` or `frd`).

`ubound = mussv(M,F,BlockStructure,'s')` adds an option to run silently. Other options are ignored for generalized μ problems.

Note that in generalized structured singular value computations, only an upper bound is calculated. `ubound` is an upper bound of the generalized structured singular value of the pair `(M,F)`, with respect to the block-diagonal uncertainty described by `BlockStructure`. Consequently `ubound` is 1-by-1 (with additional array dependence, depending on `M` and `F`). For all matrices `Delta` with block-diagonal structure defined by `BlockStructure` and `norm<1/ubound`, the matrix `[I-Delta*M;F]` is guaranteed not to lose column rank. This is verified by the matrix `Q`, which satisfies `mussv(M+Q*F,BlockStructure,'a')<=ubound`.

Examples

See `mussvextract` for a detailed example of the structured singular value.

A simple example for generalized structured singular value can be done with random complex matrices, illustrating the relationship between the upper bound for μ and generalized μ, as well as the fact that the upper bound for generalized μ comes from an optimized μ upper bound.

`M` is a complex 5-by-5 matrix and `F` is a complex 2-by-5 matrix. The block structure `BlockStructure` is an uncertain real parameter δ_1, an uncertain real parameter δ_2, an uncertain complex parameter δ_3 and a twice-repeated uncertain complex parameter δ_4.

```
rng(929,'twister')
M = randn(5,5) + sqrt(-1)*randn(5,5);
F = randn(2,5) + sqrt(-1)*randn(2,5);
BlockStructure = [-1 0;-1 0;1 1;2 0];
[ubound,Q] = mussv(M,F,BlockStructure);
bounds = mussv(M,BlockStructure);
optbounds = mussv(M+Q*F,BlockStructure);
```

The quantities `optbounds(1)` and `ubound` should be extremely close, and significantly lower than `bounds(1)` and `bounds(2)`.

```
[optbounds(1) ubound]

ans =

    2.2070    2.1749

[bounds(1)  bounds(2)]

ans =

    4.4049    4.1960
```

Algorithms

The lower bound is computed using a power method, Young and Doyle, 1990, and Packard *et al.* 1988, and the upper bound is computed using the balanced/AMI technique, Young *et al.*, 1992, for computing the upper bound from Fan *et al.*, 1991.

Peter Young and Matt Newlin wrote the original function.

The lower-bound power algorithm is from Young and Doyle, 1990, and Packard *et al.* 1988.

The upper-bound is an implementation of the bound from Fan *et al.*, 1991, and is described in detail in Young *et al.*, 1992. In the upper bound computation, the matrix is first balanced using either a variation of Osborne's method (Osborne, 1960) generalized to handle *repeated scalar* and *full* blocks, or a Perron approach. This generates the standard upper bound for the associated complex μ problem. The Perron eigenvector method is based on an idea of Safonov, (Safonov, 1982). It gives the exact computation of μ for positive matrices with scalar blocks, but is comparable to Osborne on general matrices. Both the Perron and Osborne methods have been modified to handle *repeated scalar* and *full* blocks. Perron is faster for small matrices but has a growth rate of n^3, compared with less than n^2 for Osborne. This is partly due to the `MATLAB` implementation, which greatly favors Perron. The default is to use Perron for simple block structures and Osborne for more complicated block structures. A sequence of improvements to the upper bound is then made based on various equivalent forms of the upper bound. A number of descent techniques are used that exploit the structure of the problem, concluding with general purpose LMI optimization (Boyd *et al.*), 1993, to obtain the final answer.

The optimal choice of `Q` (to minimize the upper bound) in the generalized μ problem is solved by reformulating the optimization into a semidefinite program (Packard *et al.*, 1991).

References

[1] Boyd, S. and L. El Ghaoui, “Methods of centers for minimizing generalized eigenvalues,” *Linear Algebra and Its Applications*, Vol. 188–189, 1993, pp. 63–111.

[2] Fan, M., A. Tits, and J. Doyle, “Robustness in the presence of mixed parametric uncertainty and unmodeled dynamics,” *IEEE Transactions on Automatic Control*, Vol. AC–36, 1991, pp. 25–38.

[3] Osborne, E., “On preconditioning of matrices,” *Journal of Associated Computer Machines*, Vol. 7, 1960, pp. 338–345.

[4] Packard, A.K., M. Fan and J. Doyle, “A power method for the structured singular value,” *Proc. of 1988 IEEE Conference on Control and Decision*, December 1988, pp. 2132–2137.

[5] Safonov, M., “Stability margins for diagonally perturbed multivariable feedback systems,” *IEEE Proc.*, Vol. 129, Part D, 1992, pp. 251–256.

[6] Young, P. and J. Doyle, “Computation of with real and complex uncertainties,” *Proceedings of the 29th IEEE Conference on Decision and Control*, 1990, pp. 1230–1235.

[7] Young, P., M. Newlin, and J. Doyle, “Practical computation of the mixed problem,” *Proceedings of the American Control Conference*, 1992, pp. 2190–2194.

See Also

`mussvextract` | `robgain` | `robstab` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

mussvextract

Extract `muinfo` structure returned by `mussv`

Syntax

```
[VDelta,VSigma,VLmi] = mussvextract(muinfo)
```

Description

A structured singular value computation of the form

```
[bounds,muinfo] = mussv(M,BlockStructure)
```

returns detailed information in the structure `muinfo`. `mussvextract` is used to extract the compressed information within `muinfo` into a readable form.

The most general call to `mussvextract` extracts three usable quantities: `VDelta`, `VSigma`, and `VLmi`. `VDelta` is used to verify the lower bound. `VSigma` is used to verify the Newlin/Young upper bound and has fields `DLeft`, `DRight`, `GLeft`, `GMiddle`, and `GRight`. `VLmi` is used to verify the LMI upper bound and has fields `Dr, Dc, Grc`, and `Gcr`. The relation/interpretation of these quantities with the numerical results in `bounds` is described below.

Upper Bound Information

The upper bound is based on a proof that `det(I - M*Delta)` is nonzero for all block-structured matrices `Delta` with norm smaller than `1/bounds(1)`. The Newlin/Young method consists of finding a scalar β and matrices *D* and *G*, consistent with `BlockStructure`, such that

$$\bar{\sigma}\left(\left(I+G_L^2\right)^{-\frac{1}{4}}\left(\frac{D_L M D_R^{-1}}{\beta}-jG_M\right)\left(I+G_R^2\right)^{-\frac{1}{4}}\right)\leq 1$$

Here D_L, D_R, G_L, G_M, and G_R correspond to the `DLeft`, `DRight`, `GLeft`, `GMiddle`, and `GRight` fields respectively.

Because some uncertainty blocks and `M` need not be square, the matrices *D* and *G* have a few different manifestations. In fact, in the formula above, there are a left and right *D* and *G*, as well as a middle *G*. Any such β is an upper bound of `mussv(M,BlockStructure)`.

It is true that if `BlockStructure` consists only of complex blocks, then all *G* matrices will be zero, and the expression above simplifies to

$$\bar{\sigma}(D_L M D_R^{-1}) \leq \beta\,.$$

The LMI method consists of finding a scalar β and matrices *D* and *G*, consistent with `BlockStructure`, such that

$$M'D_r M - \beta^2 D_c + j(G_{cr}M - M'G_{rc}) \leq 0$$

is negative semidefinite. Again, *D* and *G* have a few different manifestations to match the row and column dimensions of M. Any such β is an upper bound of `mussv(M,BlockStructure)`. If `BlockStructure` consists only of complex blocks, then all *G* matrices will be zero, and negative semidefiniteness of $M'D_r M$-$\beta^2 D_c$ is sufficient to derive an upper bound.

Lower Bound Information

The lower bound of `mussv(M,BlockStructure)` is based on finding a “small” (hopefully the smallest) block-structured matrix `VDelta` that causes `det(I - M*VDelta)` to equal 0. Equivalently, the matrix `M*VDelta` has an eigenvalue equal to 1. It will always be true that the lower bound `(bounds(2))` will be the reciprocal of `norm(VDelta)`.

Examples

Suppose `M` is a 4-by-4 complex matrix. Take the block structure to be two 1-by-1 complex blocks and one 2-by-2 complex block.

```
rng(0,'twister')
M = randn(4,4) + sqrt(-1)*randn(4,4);
BlockStructure = [1 1;1 1;2 2];
```

You can calculate bounds on the structured singular value using the `mussv` command and extract the scaling matrices using `mussvextract`.

```
[bounds,muinfo] = mussv(M,BlockStructure);
[VDelta,VSigma,VLmi] = mussvextract(muinfo);
```

You can first verify the Newlin/Young upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dl` and `Dr`.

```
Dl = VSigma.DLeft


Dl =

    1.0000         0         0         0
         0    0.7437         0         0
         0         0    1.0393         0
         0         0         0    1.0393


Dr = VSigma.DRight


Dr =

    1.0000         0         0         0
         0    0.7437         0         0
         0         0    1.0393         0
         0         0         0    1.0393


[norm(Dl*M/Dr) bounds(1)]

ans =

    6.2950    6.2950
```

You can first verify the LMI upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dr` and `Dc`.

```
Dr = VLmi.Dr;
Dc = VLmi.Dc;
eig(M'*Dr*M - bounds(1)^2*Dc)

ans =

  -0.0000 - 0.0000i
 -17.7242 - 0.0000i
 -33.8550 + 0.0000i
 -41.2013 - 0.0000i
```

Note that `VDelta` matches the structure defined by `BlockStructure`, and the norm of `VDelta` agrees with the lower bound,

```
VDelta

VDelta =

   0.1301 - 0.0922i        0                   0                   0          
        0             -0.0121 - 0.1590i        0                   0          
        0                  0             -0.0496 - 0.0708i   0.1272 - 0.0075i
        0                  0              0.0166 - 0.0163i   0.0076 + 0.0334i

[norm(VDelta) 1/bounds(2)]

ans =

    0.1595    0.1595
```

and that `M*VDelta` has an eigenvalue exactly at 1.

```
eig(M*VDelta)

ans =

   1.0000 - 0.0000i
  -0.2501 - 0.1109i
   0.0000 + 0.0000i
  -0.3022 + 0.2535i
```

Keep the matrix the same, but change `BlockStructure` to be a 2-by-2 repeated, real scalar block and two complex 1-by-1 blocks. Run `mussv` with the `'C'` option to tighten the upper bound.

```
BlockStructure2 = [-2 0; 1 0; 1 0];
[bounds2,muinfo2] = mussv(M,BlockStructure2,'C');
```

You can compare the computed bounds. Note that `bounds2` should be smaller than `bounds`, because the uncertainty set defined by `BlockStructure2` is a proper subset of that defined by `BlockStructure`.

```
[bounds; bounds2]
```

```
ans =

    6.2950    6.2704
    5.1840    5.1750
```

You can extract the *D*, *G* and `Delta` from `muinfo2` using `mussvextract`.

```
[VDelta2,VSigma2,VLmi2] = mussvextract(muinfo2);
```

As before, you can first verify the Newlin/Young upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dl, Dr, Gl, Gm and Gr`.

```
Dl = VSigma2.DLeft;
Dr = VSigma2.DRight;
Gl = VSigma2.GLeft;
Gm = VSigma2.GMiddle;
Gr = VSigma2.GRight;
dmd = Dl*M/Dr/bounds2(1) - sqrt(-1)*Gm;
SL = (eye(4)+Gl*Gl)^-0.25;
SR = (eye(4)+Gr*Gr)^-0.25;
norm(SL*dmd*SR)
```

```
ans =

    1.0000
```

You can first verify the LMI upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dr`, `Dc, Grc` and `Gcr`.

```
Dr = VLmi2.Dr;
Dc = VLmi2.Dc;
Grc = VLmi2.Grc;
Gcr = VLmi2.Gcr;
eig(M'*Dr*M - bounds(1)^2 *Dc + j*(Gcr*M-M'*Grc))
```

```
ans =

 -69.9757 + 0.0000i
 -11.2139 - 0.0000i
 -19.2766 - 0.0000i
 -40.2869 - 0.0000i
```

`VDelta2` matches the structure defined by `BlockStructure`, and the norm of `VDelta2` agrees with the lower bound,

```
VDelta2
```

```
VDelta2 =

   0.1932                  0                   0                   0
        0             0.1932                   0                   0
        0                  0            -0.1781 - 0.0750i          0
        0                  0                   0            0.0941 + 0.1688i
```

```
[norm(VDelta2) 1/bounds2(2)]
```

```
ans =

    0.1932    0.1932
```

and that `M*VDelta2` has an eigenvalue exactly at 1.

```
eig(M*VDelta2)
```

```
  ans =

 1.0000 + 0.0000i
  -0.4328 + 0.1586i
   0.1220 - 0.2648i
  -0.3688 - 0.3219i
```

See Also

`mussv`

Introduced before R2006a

musyn

Robust controller design using mu synthesis

Syntax

```
[K,CLperf] = musyn(P,nmeas,ncont)
[K,CLperf,info] = musyn(P,nmeas,ncont)
[K,CLperf,info] = musyn(P,nmeas,ncont,Kinit)
[K,CLperf,info] = musyn( ___ ,opts)
```

```
[CL,CLperf] = musyn(CL0)
[CL,CLperf,info] = musyn(CL0)
[CL,CLperf,info] = musyn(CL0,blockvals)
[CL,CLperf,info] = musyn( ___ ,opts)
[CL,CLperf,info,runs] = musyn( ___ ,opts)
```

Description

`musyn` designs a robust controller for an uncertain plant using D-K iteration, which combines H_∞ synthesis (K step) with μ analysis (D step) to optimize closed-loop robust performance.

You can use `musyn` to:

- Synthesize "black box" unstructured robust controllers.
- Robustly tune a fixed-order or fixed-structure controller made up of tunable components such as PID controllers, state-space models, and static gains.

For additional information about performing μ synthesis and interpreting results, see “Robust Controller Design Using Mu Synthesis”.

Full-Order Centralized Controllers

`[K,CLperf] = musyn(P,nmeas,ncont)` returns a controller `K` that optimizes the robust performance of the uncertain closed-loop system `CL = lft(P,K)`. The plant `P` is an uncertain plant with the partitioned form

$$\begin{bmatrix} z \\ y \end{bmatrix} = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \begin{bmatrix} w \\ u \end{bmatrix},$$

where:

- w represents the disturbance inputs.
- u represents the control inputs.
- z represents the error outputs to be kept small.
- y represents the measurement outputs provided to the controller.

`nmeas` and `ncont` are the numbers of signals in y and u, respectively. y and u are the last outputs and inputs of `P`, respectively. The closed-loop system `CL = lft(P,K)` achieves the robust performance `CLperf`, which is the μ upper bound, the robust performance metric calculated by `musynperf`.

For this syntax, `musyn` uses `hinfsyn` for H_∞ synthesis (the K step).

`[K,CLperf,info] = musyn(P,nmeas,ncont)` returns additional information about each D-K iteration.

`[K,CLperf,info] = musyn(P,nmeas,ncont,Kinit)` initializes the D-K iteration process with the controller `Kinit`. To restart D-K iteration using the results of the *j*th iteration from a previous run, use `Kinit = info(j).K`.

`[K,CLperf,info] = musyn( ___ ,opts)` uses additional options for the D-K iteration and underlying `hinfsyn` computations. Use `musynOptions` to create the option set. You can use this syntax with any of the previous input and output argument combinations.

Fixed-Structure Controllers

`[CL,CLperf] = musyn(CL0)` optimizes the robust performance by tuning the free parameters in the tunable, uncertain closed-loop model `CL0`. The `genss` model `CL0` is an uncertain and tunable model of the closed-loop system whose robust performance you want to optimize. The model contains:

- Uncertain control design blocks such as `ureal` and `ultidyn` to represent the uncertainty
- Tunable control design blocks such as `tunablePID`, `tunableSS`, and `tunableGain` to represent the tunable components of the control structure

`musyn` returns the closed-loop model `CL` with the tunable control design blocks set to the tuned values. The best achieved robust performance is returned as `CLperf`.

For this syntax, `musyn` uses `hinfstruct` for H_∞ synthesis (*K* step).

`[CL,CLperf,info] = musyn(CL0)` also returns additional information about each D-K iteration.

`[CL,CLperf,info] = musyn(CL0,blockvals)` initializes the D-K iteration with the tunable block values in `blockvals`. You can specify the block values as a structure or by providing a closed-loop model whose blocks are tuned to the values you want to initialize. For instance, to use the tuned values obtained in a previous `musyn` run, set `blockvalues = CL`.

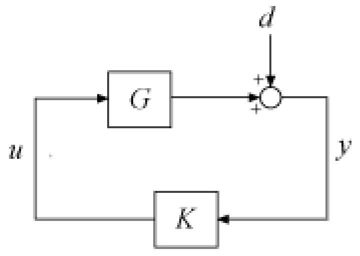
`[CL,CLperf,info] = musyn( ___ ,opts)` uses additional options for the D-K iteration and underlying `hinfstruct` computations. Use `musynOptions` to create the option set. You can use this syntax with any of the previous input and output argument combinations.

`[CL,CLperf,info,runs] = musyn( ___ ,opts)` also returns details about each independent tuning run when you use the `'RandomStart'` option of `musynOptions` to perform additional randomized runs.

Examples

Unstructured Robust Controller Synthesis

Synthesize a stabilizing robust controller K for the system in the following illustration, where the plant `G` includes some dynamic uncertainty. The controller must also reject disturbances injected at the plant output.



The nominal plant model G0 is an unstable first-order system.

```
G0 = tf(1,[1 -1]);
```

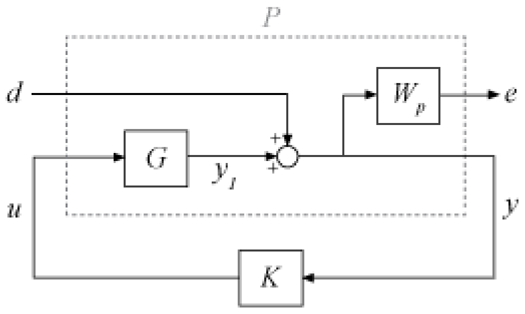
The uncertainty in G0 is as follows:

- At low frequency, below 2 rad/s, the plant can vary up to 25% from its nominal value.
- Around 2 rad/s, the percentage variation starts to increase, reaching 400% at approximately 32 rad/s.

Represent the frequency-dependent model uncertainty with the weight Wu and the uncertain LTI dynamic uncertainty InputUnc, an ultidyn control design block.

```
Wu = 0.25*tf([1/2 1],[1/32 1]);
InputUnc = ultidyn('InputUnc',[1 1]);
G = G0*(1+InputUnc*Wu);
```

musyn seeks a controller that optimizes robust performance from inputs to outputs. To set up this problem for musyn, then, insert a weighting function Wp that captures the disturbance rejection goal.

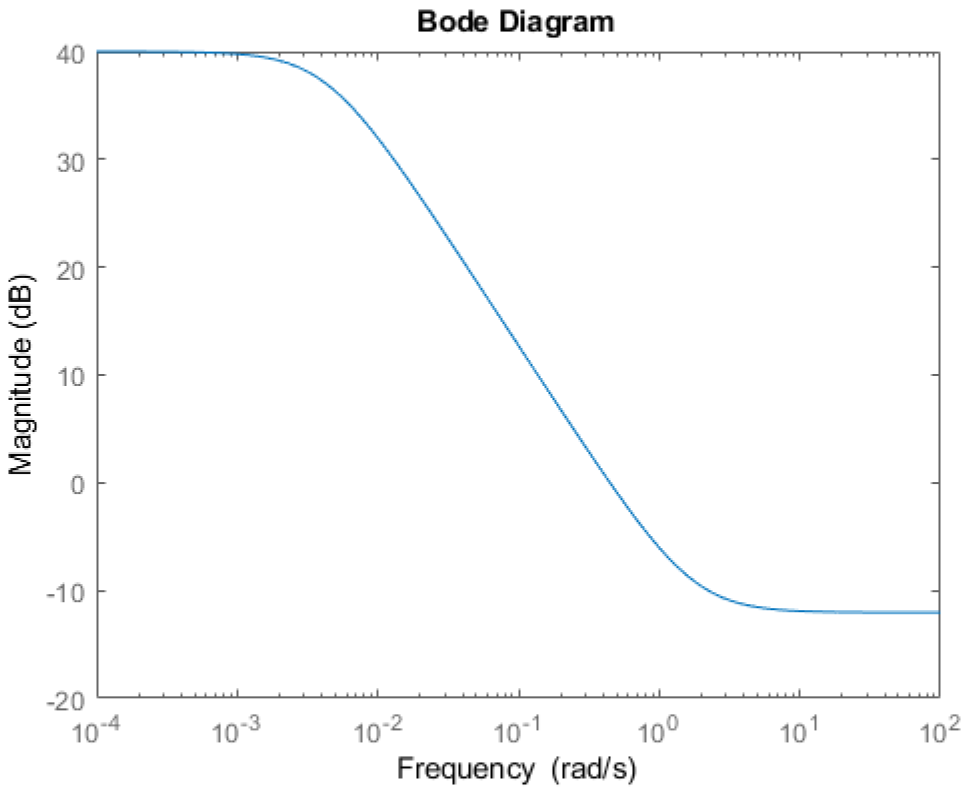


When you provide this augmented plant P to musyn, the function designs a controller that drives the transfer function from d to e below 1 at all frequencies. That transfer function is Wp/S, where S = 1 – GK is the sensitivity function. Thus, choose Wp to be the inverse of the desired sensitivity. For this example, choose Wp with:

- Low-frequency gain of 100 (40 dB)
- 0 dB crossover at 0.5 rad/s

- High-frequency gain of 0.25 (–12 dB)

```
Wp = makeweight(100,[1 0.5],0.25);
bodemag(Wp)
```



You can now construct the plant as shown in the block diagram by naming the signals, defining a sum block, and using `connect`. Construct the plant so that the control input is the last input, and the measurement output is the last output.

```
G.InputName = 'u';
G.OutputName = 'y1';
Wp.InputName = 'y';
Wp.OutputName = 'e';
SumD = sumblk('y = y1 + d');
```

```
inputs = {'d','u'};
outputs = {'e','y'};
P = connect(G,Wp,SumD,inputs,outputs);
```

Use `musyn` to design a controller `K` for this uncertain system.

```
nmeas = 1;
ncont = 1;
[K,CLperf,info] = musyn(P,nmeas,ncont);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter          K Step       Peak MU       D Fit             D
    1            1.345        1.344         1.36             8
    2           0.7923       0.7904       0.7961             4
    3           0.6789       0.6789       0.6857            10
    4           0.6572       0.6572       0.6598             8
    5           0.6538       0.6538       0.6542             8
    6           0.6532       0.6532       0.6533             8

Best achieved robust performance: 0.653
```

The display shows that the best achieved robust performance is about 0.65. This result means that the gain from `d` to `e` remains below 0.65 for up to 1/0.65 times the uncertainty specified in the plant. Thus, the controller achieves the robust performance objectives for the full range of modeled uncertainty. (For more information on interpreting `musyn` results, see “Robust Controller Design Using Mu Synthesis”.)

You can examine the robust performance using analysis commands such as `robgain` and `wcgainplot`. For instance, examine the worst-case gain of the closed-loop system.

```
CL = lft(P,K);
wcg = wcgain(CL)
```

```
wcg = struct with fields:
           LowerBound: 0.5283
           UpperBound: 0.5294
    CriticalFrequency: 0
```

This result confirms that the actual worst-case gain over the modeled uncertainty is about 0.53, which is within the robust performance of 0.65 guaranteed by `musyn`.

For this problem, the controller returned by `musyn` is fairly high order.

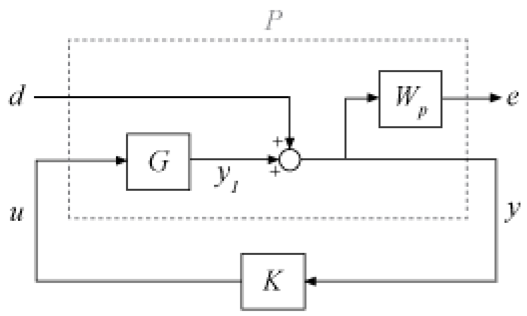
```
size(K)
```

```
State-space model with 1 outputs, 1 inputs, and 11 states.
```

You can try reducing the controller order with model-reduction commands such as `balred` or `reduce` to see whether you can maintain robust performance. (For an example, see the `musynperf` reference page.) Or, you can try specifying a lower order controller structure and use `musyn` to tune it. See “Robust Tuning of Fixed-Structure Controller” on page 1-365.

Robust Tuning of Fixed-Structure Controller

Tune a fixed-structure controller for the control system in “Unstructured Robust Controller Synthesis” on page 1-361, which shows how to use `musyn` to design an unstructured full-order centralized controller, and returns a controller of order 11. For this example, use the same control structure, as shown in the diagram, but restrict the structure of `K` to a fifth-order state-space model.



First, construct the same plant `P` as in “Unstructured Robust Controller Synthesis” on page 1-361. The plant is an uncertain plant `G` augmented by a sensitivity-weighting function `Wp`.

```
G0 = tf(1,[1 -1]);
Wu = 0.25*tf([1/2 1],[1/32 1]);
InputUnc = ultidyn('InputUnc',[1 1]);
```

```
G = G0*(1+InputUnc*Wu);
G.InputName = 'u';
G.OutputName = 'y1';

Wp = makeweight(100,[1 0.5],0.25);
Wp.InputName = 'y';
Wp.OutputName = 'e';

SumD = sumblk('y = y1 + d');
inputs = {'d','u'};
outputs = {'e','y'};
P = connect(G,Wp,SumD,inputs,outputs);
```

Create a `tunableSS` control design block to represent the fixed controller structure, a fifth-order state-space model.

```
C0 = tunableSS('K',5,1,1);
```

Form the closed-loop system, which is a generalized state-space (`genss`) model that has both a tunable block and a uncertain block.

```
CL0 = lft(P,C0)
```

```
CL0 =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs, 8 states, and
    InputUnc: Uncertain 1x1 LTI, peak gain = 1, 1 occurrences
    K: Parametric 1x1 state-space model, 5 states, 1 occurrences.

Type "ss(CL0)" to see the current value, "get(CL0)" to see all properties, and "CL0.Blo
```

Use `musyn` to tune the free entries of the controller.

```
[CL,CLperf,info] = musyn(CL0);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter          K Step        Peak MU       D Fit             D
    1           1.341          1.341        1.356            10
    2          0.7978         0.7951       0.8003             6
    3          0.6807         0.6806       0.6844             8
    4          0.6623           0.66       0.6659            10
    5          0.6601         0.6538       0.6624             6
    6          0.6592       0.6543       0.6596             6

Best achieved robust performance: 0.654
```

Even when you specify the structure of `K` as a fifth-order state-space model, `musyn` can find tuned parameter values that yield very similar robust performance to the 11th-order controller. You can try still lower controller orders to see whether the robust stability is preserved.

Input Arguments

`P` — Uncertain plant
`uss` model

Uncertain plant, specified as an uncertain state-space (`uss`) model. `P` has inputs [*w*;*u*] and outputs [*z*;*y*], where:

- *w* represents the disturbance inputs.
- *u* represents the control inputs.
- *z* represents the error outputs to be kept small.
- *y* represents the measurement outputs provided to the controller.

Construct `P` such that measurement outputs *y* are the last outputs, and the control inputs *u* are the last inputs.

`P` can optionally contain weighting functions (loop-shaping filters) that represent control objectives that you want the controller to robustly satisfy. For a detailed example that constructs such an augmented plant for μ synthesis, see “Robust Control of an Active Suspension”.

`nmeas` — Number of measurement outputs
1 (default) | nonnegative integer

Number of measurement output signals in the plant, specified as a nonnegative integer. The function takes the last `nmeas` plant outputs as the measurements *y*. The returned controller `K` has `nmeas` inputs.

`ncont` — Number of control inputs
1 (default) | nonnegative integer

Number of control input signals in the plant, specified as a nonnegative integer. The function takes the last `ncont` plant inputs as the controls *u*. The returned controller `K` has `ncont` outputs.

`Kinit` — Initial value of controller
dynamic system model

Initial value of the controller, specified as a dynamic system model, such as a state-space (`ss`) model. By default, `musyn` begins by computing an H_∞ controller for the nominal system. Use `Kinit` to start with a different controller. Setting `Kinit = info(j).K` uses the controller computed in the *j*th D-K iteration of the `musyn` run that produced `info`.

One use of this input argument is to continue iterating after `musyn` reaches the maximum number of iterations specified by the `'MaxIter'` option of `musynOptions`. If the display shows that `musyn` is still making progress when it stops iterating, you can run `musyn` again, starting with the last synthesized controller, to see how much more `musyn` can improve the robust performance.

`opts` — Additional options
`musynOptions` object

Additional options for the computation, specified as an options object you create using `musynOptions`. Available options include the following:

- Turn on a full display of algorithm progress that pauses after each D-K iteration so that you can examine intermediate results.
- Use mixed μ synthesis to treat real uncertain parameters as real rather than as complex, for a less conservative and possibly more robust controller.
- Set maximum orders for the functions used to fit the D and G scalings.
- Use parallel computing for independent optimization runs when tuning a fixed-structure controller.

For information about all available options, see `musynOptions`.

`CL0` — Closed-loop system with tunable controller elements
`genss` model

Closed-loop system with tunable controller elements, specified as a generalized state-space (`genss`) model with both uncertain and tunable control design blocks. Construct `CL0` by creating and interconnecting:

- Numeric LTI models representing the fixed components of the control system
- Uncertain control design blocks, such as `ureal` and `ultidyn` blocks, representing the uncertain components of the plant
- Optional LTI weighting functions (loop-shaping filters) that represent control objectives
- Tunable control design blocks such as `tunablePID`, `tunableSS`, and `tunableGain` to represent tunable components of the controller *C0*

For an example that shows how to build such a model, see “Build Tunable Control System Model With Uncertain Parameters”.

`blockvals` — Initial values of tunable parameters
structure | `genss` model

Initial values of the tunable parameters in `CL0`, specified as a structure or as a `genss` model. By default, `musyn` begins by tuning the controller parameters for the nominal system. Use `blockvals` to start with a different controller. Specify the initial parameter values as:

- A structure whose fields are the names of the tunable blocks, and whose values are control design blocks having the desired current values. For instance, if you have a tuned system `CL` obtained in a previous `musyn` run, you can initialize with the tuned values of the controller blocks in `CL` by setting `blockvals = CL.Blocks`.
- A `genss` model whose tunable blocks have the desired current value.

Setting `blockvals = info(j).K` uses the tuned values computed in the *j*th D-K iteration of the `musyn` run that produced `info`. For instance, if you have a tuned system `CL`, you can initialize with its tuned values by setting `blockvals = CL`. Such initialization can be useful for continuing iteration after `musyn` reaches the maximum number of iterations. If the display shows that `musyn` is still making progress when it stops iterating, you can run `musyn` again, starting with the last synthesized controller, to see how much more `musyn` can improve the robust performance.

Output Arguments

K — Controller
`ss` model

Controller that yields the robust H_∞ performance `CLperf`, returned as a state-space (`ss`) model. The controller has `nmeas` inputs and `ncont` outputs.

The order of `K` depends on the order of the fitting functions for the *D* and *G* scalings and the number of uncertain blocks in your system. For information on how to reduce the order of the returned controller, see “Improve Results of Mu Synthesis”.

`CLperf` — Best achieved robust performance
positive scalar

Best achieved robust performance, returned as a positive scalar. `musyn` attempts to optimize the controller `K` to minimize this value. The closed-loop gain from *w* to *z* remains below `CLperf` for uncertainty up 1/`CLperf` times the uncertainty specified in the plant. For instance, if `CLperf` is 1.125, then the closed-loop gain remains below 1.125 for up to 0.8 times the uncertainty specified in the plant.

For more information on the computation and interpretation of this quantity, see “Robust Performance Measure for Mu Synthesis”. For information on how to improve the best achieved robust performance, see “Improve Results of Mu Synthesis”.

`info` — Details of D-K iteration results
structure array

Details of D-K iteration results, returned as a structure array. `info(j)` contains the results of the *j*th D-K iteration in the run. `info` has the following fields.

Field	Description
`K`	Optimal controller found in the K step of this iteration, returned as a state-space (`ss`) model or as a structure containing tuned control design blocks. • For unstructured controller synthesis, `info(j).K` is a state-space model. • For fixed-structure controller tuning, `info(j).K` is a structure whose names are the names of the tunable blocks in `CL0`. The values are tunable control design blocks with current values set to the tuned value.

Field	Description
gamma	Optimized scaled H_∞ performance, returned as a scalar. This scaled performance is achieved by optimal controller info(j).K. The default command-window display shows this value in the K Step column. For details about the computation and interpretation of this quantity, see "Robust Performance Measure for Mu Synthesis".
KInfo	H_∞ synthesis data, returned as a structure. • For centralized, full-order controller synthesis, this structure is the same as the info output argument of hinfsyn. • For fixed-structure controller tuning, this structure is the same as the info output argument of hinfstruct.
PeakMu	Robust performance $\bar{\mu}$ of the closed-loop system with controller info(j).K, returned as a positive scalar value. The default command-window display shows this value in the Peak MU column . For details about the computation and interpretation of this quantity, see "Robust Performance Measure for Mu Synthesis".
DG	D and G scaling data from robust performance analysis in this iteration, returned as a structure with the following fields. • Frequency — Vector of frequencies for which μ analysis was performed • Dr, Dc, Gcr — Values of the scaling factors $D_r(\omega)$, $D_c(\omega)$, and $G_{cr}(\omega)$ at the corresponding frequencies. For more information about D and G scaling, see "Robust Performance Measure for Mu Synthesis".
dr,dc,PSI	Rational fit of the D and G scaling data, returned as ss models. For details about how musyn fits the scaling data, see "Robust Performance Measure for Mu Synthesis".
FitOrder	Orders of the functions used to fit the scaling data in this iteration, returned as a two-element vector. The two entries are the fit order for the D and G scalings, respectively. The default command-window display shows these values in the Fit Order column.

Field	Description
`PeakMuFit`	Scaled H_∞ performance achieved with `info(j).K` and the fitted D and G scalings, returned as a scalar. The default command-window display shows this value in the `DG Fit` column .

`CL` — Tuned closed-loop system
`genss` model

Tuned closed-loop system, returned as a generalized state-space `genss` model with the same uncertain and tunable control design blocks as `CL0`. The current values of the tunable blocks in `CL.Blocks` are set to the tuned values.

`runs` — Information about each independent run
structure array

Information about each independent randomized run, returned as a structure array. Use this output with fixed-structure μ synthesis when you set the `'RandomStart'` option of `musynOptions` to N > 0. That option causes `musyn` to perform multiple independent D-K iteration runs initialized from different values of controller parameters. `runs(j)` contains the results of the *j*th independent restart in the following fields.

Field	Description
`K`	Optimal controller found in this run, returned as a structure containing the tuned values of the control design blocks.
`muPerf`	Best achieved robust $H\infty$ performance for this run, returned as a positive scalar. The controller that `musyn` returns when you use multiple runs is the one for which `runs(j).muPerf` is smallest.
`Info`	Details of D-K iteration results, returned as a structure array. `runs(j).Info` contains fields corresponding to those of the `info` output of `musyn`, for the *j*th run.

Tips

- For more information on how to interpret the displays and outputs of `musyn`, see “Robust Controller Design Using Mu Synthesis”.
- For information about how to improve the results you obtain with `musyn`, see “Improve Results of Mu Synthesis”.

Algorithms

`musyn` uses an iterative process called D-K iteration. In this process, the function:

1. Uses H_∞ synthesis to find a controller that minimizes the closed-loop gain of the nominal system.
2. Performs a robustness analysis to estimate the robust H_∞ performance of the closed-loop system. This amount is expressed as a scaled H_∞ norm involving dynamic scalings called the D and G scalings (the D step).
3. Finds a new controller to minimize the scaled H_∞ norm obtained in step 2 (the K step).
4. Repeats steps 2 and 3 until the robust performance stops improving.

For more details about how this algorithm works, see “D-K Iteration Process”.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

When you use `musyn` to tune a fixed-structure robust controller, you can use parallel computing for the underlying `hinfstruct` computation. To run in parallel, set `'UseParallel'` to `true` using `musynOptions`.

See Also

`hinfstruct` | `hinfsyn` | `musynOptions` | `musynperf` | `wcgain`

Topics

“Robust Controller Design Using Mu Synthesis”
“Improve Results of Mu Synthesis”
“Simultaneous Stabilization Using Robust Control”
“Control of Aircraft Lateral Axis Using Mu Synthesis”
“Control of a Spring-Mass-Damper System Using Mixed-Mu Synthesis”

Introduced in R2019b

musynOptions

Options for `musyn`

Syntax

```
opts = musynOptions
opts = musynOptions(Name,Value)
```

Description

`opts = musynOptions` returns the default options for performing *μ* synthesis with the `musyn` command.

`opts = musynOptions(Name,Value)` creates an option set with the options specified by one or more name-value pair arguments.

Examples

Specify Algorithm Options for Mu Synthesis

Create an options set for `musyn` that turns on mixed-*μ* analysis for real uncertainty, restricts the *D* and *G* scalings for repeated `ureal` blocks so they are diagonal, and limits the maximum number of D-K iterations to 20.

```
opts = musynOptions('MixedMU','on','FullDG',false,'MaxIter',20)
```

```
opts = 
  musyn with properties:

          Display: 'short'
          MaxIter: 20
       TargetPerf: 0
          TolPerf: 0.0100
          MixedMU: 'on'
```

```
           FullDG: [0 0]
         FitOrder: [5 2]
    FrequencyGrid: [0x1 double]
        AutoScale: 'on'
       Regularize: 'on'
        LimitGain: 'on'
      RandomStart: 0
      UseParallel: 0
         MinDecay: 1.0000e-07
     MaxFrequency: Inf
```

Alternatively, start with the default options set, and use dot notation to change option values.

```
opts = musynOptions;
opts.MixedMu = 'on';
opts.FullDG  = false;
opts.MaxIter = 20;
```

You can now use `opts` as an input argument to `musyn` to perform μ synthesis using the specified options.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `opts = musynOptions('MaxIter',20,'MixedMU','on')` creates an option set for `musyn` to specify that the function take into account the presence of real uncertainty and to stop the D-K iteration process after at most 20 iterations.

General Options

`Display` — Flag to display progress of iterations
`'short'` (default) | `'full'` | `'off'`

Flag to display progress of D-K iterations and generate report in the command window, specified as the comma-separated pair consisting of `'Display'` and `'short'`, `'full'`, or `'off'`.

- `'short'` — Display a brief summary after each iteration.
- `'full'` — Pause after each iteration and display detailed results, including plots of *D* and *G* scaling data and the frequency dependence of μ.
- `'off'` — Turn off the display.

For details on how to interpret the default `'short'` display and the `'full'` display, see "Robust Performance Measure for Mu Synthesis".

Example: `opts = musynOptions('Display','off')` creates an option set for `musyn` that turns the display off.

`MaxIter` — Maximum number of D-K iterations

10 (default) | positive integer

Maximum number of D-K iterations, specified as the comma-separated pair consisting of `'MaxIter'` and a positive integer. `musyn` stops after the specified number of iterations or when the stopping tolerance specified by the '`TolPerf`' option is reached, whichever is fewer.

Example: `opts = musynOptions('MaxIter',20)` creates an option set for `musyn` that specifies a maximum of 20 iterations.

`TargetPerf` — Target robust H_∞ performance

0 (default) | nonnegative scalar

Target robust H_∞ performance, specified as the comma-separated pair consisting of `'TargetPerf'` and a nonnegative scalar. By default, `musyn` tries to drive the robust H_∞ performance (`PeakMu` in the default display) to zero in each iteration. If you set `'TargetPerf'` to a nonzero value, then D-K iteration terminates when the robust H_∞ performance drops below this target value. If you know your system can tolerate worse values of this performance metric, increasing this value can speed up the H_∞ part of the D-K iteration. For details about this performance metric, see `musynperf`.

Example: `opts = musynOptions('TargetPerf',1)` creates an option set for `musyn` that specifies a target H_∞ performance value of 1.

`TolPerf` — Stopping tolerance

0.01 (default) | 0 | nonnegative scalar

Stopping tolerance, specified as the comma-separated pair consisting of `'TolPerf'` and a nonnegative scalar. The `musyn` computation terminates when the robust H_∞ performance improves by less than this value over two consecutive iterations. Because of the limited accuracy of fitting the *D* and *G* scalings, reducing `'TolPerf'` below the default does not necessarily yield more precise results.

If `'TolPerf'` = 0, then `musyn` always performs the number of iterations specified by '`MaxIter`', regardless changes in the robust performance from iteration to iteration.

Example: `opts = musynOptions('TolPerf',0)` creates an option set for `musyn` that causes the function to always perform the number of iterations specified by `MaxIter`.

Options for D Step (μ Analysis)

`MixedMU` — Option to specify real or complex *μ* analysis

`'off'` (default) | `'on'`

Option to specify real or complex *μ* analysis, specified as the comma-separated pair consisting of `'MixedMU'` and `'off'` or `'on'`. By default, `musyn` treats all uncertainties as complex, which can result in overly conservative estimates for the upper bound on *μ*. If your plant has real uncertain parameters, try setting `'MixedMu'` to `'on'` to see if `musyn` returns a controller with better performance.

For more information, see “Improve Results of Mu Synthesis”.

Example: `opts = musynOptions('MixedMU','on')` creates an option set for `musyn` that causes the function to take into account the presence of real uncertainty.

`FullDG` — Structure of D and G scalings

`true` (default) | `false` | `[true false]` | `[false true]`

Structure of D and G scalings, specified as the comma-separated pair consisting of `'FullDG'` and `true`, `false`, `[true false]`, or `[false true]`.

By default, `musyn` uses full scalings for uncertain blocks that appear multiple times in the control system. Full scaling matrices can have frequency-dependent entries both on and off the diagonal. The alternative, diagonal scaling, is equivalent to treating each repeated block as an independent instance of the uncertain parameter. Therefore, full scaling is less conservative than diagonal scaling, and can yield better robust performance.

However, when blocks are repeated more than about four or five times, full scaling can be impractical, leading to lengthy computation, undesirably high-order controllers, or both.

In such cases, restricting scalings to diagonal can improve results. To do so, set `'FullDG'` to:

- `false` to limit both *D* and *G* scalings to diagonal.
- `[true false]` to use full *D* scaling but diagonal *G* scaling. This option is useful because fitting full *G* scalings is more likely to cause high-order controllers than full *D* scaling.
- `[false true]` to use full *G* scaling but diagonal *D* scaling. This option is useful if you need full *G* scaling to get a good fit, but observe that full *D* scaling does not improve `musyn` results.

For details about how the `musyn` algorithm uses *D* and *G* scalings, see “Robust Performance Measure for Mu Synthesis”.

Example: `opts = musynOptions('FullDG',false)` creates an option set for `musyn` that causes the function to use diagonal scalings for both D and G.

`FitOrder` — Maximum order for fitting D and G scaling data
`[5 2]` (default) | vector of two positive integers

Maximum order for fitting *D* and *G* scaling data, specified as the comma-separated pair consisting of `'FitOrder'` and a vector of two positive integers. The integers specify the maximum fit orders for the *D* and *G* scalings, respectively. (For details about how the `musyn` algorithm uses and fits scalings, see “Robust Performance Measure for Mu Synthesis”.)

For each iteration, `musyn` fits each entry in the *D* and *G* scaling matrices by a rational function whose order is automatically selected. By default, the maximum order is 5 for *D* scaling and 2 for *G* scaling. (*G* scaling is for dynamics in addition to dynamics needed to capture sign changes, so the final order of the *G* fit can be higher.) In general, the higher the order of these functions, the higher the order of the resulting controller.

To see whether you need to increase the maximum order, examine the `musyn` command-line display for a rough indication of fit quality. The `Peak MU` and `DG Fit` columns of the display give the best obtained robust performance before and after fitting, respectively. If the value for any given iteration increases drastically after fitting, you might obtain better results by increasing the maximum order.

Conversely, if the default maximum scaling order yields a good result, you can try lowering the maximum order to see if `musyn` returns a lower-order controller with similar performance.

Example: `opts = musynOptions('FitOrder',[3 2])` creates an option set for `musyn` that reduces the maximum fit order to 3 for the *D* scaling and 2 for the *G* scaling.

`FrequencyGrid` — Frequency grid used for *μ* analysis

`[]` (default) | vector of frequencies

Frequency grid used for *μ* analysis, specified as the comma-separated pair consisting of `'FrequencyGrid'` and an empty vector or a vector of frequencies in radians per second. By default, `musyn` computes an appropriate frequency grid based on system dynamics and the frequency dependence of the *D* and *G* scaling data. This default generally yields better results than a custom frequency grid, which restricts the computation to the specified frequencies regardless of the actual frequency dependence of the scaling data. Therefore, specifying frequencies is not recommended unless you know the frequency range in which *D* and *G* vary.

Options for K Step with Unstructured Controller (hinfsyn Controller Design)

`AutoScale` — Automatic plant scaling

`'on'` (default) | `'off'`

Automatic plant scaling, specified as the comma-separated pair consisting of `'AutoScale'` and one of the following:

- `'on'` — The underlying `hinfsyn` computation in the *K* step automatically scales the plant states, controls, and measurements to improve numerical accuracy. `musyn` always returns the controller in the original unscaled coordinates.
- `'off'` — `hinfsyn` does not change the plant scaling. Turning off scaling when you know your plant is well scaled can speed up the computation.

Example: `opts = musynOptions('AutoScale','off')` creates an option set for `musyn` that turns off automatic scaling for the underlying `hinfsyn` computation.

`Regularize` — Automatic regularization

`'on'` (default) | `'off'`

Automatic regularization of the plant, specified as the comma-separated pair consisting of `'Regularize'` and one of the following:

- `'on'` — The underlying `hinfsyn` computation in the *K* step automatically regularizes the plant to enforce certain nonsingularity requirements (see `hinfsyn`). Regularization is a process of adding extra disturbances and errors to handle singular problems.

- `'off'` — `hinfsyn` does not regularize the plant. Turning off regularization can speed up the computation when you know your problem is far enough from singular.

Example: `opts = musynOptions('Regularize','off')` creates an option set for `musyn` that turns off regularization for the underlying `hinfsyn` computation.

`LimitGain` — Limit on controller gains

`'on'` (default) | `'off'`

Limit on controller gains, specified as the comma-separated pair consisting of `'LimitGain'` and either `'on'` or `'off'`. For continuous-time plants, regularization of plant feedthrough matrices D_{12} or D_{21} (see `hinfsyn`) can result in controllers with large coefficients and fast dynamics. Use this option to automatically seek a controller with the same performance but lower gains and better conditioning.

Options for K Step with Structured Controller (hinfstruct Controller Design)

`RandomStart` — Number of starts with randomized parameter values

`0` (default) | positive integer

Number of additional optimization starts with randomized values of tunable controller parameters, specified as the comma-separated pair consisting of `'RandomStart'` and 0 or a positive integer.

By default, the underlying `hinfstruct` computation performs a single optimization run starting from the initial values of the tunable parameters. `hinfstruct` finds a local minimum of the gain minimization problem. To mitigate the risk of premature termination due to a local minimum that is not the best performing controller, you can perform multiple independent D-K iteration runs initialized from different values of controller parameters. Setting `RandomStart = N > 0` runs N additional `musyn` optimizations starting from N randomly generated parameter values.

Randomization only affects the initialization of the overall D-K iteration run. It does not affect each call to `hinfstruct` within a D-K iteration run.

When all runs are complete, `musyn` uses the best design that results from the multiple runs.

Use with `'UseParallel' = true` to distribute independent optimization runs among MATLAB workers (requires Parallel Computing Toolbox software).

Example: `opts = musynOptions('RandomStart',5)` creates an option set for `musyn` that runs the underlying `hinfstruct` computation a total of six times, using randomized initial values for the tunable parameters.

`UseParallel` — Option to enable parallel computing

`false` (default) | `true`

Option to enable parallel computing, specified as the comma-separated pair consisting of `'UseParallel'` and `false` or `true`. When you use `musyn` to tune a structured controller, set this option to `true` to distribute independent optimization runs among MATLAB workers in a parallel pool. If there is an available parallel pool, then the software performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

- If you select **Automatically create a parallel pool** in your Parallel Computing Toolbox preferences (Parallel Computing Toolbox), then the software starts a parallel pool using the settings in those preferences.
- If you do not select **Automatically create a parallel pool** in your preferences, then the software performs the optimization runs successively, without parallel processing.

Using parallel computing requires Parallel Computing Toolbox software.

Example: `opts = musynOptions('UseParallel',true)` creates an option set for `musyn` that turns on parallel computing for the underlying `hinfstruct` computation.

`MinDecay` — Minimum decay rate for closed-loop poles

`1e-7` (default) | positive scalar

Minimum decay rate for closed-loop poles, specified as the comma-separated pair consisting of `'MinDecay'` and a positive scalar value. The poles of the closed-loop system are constrained to satisfy `Re(p) < -MinDecay`. Increase this value to improve the stability of closed-loop poles that do not affect the closed-loop gain due to pole-zero cancellations.

Specify `MinDecay` in units of 1/`TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

`MaxFrequency` — Maximum closed-loop natural frequency

`Inf` (default) | positive scalar

Maximum closed-loop natural frequency, specified as the comma-separated pair consisting of `'MaxFrequency'` and `Inf` or a positive scalar value. Setting

`MaxFrequency` constrains the closed-loop poles to satisfy `|p| <  MaxFrequency`. To let `musyn` choose the closed-loop poles without such constraint, set `MaxFrequency = Inf`. To prevent unwanted fast dynamics or high-gain control, set `MaxFrequency` to a finite value.

Specify `MaxFrequency` in units of 1/`TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

Output Arguments

`opts` — Options for `musyn`
`musyn` options object

Options for the `musyn` computation, returned as a `musyn` options object. Use the object as an input argument to `musyn`. For example:

```
[K,CLperf,info] = musyn(P,nmeas,ncont,opts);
```

See Also
`musyn`

Topics
"Robust Controller Design Using Mu Synthesis"
"Improve Results of Mu Synthesis"
"Robust Performance Measure for Mu Synthesis"
"D-K Iteration Process"

Introduced in R2019b

musynperf

Robust H_∞ performance optimized by `musyn`

Syntax

```
[gamma,wcu] = musynperf(clp)
[gamma,wcu] = musynperf(clp,w)
[gamma,wcu] = musynperf( ___ ,opts)
[gamma,wcu,info] = musynperf( ___ )
```

Description

The robust H_∞ performance of an uncertain system is the smallest value γ such that the I/O gain of the system stays below γ for all modeled uncertainty up to size $1/\gamma$ (in normalized units). The `musyn` function synthesizes a robust controller by minimizing this quantity for the closed-loop system over all possible choices of controller. `musynperf` computes this quantity for a specified uncertain model. For a detailed discussion of robust H_∞ performance and how it is computed, see "Robust Performance Measure for Mu Synthesis".

`[gamma,wcu] = musynperf(clp)` calculates the robust H_∞ performance for an uncertain closed-loop system `clp`. The robust H_∞ performance is the smallest value γ for which the peak I/O gain stays below γ for all modeled uncertainty up to $1/\gamma$, in normalized units. For example, a value of $\gamma = 1.125$ implies the following:

- The I/O gain of `clp` remains less than 1.125 as long as the uncertain elements stay within 0.8 normalized units of their nominal values. In other words, for uncertain element values within 0.8 normalized units, the largest possible H_∞ norm is 1.125.
- For some perturbation of size 0.8 normalized units, the peak I/O gain is 1.125.

The peak I/O gain is the maximum I/O gain over all inputs, which is also the peak of the largest singular value over all frequencies and uncertainties. In other words, if Δ represents all possible values of the uncertain parameters in the closed-loop transfer function $CLP(j\omega)$, then

$$\gamma = \max_{\Delta} \max_{\omega} \sigma_{max}(CLP(j\omega)) .$$

The output structure `gamma` contains upper and lower bounds on the robust H_∞ performance and the critical frequency at which the I/O gain of `clp` reaches the lower bound. The structure `wcu` contains the uncertain-element values that drive the peak I/O gain to the lower bound.

`[gamma,wcu] = musynperf(clp,w)` computes the robust H_∞ performance at the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `musynperf` restricts the computation to the interval between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `musynperf` computes the H_∞ performance at the specified frequencies only.

`[gamma,wcu] = musynperf( ___ ,opts)` specifies additional options for the computation. Use `robOptions` to create `opts`. You can use this syntax with any of the previous input-argument combinations.

`[gamma,wcu,info] = musynperf( ___ )` returns a structure with additional information about the H_∞ performance values and the perturbations that drive the I/O gain to γ. See `info` for details about this structure. You can use this syntax with any of the previous input-argument combinations.

Examples

Reduce Synthesized Controller While Preserving Robust Performance

When you use `musyn` to synthesize an unstructured robust controller, the resulting controller often is of higher order than is necessary to achieve the desired robust performance. One way to mitigate this problem is to perform model reduction, using `musynperf` to test the robust performance of the reduced-order controller.

Create an uncertain model of the control system described in the example "Robust Tuning of Fixed-Structure Controller" on the `musyn` reference page.

```
G = tf(1,[1 -1]);
Wu = 0.25*tf([1/2 1],[1/32 1]);
InputUnc = ultidyn('InputUnc',[1 1]);
Gpert = G*(1+InputUnc*Wu);
Gpert.InputName = 'u';
Gpert.OutputName = 'y1';

Wp = makeweight(100,[1 0.5],0.25);
Wp.InputName = 'y';
Wp.OutputName = 'e';

SumD = sumblk('y = y1 + d');
inputs = {'d','u'};
outputs = {'e','y'};
P = connect(Gpert,Wp,SumD,inputs,outputs);

[K,CLperf] = musyn(P,1,1);
```

```
D-K ITERATION SUMMARY:
-----------------------------------------------------------------
                       Robust performance               Fit order
-----------------------------------------------------------------
  Iter          K Step        Peak MU       D Fit             D
    1           1.345         1.344         1.36              8
    2          0.7923        0.7904        0.7961             4
    3          0.6789        0.6789        0.6857            10
    4          0.6572        0.6572        0.6598             8
    5          0.6538        0.6538        0.6542             8
    6          0.6532        0.6532        0.6533             8

Best achieved robust performance: 0.653
```

```
N = order(K)
```

```
N = 11
```

`musyn` returns an 11th-order controller and the robust H_∞ performance `CLperf` of the closed-loop system using that controller. The best achieved robust performance of about 0.65 is good, but the controller order is high. Compute reduced-order controllers for orders ranging from 1 to full order.

```
Kred = reduce(K,1:N);
```

Find the lowest-order controller `Klow` with performance no worse than 1.05*`CLperf`, or 5% degradation compared to the full-order controller.

```
for k=1:N
   Klow = Kred(:,:,k);
   CL = lft(P,Klow);
   [gamma,~] = musynperf(CL);
   if gamma.UpperBound < 1.05*CLperf
      break
   end
end
order(Klow)

ans = 4
```

To validate the reduced-order controller, examine the robust H_∞ performance of the system using the simplified controller with that of the system using the full-order controller.

```
CLPlow = lft(P,Klow);
[gammalow,~] = musynperf(CLPlow);
gammalow.UpperBound

ans = 0.6613
```

The fourth-order controller achieves very similar robust H_∞ performance to the 11th-order controller returned by `musyn`.

Input Arguments

`clp` — Closed-loop uncertain system

`uss` | `ufrd` | `genss` | `genfrd`

Closed-loop uncertain system, specified as a `uss`, `ufrd`, `genss`, or `genfrd` model that contains uncertain elements. For `genss` or `genfrd` models, `musynperf` uses the current value of any tunable blocks and folds them into the known (not uncertain) part of the model.

`w` — Frequencies

`{wmin,wmax}` | vector

Frequencies at which to compute robust H_∞ performance, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the H_∞ performance at frequencies ranging between `wmin` and `wmax`.

- If `w` is a vector of frequencies, then the function computes the H_∞ performance at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

`opts` — Options for H_∞ performance computation
`robOptions` object

Options for computation of robust H_∞ performance, specified as `robOptions` object. Use `robOptions` to create the options object. The available options include settings that let you:

- Extract frequency-dependent H_∞ performance values.
- Examine the sensitivity of the H_∞ performance to each uncertain element.
- Improve the results of the calculation by setting certain options for the underlying `mussv` calculation. In particular, setting the option `'MussvOptions'` to `'mN'` can reduce the gap between the lower bound and upper bound. `N` is the number of restarts.

For more information about all available options, see `robOptions`.

Example: `robOptions('Sensitivity','on','MussvOptions','m3')`

Output Arguments

gamma — Robust H_∞ performance and critical frequency
structure

Robust H_∞ performance and critical frequency, returned as a structure containing the following fields:

Field	Description
LowerBound	Lower bound on the actual robust H_∞ performance γ, returned as a scalar value. The exact value of γ is guaranteed to be no smaller than LowerBound. In other words, some uncertain-element values of magnitude 1/LowerBound exist for which the I/O gain of clp reaches LowerBound. The function returns one such instance in wcu.
UpperBound	Upper bound on the actual robust H_∞ performance, returned as a scalar value. The exact value is guaranteed to be no larger than UpperBound. In other words, for all modeled uncertainty with normalized magnitude up to 1/UpperBound, the peak I/O gain of clp is less than UpperBound.
CriticalFrequency	Frequency at which the I/O gain reaches LowerBound, in rad/TimeUnit, where TimeUnit is the TimeUnit property of clp.

Use normalized2actual to convert the normalized uncertainty values 1/LowerBound or 1/UpperBound to actual deviations from nominal values.

wcu — Perturbations driving I/O gain to gamma.LowerBound
structure

Perturbations driving I/O gain to gamma.LowerBound, returned as a structure whose fields are the names of the uncertain elements of clp. Each field contains the actual value of the corresponding uncertain element. For example, if clp includes an uncertain matrix M and SISO uncertain dynamics delta, then wcu.M is a numeric matrix and wcu.delta is a SISO state-space model.

Use usubs(clp,wcu) to substitute these values for the uncertain elements in clp and obtain the corresponding dynamic system. This system has a peak gain of gamma.LowerBound.

Use actual2normalized to convert these actual uncertainty values to the normalized units in which 1/gamma.LowerBound or 1/gamma.UpperBound are expressed.

info — Additional information about γ values
structure

Additional information about the γ values, returned as a structure with the following fields.

Field	Description
`Frequency`	Frequency points at which `musynperf` returns γ values, returned as a vector. • If the `'VaryFrequency'` option of `robOptions` is `'off'`, then `info.Frequency` is the critical frequency, the frequency at which the I/O gain reaches `gamma.LowerBound`. If the smallest lower bound and the smallest upper bound on γ occur at different frequencies, then `info.Frequency` is a vector containing these two frequencies. • If the `'VaryFrequency'` option of `robOptions` is `'on'`, then `info.Frequency` contains the frequencies selected by `musynperf`. These frequencies are guaranteed to include the frequency at which the peak gain occurs. • If you specify a vector of frequencies `w` at which to compute γ, then `info.Frequency = w`. When you specify a frequency vector, these frequencies are not guaranteed to include the frequency at which the peak gain occurs. The `'VaryFrequency'` option is meaningful only for `uss` and `genss` models. `musynperf` ignores the option for `ufrd` and `genfrd` models.
`Bounds`	Lower and upper bounds on the actual γ values, returned as an array. `info.Bounds(:,1)` contains the lower bound at each corresponding frequency in `info.Frequency`, and `info.Bounds(:,2)` contains the corresponding upper bounds.

Field	Description
`WorstPerturbation`	Smallest perturbations at each frequency point in `info.Frequency`, returned as a structure array. The fields of `info.WorstPerturbation` are the names of the uncertain elements in `clp`. Each field contains the value of the corresponding element that drives the I/O gain to the corresponding lower bound at each frequency. For example, if `clp` includes an uncertain parameter `p` and SISO uncertain dynamics `delta`, then `info.WorstPerturbation.p` is a collection of numeric values and `info.WorstPerturbation.delta` is a collection of SISO state-space models.
`Sensitivity`	Sensitivity of γ to each uncertain element, returned as a structure when the `'Sensitivity'` option of `robOptions` is `'on'`. The fields of `info.Sensitivity` are the names of the uncertain elements in `clp`. Each field contains a percentage that measures how much the uncertainty in the corresponding element affects γ. For example, if `info.Sensitivity.p` is 50, then a given fractional change in the uncertainty range of `p` causes half as much fractional change in γ. If the `'Sensitivity'` option of `robOptions` is `'off'` (the default setting), then `info.Sensitivity` is `NaN`.

See Also
`actual2normalized` | `musyn` | `normalized2actual` | `robOptions` | `robgain` | `robstab` | `wcgain`

Topics
"Robust Performance Measure for Mu Synthesis"

Introduced in R2019b

ncfmargin

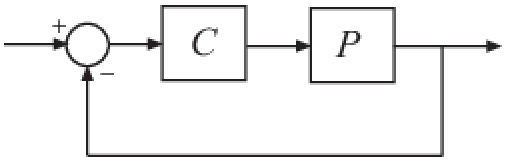
Calculate normalized coprime stability margin of plant-controller feedback loop

Syntax

```
[marg,freq] = ncfmargin(P,C)
[marg,freq] = ncfmargin(P,C,sign)
[marg,freq] = ncfmargin( ___ ,tol)
```

Description

`[marg,freq] = ncfmargin(P,C)` returns the normalized coprime stability margin of the multivariable feedback loop consisting of a controller `C` in negative feedback with a plant `P`:



The normalized coprime robust stability margin (also called the gap metric stability margin) is an indication of robustness to unstructured perturbations. Values greater than 0.3 generally indicate good robustness margins.

`[marg,freq] = ncfmargin(P,C,sign)` specifies the sign of the feedback connection assumed for the margin calculation. By default, `sign = -1`. Set `sign = +1` for positive-feedback interconnection.

`[marg,freq] = ncfmargin( ___ ,tol)` calculates the normalized coprime factor metric with the specified relative accuracy.

Examples

Normalized Coprime Stability Margin

Consider an unstable first-order plant, `p`, stabilized by high-gain and low-gain controllers, `cL` and `cH`.

```
p = tf(4,[1 -0.001]);
cL = 1;
cH = 10;
```

Compute the stability margin of the closed-loop system with the low-gain controller.

```
[margL,~] = ncfmargin(p,cL)
```

```
margL = 0.7069
```

Similarly, compute the stability margin of the closed-loop system with the high-gain controller.

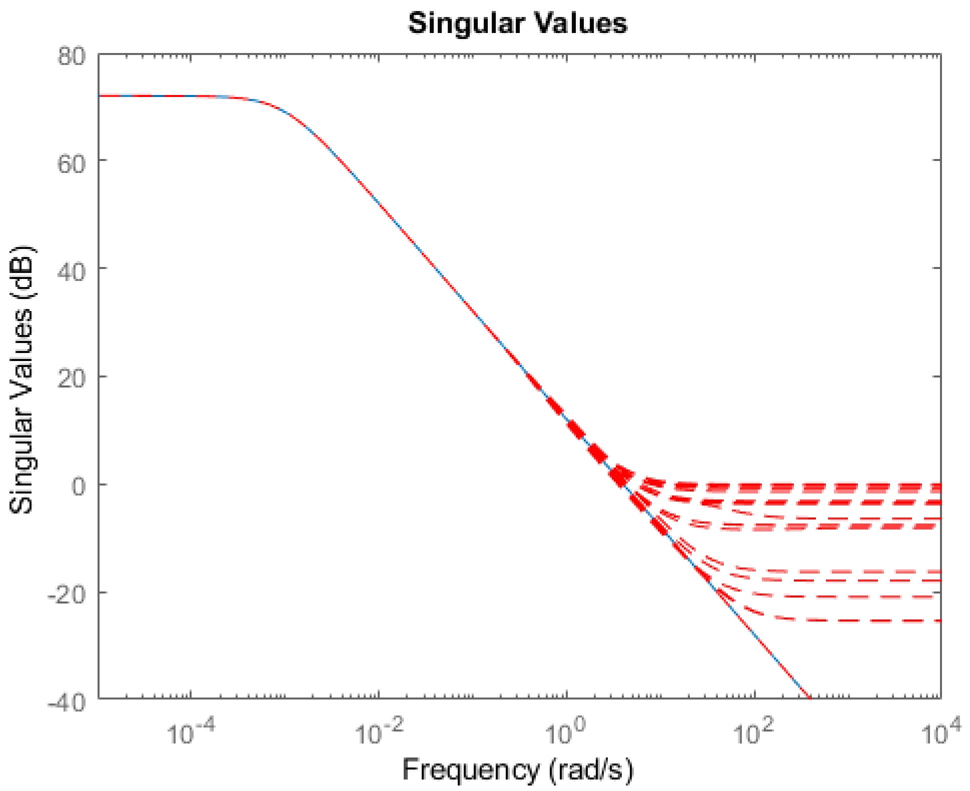
```
[margH,~] = ncfmargin(p,cH)
```

```
margH = 0.0995
```

The closed-loop systems with low-gain and high-gain controllers have normalized coprime stability margins of about 0.71 and 0.1, respectively. This result indicates that the closed-loop system with low-gain controller is more robust to unstructured perturbations than the system with the high-gain controller.

To observe this difference in robustness, construct an uncertain plant, `punc`, that has additional unmodeled dynamics at high frequency compared to the nominal plant.

```
punc = p + ultidyn('uncstruc',[1 1],'Bound',1);
sigma(p,punc,'r--')
```



Calculate the robust stability of the closed-loop systems formed by the uncertain plant and each controller.

```
[stabmargL,~] = robstab(feedback(punc,cL))
```

```
stabmargL = struct with fields:
           LowerBound: 0.9980
           UpperBound: 1
    CriticalFrequency: Inf
```

```
[stabmargH,~] = robstab(feedback(punc,cH))
```

```
stabmargH = struct with fields:
           LowerBound: 0.0998
```

```
          UpperBound: 0.1000
   CriticalFrequency: Inf
```

As expected, the robust stability analysis shows that the closed-loop system with low-gain controller is more robustly stable in the presence of the unmodeled LTI dynamics. In fact, this closed-loop system can tolerate almost 100% of the specified uncertainty. In contrast, closed-loop system with the high-gain controller can tolerate only about 10% of the specified uncertainty.

Compute Gap Metric and Stability Margin

Consider a plant and a stabilizing controller.

```
P1 = tf([1 2],[1 5 10]);
C = tf(4.4,[1 0]);
```

Compute the stability margin for this plant and controller.

```
b1 = ncfmargin(P1,C)
```

```
b1 = 0.1961
```

Next, compute the gap between `P1` and the perturbed plant, `P2`.

```
P2 = tf([1 1],[1 3 10]);
[gap,nugap] = gapmetric(P1,P2)
```

```
gap = 0.1391
```

```
nugap = 0.1390
```

Because the stability margin `b1 = b(P1,C)` is greater than the gap between the two plants, `C` also stabilizes `P2`. As discussed in “Gap Metrics and Stability Margins” on page 1-128, the stability margin `b2 = b(P2,C)` satisfies the inequality `asin(b(P2,C)) ≥ asin(b1)-asin(gap)`. Confirm this result.

```
b2 = ncfmargin(P2,C);
[asin(b2) asin(b1)-asin(gap)]
```

```
ans = 1×2
```

```
    0.0997    0.0579
```

Input Arguments

P — Plant
dynamic system model

Plant, specified as a dynamic system model. `P` can be SISO or MIMO, as long as `P*C` has the same number of inputs and outputs. `P` can be continuous time or discrete time. If `P` is a generalized state-space model (`genss` or `uss`) then `ncfmargin` uses the current or nominal value of all control design blocks in `P`.

C — Controller
dynamic system model

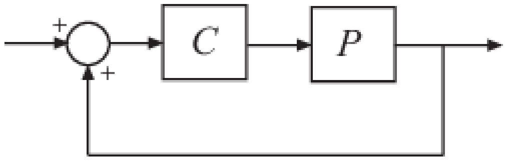
Plant, specified as a dynamic system model. `C` can be SISO or MIMO, as long as `P*C` has the same number of inputs and outputs. `C` can be continuous time or discrete time. If `C` is a generalized state-space model (`genss` or `uss`) then `ncfmargin` uses the current or nominal value of all control design blocks in `P`.

By default, `ncfmargin` assumes a negative-feedback interconnection between `P` and `C`. To compute the margins for a closed-loop system with positive feedback, use `[marg,freq] = ncfmargin(P,C,+1)`.

sign — Sign of feedback
`-1` (default) | `+1`

Sign of the feedback connection, specified as either `-1` or `+1`.

The default value, `sign = -1`, specifies negative feedback. Setting `sign = +1` assumes a positive feedback connection for the margin calculation, as in the following diagram.



`tol` — Relative accuracy
0.001 (default) | positive scalar less than 1

Relative accuracy for the computed margin, specified as a positive scalar value less than 1. The default value is 0.001, or 0.1% accuracy.

Output Arguments

`marg` — Normalized coprime robust stability margin
scalar in [0,1]

Normalized coprime robust stability margin, returned as a scalar in the range [0,1]. This quantity, also known as the gap metric stability margin, is an indicator of closed-loop robustness to unstructured perturbations. For negative-feedback control architecture, It is defined as:

$$b(P,C) = \left\| \begin{bmatrix} I \\ C \end{bmatrix} (I+PC)^{-1} [I \;\; P] \right\|_\infty^{-1} = \left\| \begin{bmatrix} I \\ P \end{bmatrix} (I+CP)^{-1} [I \;\; C] \right\|_\infty^{-1}.$$

Values greater than 0.3 generally indicate good robustness margins. If the closed-loop system is unstable, then `marg` = 0. The quantity $b(P,C)^{-1}$ is the signal gain from disturbances on the plant input and output to the input and output of the controller.

`freq` — Frequency at which margin occurs
scalar | `NaN`

Frequency at which the margin `marg` occurs, returned as a scalar. If the closed-loop system is unstable, then `freq` = `NaN`.

More About

Stability Margin and Gap Metrics

The stability margin $b(P,C)$ is related to the gap metric, which gives a numerical value $\delta(P_1,P_2)$ for the distance between two LTI systems (see `gapmetric`).

For both the gap and ν-gap metrics, the following robust performance result holds:

$$\arcsin b(P_2,C_2) \geq \arcsin b(P_1,C_1) - \arcsin \delta(P_1,P_2) - \arcsin \delta(C_1,C_2),$$

To interpret this result, suppose that a nominal plant P_1 is stabilized by controller C_1 with stability margin $b(P_1,C_1)$. Then, if P_1 is perturbed to P_2 and C_1 is perturbed to C_2, the stability margin is degraded by no more than the above formula.

The ν-gap is always less than or equal to the gap, so its predictions using the above robustness result are tighter.

Tips

- While `ncfmargin` assumes a negative-feedback loop, the `ncfsyn` command designs a controller for a positive-feedback loop. Therefore, to compute the margin using a controller designed with `ncfsyn`, use `[marg,freq] = ncfmargin(P,C,+1)`.

Algorithms

The computation of the normalized coprime stability margin is as described in Chapter 16 of [1].

References

[1] Zhou, K., Doyle, J.C., *Essentials of Robust Control*. London, UK: Pearson, 1997.

See Also

`diskmargin` | `gapmetric` | `ncfsyn` | `robstab` | `wcdiskmargin`

Introduced before R2006a

ncfmr

Balanced model truncation for normalized coprime factors

Syntax

```
GRED = ncfmr(G)
```

```
GRED = ncfmr(G,order)
```

```
[GRED,redinfo] = ncfmr(G,key1,value1,...)
```

```
[GRED,redinfo] = ncfmr(G,order,key1,value1,...)
```

Description

`ncfmr` returns a reduced order model GRED formed by a set of balanced normalized coprime factors and a struct array redinfo containing the left and right coprime factors of G and their coprime Hankel singular values.

Hankel singular values of coprime factors of such a stable system indicate the respective “state energy” of the system. Hence, reduced order can be directly determined by examining the system Hankel SV's.

With only one input argument `G`, the function will show a Hankel singular value plot of the original model and prompt for model order number to reduce.

The *left and right normalized coprime factors* are defined as [1]

- *Left Coprime Factorization*: $G = M_l^{-1}(s)N_l(s)$
- *Right Coprime Factorization*: $G = N_r(s)M_r^{-1}(s)$

where there exist stable $U_r(s)$, $V_r(s)$, $U_l(s)$ and $V_l(s)$ such that

$$U_rN_r + V_rM_r = I$$
$$N_lU_l + M_lV_l = I$$

The left/right coprime factors are stable, hence implies M_r(s) should contain as RHP-zeros all the RHP-poles of G(s). The coprimeness also implies that there should be no common RHP-zeros in N_r(s) and M_r(s), i.e., when forming $G = N_r(s)M_r^{-1}(s)$, there should be no pole-zero cancellations.

This table describes input arguments for `ncmfr`.

Argument	Description
`G`	LTI model to be reduced (without any other inputs will plot its Hankel singular values and prompt for reduced order)
`ORDER`	(Optional) Integer for the desired order of the reduced model, or optionally a vector packed with desired orders for batch runs

A batch run of a serial of different reduced order models can be generated by specifying `order = x:y`, or a vector of integers. By default, all the anti-stable part of a system is kept, because from control stability point of view, getting rid of unstable state(s) is dangerous to model a system. The `ncfmr` method allows the original model to have jω-axis singularities.

`'MaxError'` can be specified in the same fashion as an alternative for `'ORDER'`. In this case, reduced order will be determined when the sum of the tails of the Hankel singular values reaches the `'MaxError'`.

Argument	Value	Description
`'MaxError'`	A real number or a vector of different errors	Reduce to achieve H_∞ error. When present, `'MaxError'` overrides `ORDER` input.
`'Display'`	`'on'` or `'off'`	Display Hankel singular plots (default `'off'`).
`'Order'`	integer, vector or cell array	Order of reduced model. Use only if not specified as 2nd argument.

Weights on the original model input and/or output can make the model reduction algorithm focus on some frequency range of interests. But weights have to be stable, minimum phase, and invertible.

This table describes output arguments.

Argument	Description
GRED	LTI reduced order model, that becomes multi-dimensional array when input is a serial of different model order array.
REDINFO	A STRUCT array with 3 fields: • REDINFO.GL (left coprime factor) • REDINFO.GR (right coprime factor) • REDINFO.hsv (Hankel singular values)

G can be stable or unstable, continuous or discrete.

Examples

Given a continuous or discrete, stable or unstable system, G, the following commands can get a set of reduced order models based on your selections:

```
rng(1234,'twister');
G = rss(30,5,4);
G.D = zeros(5,4);
[g1, redinfo1] = ncfmr(G); % display Hankel SV plot
                           % and prompt for order (try 15:20)
[g2, redinfo2] = ncfmr(G,20);
[g3, redinfo3] = ncfmr(G,[10:2:18]);
[g4, redinfo4] = ncfmr(G,'MaxError',[0.01, 0.05]);
for i = 1:4
    figure(i)
    eval(['sigma(G,g' num2str(i) ');']);
end
```

Algorithms

Given a state space (A,B,C,D) of a system and k, the desired reduced order, the following steps will produce a similarity transformation to truncate the original state-space system to the k^{th} order reduced model.

1 Find the normalized coprime factors of G by solving Hamiltonian described in [1].

$$G_l = [N_l \; M_l]$$

$$G_r = \begin{bmatrix} N_r \\ M_r \end{bmatrix}$$

2 Perform k^{th} order square root balanced model truncation on G_l (or G_r) [2].

3 The reduced model `GRED` is:

$$\begin{bmatrix} \widehat{A} & \widehat{B} \\ \widehat{C} & \widehat{D} \end{bmatrix} = \begin{bmatrix} A_c - B_m C_l & B_n - B_m D_l \\ C_l & D_l \end{bmatrix}$$

where

$N_l(:= \quad A_c, \quad B_n, \quad C_c, \quad D_n)$

$M_l := (A_c, B_m, C_c, D_m)$

$C_l = (D_m)^{-1} C_c$

$D_l = (D_m)^{-1} D_n$

References

[1] M. Vidyasagar. *Control System Synthesis - A Factorization Approach.* London: The MIT Press, 1985.

[2] M. G. Safonov and R. Y. Chiang, “A Schur Method for Balanced Model Reduction,” *IEEE Trans. on Automat. Contr.*, vol. AC-2, no. 7, July 1989, pp. 729-733.

See Also

`balancmr` | `bstmr` | `hankelmr` | `hankelsv` | `reduce` | `schurmr`

Introduced before R2006a

ncfsyn

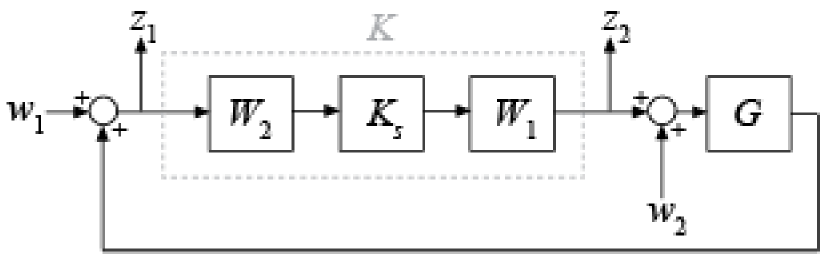
Loop shaping design using Glover-McFarlane method

Syntax

```
[K,CL,gamma,info] = ncfsyn(G)
[K,CL,gamma,info] = ncfsyn(G,W1)
[K,CL,gamma,info] = ncfsyn(G,W1,W2)
[K,CL,gamma,info] = ncfsyn(G,W1,W2,'ref')
```

Description

`ncfsyn` implements a method for designing controllers that uses a combination of loop shaping and robust stabilization as proposed in [1]-[2]. The function computes the Glover-McFarlane H_∞ normalized coprime factor loop-shaping controller K for a plant G with pre-compensator and post-compensator weights W_1 and W_2. The function assumes the positive feedback configuration of the following illustration.



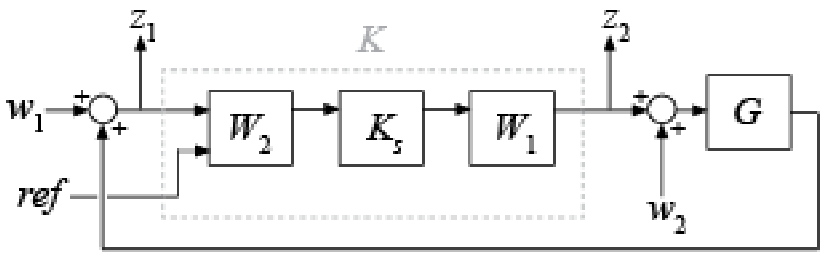
To specify negative feedback, replace G by $–G$. The controller K_s stabilizes a family of systems given by a ball of uncertainty in the normalized coprime factors of the shaped plant $G_s = W_2GW_1$. The final controller K returned by `ncfsyn` is obtained as $K = W_1K_sW_2$.

`[K,CL,gamma,info] = ncfsyn(G)` computes the Glover-McFarlane H_∞ normalized coprime factor loop-shaping controller `K` for the plant `G`, with $W_1 = W_2 = I$. `CL` is the closed-loop system from the disturbances w_1 and w_2 to the outputs z_1 and z_2. The function also returns the H_∞ optimal cost `gamma`, and a structure containing additional information about the result.

`[K,CL,gamma,info] = ncfsyn(G,W1)` computes the controller using the pre-compensator weight you specify in `W1`, and $W_2 = I$.

`[K,CL,gamma,info] = ncfsyn(G,W1,W2)` computes the controller using the specified pre-compensator weight `W1` and post-compensator weight `W2`.

`[K,CL,gamma,info] = ncfsyn(G,W1,W2,'ref')` computes the controller with a reference command, as in the system of the following illustration.



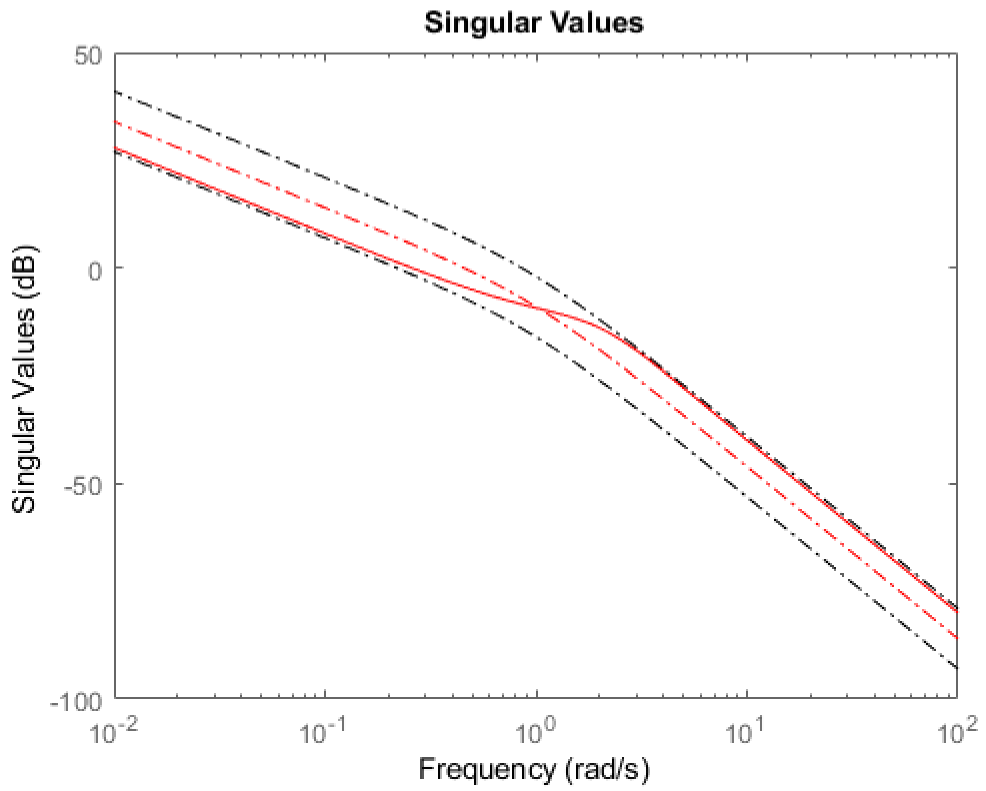
When you use this syntax, the closed-loop system `CL` represents the response from [w_1; w_2; *ref*] to [z_1; z_2].

Examples

Loop Shaping With ncfsyn

The following code shows how `ncfsyn` can be used for loop-shaping.

```
s = zpk('s');
G = (s-1)/(s+1)^2;
W1 = 0.5/s;
[K,CL,GAM] = ncfsyn(G,W1);
sigma(G*K,'r',G*W1,'r-.',G*W1*GAM,'k-.',G*W1/GAM,'k-.')
```



The singular value plot of the achieved loop `G*K` is equal to that of the target loop `G*W1` to within plus or minus `GAM` (in dB).

Input Arguments

G — Plant
dynamic system model

Plant, specified as a dynamic system model such as a state-space (`ss`) model. If `G` is a generalized state-space model with uncertain or tunable control design blocks, then `ncfsyn` uses the nominal or current value of those elements. `G` must have the same number of inputs and outputs.

W1 — Pre-compensator weight
`eye(N)` (default) | LTI model

Pre-compensator weight, specified as:

- Identity matrix `eye(N)`, where `N` is the number of inputs or outputs in `G`.
- SISO minimum-phase LTI model. In this case, `ncfsyn` uses the same weight for every loop channel.
- MIMO minimum-phase LTI model of the same I/O dimensions as `G`.

Select a pre-compensator and post-compensator weights W_1 and W_2 such that the gain of the shaped plant $G_s = W_2GW_1$ is sufficiently high at frequencies where good disturbance attenuation is required, and sufficiently low at frequencies where good robust stability is required.

W2 — Post-compensator weight
I (default) | LTI model

Post-compensator weight, specified as the identity matrix `eye(N)` or a SISO or MIMO LTI model. The considerations for specifying `W2` are the same as those for `W1`.

Output Arguments

K — H_∞-optimal loop-shaping controller
state-space (`ss`) model

H_∞-optimal loop-shaping controller, returned as a state-space (`ss`) model with the same I/O dimensions as `G`. The optimal controller $K = W_1K_sW_2$. See “Algorithms” on page 1-408.

`CL` — Optimal closed-loop system
state-space (`ss`) model

Optimal closed-loop system from the disturbances w_1 and w_2 to the outputs z_1 and z_2, returned as a state-space model. The closed-loop system is given by:

$$\begin{bmatrix} I \\ K \end{bmatrix}(I - GK)^{-1}[I,\ G].$$

`gamma` — H_∞ optimal cost
positive scalar value

H_∞ optimal cost, returned as a positive scalar value greater than 1. The optimal cost is `hinfnorm(CL)`. The optimal controller K_s is such that the singular-value plot of the shaped loop $L_s = W_2GW_1K_s$ optimally matches the target loop shape G_s to within a factor of `gamma`.

`gamma` is related to the normalized coprime stability margin of the system by `gamma = 1/ncfmargin(Gs,-K)`. Thus, `gamma` gives a good indication of robustness of stability to a wide class of unstructured plant variations, with values in the range 1 < `gamma` < 3 corresponding to satisfactory stability margins for most typical control system designs.

`info` — Additional information
structure

- `emax` — `nugap` robustness metric, `emax = 1/ gamma` (see `gapmetric`)
- `Gs` — Shaped plant $G_s = W_2GW_1$
- `Ks` — Optimal controller for shaped plant `Gs`. The final controller is $K = W_1K_sW_2$. See “Algorithms” on page 1-408 for details.

Tips

- While `ncfmargin` assumes a negative-feedback loop, the `ncfsyn` command designs a controller for a positive-feedback loop. Therefore, to compute the margin using a controller designed with `ncfsyn`, use `[marg,freq] = ncfmargin(G,K,+1)`.

Algorithms

The returned controller $K = W_1K_sW_2$, where K_s is an optimal H_∞ controller that minimizes the H_∞ cost

$$\gamma(K_S) = \left\| \begin{bmatrix} I \\ K_S \end{bmatrix} (I - G_S K_S)^{-1} [I, G_S] \right\|_\infty = \left\| \begin{bmatrix} I \\ G_S \end{bmatrix} (I - K_S G_S)^{-1} [I, K_S] \right\|_\infty .$$

The optimal performance is the minimal cost

$$\gamma: = \min_{K_S} \gamma(K_S) .$$

Suppose that $G_s = NM^{-1}$, where $N(j\omega)^*N(j\omega) + M(j\omega)^*M(j\omega) = I$, is a normalized coprime factorization (NCF) of the weighted plant model G_s. Then, theory ensures that the control system remains robustly stable for any perturbation $\tilde{G}_S$ to G_s of the form

$$\tilde{G}_S = (N + \Delta_1)(M + \Delta_2)^{-1}$$

where Δ_1, Δ_2 are a stable pair satisfying

$$\left\| \begin{bmatrix} \Delta_1 \\ \Delta_2 \end{bmatrix} \right\|_\infty < MARG: = \frac{1}{\gamma} .$$

The closed-loop H_∞-norm objective has the standard signal gain interpretation. Finally it can be shown that the controller, K_s, does not substantially affect the loop shape in frequencies where the gain of W_2GW_1 is either high or low, and will guarantee satisfactory stability margins in the frequency region of gain cross-over. In the regulator set-up, the final controller to be implemented is $K=W_1K_sW_2$.

See McFarlane and Glover [1]–[2] for details.

References

[1] McFarlane, D.C., and K. Glover, Robust Controller Design using Normalised Coprime Factor Plant Descriptions, Springer Verlag, *Lecture Notes in Control and Information Sciences,* vol. 138, 1989.

[2] McFarlane, D.C., and K. Glover, "A Loop Shaping Design Procedure using Synthesis," *IEEE Transactions on Automatic Control,* vol. 37, no. 6, pp. 759– 769, June 1992.

[3] Vinnicombe, G., “Measuring Robustness of Feedback Systems,” PhD dissertation, Department of Engineering, University of Cambridge, 1993.

[4] Zhou, K., and J.C. Doyle, Essentials of Robust Control. NY: Prentice-Hall, 1998.

See Also

`gapmetric` | `hinfsyn` | `loopsyn` | `ncfmargin`

Introduced before R2006a

newlmi

Attach identifying tag to LMIs

Syntax

```
tag = newlmi
```

Description

`newlmi` adds a new LMI to the LMI system currently described and returns an identifier tag for this LMI. This identifier can be used in `lmiterm`, `showlmi`, or `dellmi` commands to refer to the newly declared LMI. Tagging LMIs is *optional* and only meant to facilitate code development and readability.

Identifiers can be given mnemonic names to help keep track of the various LMIs. Their value is simply the ranking of each LMI in the system (in the order of declaration). They prove useful when some LMIs are deleted from the LMI system. In such cases, the identifiers are the safest means of referring to the remaining LMIs.

See Also

`dellmi` | `getlmis` | `lmiedit` | `lmiterm` | `lmivar` | `setlmis`

Introduced before R2006a

normalized2actual

Convert value for atom in normalized coordinates to corresponding actual value

Syntax

```
avalue = normalized2actual(A,NV)
```

Description

Converts a normalized value NV of an atom to its corresponding actual (unnormalized) value.

If `NV` is an array of values, then `avalue` will be an array of the same dimension.

Examples

Create uncertain real parameters with a range that is symmetric about the nominal value, where each endpoint is 1 unit from the nominal. Points that lie inside the range are less than 1 unit from the nominal, while points that lie outside the range are greater than 1 unit from the nominal.

```
a = ureal('a',3,'range',[1 5]);
actual2normalized(a,[1 3 5])
ans =
   -1.0000   -0.0000    1.0000
normalized2actual(a,[-1 1])
ans =
   1.0000    5.0000
normalized2actual(a,[-1.5 1.5])
ans =
   0.0000    6.0000
```

See Also

`actual2normalized` | `getLimits` | `robgain` | `robstab`

Introduced before R2006a

pdlstab

Assess robust stability of polytopic or parameter-dependent system

Syntax

```
[tau,Q0,Q1,...] = pdlstab(pds,options)
```

Description

`pdlstab` uses parameter-dependent Lyapunov functions to establish the stability of uncertain state-space models over some parameter range or polytope of systems. Only sufficient conditions for the existence of such Lyapunov functions are available in general. Nevertheless, the resulting robust stability tests are always less conservative than quadratic stability tests when the parameters are either time-invariant or slowly varying.

For an affine parameter-dependent system

$E(p)\dot{x} = A(p)x + B(p)u$

$y = C(p)x + D(p)u$

with $p = (p_1, . . ., p_n) \in R^n$, `pdlstab` seeks a Lyapunov function of the form

$V(xp,) = x^TQ(p)–1x, Q(p) = Q_0 + p_1Q_1 + . . .p_nQ_n$

such that $dV(x, p)/dt < 0$ along all admissible parameter trajectories. The system description `pds` is specified with `psys` and contains information about the range of values and rate of variation of each parameter p_i.

For a *time-invariant* polytopic system

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

$y = Cx + Du$

with

$$\begin{pmatrix} A + jE & B \\ C & D \end{pmatrix} = \sum_{i=1}^{n} \alpha_i \begin{pmatrix} A + jE_i & B_i \\ C_i & D_i \end{pmatrix}, \ \alpha_i \geq 0, \ \sum_{i=1}^{n} \alpha_i = 1 \tag{1-14}$$

`pdlstab` seeks a Lyapunov function of the form

$V(x, \alpha) = x^T Q(\alpha)–1x,\ Q(\alpha) = \alpha_1 Q_1 + \ldots + \alpha_n Q_n$

such that $dV(x, \alpha)/dt < 0$ for all polytopic decompositions of the form “Equation 1-14”.

Several options and control parameters are accessible through the optional argument `options`:

- Setting `options(1)=0` tests robust stability (default)
- When `options(2)=0`, `pdlstab` uses simplified sufficient conditions for faster running times. Set `options(2)=1` to use the least conservative conditions

Tips

For affine parameter-dependent systems with *time-invariant* parameters, there is equivalence between the robust stability of

$$E(p)\dot{x} = A(p)x \tag{1-15}$$

and that of the dual system

$$E(p)^T \dot{z} = A(p)^T z \tag{1-16}$$

However, the second system may admit an affine parameter-dependent Lyapunov function while the first does not.

In such case, `pdlstab` automatically restarts and tests stability on the dual system “Equation 1-16” when it fails on “Equation 1-15”.

See Also

`quadstab`

Introduced before R2006a

pdsimul

Time response of parameter-dependent system along given parameter trajectory

Syntax

```
pdsimul(pds,'traj',tf,'ut',xi,options)
```

```
[t,x,y] = pdsimul(pds,pv,'traj',tf,'ut',xi,options)
```

Description

`pdsimul` simulates the time response of an affine parameter-dependent system

$$E(p)\dot{x} = A(p)x + B(p)u$$

$$y = C(p)x + D(p)u$$

along a parameter trajectory $p(t)$ and for an input signal $u(t)$. The parameter trajectory and input signals are specified by two time functions `p=traj(t)` and `u=ut(t)`. If `'ut'` is omitted, the response to a step input is computed by default.

The affine system `pds` is specified with `psys`. The function `pdsimul` also accepts the polytopic representation of such systems as returned by `aff2pol(pds)` or `hinfgs`. The final time and initial state vector can be reset through `tf` and `xi` (their respective default values are 5 seconds and 0). Finally, `options` gives access to the parameters controlling the ODE integration (type `help gear` for details).

When invoked without output arguments, `pdsimul` plots the output trajectories $y(t)$. Otherwise, it returns the vector of integration time points `t` as well as the state and output trajectories `x,y`.

See Also

`psys` | `pvec`

Introduced before R2006a

polydec

Compute polytopic coordinates with respect to box corners

Syntax

```
vertx = polydec(PV)
```

```
[C,vertx] = polydec(PV,P)
```

Description

`vertx = polydec(PV)` takes an uncertain parameter vector `PV` taking values ranging in a box, and returns the corners or vertices of the box as columns of the matrix `vertx`.

`[C,vertx] = polydec(PV,P)` takes an uncertain parameter vector `PV` and a value `P` of the parameter vector `PV`, and returns the convex decomposition `C` of `P` over the set `VERTX` of box corners:

```
P = c1*VERTX(:,1) + ... + cn*VERTX(:,n)
cj >=0 ,              c1 + ... + cn = 1
```

The list `vertx` of corners can be obtained directly by typing

```
vertx = polydec(PV)
```

See Also

`aff2pol` | `hinfgs` | `pvec` | `pvinfo`

Introduced before R2006a

popov

Perform Popov robust stability test

Syntax

```
[t,P,S,N] = popov(sys,delta,flag)
```

Description

`popov` uses the Popov criterion to test the robust stability of dynamical systems with possibly nonlinear and/or time-varying uncertainty. The uncertain system must be described as the interconnection of a nominal LTI system `sys` and some uncertainty `delta`.

The command

```
[t,P,S,N] = popov(sys,delta)
```

tests the robust stability of this interconnection. Robust stability is guaranteed if `t` < 0. Then `P` determines the quadratic part x^TPx of the Lyapunov function and `D` and `S` are the Popov multipliers.

If the uncertainty `delta` contains real parameter blocks, the conservatism of the Popov criterion can be reduced by first performing a simple loop transformation. To use this refined test, call `popov` with the syntax

```
[t,P,S,N] = popov(sys,delta,1)
```

See Also

`pdlstab` | `quadstab`

Introduced before R2006a

psinfo

Inquire about polytopic or parameter-dependent systems created with `psys`

Syntax

```
psinfo(ps)
```

```
[type,k,ns,ni,no] = psinfo(ps)
```

```
pv = psinfo(ps,'par')
```

```
sk = psinfo(ps,'sys',k)
```

```
sys = psinfo(ps,'eval',p)
```

Description

`psinfo` is a multi-usage function for queries about a polytopic or parameter-dependent system `ps` created with `psys`. It performs the following operations depending on the calling sequence:

- `psinfo(ps)` displays the type of system (affine or polytopic); the number `k` of `SYSTEM` matrices involved in its definition; and the numbers of `ns`, `ni`, `no` of states, inputs, and outputs of the system. This information can be optionally stored in MATLAB variables by providing output arguments.
- `pv = psinfo(ps,'par')` returns the parameter vector description (for parameter-dependent systems only).
- `sk = psinfo(ps,'sys',k)` returns the *k*-th `SYSTEM` matrix involved in the definition of `ps`. The ranking k is relative to the list of systems `syslist` used in `psys`.
- `sys = psinfo(ps,'eval',p)` instantiates the system for a given vector *p* of parameter values or polytopic coordinates.

  For *affine parameter-dependent* systems defined by the `SYSTEM` matrices $S_0, S_1, \ldots,$ S_n, the entries of `p` should be real parameter values $p_1, \ldots, p_n$ and the result is the LTI system of `SYSTEM` matrix

  $$S(p) = S_0 + p_1 S_1 + \ldots + p_n S_n$$

For *polytopic* systems with `SYSTEM` matrix ranging in

$\mathrm{Co}\{S_1, . . ., S_n\}$,

the entries of `p` should be polytopic coordinates $p_1, . . ., p_n$ satisfying $p_j \geq 0$ and the result is the interpolated LTI system of `SYSTEM` matrix

$$S = \frac{p_1 S_1 + \cdots + p_n S_n}{p_1 + \cdots + p_n}$$

See Also

`psys`

Introduced before R2006a

psys

Specify polytopic or parameter-dependent linear systems

Syntax

```
pols = psys(syslist)
```

```
affs = psys(pv,syslist)
```

Description

`psys` specifies state-space models where the state-space matrices can be uncertain, time-varying, or parameter-dependent.

`psys` supports two types of uncertain state-space models:

- *Polytopic* systems

  $$E(t)\dot{x} = A(t)x + B(t)u$$

  $$y = C(t)x + D(t)u$$

  whose `SYSTEM` matrix takes values in a fixed polytope:

  $$\underbrace{\begin{bmatrix} A(t) + jE(t) & B(t) \\ C(t) & D(t) \end{bmatrix}}_{S(t)} \in \mathrm{Co}\left\{ \underbrace{\begin{bmatrix} A_1 + jE_1 & B_1 \\ C_1 & D_1 \end{bmatrix}}_{S_1}, \ldots, \underbrace{\begin{bmatrix} Ak + jE_k & B_k \\ C_k & D_k \end{bmatrix}}_{S_k} \right\}$$

  where $S_1, \ldots, S_k$ are given “vertex” systems and

  $$\mathrm{Co}\left\{S_1, \ldots, S_k\right\} = \left\{ \sum_{i=1}^{k} \alpha_i S_i : \alpha_i \geq 0, \sum_{i=1}^{k} \alpha_i = 1 \right\}$$

  denotes the convex hull of $S_1, \ldots, S_k$ (polytope of matrices with vertices $S_1, \ldots, S_k$)
- *Affine parameter-dependent* systems

$$E(p)\dot{x} = A(p)x + B(p)u$$

$$y = C(p)x + D(p)u$$

where $A(\cdot\ ); B(\cdot\ ), \ldots, E(\cdot\ )$ are fixed affine functions of some vector $p = (p_1, \ldots, p_n)$ of real parameters, i.e.,

$$\underbrace{\begin{bmatrix} A(p) + jE(p) & B(p) \\ C(p) & D(p) \end{bmatrix}}_{S(p)} =$$

$$\underbrace{\begin{bmatrix} A_0 + jE_0 & B_0 \\ C_0 & D_0 \end{bmatrix}}_{S_0} + p1\underbrace{\begin{bmatrix} A_1 + jE_1 & B_1 \\ C_1 & D_1 \end{bmatrix}}_{S_1} + \ldots + p_n\underbrace{\begin{bmatrix} A_n + jE_n & B_n \\ C_n & D_n \end{bmatrix}}_{S_n}$$

where $S_0, S_1, \ldots, S_n$ are given SYSTEM matrices. The parameters p_i can be time-varying or constant but uncertain.

The argument syslist lists the SYSTEM matrices S_i characterizing the polytopic value set or parameter dependence. In addition, the description pv of the parameter vector (range of values and rate of variation) is required for affine parameter- dependent models (see pvec for details). Thus, a polytopic model with vertex systems $S_1, \ldots, S_4$ is created by

```
pols = psys([s1,s2,s3,s4])
```

while an affine parameter-dependent model with 4 real parameters is defined by

```
affs = psys(pv,[s0,s1,s2,s3,s4])
```

The output is a structured matrix storing all the relevant information.

See Also

aff2pol | psinfo | pvec

Introduced before R2006a

pvec

Specify range and rate of variation of uncertain or time-varying parameters

Syntax

```
pv = pvec('box',range,rates)
```

```
pv = pvec('pol',vertices)
```

Description

`pvec` is used in conjunction with psys to specify parameter-dependent systems. Such systems are parametrized by a vector $p = (p_1, \ldots, p_n)$ of uncertain or time-varying real parameters p_i. The function `pvec` defines the range of values and the rates of variation of these parameters.

The type `'box'` corresponds to independent parameters ranging in intervals

$$\underline{p}_j \le p_j \le \bar{p}_j$$

The parameter vector p then takes values in a hyperrectangle of R^n called the parameter box. The second argument range is an n-by-2 matrix that stacks up the extremal values $\underline{p}_j$ and $\bar{p}_j$ of each p_j. If the third argument `rates` is omitted, all parameters are assumed time-invariant. Otherwise, `rates` is also an n-by-2 matrix and its j-th row specifies lower and upper bounds $\underline{\nu}_j$ and $\bar{\nu}_j$ on $\frac{dp_j}{dt}$:

$$\underline{\nu}_j \le \frac{dp_j}{dt} \le \bar{\nu}_j$$

Set $\underline{\nu}_j$ = `–Inf` and $\bar{\nu}_j$ = `Inf` if $p_j(t)$ can vary arbitrarily fast or discontinuously.

The type `'pol'` corresponds to parameter vectors p ranging in a polytope of the parameter space R^n. This polytope is defined by a set of vertices $V_1, \ldots, V_n$ corresponding to “extremal” values of the vector p. Such parameter vectors are declared by the command

```
pv = pvec('pol',[v1,v2, . . ., vn])
```

where the second argument is the concatenation of the vectors `v1,...,vn`.

The output argument `pv` is a structured matrix storing the parameter vector description. Use `pvinfo` to read the contents of `pv`.

Examples

Consider a problem with two time-invariant parameters

$p_1 \in [-1, 2], \quad p_2 \in [20, 50]$

The corresponding parameter vector $p = (p_1, p_2)$ is specified by

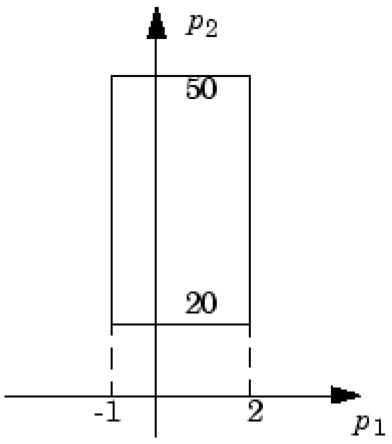
```
pv = pvec('box',[-1 2;20 50])
```

Alternatively, this vector can be regarded as taking values in the rectangle drawn in the following figure. The four corners of this rectangle are the four vectors

$$v_1 = \begin{pmatrix} -1 \\ 20 \end{pmatrix}, \ v_2 = \begin{pmatrix} -1 \\ 50 \end{pmatrix}, \ v_3 = \begin{pmatrix} 2 \\ 20 \end{pmatrix}, \ v_4 = \begin{pmatrix} 2 \\ 50 \end{pmatrix}$$

Hence, you could also specify p by

```
pv = pvec('pol',[v1,v2,v3,v4])
```



Parameter box

See Also

`psys` | `pvinfo`

Introduced before R2006a

pvinfo

Describe parameter vector specified with `pvec`

Syntax

```
[typ,k,nv] = pvinfo(pv)
```

```
[pmin,pmax,dpmin,dpmax] = pvinfo(pv,'par',j)
```

```
vj = pvinfo(pv,'par',j)
```

```
p = pvinfo(pv,'eval',c)
```

Description

`pvinfo` retrieves information about a vector $p = (p_1, \ldots, p_n)$ of real parameters declared with `pvec` and stored in `pv`. The command `pvinfo(pv)` displays the type of parameter vector (`'box'` or `'pol'`), the number n of scalar parameters, and for the type `'pol'`, the number of vertices used to specify the parameter range.

For the type `'box'`:

```
[pmin,pmax,dpmin,dpmax] = pvinfo(pv,'par',j)
```

returns the bounds on the value and rate of variations of the j-th real parameter p_j. Specifically,

$$p\text{min} \le p_j(t) \le p\text{max}, dp\text{min} \le \frac{dp_j}{dt} \le dp\text{max}$$

For the type `'pol'`:

```
pvinfo(pv,'par',j)
```

returns the j-th vertex of the polytope of R^n in which p ranges, while

```
pvinfo(pv,'eval',c)
```

returns the value of the parameter vector *p* given its barycentric coordinates c with respect to the polytope vertices ($V_1, . . ., V_k$). The vector c must be of length *k* and have nonnegative entries. The corresponding value of *p* is then given by

$$p = \frac{\sum_{i=1}^{k} c_i V_i}{\sum_{i=1}^{k} c_i}$$

See Also

psys | pvec

Introduced before R2006a

quadperf

Compute quadratic H_∞ performance of polytopic or parameter-dependent system

Syntax

```
[perf,P] = quadperf(ps,g,options)
```

Description

The RMS gain of the time-varying system

$$E(t)\dot{x} = A(t)x + B(t)u, \quad y = C(t)X + D(t)u \tag{1-17}$$

is the smallest $\gamma > 0$ such that

$$\|y\|_{L_2} \le \gamma \|u\|_{L_2} \tag{1-18}$$

for all input $u(t)$ with bounded energy. A sufficient condition for “Equation 1-18” is the existence of a quadratic Lyapunov function

$$\mathrm{V}(x) = x^T P x, \quad P > 0$$

such that

$$\forall u \in L_2,\ \frac{dV}{dt} + y^T y - \gamma^2 u^T u < 0$$

Minimizing γ over such quadratic Lyapunov functions yields the quadratic H_∞ performance, an upper bound on the true RMS gain.

The command

```
[perf,P] = quadperf(ps)
```

computes the quadratic H_∞ performance `perf` when “Equation 1-17” is a polytopic or affine parameter-dependent system `ps` (see `psys`). The Lyapunov matrix P yielding the performance `perf` is returned in `P`.

The optional input `options` gives access to the following task and control parameters:

- If `options(1)=1`, `perf` is the largest portion of the parameter box where the quadratic RMS gain remains smaller than the positive value `g` (for affine parameter-dependent systems only). The default value is 0.
- If `options(2)=1`, `quadperf` uses the least conservative quadratic performance test. The default is `options(2)=0` (fast mode)
- `options(3)` is a user-specified upper bound on the condition number of *P* (the default is 109).

See Also

`psys` | `quadstab`

Introduced before R2006a

quadstab

Quadratic stability of polytopic or affine parameter-dependent systems

Syntax

```
[tau,P] = quadstab(ps,options)
```

Description

For affine parameter-dependent systems

$$E(p)\dot{x} = A(p)x, \quad p(t) = (p_1(t), \ldots, p_n(t))$$

or polytopic systems

$$E(t)\dot{x} = A(t)x, \quad (A, E) \in \mathrm{Co}\{(A_1, E_1), \ldots, (A_n, E_n)\},$$

`quadstab` seeks a fixed Lyapunov function $V(x) = x^TPx$ with $P > 0$ that establishes quadratic stability. The affine or polytopic model is described by `ps` (see `psys`).

The task performed by `quadstab` is selected by `options(1)`:

- if `options(1)=0` (default), `quadstab` assesses quadratic stability by solving the LMI problem

  Minimize τ over $Q = Q^T$ such that

  $$A^TQE + EQA^T < \tau I \quad \text{for all admissible values of } (A, E)$$

  $$Q > I$$

  The global minimum of this problem is returned in `tau` and the system is quadratically stable if `tau` < 0.
- if `options(1)=1`, `quadstab` computes the largest portion of the specified parameter range where quadratic stability holds (only available for affine models). Specifically, if each parameter p_i varies in the interval

$p_i \in [p_{i0} - \delta_i, p_{i0} + \delta_i],$

`quadstab` computes the largest $\Theta > 0$ such that quadratic stability holds over the parameter box

$p_i \in [p_{i0} - \Theta\delta_i, p_{i0} + \Theta\delta_i]$

This “quadratic stability margin” is returned in `tau` and `ps` is quadratically stable if `tau` ≥ 1.

Given the solution Q_{opt} of the LMI optimization, the Lyapunov matrix P is given by $P = Q_{\text{opt}}^{-1}$. This matrix is returned in `P`.

Other control parameters can be accessed through `options(2)` and `options(3)`:

- if `options(2)=0` (default), `quadstab` runs in fast mode, using the least expensive sufficient conditions. Set `options(2)=1` to use the least conservative conditions
- `options(3)` is a bound on the condition number of the Lyapunov matrix P. The default is 10^9.

See Also

`decay` | `pdlstab` | `psys` | `quadperf`

Introduced before R2006a

randatom

Generate random uncertain `atom` objects

Syntax

```
A = randatom(Type)
```

```
A = randatom(Type,sz)
```

```
A = randatom
```

Description

`A = randatom(Type)` generates a 1-by-1 `type` uncertain object. Valid values for `Type` include `'ureal'`, `'ultidyn'`, `'ucomplex'`, and `'ucomplexm'`.

`A = randatom(Type,sz)` generates an `sz(1)`-by-`sz(2)` uncertain object. Valid values for `Type` include `'ultidyn'` or `'ucomplexm'`. If `Type` is set to `'ureal'` or `'ucomplex'`, the size variable is ignored and A is a 1-by-1 uncertain object.

`A = randatom`, where `randatom` has no input arguments, results in a 1-by-1 uncertain object. The class is of this object is randomly selected between `'ureal'`,`'ultidyn'` and `'ucomplex'`.

In general, both `rand` and `randn` are used internally. You can control the result of `randatom` by setting seeds for both random number generators before calling the function.

Examples

The following statement creates the `ureal` uncertain object `xr`. Note that your display can differ because a random seed is used.

```
xr = randatom('ureal')
xr =

  Uncertain real parameter "NMGXC" with nominal value 5.34 and variability [-2.99,1.92].
```

The following statement creates the variable `ultidyn` uncertain object `xlti` with three inputs and four outputs. You will get the results shown below if you set the random variable seed to 29.

```
rng(29,'twister');
xlti = randatom('ultidyn',[4 3])
```

```
xlti =

  Uncertain LTI dynamics "LOSWT" with 4 outputs, 3 inputs, and gain less than 0.293.
```

See Also

`rand` | `randn` | `randumat` | `randuss` | `ucomplex` | `ucomplexm` | `ultidyn`

Introduced before R2006a

randumat

Generate random uncertain `umat` objects

Syntax

`um = randumat(ny,nu)`

`um = randumat`

Description

`um = randumat(ny,nu)` generates an uncertain matrix of size `ny`-by-`nu`. `randumat` randomly selects from uncertain atoms of type `'ureal'`, `'ultidyn'`, and `'ucomplex'`.

`um = randumat` results in a 1-by-1 `umat` uncertain object, including up to four uncertain objects.

Examples

The following statement creates the `umat` uncertain object `x1` of size 2-by-3. Note that your result can differ because a random seed is used.

```
x1 = randumat(2,3)
```

```
x1 =

  Uncertain matrix with 2 rows and 3 columns.
  The uncertainty consists of the following blocks:
    AWYRT: Uncertain real, nominal = 7.09, variability = [-7.84,16.4]%, 2 occurrences
    HRRED: Uncertain complex, nominal = 3.14+5.47i, radius = 1.92, 1 occurrences
    VSIYA: Uncertain real, nominal = -4.05, variability = [-1.53,3.83], 3 occurrences
    YZEZY: Uncertain complex, nominal = -6.54-2.17i, variability = 24%, 1 occurrences

Type "x1.NominalValue" to see the nominal value, "get(x1)" to see all properties, and
"x1.Uncertainty" to interact with the uncertain elements.
```

The following statement creates the `umat` uncertain object `x2` of size 4-by-2 with the seed 91.

```
rng(91,'twister');
x2 = randumat(4,2)
```

```
x2 =

  Uncertain matrix with 4 rows and 2 columns.
  The uncertainty consists of the following blocks:
    YQZBI: Uncertain complex, nominal = 3.61+1.88i, radius = 1.42, 1 occurrences

Type "x2.NominalValue" to see the nominal value, "get(x2)" to see all properties,
and "x2.Uncertainty" to interact with the uncertain elements.
```

See Also

`rand` | `randatom` | `randn` | `randuss` | `ucomplex` | `ultidyn`

Introduced before R2006a

randuss

Generate stable, random `uss` objects

Syntax

```
usys = randuss(n)
```

```
usys = randuss(n,p)
```

```
usys = randuss(n,p,m)
```

```
usys = randuss(n,p,m,Ts)
```

```
usys = randuss
```

Description

`usys = randuss(n)` generates an `n`th order single-input/single-output uncertain continuous-time system. `randuss` randomly selects from uncertain atoms of type `'ureal'`, `'ultidyn'`, and `'ucomplex'`.

`usys = randuss(n,p)` generates an `n`th order single-input uncertain continuous-time system with `p` outputs.

`usys = randuss(n,p,m)` generates an `n`th order uncertain continuous-time system with `p` outputs and `m` inputs.

`usys = randuss(n,p,m,Ts)` generates an `n`th order uncertain discrete-time system with `p` outputs and `m` inputs. The sample time is `Ts`.

`usys = randuss` (without arguments) results in a 1-by-1 uncertain continuous-time `uss` object with up to four uncertain objects.

In general, both `rand` and `randn` are used internally. You can control the result of `randuss` by setting seeds for both random number generators before calling the function.

Examples

The statement creates a fifth order, continuous-time uncertain system `s1` of size 2-by-3. Note your display can differ because a random seed is used.

```
s1 = randuss(5,2,3)
USS: 5 States, 2 Outputs, 3 Inputs, Continuous System
  CTPQV: 1x1 LTI, max. gain = 2.2, 1 occurrence
  IGDHN: real, nominal = -4.03, variability =
[-3.74667  22.7816]%, 1 occurrence
  MLGCD: complex, nominal = 8.36+3.09i,  +/- 7.07%, 1 occurrence
  OEDJK: complex, nominal = -0.346-0.296i, radius = 0.895,
1 occurrence
```

See Also

`rand` | `randatom` | `randn` | `randumat` | `ucomplex` | `ultidyn`

Introduced before R2006a

reduce

Simplified access to Hankel singular value based model reduction functions

Syntax

```
GRED = reduce(G)
```

```
GRED = reduce(G,order)
```

```
[GRED,redinfo] = reduce(G,'key1','value1',...)
```

```
[GRED,redinfo] = reduce(G,order,'key1','value1',...)
```

Description

`reduce` returns a reduced order model `GRED` of `G` and a struct array `redinfo` containing the error bound of the reduced model, Hankel singular values of the original system and some other relevant model reduction information.

An error bound is a measure of how close `GRED` is to `G` and is computed based on either *additive error,* $\| G-GRED \|_\infty$, *multiplicative error,* $\|G^{-1}(G-GRED) \|_\infty$, or *nugap error* (ref.: `ncfmr`) [1],[4],[5].

Hankel singular values of a stable system indicate the respective state energy of the system. Hence, reduced order can be directly determined by examining the system Hankel SV's. Model reduction routines, which based on Hankel singular values are grouped by their error bound types. In many cases, the additive error method `GRED=reduce(G,ORDER)` is adequate to provide a good reduced order model. But for systems with lightly damped poles and/or zeros, a multiplicative error method (namely, `GRED=reduce(G,ORDER,'ErrorType','mult')`) that minimizes the relative error between `G` and `GRED` tends to produce a better fit.

This table describes input arguments for `reduce`.

Argument	**Description**
`G`	LTI model to be reduced (without any other inputs will plot its Hankel singular values and prompt for reduced order).
`ORDER`	(Optional) Integer for the desired order of the reduced model, or optionally a vector packed with desired orders for batch runs.

A batch run of a serial of different reduced order models can be generated by specifying `order = x:y`, or a vector of integers. By default, all the anti-stable part of a physical system is kept, because from control stability point of view, getting rid of unstable state(s) is dangerous to model a system.

`'MaxError'` can be specified in the same fashion as an alternative for `' ORDER '` after an *`'ErrorType'`* is selected. In this case, reduced order will be determined when the sum of the tails of the Hankel SV's reaches the *`'MaxError'`*.

Argument	**Value**	**Description**
`'Algorithm'`	*`'balance'`*	Default for `'add'` (`balancmr`)
	`'schur'`	Option for `'add'` (`schurmr`)
	`'hankel'`	Option for `'add'` (`hankelmr`)
	`'bst'`	Default for `'mult'` (`bstmr`)
	`'ncf'`	Default for `'ncf'` (`ncfmr`)
`'ErrorType'`	*`'add'`*	Additive error (default)
	`'mult'`	Multiplicative error at model output
	`'ncf'`	NCF nugap error
`'MaxError'`	A real number or a vector of different errors	Reduce to achieve H_∞ error. When present, *`'MaxError'`* overrides `ORDER` input.
`'Weights'`	`{Wout,Win}` cell array	Optimal 1x2 cell array of LTI weights `Wout` (output) and `Win` (input); default is both identity; used only with *`'ErrorType'`*, *`'add'`*. Weights must be invertible.
`'Display'`	*`'on'`* or *`'off'`*	Display Hankel singular plots (default *`'off'`*).

Argument	Value	Description
'Order'	Integer, vector or cell array	Order of reduced model. Use only if not specified as 2nd argument.

Weights on the original model input and/or output can make the model reduction algorithm focus on some frequency range of interests. But weights have to be stable, minimum phase and invertible.

This table describes output arguments.

Argument	Description
GRED	LTI reduced order model. Becomes multi-dimensional array when input is a serial of different model order array.
REDINFO	A STRUCT array with 3 fields: • REDINFO.ErrorBound • REDINFO.StabSV • REDINFO.UnstabSV For 'hankel' algorithm, STRUCT array becomes: • REDINFO.ErrorBound • REDINFO.StabSV • REDINFO.UnstabSV • REDINFO.Ganticausal For 'ncf' option, STRUCT array becomes: • REDINFO.GL • REDINFO.GR • REDINFO.hsv

G can be stable or unstable. G and GRED can be either continuous or discrete.

A successful model reduction with a well-conditioned original model G will ensure that the reduced model GRED satisfies the infinity norm error bound.

Examples

Reduce Model Order

Given a continuous or discrete, stable or unstable system, G, create a set of reduced-order models based on your selections.

```
rng(1234,'twister'); % For reproducibility
G = rss(30,5,4);
```

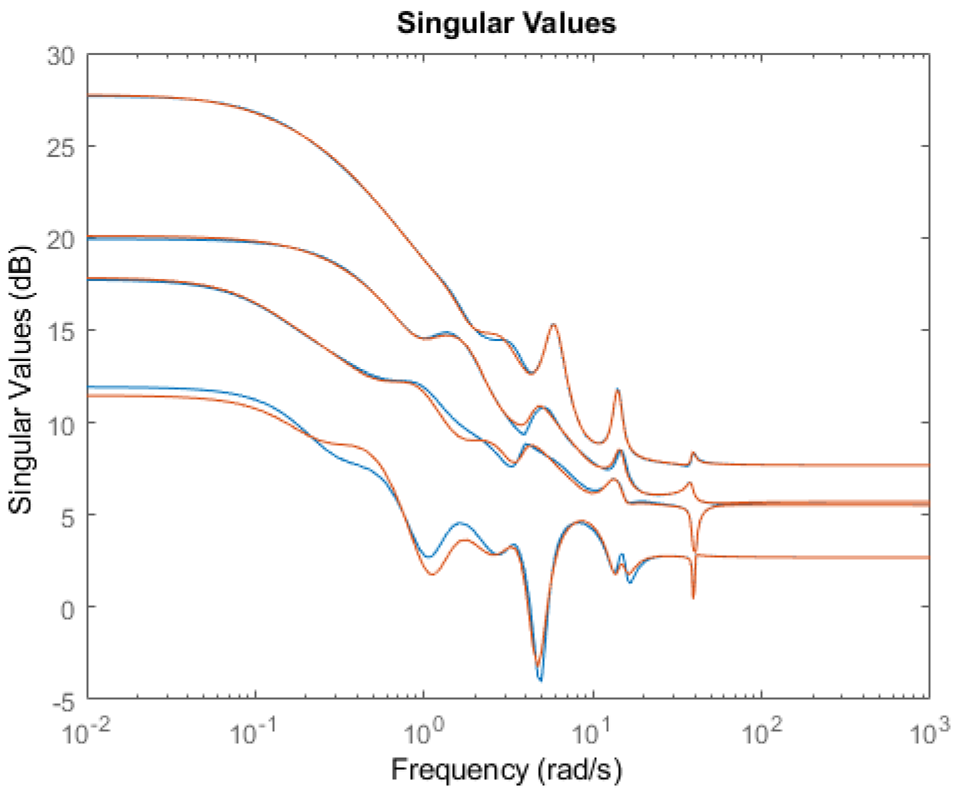
If you call reduce without specifying an order for the reduced model, the software displays a Hankel singular-value plot and prompts you to select an order.

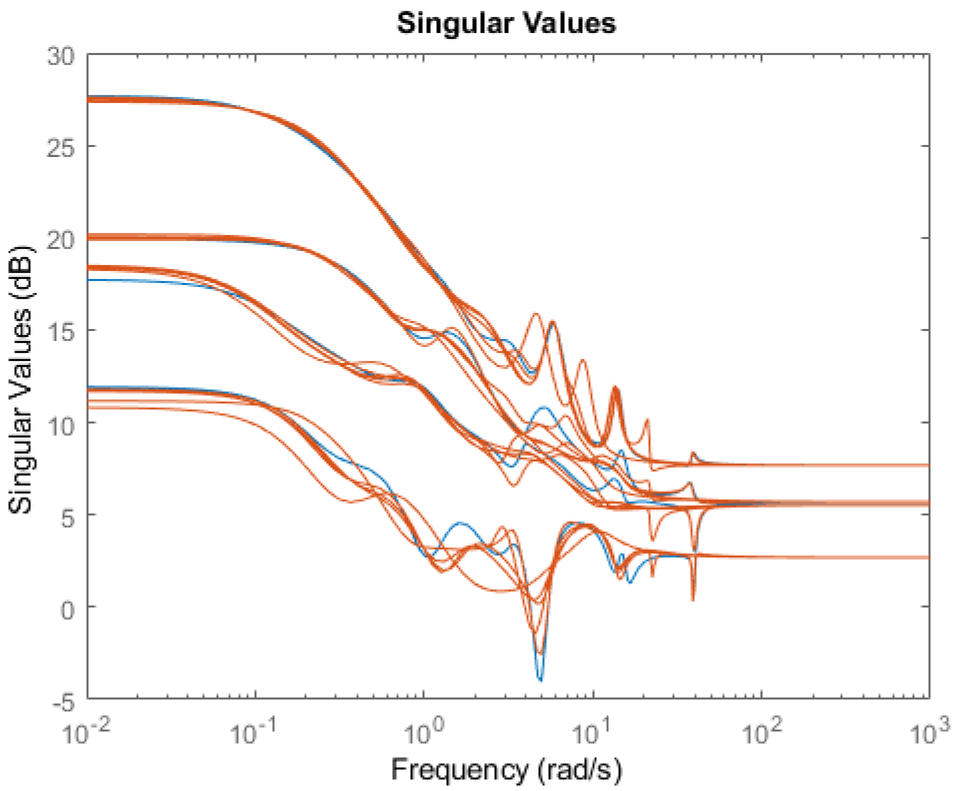
If you specify a reduced-model order, reduce defaults to the balancmr algorithm for model reduction.

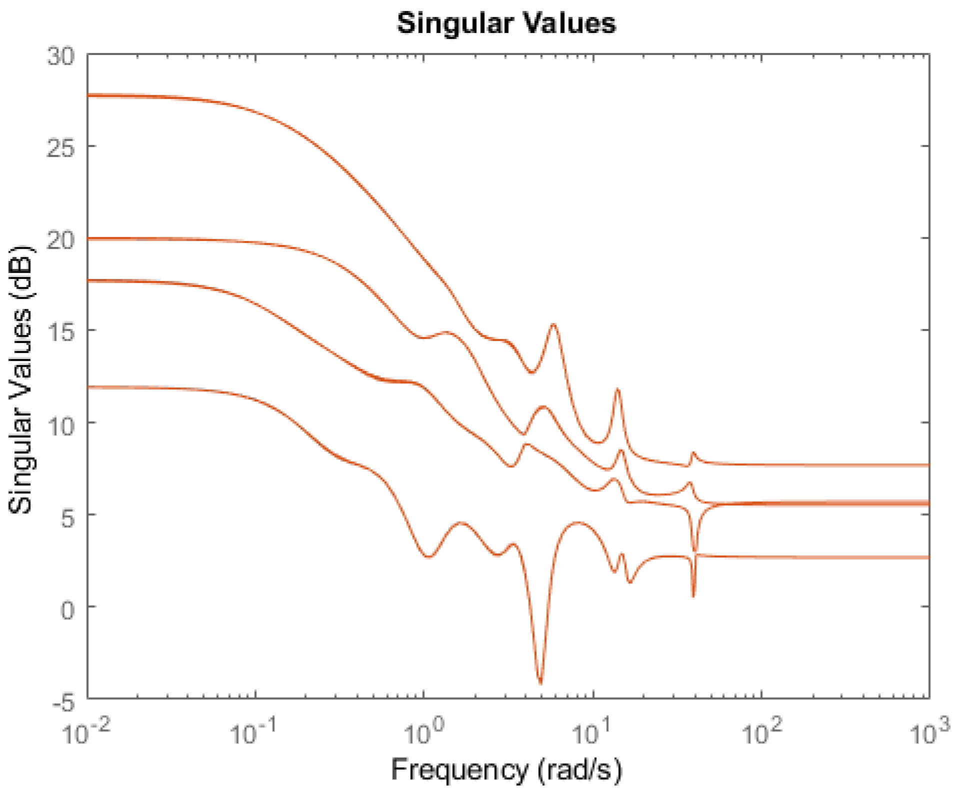
```
[g1,redinfo1] = reduce(G,20);
```

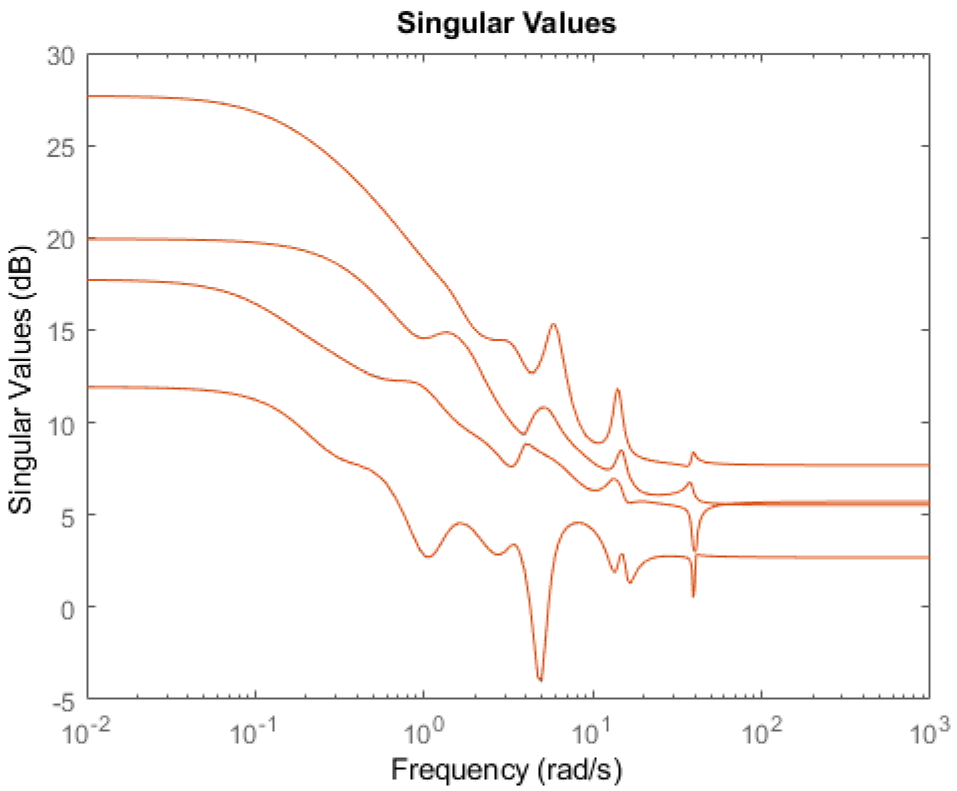
Specify other algorithms using the Algorithm argument. Use the ErrorType argument to specify whether the algorithm uses multiplicative or additive error, and the maximum permissible error in the reduced model.

```
[g2,redinfo2] = reduce(G,[10:2:18],'Algorithm','schur');
[g3,redinfo3] = reduce(G,'ErrorType','mult','MaxError',[0.01 0.05]);
[g4,redinfo4] = reduce(G,'ErrorType','add','Algorithm','hankel','MaxError',[0.01]);
for i = 1:4
    figure(i); eval(['sigma(G,g' num2str(i) ');']);
end
```


Singular Values
Singular Values (dB)
Frequency (rad/s)
30
25
20
15
10
5
0
-5
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}


Singular Values
Singular Values (dB)
Frequency (rad/s)
30
25
20
15
10
5
0
-5
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}


Singular Values
Singular Values (dB)
Frequency (rad/s)
30
25
20
15
10
5
0
-5
10^-2
10^-1
10^0
10^1
10^2
10^3



References

[1] K. Glover, “All Optimal Hankel Norm Approximation of Linear Multivariable Systems, and Their L_∞- error Bounds,” Int. J. Control, vol. 39, no. 6, pp. 1145-1193, 1984.

[2] M. G. Safonov and R. Y. Chiang, “A Schur Method for Balanced Model Reduction,” *IEEE Trans. on Automat. Contr.*, vol. AC-2, no. 7, July 1989, pp. 729-733.

[3] M. G. Safonov, R. Y. Chiang and D. J. N. Limebeer, “Optimal Hankel Model Reduction for Nonminimal Systems,” *IEEE Trans. on Automat. Contr.*, vol. 35, No. 4, April, 1990, pp. 496-502.

[4] M. G. Safonov and R. Y. Chiang, “Model Reduction for Robust Control: A Schur Relative-Error Method,” *International Journal of Adaptive Control and Signal Processing*, vol. 2, pp. 259-272, 1988.

[5] K. Zhou, “Frequency weighted L[[BULLET]] error bounds,” Syst. Contr. Lett., Vol. 21, 115-125, 1993.

See Also

`balancmr` | `bstmr` | `hankelmr` | `hankelsv` | `ncfmr` | `schurmr`

Introduced before R2006a

repmat

Replicate and tile array

Syntax

```
B = repmat(A,M,N)
```

Description

`B = repmat(A,M,N)` creates a large matrix `B` consisting of an `M`-by-`N` tiling of copies of `A`.

`B = repmat(A,[M N])` accomplishes the same result as `repmat(A,M,N)`.

`B = repmat(A,[M N P ...])` tiles the array `A` to produce an `M`-by-`N`-by`-P`-by-... block array. `A` can be N-D.

`repmat(A,M,N)` for scalar `A` is commonly used to produce an `M`-by-`N` matrix filled with values of `A`.

Examples

Simple examples of using `repmat` are

```
repmat(randumat(2,2),2,3)
repmat(ureal('A',6),[4 2])
```

Introduced before R2006a

rncf

Right normalized coprime factorization

Syntax

```
fact = rncf(sys)
[fact,Mr,Nr] = rncf(sys)
```

Description

`fact = rncf(sys)` computes the right normalized coprime factorization of the dynamic system model `sys`. The factorization is given by:

$$sys = N_r M_r^{-1}, \quad M_r^* M_r + N_r^* N_r = I\,.$$

Here, M_r^* denotes the conjugate of M_r (see `ctranspose`). The returned model `fact` is a minimal state-space realization of the stable system $[M_r;N_r]$. This factorization is used in other normalized coprime factor computations such as model reduction (`ncfmr`) and controller synthesis (`ncfsyn`).

`[fact,Mr,Nr] = rncf(sys)` also returns the coprime factors M_r and N_r.

Examples

Right Normalized Coprime Factorization of SISO System

Compute the right normalized coprime factorization of a SISO system.

```
sys = zpk([1 -1+2i -1-2i],[-1 2+1i 2-1i],1);
[fact,Mr,Nr] = rncf(sys);
```

Examine the original system and its factors.

```
sys
```

```
sys =

  (s-1) (s^2 + 2s + 5)
  --------------------
  (s+1) (s^2 - 4s + 5)

Continuous-time zero/pole/gain model.
```

```
zpk(Mr)
```

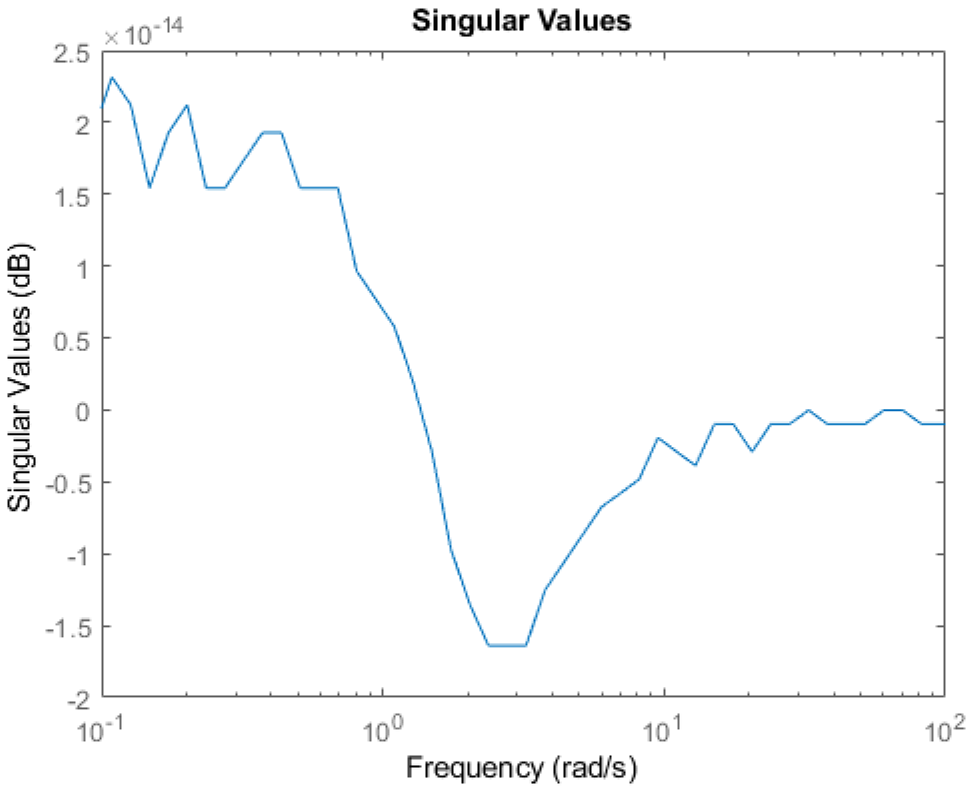
```
ans =

  0.70711 (s+1) (s^2 - 4s + 5)
  ----------------------------
    (s+1) (s^2 + 3.162s + 5)

Continuous-time zero/pole/gain model.
```

```
zpk(Nr)
```

```
ans =

  0.70711 (s-1) (s^2 + 2s + 5)
  ----------------------------
    (s+1) (s^2 + 3.162s + 5)

Continuous-time zero/pole/gain model.
```

The numerators of the factors `Mr` and `Nr` are the denominator and numerator of `sys`, respectively. Thus, `sys = Nr/Mr`. `rncf` chooses the denominators of the factors such that the system $[M_r(j\omega); N_r(j\omega)]$ is a unit vector at all frequencies. To confirm that property of the factorization, examine the singular values of `fact`, which is a stable minimal realization of $[M_r(j\omega); N_r(j\omega)]$.

```
sigma(fact)
```



Within a small numerical error, the singular value of `fact` is 1 (0 dB) at all frequencies.

Right Normalized Coprime Factorization of MIMO System

Compute the right normalized coprime factorization of a state-space model that has two outputs, two inputs, and three states.

```
rng(0); % for reproducibility
sys = rss(3,2,2);
[fact,Mr,Nr] = rncf(sys);
```

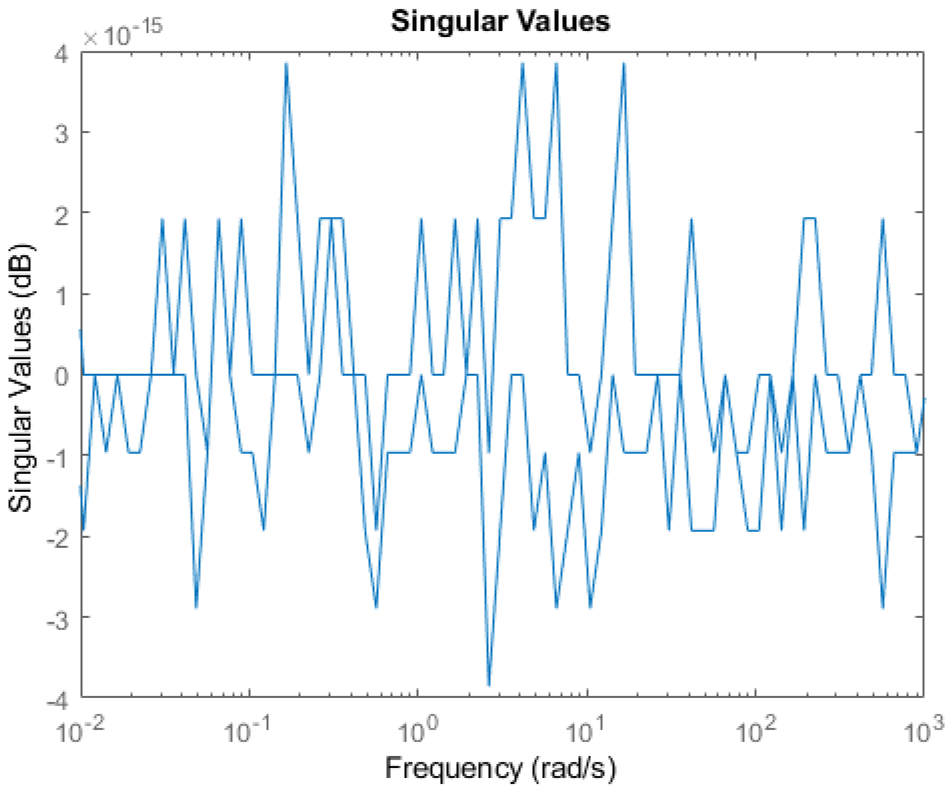
`fact` is a stable minimal realization of the factorization given by `[Mr;Nr]`.

```
isstable(fact)
```

```
ans = logical
   1
```

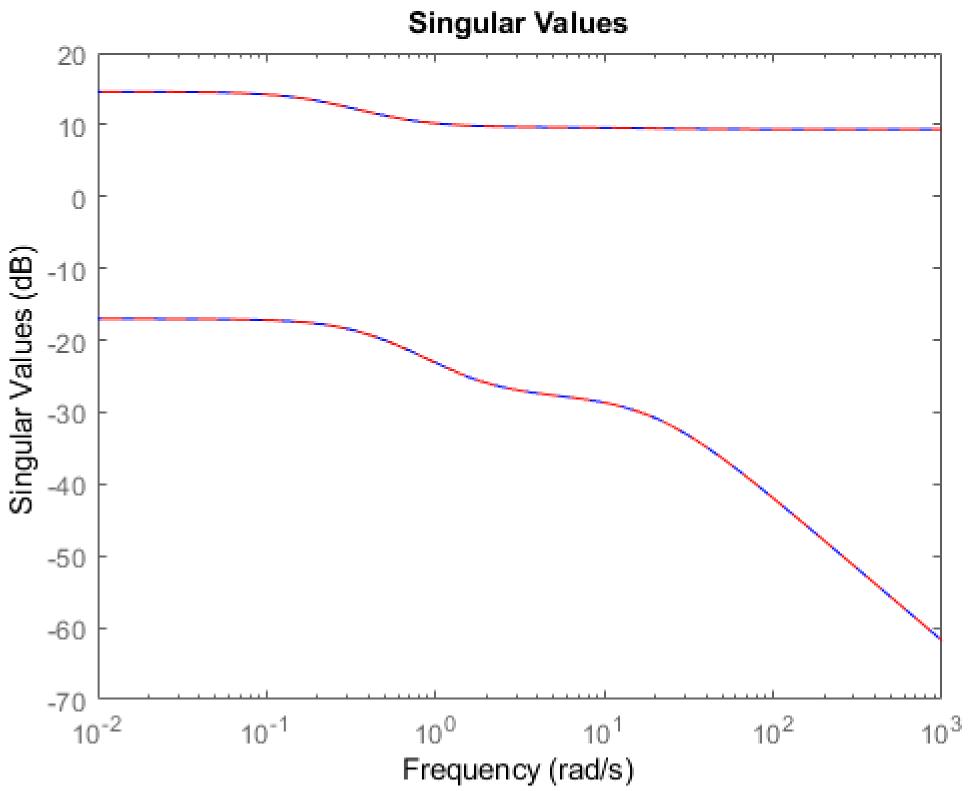
Another property of `fact` is that its frequency response $F(j\omega)$ is an orthogonal matrix at all frequencies ($F(j\omega)'F(j\omega) = I$). Confirm this property by examining the singular values of `fact`. Within a small numerical error, the singular values are 1 (0 dB) at all frequencies.

```
sigma(fact)
```



Confirm that the factors satisfy `sys = Nr/Mr` by examining the singular values of both.

```
sigma(sys,'b-',Nr/Mr,'r--')
```



Input Arguments

sys — Input system
dynamic system model

Input system to factorize, specified as a dynamic system model such as a state-space (`ss`) model. If `sys` is a generalized state-space model with uncertain or tunable control design blocks, then the function uses the nominal or current value of those elements. `sys` cannot be an `frd` model or a model with time delays.

Output Arguments

`fact` — Minimal realization of `[Mr;Nr]`
`ss` model

Minimal realization of `[Mr;Nr]`, returned as a state-space model. `fact` is stable and its frequency response is an orthogonal matrix at all frequencies. If `sys` has `p` outputs and `m` inputs, then `fact` has `m+p` outputs and `m` inputs. `fact` has the same number of states as `sys`.

`Mr,Nr` — Right coprime factors
`ss` models

Right coprime factors of `sys`, returned as state-space models. If `sys` has `p` outputs and `m` inputs, then:

- `Mr` has `m` outputs and `m` inputs.
- `Nr` has `p` outputs and `m` inputs.

Both factors have the same number of states as `sys` and the same `A` and `B` matrices as `fact`.

Tips

- `fact` is a minimal realization of `[Mr;Nr]`. If you need to use `[Mr;Nr]` or `[Mr;Nr]'` in a computation, it is better to use `fact` than to concatenate the factors yourself. Such manual concatenation results in extra (nonminimal) states, which can lead to decreased numerical accuracy.

See Also

`lncf` | `ncfmr` | `ncfsyn`

Introduced in R2019a

robgain

Robust performance of uncertain system

Syntax

```
[perfmarg,wcu] = robgain(usys,gamma)
[perfmarg,wcu] = robgain(usys,gamma,w)
[perfmarg,wcu] = robgain( ___ ,opts)
[perfmarg,wcu,info] = robgain( ___ )
```

Description

`[perfmarg,wcu] = robgain(usys,gamma)` calculates the robust performance margin for an uncertain system and the performance level `gamma`. The performance of `usys` is measured by its peak gain or peak singular value (see “Robustness and Worst-Case Analysis”). The performance margin is relative to the uncertainty level specified in `usys`. A margin greater than 1 means that the gain of `usys` remains below `gamma` for all values of the uncertainty modeled in `usys`. A margin less than 1 means that at some frequency, the gain of `usys` exceeds `gamma` for some values of the uncertain elements within their specified ranges. For example, a margin of 0.5 implies the following:

- The gain of `usys` remains below `gamma` as long as the uncertain element values stay within 0.5 normalized units of their nominal values.
- There is a perturbation of size 0.5 normalized units that drives the peak gain to the level `gamma`.

The structure `perfmarg` contains upper and lower bounds on the actual performance margin and the critical frequency at which the margin upper bound is smallest. The structure `wcu` contains the uncertain-element values that drive the peak gain to the level `gamma`.

`[perfmarg,wcu] = robgain(usys,gamma,w)` assesses the robust performance margin for the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `robgain` restricts the performance margin computation to the interval between `wmin` and `wmax`.

- If w is a vector of frequencies, then robgain computes the performance margin at the specified frequencies only.

[perfmarg,wcu] = robgain(___ ,opts) specifies additional options for the computation. Use robOptions to create opts. You can use this syntax with any of the previous input-argument combinations.

[perfmarg,wcu,info] = robgain(___) returns a structure with additional information about the performance margins and the perturbations that drive the gain to gamma. See info for details about this structure. You can use this syntax with any of the previous input-argument combinations.

Examples

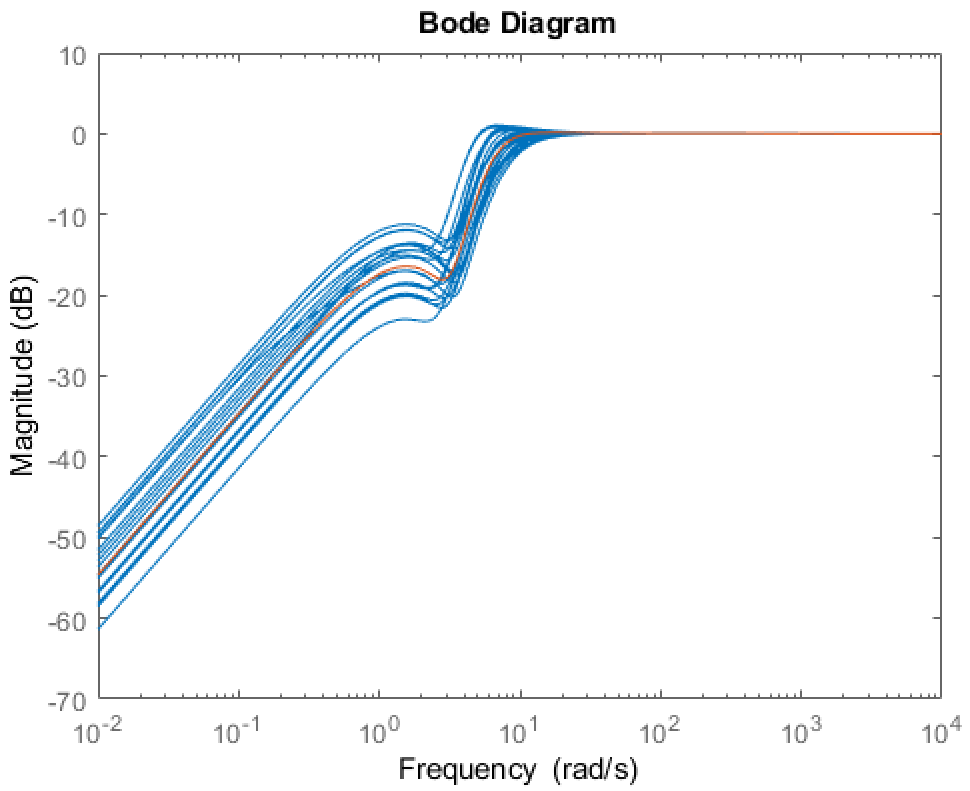
Robust Performance of Closed-Loop System

Consider a control system whose plant contains both parametric uncertainty and dynamic uncertainty. Create a model of the plant using uncertain elements.

```
k = ureal('k',10,'Percent',40);
delta = ultidyn('delta',[1 1]);
G = tf(18,[1 1.8 k]) * (1 + 0.5*delta);
```

Create a model of the controller, and build the closed-loop sensitivity function, S. The sensitivity measures the closed-loop response at the plant output to a disturbance at the plant input.

```
C = pid(2.3,3,0.38,0.001);
S = feedback(1,G*C);
bodemag(S,S.NominalValue)
```



The peak gain of the nominal response is very nearly 1, but some of the sampled systems within the uncertainty range exceed that level. Suppose that you can tolerate some ringdown in the response but do not want the peak gain to exceed 1.5. Use `robgain` to find out how much uncertainty the system can have while the peak gain remains below 1.5.

```
[perfmarg,wcu] = robgain(S,1.5);
perfmarg
```

```
perfmarg = struct with fields:
           LowerBound: 0.7821
           UpperBound: 0.7837
    CriticalFrequency: 7.8566
```

The LowerBound and UpperBound fields of perfmarg show that the robust performance margin is around 0.78. This result means that there is a perturbation of only about 78% of the uncertainty specified in S with peak gain exceeding 1.5.

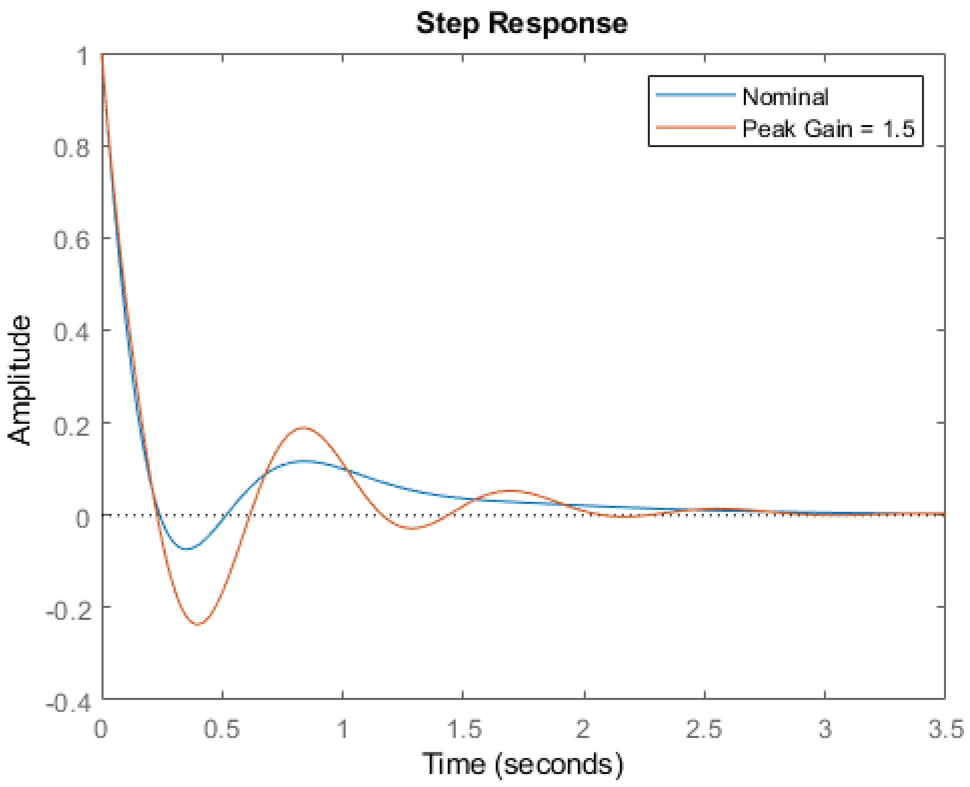
The output wcu is a structure that contains the corresponding perturbations to k and delta. Verify that the values in wcu cause Smax to achieve the gain level of 1.5 by substituting them into S.

```
Smax = usubs(S,wcu);
getPeakGain(Smax,1e-6)
```

```
ans = 1.5001
```

Examine the disturbance rejection of the system with these values.

```
step(S.NominalValue,Smax)
legend('Nominal','Peak Gain = 1.5')
```



The `CriticalFrequency` field of `perfmarg` contains the frequency at which the peak gain reaches 1.5.

Sensitivity of Performance to Uncertain Elements

Examine the relative sensitivity of the robust performance margin to the uncertain elements of the system. Consider a model of a control system containing uncertain elements.

```
k = ureal('k',10,'Percent',50);
delta = ultidyn('delta',[1 1]);
```

```
G = tf(18,[1 1.8 k]) * (1 + 0.15*delta);
C = pid(2.3,3,0.38,0.001);
S = feedback(1,G*C);
```

Create an options set for `robgain` that enables the sensitivity calculation.

```
opts = robOptions('Sensitivity','On');
```

Calculate the robust performance margin of the system relative to a peak gain of 1.5, specifying the `info` output to access additional information about the calculation.

```
[perfmarg,wcu,info] = robgain(S,1.5,opts);
```

Examine the `Sensitivity` field of `info`.

```
info.Sensitivity
```

```
ans = struct with fields:
    delta: 75
        k: 28
```

The values in this field indicate how much a change in the normalized perturbation on each element affects the performance margin. For example, the sensitivity for `k` is 28. This value means that a given change `dk` in the normalized uncertainty range of `k` causes a change of about 28% of that, or `0.28*dk`, in the performance margin. The margin in this case is more sensitive to `delta`, for which the margin changes by about 75% of the change in the normalized uncertainty range.

Robust Performance Margin as a Function of Frequency

Consider a model of a control system containing uncertain elements.

```
k = ureal('k',10,'Percent',40);
delta = ultidyn('delta',[1 1]);
G = tf(18,[1 1.8 k]) * (1 + 0.5*delta);
C = pid(2.3,3,0.38,0.001);
S = feedback(1,G*C);
```

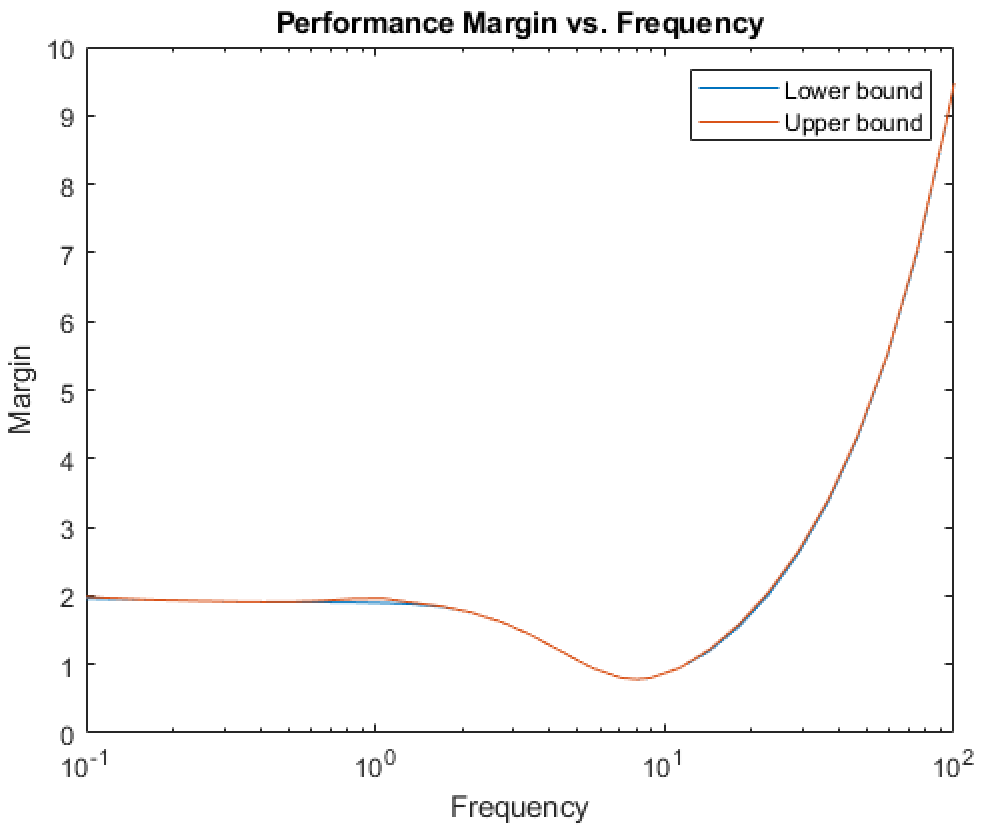
By default, `robgain` computes only the weakest performance margin over all frequencies. To see how the margin varies with frequency, use the `'VaryFrequency'` option of `robOptions`. For example, compute the performance margin of the system for a performance level of 1.5, at frequency points between 0.1 and 100 rad/s.

```
opts = robOptions('VaryFrequency','on');
[perfmarg,wcu,info] = robgain(S,1.5,{0.1,100},opts);
info
```

```
info = struct with fields:
                Model: 1
            Frequency: [32x1 double]
               Bounds: [32x2 double]
    WorstPerturbation: [32x1 struct]
          Sensitivity: [1x1 struct]
```

`robgain` returns the vector of frequencies in the `info` output, in the `Frequencies` field. `info.Bounds` contains the upper and lower bounds on the performance margin at each frequency. Use these values to plot the frequency dependence of the performance margin.

```
semilogx(info.Frequency,info.Bounds)
title('Performance Margin vs. Frequency')
ylabel('Margin')
xlabel('Frequency')
legend('Lower bound','Upper bound')
```



When you use the `'VaryFrequency'` option, `robgain` chooses frequency points automatically. The frequencies it selects are guaranteed to include the frequency at which the margin is smallest (within the specified range). Display the returned frequency values to confirm that they include the critical frequency.

```
info.Frequency
```

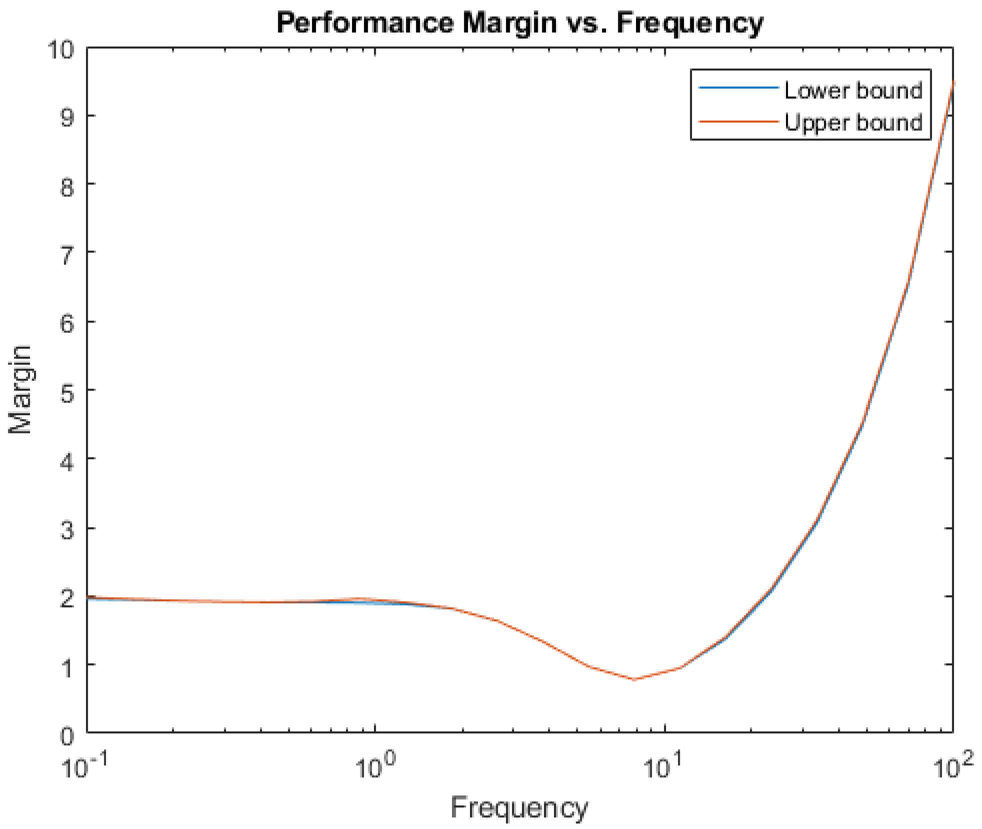
```
ans = 32×1

    0.1000
    0.1266
    0.1604
    0.2031
    0.2572
    0.3257
    0.4125
    0.5223
    0.6615
    0.8377
       ⋮
```

```
perfmarg.CriticalFrequency
```

```
ans = 7.9966
```

Alternatively, instead of using `'VaryFrequency'`, you can specify particular frequencies at which to compute the robust performance margins. `info.Bounds` contains the margins at all specified frequencies. However, these results are not guaranteed to include the weakest margin, which might fall between specified frequency points.

```
w = logspace(-1,2,20);
[perfmarg,wcu,info] = robgain(S,1.5,w);
semilogx(w,info.Bounds)
title('Performance Margin vs. Frequency')
ylabel('Margin')
xlabel('Frequency')
legend('Lower bound','Upper bound')
```



Input Arguments

usys — Dynamic system with uncertainty
`uss` | `ufrd` | `genss` | `genfrd`

Dynamic system with uncertainty, specified as a `uss`, `ufrd`, `genss`, or `genfrd` model that contains uncertain elements. For `genss` or `genfrd` models, `robgain` uses the current value of any tunable blocks and folds them into the known (not uncertain) part of the model.

`usys` can also be an array of uncertain models. In that case, `robgain` returns the smallest margin across all models in the array, and the `info` output contains the index of the corresponding model.

gamma — Performance level
positive scalar

Performance level, specified as a positive scalar. The performance level is the peak gain of a system or, for MIMO systems, the peak singular value (H_∞ norm). Generally, the lower this value, the better the system performance. `robgain` computes the amount of uncertainty the system can tolerate while keeping the peak gain below this level. For more information about this performance measure, see “Robustness and Worst-Case Analysis”.

`w` — Frequencies
`{wmin,wmax}` | vector

Frequencies at which to compute robust performance margins, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the margins at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the margins at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

`opts` — Options for margin computation
`robOptions` object

Options for computation of robust performance margins, specified as an object you create with `robOptions`. The available options include settings that let you:

- Extract frequency-dependent performance margins.
- Examine the sensitivity of the margins to each uncertain element.
- Improve the results of the performance-margin calculation by setting certain options for the underlying `mussv` calculation. In particular, setting the option `'MussvOptions'` to `'mN'` can reduce the gap between the lower bound and upper bound. `N` is the number of restarts.

For more information about all available options, see `robOptions`.

Example: `robOptions('Sensitivity','on','MussvOptions','m3')`

Output Arguments

`perfmarg` — Robust performance margin and critical frequency
structure

Robust performance margin and critical frequency, returned as a structure containing the following fields:

Field	Description
`LowerBound`	Lower bound on the actual robust performance margin of the model with respect to `gamma`, returned as a scalar value. The exact margin is guaranteed to be no smaller than `LowerBound`. In other words, for all modeled uncertainty with normalized magnitude up to `LowerBound`, the system is guaranteed to have peak gain below `gamma`.
`UpperBound`	Upper bound on the actual robust performance margin, returned as a scalar value. The exact margin is guaranteed to be no larger than `UpperBound`. In other words, there exist some uncertain-element values associated with this magnitude that drive the peak gain above `gamma`. `robgain` returns one such instance in `wcu`.
`CriticalFrequency`	Frequency at which the performance margin is the smallest, in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `usys`.

A margin greater than 1 means that the gain of `usys` remains below `gamma` for all values of the uncertainty modeled in `usys`. A margin less than 1 means that at some frequency, the gain of `usys` exceeds `gamma` for some values of the uncertain elements within their specified ranges. For example, a margin of 0.5 implies the following:

- The gain of `usys` remains below `gamma` as long as the uncertain element values stay within 0.5 normalized units of their nominal values.

- There is a perturbation of size 0.5 normalized units that drives the peak gain above `gamma`.

Use `normalized2actual` to convert the normalized uncertainty value expressed in the performance margin to actual deviations from nominal values.

If the nominal value of `usys` has peak gain greater than `gamma`, the performance margin is 0.

If `usys` is an array of uncertain models, `perfmarg` contains the smallest margin across all models in the array. In that case, the `info` output contains the index of the corresponding model in its `Model` field.

`wcu` — Perturbations driving system gain to gamma
structure

Smallest perturbations of uncertain elements that drive the peak gain of `usys` to the level `gamma`, returned as a structure whose fields are the names of the uncertain elements of `usys`. Each field contains the actual value of the corresponding uncertain element. For example, if `usys` includes an uncertain matrix `M` and SISO uncertain dynamics `delta`, then `wcu.M` is a numeric matrix and `wcu.delta` is a SISO state-space model.

Use `usubs(usys,wcu)` to substitute these values for the uncertain elements in `usys` and obtain the corresponding dynamic system. This system has peak gain `gamma`.Use `actual2normalized` to convert these actual uncertainty values to the normalized units in which the performance margin is expressed.

For `ureal` parameters in `usys` whose range is not centered around the nominal value, `robgain` makes the following adjustments for the purposes of its analysis:

- When the worst perturbation (the smallest perturbation achieving target gain) lies outside the range of validity of the actual-to-normalized transformation (see `getLimits`), then `robgain` sets the corresponding entry of `wcu` to the nearest valid value. In other words, if `actpert` is the worst perturbation in actual units, `robgain` sets `wcu` to the nearest value inside the interval `ActLims` returned by `getLimits`.
- When there is no perturbation causing the system to exceed the target gain, then `robgain` sets the corresponding entry of `wcu` to the nominal value of the `ureal` parameter.

`info` — Additional information about performance margins
structure

Additional information about the performance margins, returned as a structure with the following fields:

Field	Description
`Model`	Index of the model that has the weakest performance margin, when `usys` is an array of models.
`Frequency`	Frequency points at which `robgain` returns the robust performance margin, returned as a vector. • If the `'VaryFrequency'` option of `robOptions` is `'off'`, then `info.Frequency` is the critical frequency, the frequency at which the smallest margin occurs. If the smallest lower bound and the smallest upper bound on the performance margin occur at different frequencies, then `info.Frequency` is a vector containing these two frequencies. • If the `'VaryFrequency'` option of `robOptions` is `'on'`, then `info.Frequency` contains the frequencies selected by `robgain`. These frequencies are guaranteed to include the frequency at which the performance margin is smallest. • If you specify a vector of frequencies `w` at which to compute the performance margins, then `info.Frequency = w`. When you specify a frequency vector, these frequencies are not guaranteed to include the frequency at which the margin is smallest. The `'VaryFrequency'` option is meaningful only for `uss` and `genss` models. `robgain` ignores the option for `ufrd` and `genfrd` models.

Field	Description
`Bounds`	Lower and upper bounds on the actual robust performance margin of the model, returned as an array. `info.Bounds(:,1)` contains the lower bound at each corresponding frequency in `info.Frequency`, and `info.Bounds(:,2)` contains the corresponding upper bounds.
`WorstPerturbation`	Smallest perturbations at each frequency point in `info.Frequency`, returned as a structure array. The fields of `info.WorstPerturbation` are the names of the uncertain elements in `usys`. Each field contains the value of the corresponding element that drives the peak gain above `gamma` at each frequency. For example, if `usys` includes an uncertain parameter `p` and SISO uncertain dynamics `delta`, then `info.WorstPerturbation.p` is a collection of numeric values and `info.WorstPerturbation.delta` is a collection of SISO state-space models.
`Sensitivity`	Sensitivity of the performance margin to each uncertain element, returned as a structure when the `'Sensitivity'` option of `robOptions` is `'on'`. The fields of `info.Sensitivity` are the names of the uncertain elements in `usys`. Each field contains a percentage that measures how much the uncertainty in the corresponding element affects the performance margin. For example, if `info.Sensitivity.p` is 50, then a given fractional change in the uncertainty range of `p` causes half as much fractional change in the performance margin. If the `'Sensitivity'` option of `robOptions` is off (the default setting), then `info.Sensitivity` is `NaN`.

Algorithms

Computing the robustness margin at a particular frequency is equivalent to computing the structured singular value, μ, for some appropriate block structure (μ-analysis).

For `uss` and `genss` models, `robgain(usys)` and `robgain(usys,{wmin,wmax})` use an algorithm that finds the smallest margin across frequency. This algorithm does not rely on frequency gridding and is not adversely affected by discontinuities of the μ structured singular value. See “Getting Reliable Estimates of Robustness Margins” for more information.

For `ufrd` and `genfrd` models, `robgain` computes the μ lower and upper bounds at each frequency point. This computation offers no guarantee between frequency points and can be inaccurate if there are discontinuities or sharp peaks in μ. The syntax `robgain(uss,w)`, where `w` is a vector of frequency points, is the same as `robgain(ufrd(uss,w))` and also relies on frequency gridding to compute the margin.

In general, the algorithm for state-space models is faster and safer than the frequency-gridding approach. In some cases, however, the state-space algorithm requires many μ calculations. In those cases, specifying a frequency grid as a vector `w` can be faster, provided that the robustness margin varies smoothly with frequency. Such smooth variation is typical for systems with dynamic uncertainty.

See Also

`robOptions` | `robstab` | `wcgain`

Topics

“Robust Stability, Robust Performance and Mu Analysis”
“Robustness and Worst-Case Analysis”

Introduced in R2016b

robOptions

Option set for robustness analysis

Syntax

```
opts = robOptions
opts = robOptions(Name,Value,...)
```

Description

`opts = robOptions` returns the default option set for robustness analysis commands `robstab` and `robgain`, and for `musynperf`.

`opts = robOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Examples

Options for Robustness Margin Calculation

Create an options set for a `robstab`, `robgain`, or `musynperf` calculation that displays the progress of the underlying `mussv` calculation. Also, turn on the element-by-element sensitivity calculation.

```
opts = robOptions('Display','on','Sensitivity','on');
```

Alternatively, create a default option set, and use dot notation to set the values of particular options.

```
opts = robOptions;
opts.Display = 'on';
opts.Sensitivity = 'on';
```

Use `opts` as an input argument to `robstab`, `robgain`, or `musynperf`.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Display','on','Sensitivity','on'`

`Display` — Display progress of computation and summary report
`'off'` (default) | `'on'`

Display progress and summary of the robustness computation, specified as the comma-separated pair consisting of `'Display'` and one of these values:

- `'off'` — Do not display progress and report.
- `'on'` — Display progress and report. When you use this option, a progress indicator and summary of results is displayed in the command window, similar to the following.

```
points completed (of 28) ... 28
System is robustly stable for the modeled uncertainty.
 -- It can tolerate up to 116% of the modeled uncertainty.
 -- There is a destabilizing perturbation amounting to 117% of the modeled uncertair
 -- This perturbation causes an instability at the frequency 5.9 rad/seconds.
```

This setting overrides the silent (`'s'`) option in the `MussvOptions` option.

`VaryFrequency` — Compute robustness margin as function of frequency
`'off'` (default) | `'on'`

Return robustness margin as a function of frequency, specified as the comma-separated pair consisting of `'VaryFrequency'` and one of these values:

- `'off'` — Only return margins at frequencies where robustness is weakest.
- `'on'` — Compute robustness margins over a frequency grid suitable for plotting. The frequency grid is chosen automatically based on system dynamics. This calculation is done in addition to identifying the critical frequency where the margin is weakest.

Access the frequency values and corresponding margins in the `info` output of `robstab` and `robgain`.

This option is ignored for `ufrd` and `genfrd` models.

`Sensitivity` — Calculate sensitivity of robustness margin

`'off'` (default) | `'on'`

Calculate the sensitivity of the robustness margin to each uncertain element in the model, specified as the comma-separated pair consisting of `'Sensitivity'` and either `'off'` or `'on'`.

Each uncertain element contributes to the overall stability margin in a coupled manner. Set this option to `'on'` to estimate the sensitivity of the margin to each element. This element-by-element sensitivity provides an indication of which elements are most problematic for robustness. Access the sensitivity estimates in the `info` output of `robstab` and `robgain`.

`SensitivityPercent` — Percentage variation of uncertainty for computing sensitivity

25 (default) | positive scalar value

Percentage variation of uncertainty level for computing sensitivity, specified as the comma-separated pair consisting of `'SensitivityPercent'` and a positive scalar value. The sensitivity to a particular uncertain element is estimated using a finite difference calculation. This calculation increases the (normalized) amount of uncertainty on this element by some percentage, computes the resulting robustness, and computes the ratio of percent variations. This option specifies the percentage increase in uncertainty level applied to each element. The default value is 25%.

`MussvOptions` — Options for `mussv` calculation

`''` (default) | character vector

Options for the underlying `mussv` calculation that `robstab` and `robgain` perform, specified as the comma-separated pair consisting of `'MussvOptions'` and a character vector such as `'sm3'` or `'ad'`.

Some `MussvOptions` values that are especially useful for improving robustness-margin calculations include:

- `'a'` — Force the use of LMI optimization to compute the μ upper bound, which yields better results in general but can be expensive when some `ureal` elements are repeated multiple times.
- `'mN'` — Use multiple restarts when computing the μ lower bound, which corresponds to the upper bound for robustness margins. This option can reduce the gap between the lower bound and upper bound on the robustness margins. `N` is the number of restarts. For example, setting `'MussvOptions'` to `'m3'` causes three restarts.

See `mussv` for the remaining available options and corresponding characters. The default, `''`, uses the default options for `mussv`.

Output Arguments

`opts` — Options for robustness commands
`robOptions` object

Options for robustness commands `robstab`, `robgain`, and `musynperf`, returned as a `robOptions` object. Use the options as an input argument to `robstab`, `robgain`, or `musynperf`. For example:

```
[stabmarg,wcu,info] = robstab(usys,opts)
```

See Also

`musynperf` | `robgain` | `robstab`

Topics

“Robust Stability, Robust Performance and Mu Analysis”
“Robustness and Worst-Case Analysis”

Introduced in R2016b

robstab

Robust stability of uncertain system

Syntax

```
[stabmarg,wcu] = robstab(usys)
[stabmarg,wcu] = robstab(usys,w)
[stabmarg,wcu] = robstab( ___ ,opts)
[stabmarg,wcu,info] = robstab( ___ )
```

Description

`[stabmarg,wcu] = robstab(usys)` calculates the robust stability margin for an uncertain system. This stability margin is relative to the uncertainty level specified in `usys`. A robust stability margin greater than 1 means that the system is stable for all values of its modeled uncertainty. A robust stability margin less than 1 means that the system becomes unstable for some values of the uncertain elements within their specified ranges. For example, a margin of 0.5 implies the following:

- `usys` remains stable as long as the uncertain element values stay within 0.5 normalized units of their nominal values.
- There is a destabilizing perturbation of size 0.5 normalized units.

The structure `stabmarg` contains upper and lower bounds on the actual stability margin and the critical frequency at which the stability margin is smallest. The structure `wcu` contains the destabilizing values of the uncertain elements.

`[stabmarg,wcu] = robstab(usys,w)` restricts the robust stability margin computation to the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `robstab` restricts the stability margin computation to the interval between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `robstab` computes the robust stability margin at the specified frequencies only.

`[stabmarg,wcu] = robstab(___,opts)` specifies additional options for the computation. Use `robOptions` to create `opts`. You can use this syntax with any of the previous input-argument combinations.

`[stabmarg,wcu,info] = robstab(___)` returns a structure with additional information about the stability margins and destabilizing perturbations. See `info` for details about this structure. You can use this syntax with any of the previous input-argument combinations.

Examples

Robust Stability Margin of Closed-Loop System

Consider a control system whose plant contains both parametric uncertainty and dynamic uncertainty. Create a model of the plant using uncertain elements.

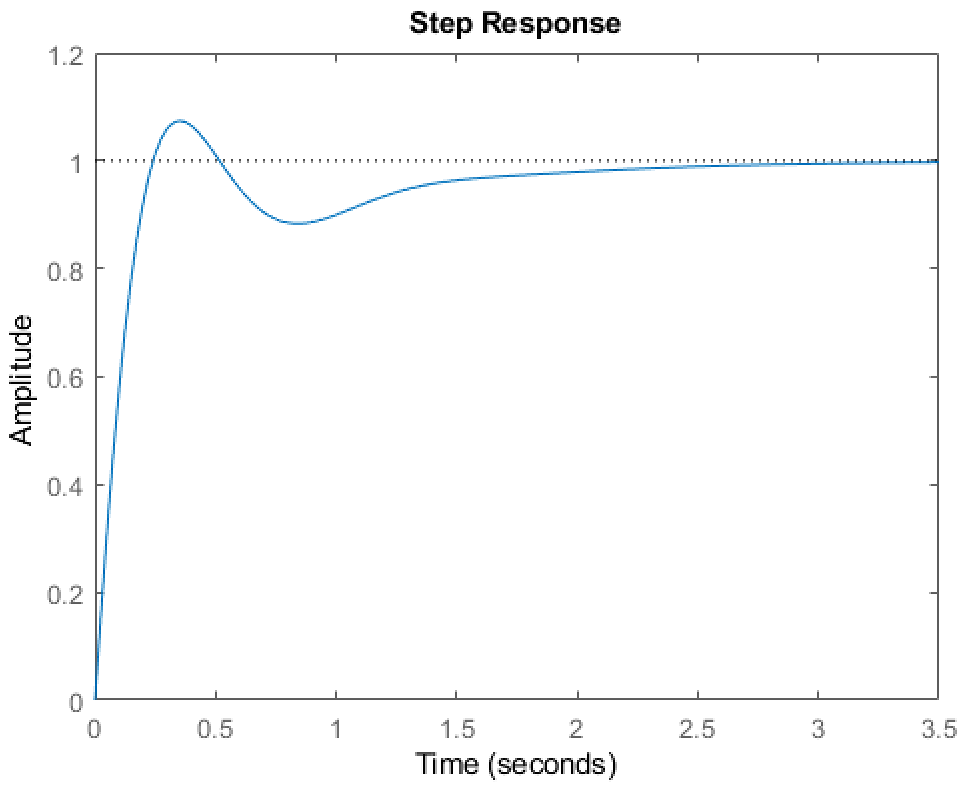
```
k = ureal('k',10,'Percent',40);
delta = ultidyn('delta',[1 1]);
G = tf(18,[1 1.8 k]) * (1 + 0.5*delta);
```

Create a model of the controller, and build the closed-loop transfer function.

```
C = pid(2.3,3,0.38,0.001);
CL = feedback(G*C,1);
```

A step response plot shows that the closed-loop system is nominally stable.

```
step(CL.NominalValue)
```



Examine the robust stability of the closed-loop system.

```
[stabmarg,wcu] = robstab(CL);
stabmarg
```

```
stabmarg = struct with fields:
           LowerBound: 1.5960
           UpperBound: 1.5993
    CriticalFrequency: 4.8627
```

The `LowerBound` and `UpperBound` fields of `stabmarg` show the robust stability margin of the closed-loop system is around 1.6. This result means that the system can withstand about 60% more uncertainty than is specified in the uncertain elements without going unstable.

The output `wcu` is a structure that contains the smallest perturbation to `k` and `delta` that make the system unstable. Confirm the instability by substituting these values into the closed-loop model and examining the pole locations.

```
CLunst = usubs(CL,wcu);
pole(CLunst)
```

```
ans = 8×1 complex
10^2 ×

  -9.9314 + 0.0000i
  -0.1027 + 0.1009i
  -0.1027 - 0.1009i
   0.0000 + 0.0486i
   0.0000 - 0.0486i
  -0.0115 + 0.0000i
  -0.0216 + 0.0000i
  -0.0403 + 0.0000i
```

The resulting system has an undamped pair of complex poles with natural frequency 4.89, which renders it unstable. The `CriticalFrequency` field of `stabmarg` contains the same value, which is the frequency at which the `CL` is closest to instability.

Sensitivity to Uncertain Elements

Examine the relative sensitivity of the robust stability margin to the uncertain elements of the system. Consider a model of a control system containing uncertain elements.

```
k = ureal('k',10,'Percent',40);
delta = ultidyn('delta',[1 1]);
G = tf(18,[1 1.8 k]) * (1 + 0.25*delta);
C = pid(2.3,3,0.38,0.001);
CL = feedback(G*C,1);
```

Create an options set for `robstab` that enables the sensitivity calculation.

```
opts = robOptions('Sensitivity','On');
```

Calculate the robust stability margin, specifying the `info` output to access additional information about the calculation.

```
[stabmarg,wcu,info] = robstab(CL,opts);
```

Examine the `Sensitivity` field of `info`.

```
info.Sensitivity
```

```
ans = struct with fields:
    delta: 80
        k: 20
```

The values in this field indicate how much a change in the normalized perturbation on each element affects the stability margin. For example, the sensitivity for `k` is 21. This value means that a given change `dk` in the normalized uncertainty range of `k` causes a change of about 21% percent of that, or `0.21*dk`, in the stability margin. The margin in this case is much more sensitive to `delta`, for which the margin changes by about 81% of the change in the normalized uncertainty range.

Robust Stability Margin as a Function of Frequency

Consider a model of a control system containing uncertain elements.

```
k = ureal('k',10,'Percent',40);
delta = ultidyn('delta',[1 1]);
G = tf(18,[1 1.8 k]) * (1 + 0.5*delta);
C = pid(2.3,3,0.38,0.001);
CL = feedback(G*C,1);
```

By default, `robstab` computes only the weakest stability margin over all frequencies. To see how the stability margin varies with frequency, use the `'VaryFrequency'` option of `robOptions`. For example, compute the stability margin of the system at frequency points between 0.1 and 10 rad/s.

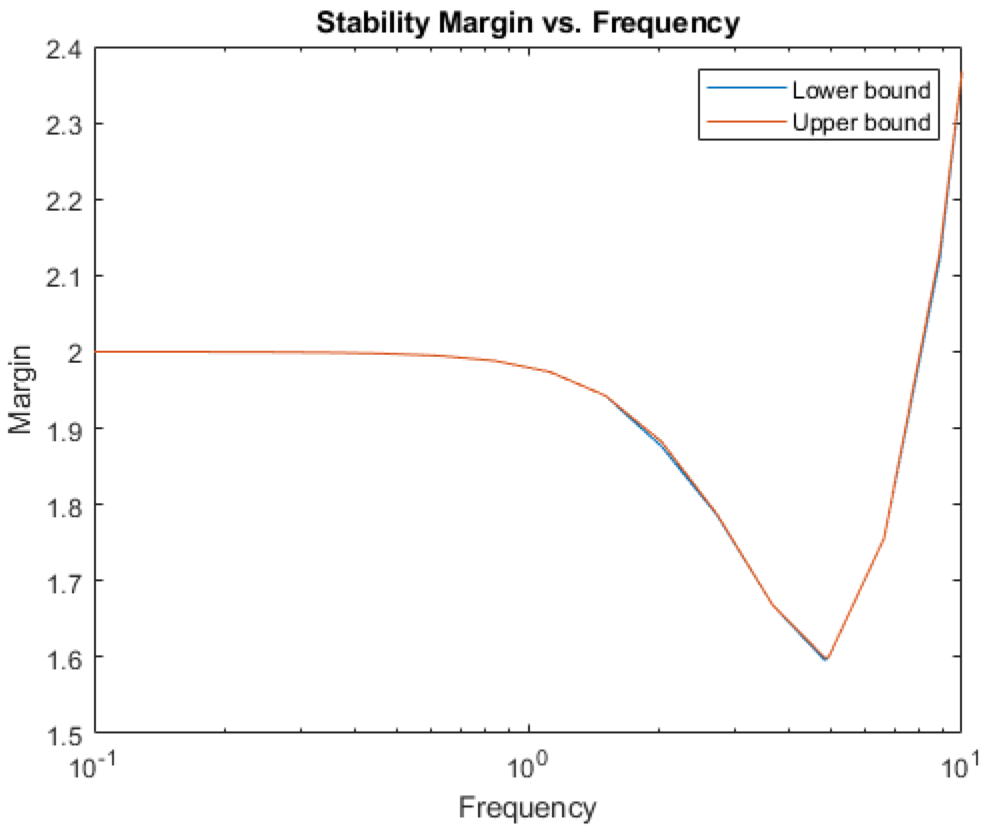
```
opts = robOptions('VaryFrequency','on');
[stabmarg,wcu,info] = robstab(CL,{0.1,10},opts);
info
```

```
info = struct with fields:
                Model: 1
```

```
              Frequency: [19x1 double]
                 Bounds: [19x2 double]
    WorstPerturbation: [19x1 struct]
            Sensitivity: [1x1 struct]
```

`robstab` returns the vector of frequencies in the `info` output, in the `Frequencies` field. `info.Bounds` contains the upper and lower bounds on the stability margin at each frequency. Use these values to plot the frequency dependence of the stability margin.

```
semilogx(info.Frequency,info.Bounds)
title('Stability Margin vs. Frequency')
ylabel('Margin')
xlabel('Frequency')
legend('Lower bound','Upper bound')
```



When you use the `'VaryFrequency'` option, `robstab` chooses frequency points automatically. The frequencies it selects are guaranteed to include the frequency at which the stability margin is weakest (within the specified range). Display the returned frequency values to confirm that they include the critical frequency.

```
info.Frequency
```

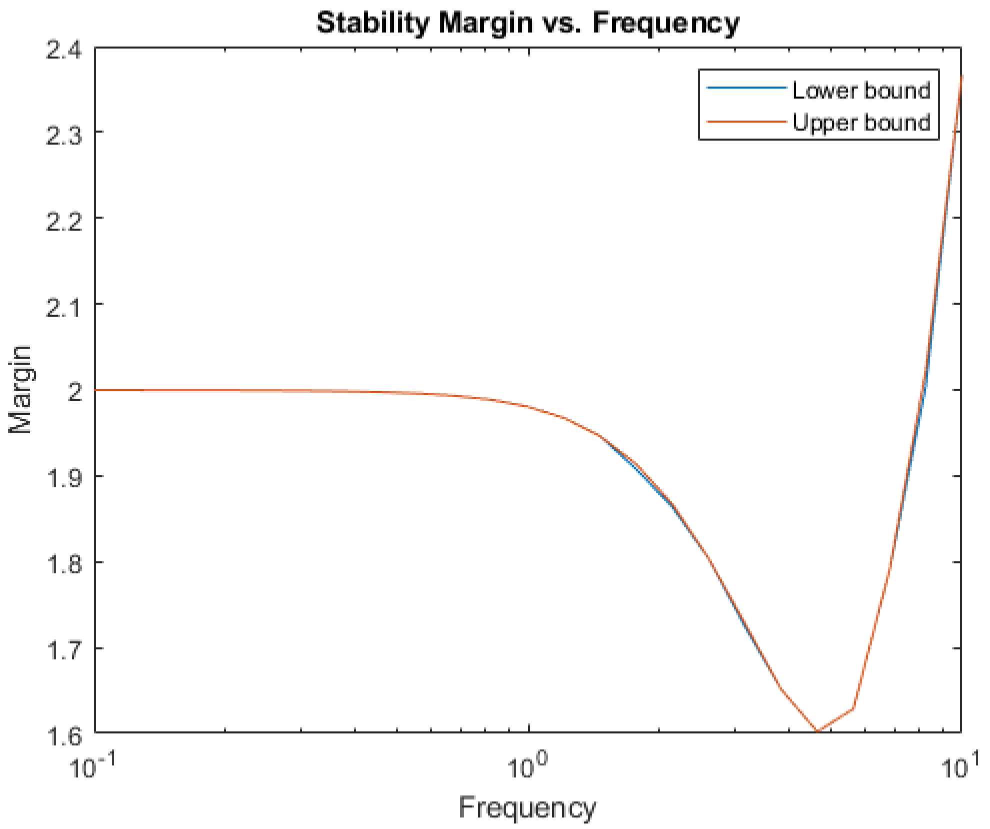
```
ans = 19×1

    0.1000
    0.1061
    0.1425
    0.1914
    0.2572
    0.3455
    0.4642
    0.6236
    0.8377
    1.1253
       ⋮
```

```
stabmarg.CriticalFrequency
```

```
ans = 4.8280
```

Alternatively, instead of using `'VaryFrequency'`, you can specify particular frequencies at which to compute the robust stability margins. `info.Bounds` contains the margins at all specified frequencies. However, these results are not guaranteed to include the weakest margin, which might fall between specified frequency points.

```
w = logspace(-1,1,25);
[stabmarg,wcu,info] = robstab(CL,w);
semilogx(w,info.Bounds)
title('Stability Margin vs. Frequency')
ylabel('Margin')
xlabel('Frequency')
legend('Lower bound','Upper bound')
```



Input Arguments

usys — Dynamic system with uncertainty
`uss` | `ufrd` | `genss` | `genfrd`

Dynamic system with uncertainty, specified as a `uss`, `ufrd`, `genss`, or `genfrd` model that contains uncertain elements. For `genss` or `genfrd` models, `robstab` uses the current value of any tunable blocks and folds them into the known (not uncertain) part of the model.

For frequency-response models, `ufrd` or `genfrd`, `robstab` assumes that the system is nominally stable.

`usys` can also be an array of uncertain models. In that case, `robstab` returns the smallest margin across all models in the array, and the `info` output contains the index of the corresponding model.

`w` — Frequencies
`{wmin,wmax}` | vector

Frequencies at which to compute robust stability margins, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function computes the margins at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the margins at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values.

  For `uss` and `genss` models, when `w` is a vector, `robstab(usys,w)` is equivalent to `robstab(ufrd(usys,w))`. Therefore, `usys` must be nominally stable.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

`opts` — Options for margin computation
`robOptions` object

Options for computation of robust stability margins, specified as an object you create with `robOptions`. The available options include settings that let you:

- Extract frequency-dependent stability margins.
- Examine the sensitivity of the margins to each uncertain element.
- Improve the results of the stability-margin calculation by setting certain options for the underlying `mussv` calculation. In particular, setting the option `'MussvOptions'` to `'mN'` can reduce the gap between the lower bound and upper bound. `N` is the number of restarts.

For more information about all available options, see `robOptions`.

Example: `robOptions('Sensitivity','on','MussvOptions','m3')`

Output Arguments

stabmarg — Robust stability margin and critical frequency

structure

Robust stability margin and critical frequency, returned as a structure containing the following fields:

Field	Description
LowerBound	Lower bound on the actual robust stability margin of the model, returned as a scalar value. The exact stability margin is guaranteed to be no smaller than LowerBound. In other words, for all modeled uncertainty with normalized magnitude up to LowerBound, the system is guaranteed stable.
UpperBound	Upper bound on the actual robust stability margin of the model, returned as a scalar value. The exact stability margin is guaranteed to be no larger than UpperBound. In other words, there exist some uncertain-element values associated with this magnitude that cause instability. robstab returns one such instance in wcu.
CriticalFrequency	Frequency at which the stability margin is the smallest, in rad/TimeUnit, where TimeUnit is the TimeUnit property of usys.

A robust stability margin greater than 1 means that usys is stable for all values of its modeled uncertainty. A robust stability margin less than 1 implies that usys becomes unstable for some values of its uncertain elements within their specified ranges. For example, a margin of 0.5 implies the following:

- usys remains stable as long as the uncertain element values stay within 0.5 normalized units of their nominal values.
- There is a destabilizing perturbation of size 0.5 normalized units.

Use normalized2actual to convert the normalized uncertainty value expressed in the stability margin to actual deviations from nominal values.

If the nominal value of usys is unstable, the stability margin is 0. If usys is a ufrd or genfrd model, robstab assumes it is nominally stable.

If `usys` is an array of uncertain models, `stabmarg` contains the smallest margin across all models in the array. In that case, the `info` output contains the index of the corresponding model in its `Model` field.

`wcu` — Perturbations causing instability

structure

Smallest perturbations of uncertain elements that cause instability in `usys`, returned as a structure whose fields are the names of the uncertain elements of `usys`. Each field contains the actual destabilizing value for each uncertain element of `usys`. For example, if `usys` includes an uncertain matrix `M` and SISO uncertain dynamics `delta`, then `wcu.M` is a numeric matrix and `wcu.delta` is a SISO state-space model.

Use `usubs(usys,wcu)` to substitute these values for the uncertain elements in `usys`, to obtain the unstable dynamic system that deviates the least from the nominal system. Use `actual2normalized` to convert these actual uncertainty values to the normalized units in which the stability margin is expressed.

For `ureal` parameters in `usys` whose range is not centered around the nominal value, `robstab` makes the following adjustments for the purposes of its analysis:

- When the worst perturbation (the smallest destabilizing perturbation) lies outside the range of validity of the actual-to-normalized transformation (see `getLimits`), then `robstab` sets the corresponding entry of `wcu` to the nearest valid value. In other words, if `actpert` is the worst perturbation in actual units, `robgain` sets `wcu` to the nearest value inside the interval `ActLims` returned by `getLimits`.
- When there is no destabilizing perturbation, then `robstab` sets the corresponding entry of `wcu` to the nominal value of the `ureal` parameter.

`info` — Additional information about stability margins

structure

Additional information about the robust stability margins, returned as a structure with the following fields:

Field	Description
`Model`	Index of the model that has the weakest stability margin, when `usys` is an array of models.

Field	Description
`Frequency`	Frequency points at which `robstab` returns the robust stability margin, returned as a vector. • If the `'VaryFrequency'` option of `robOptions` is `'off'`, then `info.Frequency` is the critical frequency, the frequency at which the smallest margin occurs. If the smallest lower bound and the smallest upper bound on the stability margin occur at different frequencies, then `info.Frequency` is a vector containing these two frequencies. • If the `'VaryFrequency'` option of `robOptions` is `'on'`, then `info.Frequency` contains the frequencies selected by `robstab`. These frequencies are guaranteed to include the frequency at which the stability margin is smallest. • If you specify a vector of frequencies `w` at which to compute the stability margins, then `info.Frequency = w`. When you specify a frequency vector, these frequencies are not guaranteed to include the frequency at which the stability margin is smallest. The `'VaryFrequency'` option is meaningful only for `uss` and `genss` models. `robstab` ignores the option for `ufrd` and `genfrd` models.
`Bounds`	Lower and upper bounds on the actual robust stability margin of the model, returned as an array. `info.Bounds(:,1)` contains the lower bound at each corresponding frequency in `info.Frequency`, and `info.Bounds(:,2)` contains the corresponding upper bounds.

Field	Description
`WorstPerturbation`	Smallest destabilizing perturbations at each frequency point in `info.Frequency`, returned as a structure array. The fields of `info.WorstPerturbation` are the names of the uncertain elements in `usys`, and each field contains the destabilizing value of the corresponding element at each frequency. For example, if `usys` includes an uncertain parameter `p` and SISO uncertain dynamics `delta`, then `info.WorstPerturbation.p` is a collection of numeric values and `info.WorstPerturbation.delta` is a collection of SISO state-space models.
`Sensitivity`	Sensitivity of the stability margin to each uncertain element, returned as a structure when the `'Sensitivity'` option of `robOptions` is `'on'`. The fields of `info.Sensitivity` are the names of the uncertain elements in `usys`. Each field contains a percentage that measures how much the uncertainty in the corresponding element affects the stability margin. For example, if `info.Sensitivity.p` is 50, then a given fractional change in the uncertainty range of `p` causes half as much fractional change in the stability margin. If the `'Sensitivity'` option of `robOptions` is off (the default setting), then `info.Sensitivity` is `NaN`.

Algorithms

Computing the robustness margin at a particular frequency is equivalent to computing the structured singular value, μ, for some appropriate block structure (μ-analysis).

For `uss` and `genss` models, `robstab(usys)` and `robstab(usys,{wmin,wmax})` use an algorithm that finds the smallest margin across frequency. This algorithm does not rely on frequency gridding and is not adversely affected by discontinuities of the μ structured singular value. See “Getting Reliable Estimates of Robustness Margins” for more information.

For `ufrd` and `genfrd` models, `robstab` computes the μ lower and upper bounds at each frequency point. This computation offers no guarantee between frequency points and can be inaccurate if there are discontinuities or sharp peaks in μ. The syntax `robstab(uss,w)`, where `w` is a vector of frequency points, is the same as `robstab(ufrd(uss,w))` and also relies on frequency gridding to compute the margin.

In general, the algorithm for state-space models is faster and safer than the frequency-gridding approach. In some cases, however, the state-space algorithm requires many μ calculations. In those cases, specifying a frequency grid as a vector `w` can be faster, provided that the robustness margin varies smoothly with frequency. Such smooth variation is typical for systems with dynamic uncertainty.

See Also

`actual2normalized` | `mussv` | `normalized2actual` | `robOptions` | `robgain` | `wcgain`

Topics

“Robust Stability and Worst-Case Gain of Uncertain System”
“Robust Stability, Robust Performance and Mu Analysis”
“Robustness and Worst-Case Analysis”

Introduced in R2016b

robustperf

(Not recommended) Robust performance margin of uncertain multivariable system

Note `robustperf` is not recommended. Use `robgain` instead.

Syntax

```
perfmarg = robustperf(usys)
```

```
[perfmarg,wcu,report,info] = robustperf(usys)
```

```
[perfmarg,wcu,report,info] = robustperf(usys,opt)
```

Description

The performance of a nominally stable uncertain system model will generally degrade for specific values of its uncertain elements. `robustperf`, largely included for historical purposes, computes the robust performance margin, which is one measure of the level of degradation brought on by the modeled uncertainty.

As with other *uncertain-system* analysis tools, only bounds on the performance margin are computed. The exact robust performance margin is guaranteed to lie between these upper and lower bounds.

The computation used in `robustperf` is a frequency-domain calculation. Coupled with stability of the nominal system, this frequency domain calculation determines robust performance of `usys`. If the input system `usys` is a `ufrd`, then the analysis is performed on the frequency grid within the `ufrd`. Note that the stability of the nominal system is not verified by the computation. If the input system sys is a `uss`, then the stability of the nominal system is first checked, an appropriate frequency grid is generated (automatically), and the analysis performed on that frequency grid. In all discussion that follows, *N* denotes the number of points in the frequency grid.

Basic Syntax

Suppose `usys` is a `ufrd` or `uss` with *M* uncertain elements. The results of

```
[perfmarg,perfmargunc,Report] = robustperf(usys)
```

are such that `perfmarg` is a structure with the following fields:

Field	Description
`LowerBound`	Lower bound on robust performance margin, positive scalar.
`UpperBound`	Upper bound on robust performance margin, positive scalar.
`CriticalFrequency`	The value of frequency at which the performance degradation curve crosses the $y = 1/x$ curve. See "Robustness and Worst-Case Analysis".

`perfmargunc` is a `struct` of values of uncertain elements associated with the intersection of the performance degradation curve and the $y = 1/x$ curve. See "Robustness and Worst-Case Analysis". There are *M* field names, which are the names of uncertain elements of `usys`.

`Report` is a text description of the robust performance analysis results.

If `usys` is an array of uncertain models, the outputs are struct arrays whose entries correspond to each model in the array.

Examples

Create a plant with a nominal model of an integrator, and include additive unmodeled dynamics uncertainty of a level of 0.4 (this corresponds to 100% model uncertainty at 2.5 rads/s).

```
P = tf(1,[1 0]) + ultidyn('delta',[1 1],'bound',0.4);
```

Design a "proportional" controller *K* that puts the nominal closed-loop bandwidth at 0.8 rad/s. Roll off *K* at a frequency 25 times the nominal closed-loop bandwidth. Form the closed-loop sensitivity function.

```
BW = 0.8;
K = tf(BW,[1/(25*BW) 1]);
S = feedback(1,P*K);
```

Assess the performance margin of the closed-loop sensitivity function. Because the nominal gain of the sensitivity function is 1, and the performance degradation curve is monotonically increasing (see “Robustness and Worst-Case Analysis”), the performance margin should be less than 1.

```
[perfmargin,punc] = robustperf(S);
perfmargin
perfmargin =
           UpperBound: 7.4305e-001
           LowerBound: 7.4305e-001
    CriticalFrequency: 5.3096e+000
```

You can verify that the upper bound of the performance margin corresponds to a point on or above the *y*=1/*x* curve. First, compute the normalized size of the value of the uncertain element, and check that this agrees with the upper bound.

```
nsize = actual2normalized(S.Uncertainty.delta, punc.delta)
nsize =
perfmargin.UpperBound
ans =
  7.4305e-001
```

Compute the system gain with that value substituted, and verify that the product of the normalized size and the system gain is greater than or equal to 1.

```
gain = norm(usubs(S,punc),inf,.00001);
nsize*gain
ans =
  1.0000e+000
```

Finally, as a sanity check, verify that the robust performance margin is less than the robust stability margin.

```
[stabmargin] = robuststab(S);
stabmargin
stabmargin =
                UpperBound: 3.1251e+000
                LowerBound: 3.1251e+000
    DestabilizingFrequency: 4.0862e+000
```

While the robust stability margin is easy to describe (poles migrating from stable region into unstable region), describing the robust performance margin is less elementary. See the diagrams and figures in “Robustness and Worst-Case Analysis”. Rather than finding values for uncertain elements that lead to instability, the analysis finds values of uncertain elements “corresponding to the intersection point of the performance degradation curve with a *y*=1/*x* hyperbola.” This characterization, mentioned above in the description of perfmarg.CriticalFrequency and perfmargunc, is used often in the descriptions below.

Basic Syntax with Fourth Output Argument

A fourth output argument yields more specialized information, including sensitivities and frequency-by-frequency information.

```
[perfmarg,perfmargunc,Report,Info] = robustperf(usys)
```

In addition to the first 3 output arguments, described previously, Info is a structure with the following fields:

Field	Description
Sensitivity	A struct with *M* fields, field names are names of uncertain elements of usys. Values of fields are positive and contain the local sensitivity of the overall Stability Margin to that element's uncertainty range. For instance, a value of 25 indicates that if the uncertainty range is enlarged by 8%, then the stability margin should drop by about 2% (25% of 8). If the Sensitivity property of the robustperfOptions object is 'off', the values are set to NaN.
Frequency	*N*-by-1 frequency vector associated with analysis.
BadUncertainValues	*N*-by-1 struct array containing the worst uncertain element values at each frequency.
MussvBnds	A 1-by-2 frd, with upper and lower bounds from mussv. The (1,1) entry is the µ-upper bound (corresponds to perfmarg.LowerBound) and the (1,2) entry is the µ-lower bound (for perfmarg.UpperBound).
MussvInfo	Structure of compressed data from mussv.

Specifying Additional Options

Use robustperfOptions to specify additional options for the robustperf computation. For example, you can control what is displayed during the computation, turn the sensitivity computation on or off, set the step size in the sensitivity computation, or control the option argument used in the underlying call to mussv. For example, you can turn the display on and turn off the sensitivity by executing

```
opt = robustperfOptions('Sensitivity','off','Display','on');
[PerfMarg,Destabunc,Report,Info] = robustperf(usys,opt)
```

See the robustperfOptions reference page for more information about available options.

Limitations

Because the calculation is carried out with a frequency gridding, it is possible (likely) that the true critical frequency is missing from the frequency vector used in the analysis. This is similar to the problem in robuststab. However, in comparing to robuststab, the problem in robustperf is less acute. The robust performance margin, considered a function of problem data and frequency, is typically a continuous function (unlike the robust stability margin, described in “Getting Reliable Estimates of Robustness Margins”). Hence, in robust performance margin calculations, increasing the density of the frequency grid will always increase the accuracy of the answers, and in the limit, answers arbitrarily close to the actual answers are obtainable with finite frequency grids.

Algorithms

A rigorous robust performance analysis consists of two steps:

1. Verify that the nominal system is stable.
2. Robust performance analysis on an augmented system.

The algorithm in robustperf follows this in spirit, with the following limitations:

- If usys is a uss object, then robustperf explicitly checks the stability of the nominal value. However, if usys is a ufrd model, robustperf instead assumes that the nominal value is stable, and does not perform this check.
- The exact performance margin is guaranteed to be no larger than UpperBound (some uncertain elements associated with this magnitude cause instability – one instance is returned in the structure perfmargunc). The instability created by perfmargunc occurs at the frequency value in CriticalFrequency.

- Similarly, the exact performance margin is guaranteed to be no smaller than `LowerBound`.

See Also

`actual2normalized` | `mussv` | `norm` | `robgain` | `robstab` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

robustperfOptions

(Not recommended) Option set for `robustperf`

Syntax

```
options = robustperfOptions
options = robustperfOptions(Name,Value,...)
```

Note `robustperfOptions` is not recommended. Use `robOptions` instead.

Description

`options = robustperfOptions` returns the default option set for the `robustperf` command.

`options = robustperfOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`robustperfOptions` takes the following `Name` arguments:

`Display`

Specifies whether `robustperf` displays progress of `mussv` computations.

- `'off'` — Do not display progress.
- `'on'` — Display progress. This setting overrides the silent (`'s'`) option in the `Mussv` option.

Default: `'off'`

`Sensitivity`

Specifies whether `robustperf` computes the sensitivity of the performance margin with respect to each individual uncertain element. This element-by-element sensitivity provides an indication of which elements the performance margin is most sensitive to. Turning off the element-by-element sensitivity calculation speeds up `robustperf`.

- `'on'` — Compute the sensitivity for each uncertain element.
- `'off'` — Do not compute the sensitivity for each uncertain element.

Default: `'on'`

`VaryUncertainty`

Percentage variation of uncertainty for computing sensitivity. The sensitivity estimate uses a finite difference calculation.

Default: 25

`Mussv`

Options for the `mussv` calculation that `robustperf` performs. See `mussv` for the available options.

Default: `''` (default behavior of `mussv`)

Output Arguments

`options`

Option set containing the specified options for the `robustperf` command.

Examples

Create an options set for a `robustperf` calculation that displays the progress of the `mussv` calculation. Also, turn off the element-by-element sensitivity calculation.

```
 options = robustperfOptions('Display','on','Sensitivity','off');
```

Alternatively, use dot notation to set the values of `options`.

```
options = robustperfOptions;
options.Display = 'on';
options.Sensitivity = 'off';
```

See Also

`robOptions`

Introduced in R2011b

robuststab

(Not recommended) Calculate robust stability margins of uncertain multivariable system

Note `robuststab` is not recommended. Use `robstab` instead.

Syntax

```
[stabmarg,destabunc,report,info] = robuststab(sys)
```

```
[stabmarg,destabunc,report,info] = robuststab(sys,opt)
```

Description

A nominally stable uncertain system is generally unstable for specific values of its uncertain elements. Determining the values of the uncertain elements closest to their nominal values for which instability occurs is a *robust stability* calculation.

If the uncertain system is stable for all values of uncertain elements within their allowable ranges (ranges for `ureal`, norm bound or positive-real constraint for `ultidyn`, radius for `ucomplex`, weighted ball for `ucomplexm`), the uncertain system is *robustly stable*. Conversely, if there is a combination of element values that cause instability, and all lie within their allowable ranges, then the uncertain system is not robustly stable.

`robuststab` computes the margin of stability robustness for an uncertain system. A stability robustness margin greater than 1 means that the uncertain system is stable for all values of its modeled uncertainty. A stability robustness margin less than 1 implies that certain allowable values of the uncertain elements, within their specified ranges, lead to instability.

Numerically, a margin of 0.5 (for example) implies two things: the uncertain system remains stable for all values of uncertain elements that are less than 0.5 normalized units away from their nominal values and, there is a collection of uncertain elements that are less than or equal to 0.5 normalized units away from their nominal values that results in instability. Similarly, a margin of 1.3 implies that the uncertain system remains stable for all values of uncertain elements up to 30% outside their modeled uncertain ranges. See `actual2normalized` for converting between actual and normalized deviations from the nominal value of an uncertain element.

As with other *uncertain-system* analysis tools, only bounds on the exact stability margin are computed. The exact robust stability margin is guaranteed to lie in between these upper and lower bounds.

The computation used in `robuststab` is a frequency-domain calculation, which determines whether poles can migrate (due to variability of the uncertain atoms) across the stability boundary (imaginary axis for continuous-time, unit circle for discrete-time). Coupled with stability of the nominal system, determining that no migration occurs constitutes robust stability. If the input system `sys` is a `ufrd`, then the analysis is performed on the frequency grid within the `ufrd`. Note that the stability of the nominal system is not verified by the computation. If the input system sys is a `uss`, then the stability of the nominal system is first checked, an appropriate frequency grid is generated (automatically), and the analysis performed on that frequency grid. In all discussion that follows, *N* denotes the number of points in the frequency grid.

Basic Syntax

Suppose `sys` is a `ufrd` or `uss` with *M* uncertain elements. The results of

```
[stabmarg,destabunc,Report] = robuststab(sys)
```

are that `stabmarg` is a structure with the following fields

Field	Description
`LowerBound`	Lower bound on stability margin, positive scalar. If greater than 1, then the uncertain system is guaranteed stable for all values of the modeled uncertainty. If the nominal value of the uncertain system is unstable, then `stabmarg.UpperBound` and `stabmarg.LowerBound` both equal 0.
`UpperBound`	Upper bound on stability margin, positive scalar. If less than 1, the uncertain system is not stable for all values of the modeled uncertainty.

Field	Description
`DestabilizingFrequency`	The critical value of frequency at which instability occurs, with uncertain elements closest to their nominal values. At a particular value of uncertain elements (see `destabunc` below), the poles migrate across the stability boundary (imaginary-axis in continuous-time systems, unit-disk in discrete-time systems) at the frequency given by `DestabilizingFrequency`.

`destabunc` is a structure of values of uncertain elements, closest to nominal, that cause instability. There are *M* field names, which are the names of uncertain elements of `sys`. The value of each field is the corresponding value of the uncertain element, such that when jointly combined, lead to instability. The command `pole(usubs(sys,destabunc))` shows the instability. If `A` is an uncertain element of sys, then

```
actual2normalized(destabunc.A,sys.Uncertainty.A)
```

will be less than or equal to `UpperBound`, and for at least one uncertain element of `sys`, this normalized distance will be equal to `UpperBound`, proving that `UpperBound` is indeed an upper bound on the robust stability margin.

`Report` is a text description of the arguments returned by `robuststab`.

If `sys` is an array of uncertain models, the outputs are struct arrays whose entries correspond to each model in the array.

Examples

Construct a feedback loop with a second-order plant and a PID controller with approximate differentiation. The second-order plant has frequency-dependent uncertainty, in the form of additive unmodeled dynamics, introduced with an `ultidyn` object and a shaping filter.

`robuststab` is used to compute the stability margins of the closed-loop system with respect to the plant model uncertainty.

```
P = tf(4,[1 .8 4]);
delta = ultidyn('delta',[1 1],'SampleStateDimension',5);
Pu = P + 0.25*tf([1],[.15 1])*delta;
C = tf([1 1],[.1 1]) + tf(2,[1 0]);
```

```
S = feedback(1,Pu*C);
[stabmarg,destabunc,report,info] = robuststab(S);
```

You can view the `stabmarg` variable.

```
stabmarg
stabmarg =
                UpperBound: 0.8181
                LowerBound: 0.8181
    DestabilizingFrequency: 9.1321
```

As the margin is less than 1, the closed-loop system is not stable for plant models covered by the uncertain model `Pu`. There is a specific plant within the uncertain behavior modeled by `Pu` (actually about 82% of the modeled uncertainty) that leads to closed-loop instability, with the poles migrating across the stability boundary at 9.1 rads/s.

The `report` variable is specific, giving a plain-language version of the conclusion.

```
report
report =
Uncertain System is NOT robustly stable to modeled uncertainty.
 -- It can tolerate up to 81.8% of modeled uncertainty.
 -- A destabilizing combination of 81.8% the modeled uncertainty
exists, causing an instability at 9.13 rad/s.
 -- Sensitivity with respect to uncertain element ...
   'delta' is 100%.  Increasing 'delta' by 25% leads to a
25% decrease in the margin.
```

Because the problem has only one uncertain element, the stability margin is completely determined by this element, and hence the margin exhibits 100% sensitivity to this uncertain element.

You can verify that the destabilizing value of `delta` is indeed about 0.82 normalized units from its nominal value.

```
actual2normalized(S.Uncertainty.delta,destabunc.delta)
ans =
    0.8181
```

Use `usubs` to substitute the specific value into the closed-loop system. Verify that there is a closed-loop pole near `j9.1`, and plot the unit-step response of the nominal closed-loop system, as well as the unstable closed-loop system.

```
Sbad = usubs(S,destabunc);
pole(Sbad)
```

```
ans =
  1.0e+002 *
  -3.2318
  -0.2539
  -0.0000 + 0.0913i
  -0.0000 - 0.0913i
  -0.0203 + 0.0211i
  -0.0203 - 0.0211i
  -0.0106 + 0.0116i
  -0.0106 - 0.0116i
step(S.NominalValue,'r--',Sbad,'g',4);
```

Finally, as an ad-hoc test, set the gain bound on the uncertain `delta` to 0.81 (slightly less than the stability margin). Sample the closed-loop system at 100 values, and compute the poles of all these systems.

```
S.Uncertainty.delta.Bound = 0.81;
S100 = usample(S,100);
p100 = pole(S100);
max(real(p100(:)))
ans =
 -6.4647e-007
```

As expected, all poles have negative real parts.

Basic Syntax with Fourth Output Argument

A fourth output argument yields more specialized information, including sensitivities and frequency-by-frequency information.

```
[StabMarg,Destabunc,Report,Info] = robuststab(sys)
```

In addition to the first 3 output arguments, described previously, `Info` is a structure with the following fields

Field	Description
Sensitivity	A struct with *M* fields, Field names are names of uncertain elements of sys. Values of fields are positive, each the local sensitivity of the overall stability margin to that element's uncertainty range. For instance, a value of 25 indicates that if the uncertainty range is enlarged by 8%, then the stability margin should drop by about 2% (25% of 8). If the Sensitivity property of the robuststabOptions object is 'off', the values are set to NaN.
Frequency	*N*-by-1 frequency vector associated with analysis.
BadUncertainValues	*N*-by-1 struct array containing the destabilizing uncertain element values at each frequency.
MussvBnds	A 1-by-2 frd, with upper and lower bounds from mussv. The (1,1) entry is the µ-upper bound (corresponds to stabmarg.LowerBound) and the (1,2) entry is the µ-lower bound (for stabmarg.UpperBound).
MussvInfo	Structure of compressed data from mussv.

Specifying Additional Options

Use robuststabOptions to specify additional options for the robuststab computation. For example, you can control what is displayed during the computation, turning the sensitivity computation on or off, set the step-size in the sensitivity computation, or control the option argument used in the underlying call to mussv. For instance, you can turn the display on, and the sensitivity calculation off by executing

```
opt = robuststabOptions('Sensitivity','off','Display','on');
[StabMarg,Destabunc,Report,Info] = robuststab(sys,opt)
```

See the robuststabOptions reference page for more information about available options.

Limitations

Under most conditions, the robust stability margin at each frequency is a continuous function of the problem data at that frequency. Because the problem data, in turn, is a continuous function of frequency, it follows that finite frequency grids are usually adequate in correctly assessing robust stability bounds, assuming the frequency grid is dense enough.

Nevertheless, there are simple examples that violate this. In some problems, the migration of poles from stable to unstable *only* occurs at a finite collection of specific frequencies (generally unknown to you). Any frequency grid that excludes these critical frequencies (and almost every grid will exclude them) will result in undetected migration and misleading results, namely stability margins of ∞.

See “Getting Reliable Estimates of Robustness Margins” for more information about circumventing the problem in an engineering-relevant fashion.

Algorithms

A rigorous robust stability analysis consists of two steps:

1. Verify that the nominal system is stable;
2. Verify that no poles cross the stability boundary as the uncertain elements vary within their ranges.

Because the stability boundary is also associated with the frequency response, the second step can be interpreted (and carried out) as a frequency domain calculation. This amounts to a classical μ-analysis problem.

The algorithm in `robuststab` follows this in spirit, with the following limitations.

- If `sys` is a `uss` object, then the first requirement of stability of nominal value is explicitly checked within `robuststab`. However, if `sys` is an `ufrd`, then the verification of nominal stability from the nominal frequency response data is not performed, and is instead assumed.
- In the second step (monitoring the stability boundary for the migration of poles), rather than check all points on stability boundary, the algorithm only detects migration of poles across the stability boundary at the frequencies in `info.Frequency`.

See “Limitations” on page 1-503 for information about issues related to migration detection.

The exact stability margin is guaranteed to be no larger than `UpperBound` (some uncertain elements associated with this magnitude cause instability – one instance is returned in the structure `destabunc`). The instability created by `destabunc` occurs at the frequency value in `DestabilizingFrequency`.

Similarly, the exact stability margin is guaranteed to be no smaller than `LowerBound`. In other words, for all modeled uncertainty with magnitude up to `LowerBound`, the system is guaranteed stable. These bounds are derived using the upper bound for the structured singular value, which is essentially optimally-scaled, small-gain theorem analysis.

See Also

`diskmargin` | `mussv` | `robgain` | `robstab` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

robuststabOptions

(Not recommended) Option set for `robuststab`

Syntax

```
options = robuststabOptions
options = robuststabOptions(Name,Value,...)
```

Note `robuststabOptions` is not recommended. Use `robOptions` instead.

Description

`options = robuststabOptions` returns the default option set for the `robuststab` command.

`options = robuststabOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`robuststabOptions` takes the following `Name` arguments:

`Display`

Specifies whether `robuststab` displays progress of `mussv` computations.

- `'off'` — Do not display progress.
- `'on'` — Display progress. This setting overrides the silent (`'s'`) option in the `Mussv` option.

Default: `'off'`

`Sensitivity`

Specifies whether `robuststab` computes the sensitivity of the stability margin with respect to each individual uncertain element. This element-by-element sensitivity provides an indication of which elements the stability margin is most sensitive to. Turning off the element-by-element sensitivity calculation speeds up `robuststab`.

- `'on'` — Compute the sensitivity for each uncertain element.
- `'off'` — Do not compute the sensitivity for each uncertain element.

Default: `'on'`

`VaryUncertainty`

Percentage variation of uncertainty for computing sensitivity. The sensitivity estimate uses a finite difference calculation.

Default: 25

`Mussv`

Options for the `mussv` calculation that `robustperf` performs. See `mussv` for the available options.

Default: `''` (default behavior of `mussv`)

Output Arguments

`options`

Option set containing the specified options for the `robuststab` command.

Examples

Create an options set for a `robuststab` calculation that displays the progress of the `mussv` calculation. Also, turn off the element-by-element sensitivity calculation.

```
 options = robuststabOptions('Display','on','Sensitivity','off');
```

Alternatively, use dot notation to set the values of `options`.

```
options = robuststabOptions;
options.Display = 'on';
options.Sensitivity = 'off';
```

See Also

`robOptions`

Introduced in R2011b

schurmr

Balanced model truncation via Schur method

Syntax

```
GRED = schurmr(G)
```

```
GRED = schurmr(G,order)
```

```
[GRED,redinfo] = schurmr(G,key1,value1,...)
```

```
[GRED,redinfo] = schurmr(G,order,key1,value1,...)
```

Description

`schurmr` returns a reduced order model `GRED` of G and a struct array redinfo containing the error bound of the reduced model and Hankel singular values of the original system.

The error bound is computed based on Hankel singular values of `G`. For a stable system Hankel singular values indicate the respective state energy of the system. Hence, reduced order can be directly determined by examining the system Hankel SV's, $\sigma\iota$.

With only one input argument `G`, the function will show a Hankel singular value plot of the original model and prompt for model order number to reduce.

This method guarantees an error bound on the infinity norm of the *additive error* ‖ `G-GRED` ‖∞ for well-conditioned model reduced problems [1]:

$$\|G - Gred\|_\infty \le 2 \sum_{k+1}^{n} \sigma_i$$

This table describes input arguments for `schurmr`.

Argument	Description
`G`	LTI model to be reduced (without any other inputs will plot its Hankel singular values and prompt for reduced order).

Argument	Description
`ORDER`	(Optional) an integer for the desired order of the reduced model, or optionally a vector packed with desired orders for batch runs

A batch run of a serial of different reduced order models can be generated by specifying `order = x:y`, or a vector of integers. By default, all the anti-stable part of a system is kept, because from control stability point of view, getting rid of unstable state(s) is dangerous to model a system.

`'MaxError'` can be specified in the same fashion as an alternative for `' ORDER '`. In this case, reduced order will be determined when the sum of the tails of the Hankel sv's reaches the `'MaxError'`.

Argument	Value	Description
`'MaxError'`	A real number or a vector of different errors	Reduce to achieve H_∞ error. When present, `'MaxError'` overrides `ORDER` input.
`'Weights'`	`{Wout,Win}` cell array	Optimal 1x2 cell array of LTI weights `Wout` (output) and `Win` (input); default is both identity; Weights must be invertible.
`'Display'`	`'on'` or `'off'`	Display Hankel singular plots (default `'off'`).
`'Order'`	Integer, vector or cell array	Order of reduced model. Use only if not specified as 2nd argument.

Weights on the original model input and/or output can make the model reduction algorithm focus on some frequency range of interests. But weights have to be stable, minimum phase and invertible.

This table describes output arguments.

Argument	Description
`GRED`	LTI reduced order model. Becomes multi-dimensional array when input is a serial of different model order array.

Argument	Description
`REDINFO`	A STRUCT array with 3 fields: • `REDINFO.ErrorBound` • `REDINFO.StabSV` • `REDINFO.UnstabSV`

`G` can be stable or unstable. `G` and `GRED` can be either continuous or discrete.

Examples

Given a continuous or discrete, stable or unstable system, `G`, the following commands can get a set of reduced order models based on your selections:

```
rng(1234,'twister');
G = rss(30,5,4);
[g1, redinfo1] = schurmr(G); % display Hankel SV plot
                             % and prompt for order (try 15:20)
[g2, redinfo2] = schurmr(G,20);
[g3, redinfo3] = schurmr(G,[10:2:18]);
[g4, redinfo4] = schurmr(G,'MaxError',[0.01, 0.05]);
for i = 1:4
     figure(i); eval(['sigma(G,g' num2str(i) ');']);
end
```

Algorithms

Given a state space (A,B,C,D) of a system and k, the desired reduced order, the following steps will produce a similarity transformation to truncate the original state-space system to the k^{th} order reduced model [16].

1 Find the controllability and observability grammians P and Q.

2 Find the Schur decomposition for PQ in both ascending and descending order, respectively,

$$V_A^T PQV_A = \begin{bmatrix} \lambda_1 & \dots & \dots \\ 0 & \dots & \dots \\ 0 & 0 & \lambda_n \end{bmatrix}$$

$$V_D^T PQV_D = \begin{bmatrix} \lambda n & \dots & \dots \\ 0 & \dots & \dots \\ 0 & 0 & \lambda_1 \end{bmatrix}$$

3 Find the left/right orthonormal eigen-bases of PQ associated with the k^{th} big Hankel singular values.

$$V_A = [V_{R,SMALL}, V_{L,BIG}]$$

4 Find the SVD of $(V^T_{L,BIG}\ V_{R,BIG}) = U\ \Sigma\ V^T$

$$V_D = [V_{R,BIG}, V_{L,SMALL}]$$

5 Form the left/right transformation for the final k^{th} order reduced model

$$S_{L,BIG} = V_{L,BIG}\ U\Sigma(1{:}k,1{:}k)^{-½}$$

$$S_{\mathrm{R,BIG}} = V_{R,BIG}V\Sigma(1{:}k,1{:}k)^{-½}$$

6 Finally,

$$\begin{bmatrix} \widehat{A} & \widehat{B} \\ \widehat{C} & \widehat{D} \end{bmatrix} = \begin{bmatrix} S^T_{L,BIG}AS_{R,BIG} & S^T_{L,BIG}B \\ CS_{R,BIG} & D \end{bmatrix}$$

The proof of the Schur balance truncation algorithm can be found in [2].

References

[1] K. Glover, "All Optimal Hankel Norm Approximation of Linear Multivariable Systems, and Their L_∞- error Bounds," Int. J. Control, vol. 39, no. 6, pp. 1145-1193, 1984.

[2] M. G. Safonov and R. Y. Chiang, "A Schur Method for Balanced Model Reduction," *IEEE Trans. on Automat. Contr.*, vol. 34, no. 7, July 1989, pp. 729-733.

See Also

`balancmr` | `bstmr` | `hankelmr` | `hankelsv` | `ncfmr` | `reduce`

Introduced before R2006a

sdhinfnorm

Compute L_2 norm of continuous-time system in feedback with discrete-time system

Syntax

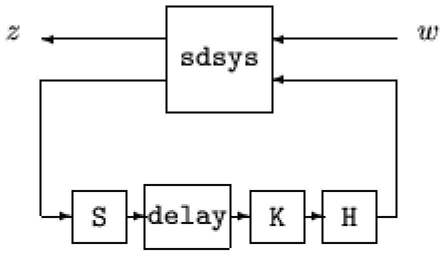
```
[gaml,gamu] = sdhinfnorm(sdsys,k)
```

```
[gaml,gamu] = sdhinfnorm(sdsys,k,delay)
```

```
[gaml,gamu] = sdhinfnorm(sdsys,k,delay,tol)
```

Description

`[gaml,gamu] = sdhinfnorm(sdsys,k)` computes the L_2 induced norm of a continuous-time LTI plant, `sdsys`, in feedback with a discrete-time controller, `k`, connected through an ideal sampler and a zero-order hold (see figure below). `sdsys` must be strictly proper, such that the constant feedback gain must be zero. The outputs, `gamu` and `gaml`, are upper and lower bounds on the induced L_2 norm of the sampled-data closed-loop system.



`[gaml,gamu] = sdhinfnorm(sdsys,k,h,delay)` includes the input argument `delay`. `delay` is a nonnegative integer associated with the number of computational delays of the controller. The default value of the delay is 0.

`[gaml,gamu] = sdhinfnorm(sdsys,k,h,delay,tol)` includes the input argument, `tol`, which defines the difference between upper and lower bounds when search terminates. The default value of `tol` is 0.001.

Examples

Consider an open-loop, continuous-time transfer function `p = 30/s(s+30)` and a continuous-time controller `k = 4/(s+4)`. The closed-loop continuous-time system has a peak magnitude across frequency of 1.

```
p = ss(tf(30,[1 30])*tf([1],[1 0]));
k = ss(tf(4,[1 4]));
cl = feedback(p,k);
norm(cl,'inf')
ans =
     1
```

Initially the controller is to be implemented at a sample rate of 1.5 Hz. The sample-data norm of the closed-loop system with the discrete-time controller is 1.0.

```
kd = c2d(k,0.75,'zoh');
[gu,gl] = sdhinfnorm([1; 1]*p*[1 1],-kd);
[gu gl]
ans =
    3.7908    3.7929
```

Because of the large difference in norm between the continuous-time and sampled-data closed-loop system, the sample rate of the controller is increased from 1.5 Hz to 5 Hz. The sample-data norm of the new closed-loop system is 3.79.

```
kd = c2d(k,0.2,'zoh');
[gu,gl] = sdhinfnorm([1; 1]*p*[1 1],-kd);
[gu gl]
ans =
    1.0044    1.0049
```

Algorithms

`sdhinfnorm` uses variations of the formulas described in the Bamieh and Pearson paper to obtain an equivalent discrete-time system. (These variations are done to improve the numerical conditioning of the algorithms.) A preliminary step is to determine whether the norm of the continuous-time system over one sampling period without control is less than the given value. This requires a search and is, computationally, a relatively expensive step.

References

Bamieh, B.A., and J.B. Pearson, “A General Framework for Linear Periodic Systems with Applications to Sampled-Data Control,” *IEEE Transactions on Automatic Control,* Vol. AC–37, 1992, pp. 418-435.

See Also

`gapmetric` | `hinfsyn` | `norm` | `sdhinfsyn` | `sdlsim`

Introduced before R2006a

sdhinfsyn

Compute H_∞ controller for sampled-data system

Syntax

```
[K,GAM]=sdhinfsyn(P,NMEAS,NCON)
```

```
[K,GAM]=sdhinfsyn(P,NMEAS,NCON, KEY1,VALUE1,KEY2,VALUE2,...)
```

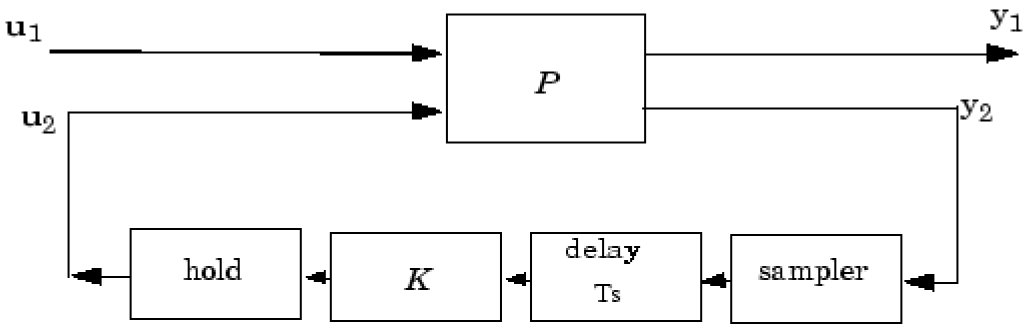
Description

`sdhinfsyn` controls a continuous-time LTI system P with a discrete-time controller `K`. The continuous-time LTI plant `P` has a state-space realization partitioned as follows:

$$P = \begin{bmatrix} A & B_1 & B_2 \\ C_1 & 0 & 0 \\ C_2 & 0 & 0 \end{bmatrix}$$

where the continuous-time disturbance inputs enter through B_1, the outputs from the controller are held constant between sampling instants and enter through B_2, the continuous-time errors (to be kept small) correspond to the C_1 partition, and the output measurements that are sampled by the controller correspond to the C_2 partition. B_2 has column size `ncon` and C_2 has row size `nmeas`. Note that the D matrix must be zero.

`sdhinfsyn` synthesizes a discrete-time LTI controller `K` to achieve a given norm (if possible) or find the minimum possible norm to within tolerance `TOLGAM`.



Similar to `hinfsyn`, the function `sdhinfsyn` employs a γ iteration. Given a high and low value of γ, `GMAX` and `GMIN`, the bisection method is used to iterate on the value of γ in an effort to approach the optimal H_∞ control design. If `GMAX = GMIN`, only one γ value is tested. The stopping criterion for the bisection algorithm requires that the relative difference between the last γ value that failed and the last γ value that passed be less than `TOLGAM`.

Input arguments

`P`	LTI plant
`NMEAS`	Number of measurements output to controller
`NCON`	Number of control inputs

Optional input arguments (`KEY, VALUE`) pairs are similar to `hinfsyn`, but with additional `KEY` values `'Ts'` and `'DELAY'`.

KEY	**VALUE**	**Meaning**
`'GMAX'`	real	Initial upper bound on GAM (default=`Inf`)
`'GMIN'`	real	Initial lower bound on GAM (default=0)
`'TOLGAM'`	real	Relative error tolerance for GAM (default=.01)
`'Ts'`	real	(Default=1) sample time of the controller to be designed
`'DELAY'`	integer	(Default=0) a nonnegative integer giving the number of sample periods delay for the control computation

KEY	VALUE	Meaning

`'DISPLAY'`	`'off'`
	
	`'on'`
		(Default) no command window display, or the command window displays synthesis progress information

Output arguments

`K`	H_∞ controller
`GAM`	Final γ value of H_∞ cost achieved

Algorithms

`sdhinfsyn` uses a variation of the formulas described in the Bamieh and Pearson paper [1] to obtain an equivalent discrete-time system. (This is done to improve the numerical conditioning of the algorithms.) A preliminary step is to determine whether the norm of the continuous-time system over one sampling period without control is less than the given γ-value. This requires a search and is computationally a relatively expensive step.

References

[1] Bamieh, B.A., and J.B. Pearson, “A General Framework for Linear Periodic Systems with Applications to Sampled-Data Control,” *IEEE Transactions on Automatic Control,* Vol. AC–37, 1992, pp. 418-435.

See Also

`hinfsyn` | `norm` | `sdhinfnorm`

Introduced before R2006a

sdlsim

Time response of sampled-data feedback system

Syntax

```
sdlsim(p,k,w,t,tf)
```

```
sdlsim(p,k,w,t,tf,x0,z0)
```

```
sdlsim(p,k,w,t,tf,x0,z0,int)
```

```
[vt,yt,ut,t] = sdlsim(p,k,w,t,tf)
```

```
[vt,yt,ut,t] = sdlsim(p,k,w,t,tf,x0,z0,int)
```

Description

`sdlsim(p,k,w,t,tf)` plots the time response of the hybrid feedback system. `lft(p,k)`, is forced by the continuous input signal described by `w` and `t` (values and times, as in `lsim`). `p` must be a continuous-time LTI system, and `k` must be discrete-time LTI system with a specified sample time (the unspecified sample time –1 is not allowed). The final time is specified with `tf`.

`sdlsim(p,k,w,t,tf,x0,z0)` specifies the initial state vector `x0` of `p`, and `z0` of `k`, at time `t(1)`.

`sdlsim(p,k,w,t,tf,x0,z0,int)` specifies the continuous-time integration step size `int`. `sdlsim` forces `int = (k.Ts)/N int` where N>4 is an integer. If any of these optional arguments is omitted, or passed as empty matrices, then default values are used. The default value for `x0` and `z0` is zero. Nonzero initial conditions are allowed for `p` (and/or `k`) only if `p` (and/or `k`) is an `ss` object.

If `p` and/or `k` is an LTI array with consistent array dimensions, then the time simulation is performed pointwise across the array dimensions.

`[vt,yt,ut,t] = sdlsim(p,k,w,t,tf)` computes the continuous-time response of the hybrid feedback system `lft(p,k)` forced by the continuous input signal defined by `w` and `t` (values and times, as in `lsim`). `p` must be a continuous-time system, and `k` must be discrete-time, with a specified sample time (the unspecified sample time –1 is not allowed). The final time is specified with `tf`. The outputs `vt, yt` and `ut` are 2-by-1 cell arrays: in each the first entry is a time vector, and the second entry is the signal values. Stored in this manner, the signal `vt` is plotted by using one of the following commands:

```
plot(vt{1},vt{2})
plot(vt{:})
```

Signals `yt` and `ut` are respectively the input to `k` and output of `k`.

If `p` and/or `k` are LTI arrays with consistent array dimensions, then the time simulation is performed pointwise across the array dimensions. The outputs are 2-by-1-by-array dimension cell arrays. All responses can be plotted simultaneously, for example, `plot(vt)`.

`[vt,yt,ut,t] = sdlsim(p,k,w,t,tf,x0,z0,int)` The optional arguments are `int` (integration step size), `x0` (initial condition for `p`), and `z0` (initial condition for `k`). `sdlsim` forces `int = (k.Ts)/N`, where N>4 is an integer. If any of these arguments is omitted, or passed as empty matrices, then default values are used. The default value for `x0` and `z0` is zero. Nonzero initial conditions are allowed for `p` (and/or `k`) only if `p` (and/or `k`) is an `ss` object.

Examples

Time Response of Continuous Plant with Discrete Controller

To illustrate the use of `sdlsim`, consider the application of a discrete controller to a plant with an integrator and near integrator. A continuous plant and a discrete controller are created. A sample-and-hold equivalent of the plant is formed and the discrete closed-loop system is calculated. Simulating this gives the system response at the sample points. `sdlsim` is then used to calculate the intersample behavior.

```
P = tf(1,[1, 1e-5,0]);
T = 1.0/20;
C = ss([-1.5 T/4; -2/T -.5],[ .5 2;1/T 1/T],...
   [-1/T^2  -1.5/T], [1/T^2  0],T);
Pd = c2d(P,T,'zoh');
```

The closed-loop digital system is now set up. You can use `sysic` to construct the interconnected feedback system.

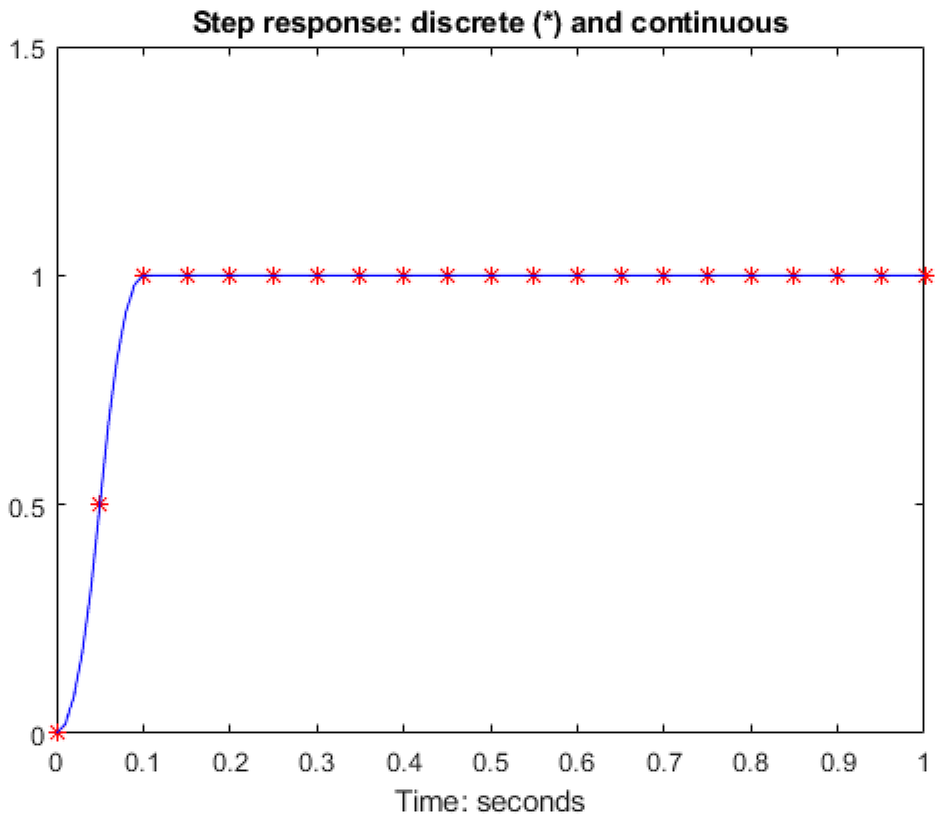
```
systemnames = 'Pd C';
inputvar = '[ref]';
outputvar = '[Pd]';
input_to_Pd = '[C]';
input_to_C = '[ref ; Pd]';
sysoutname = 'dclp';
cleanupsysic = 'yes';
sysic;
```

Use `step` to simulate the digital step response.

```
[yd,td] = step(dclp,20*T);
```

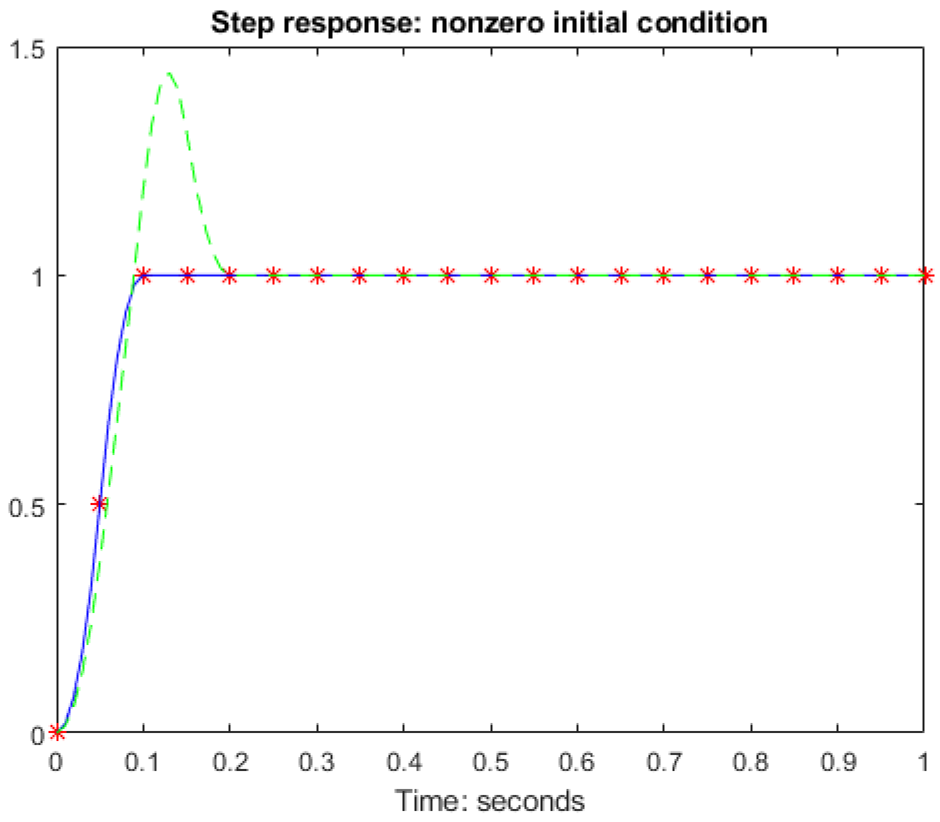
Set up the continuous interconnection and calculate the sampled data response with `sdlsim`.

```
M = [0,1;1,0;0,1]*blkdiag(1,P);
t = [0:.01:1]';
u = ones(size(t));
y1 = sdlsim(M,C,u,t);
plot(td,yd,'r*',y1{:},'b-')
axis([0,1,0,1.5])
xlabel('Time: seconds')
title('Step response: discrete (*) and continuous')
```



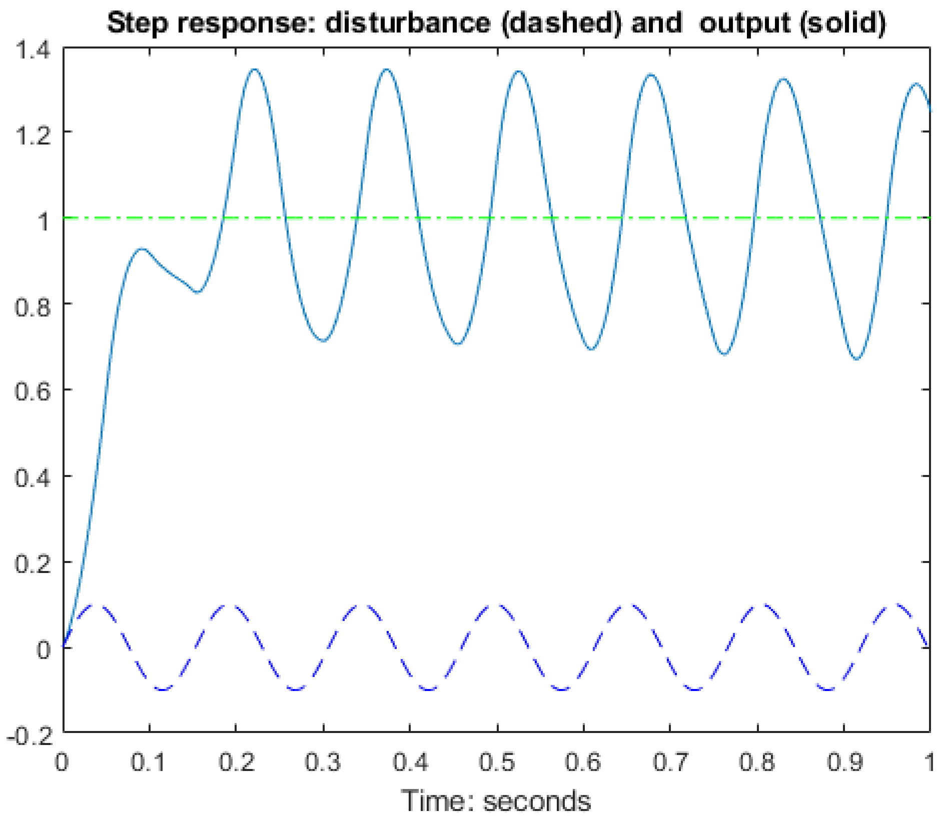
You can see the effect of a nonzero initial condition in the continuous-time system. Note how examining the system at only the sample points will underestimate the amplitude of the overshoot.

```
y2 = sdlsim(M,C,u,t,1,0,[0.25;0]);
plot(td,yd,'r*',y1{:},'b-',y2{:},'g--')
axis([0,1,0,1.5])
xlabel('Time: seconds')
title('Step response: nonzero initial condition')
```



Finally, you can examine the effect of a sinusoidal disturbance at the continuous-time plant output. This controller is not designed to reject such a disturbance and the system does not contain antialiasing filters. Simulating the effect of antialiasing filters is easily accomplished by including them in the continuous interconnection structure.

```
M2 = [0,1,1;1,0,0;0,1,1]*blkdiag(1,1,P);
t = [0:.001:1]';
dist = 0.1*sin(41*t);
u = ones(size(t));
[y3,meas,act] = sdlsim(M2,C,[u dist],t,1);
plot(y3{:},'-',t,dist,'b--',t,u,'g-.')
xlabel('Time: seconds')
title('Step response: disturbance (dashed) and  output (solid)')
```



Algorithms

`sdlsim` oversamples the continuous-time, *N* times the sample rate of the controller *k*.

See Also

`gapmetric` | `hinfsyn` | `norm` | `sdhinfnorm` | `sdhinfsyn` | `sysic`

Introduced before R2006a

sectf

State-space sector bilinear transformation

Syntax

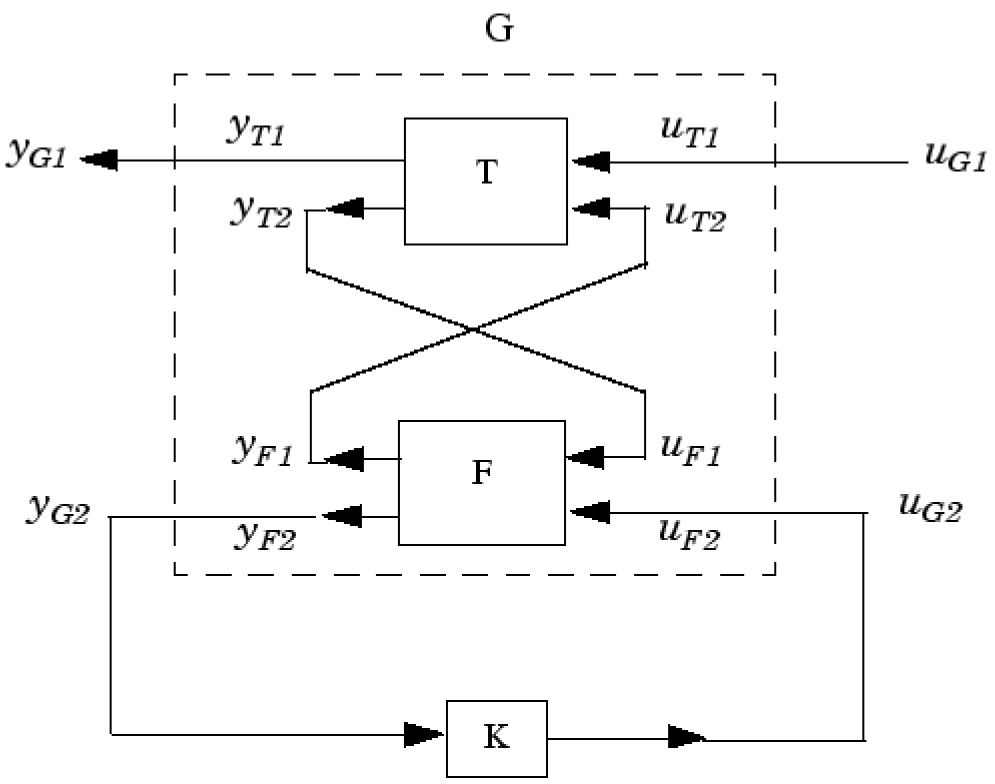
```
[G,T] = sectf(F,SECF,SECG)
```

Description

`[G,T] = sectf(F,SECF,SECG)` computes a linear fractional transform `T` such that the system `lft(F,K)` is in sector `SECF` if and only if the system `lft(G,K)` is in sector `SECG` where

```
G=lft(T,F,NU,NY)
```

where `NU` and `NY` are the dimensions of u_{T2} and y_{T2}, respectively—see the following figure.



Sector transform `G=lft(T,F,NU,NY)`.

`sectf` are used to transform general conic-sector control system performance specifications into equivalent H_∞-norm performance specifications.

Input Arguments		
`F`	LTI state-space plant	
`SECG`, `SECF`:	Conic Sector:	
	`[-1,1]` or `[-1;1]`	$\|y\|^2 \le \|u\|^2$
	`[0,Inf]` or `[0;Inf]`	$0 \le \mathrm{Re}[y * u]$

Input Arguments		
	`[A,B]` or `[A;B]`	$0 \geq \mathrm{Re}[(y - Au) * (y - Bu)]$
	`[a,b]` or `[a;b]`	$0 \geq \mathrm{Re}[(y - diag(a)u) * (y - diag(b)u)]$
	`S`	$0 \geq \mathrm{Re}[(S_{11}u + S_{12}y) * (S_{21}u + S_{22}y)]$
	`S`	$0 \geq \mathrm{Re}[(S_{11}u + S_{12}y) * (S_{21}u + S_{22}y)]$

where `A`,`B` are scalars in [–∞, ∞] or square matrices; `a,b` are vectors; `S=[S11 S12;S21,S22]` is a square matrix whose blocks `S11,S12,S21,S22` are either scalars or square matrices; `S` is a two-port system `S=mksys(a`,`b1`,`b2`,`...`,`'tss')` with transfer function

$$S(s) = \begin{bmatrix} S_{11}(s) & S_{12}(s) \\ S_{21}(s) & S_{22}(s) \end{bmatrix}$$

Output Arguments	Description
`G`	Transformed plant $G(s)$=`lftf(T,F)`
`T`	LFT sector transform, maps conic sector `SECF` into conic sector `SECG`

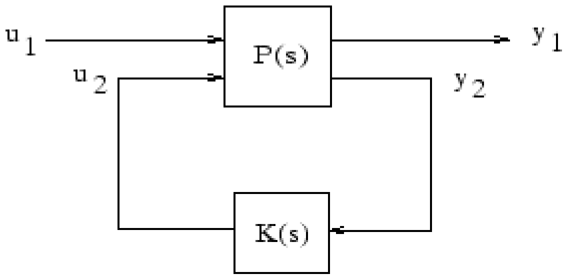
Output Variables	
`G`	The transformed plant $G(s)$ = `lftf(T,F)`:
`T`	The linear fractional transformation $T(s)$ = `T`

Examples

The statement $G(j\omega)$ inside sector[–1, 1] is equivalent to the H_∞ inequality

$$\sup_{\omega} \bar{\sigma}(G(j\omega)) = \|G\|_\infty \leq 1$$

Given a two-port open-loop plant $P(s)$:= `P`, the command `P1 = sectf(P,[0,Inf], [-1,1])`computes a transformed $P_1(s)$:= `P1` such that if `lft(G,K)` is inside *sector*[–1, 1] if and only if `lft(F,K)` is inside *sector*[0, ∞]. In other words, `norm(lft(G,K), inf)`<1 if and only if `lft(F,K)` is strictly positive real. See “Example of Sector Transform” on page 1-529.



Sector Transform Block Diagram

Here is a simple example of the sector transform.

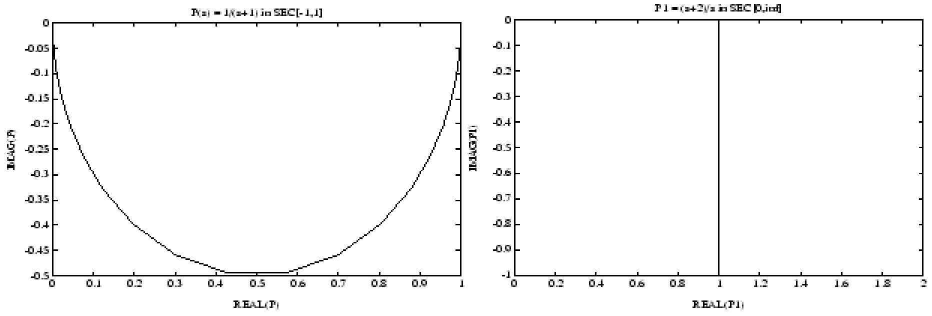
$$P(s) = \frac{1}{s+1} \in \text{sector}[-1,1] \rightarrow P_1(s) = \frac{s+2}{2} \in \text{sector}[0,\infty]\,.$$

You can compute this by simply executing the following commands:

```
P = ss(tf(1,[1 1]));
P1 = sectf(P,[-1,1],[0,Inf]);
```

The Nyquist plots for this transformation are depicted in “Example of Sector Transform” on page 1-529. The condition $P_1(s)$ inside $[0, \infty]$ implies that $P_1(s)$ is stable and $P_1(j\omega)$ is *positive real*, i.e.,

$$P_1^*(j\omega) + P_1(j\omega) \geq 0 \quad \forall\omega$$



Example of Sector Transform

Limitations

A well-posed conic sector must have det(B–A)≠ 0 or

$$\det\left(\begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix}\right) \neq 0.$$

Also, you must have dim(u_{F1}) = dim(y_{F1}) since sectors are only defined for square systems.

Algorithms

`sectf` uses the generalization of the sector concept of [3] described by [1]. First the sector input data `Sf= SECF` and `Sg=SECG` is converted to two-port state-space form; non-dynamical sectors are handled with empty *a*, *b1*, *b2*, *c1*, *c2* matrices. Next the equation

$$S_{g(}s)\begin{bmatrix} u_{g_1} \\ y_{g_1} \end{bmatrix} = S_f(s)\begin{bmatrix} u_{f_1} \\ y_{f_1} \end{bmatrix}$$

is solved for the two-port transfer function $T(s)$ from $u_{g_1}y_{f_1}$ to $u_{f_1}y_{g_1}$. Finally, the function `lftf` is used to compute $G(s)$ as `G = lftf(T,F)`.

References

[1] Safonov, M.G., *Stability and Robustness of Multivariable Feedback Systems.* Cambridge, MA: MIT Press, 1980.

[2] Safonov, M.G., E.A. Jonckheere, M. Verma and D.J.N. Limebeer, "Synthesis of Positive Real Multivariable Feedback Systems," *Int. J. Control*, vol. 45, no. 3, pp. 817-842, 1987.

[3] Zames, G., "On the Input-Output Stability of Time-Varying Nonlinear Feedback Systems ≥— Part I: Conditions Using Concepts of Loop Gain, Conicity, and Positivity," *IEEE Trans. on Automat. Contr.*, AC-11, pp. 228-238, 1966.

See Also

`hinfsyn` | `lft`

Introduced before R2006a

setlmis

Initialize description of LMI system

Syntax

```
setlmis(lmi0)
```

Description

Before starting the description of a new LMI system with `lmivar` and `lmiterm`, type

```
setlmis([])
```

to initialize its internal representation.

To add on to an existing LMI system, use the syntax

```
setlmis(lmi0)
```

where `lmi0` is the internal representation of this LMI system. Subsequent `lmivar` and `lmiterm` commands will then add new variables and terms to the initial LMI system `lmi0`.

See Also

`getlmis` | `lmiterm` | `lmivar` | `newlmi`

Introduced before R2006a

setmvar

Instantiate matrix variable and evaluate all LMI terms involving this matrix variable

Syntax

```
mnewsys = setmvar(lmisys,X,Xval)
```

Description

`setmvar` sets the matrix variable X with identifier `X` to the value `Xval`. All terms involving X are evaluated, the constant terms are updated accordingly, and X is removed from the list of matrix variables. A description of the resulting LMI system is returned in `newsys`.

The integer `X` is the identifier returned by `lmivar` when X is declared. Instantiating X with `setmvar` does not alter the identifiers of the remaining matrix variables.

The function `setmvar` is useful to freeze certain matrix variables and optimize with respect to the remaining ones. It saves time by avoiding partial or complete redefinition of the set of LMI constraints.

Examples

Consider the system

$$x^{\cdot} = Ax + Bu$$

and the problem of finding a stabilizing state-feedback law $u = Kx$ where K is an unknown matrix.

By the Lyapunov Theorem, this is equivalent to finding $P > 0$ and K such that

$$(A + BK)P + P(A + BK^T) + I < 0.$$

With the change of variable $Y := KP$, this condition reduces to the LMI

$$AP + PA^T + BY + Y^TB^T + I < 0.$$

This LMI is entered by the commands

```
n = size(A,1)                  % number of states
ncon = size(B,2)               % number of inputs

setlmis([])
P = lmivar(1,[n 1])            % P full symmetric
Y = lmivar(2,[ncon n])         % Y rectangular

lmiterm([1 1 1 P],A,1,'s')     % AP+PA'
lmiterm([1 1 1 Y],B,1,'s')     % BY+Y'B'
lmiterm([1 1 1 0],1)           % I
lmis = getlmis
```

To find out whether this problem has a solution K for the particular Lyapunov matrix $P = I$, set P to I by typing

```
news = setmvar(lmis,P,1)
```

The resulting LMI system `news` has only one variable $Y = K$. Its feasibility is assessed by calling `feasp`:

```
[tmin,xfeas] = feasp(news)
Y = dec2mat(news,xfeas,Y)
```

The computed Y is feasible whenever `tmin` < 0.

See Also

`delmvar` | `evallmi`

Introduced before R2006a

showlmi

Return left and right sides of LMI after evaluation of all variable terms

Syntax

```
[lhs,rhs] = showlmi(evalsys,n)
```

Description

For given values of the decision variables, the function `evallmi` evaluates all variable terms in a system of LMIs. The left and right sides of the *n*-th LMI are then constant matrices that can be displayed with `showlmi`. If `evalsys` is the output of `evallmi`, the values `lhs` and `rhs` of these left and right sides are given by

```
[lhs,rhs] = showlmi(evalsys,n)
```

An error is issued if `evalsys` still contains variable terms.

Examples

See the description of `evallmi`.

See Also

`evallmi` | `setmvar`

Introduced before R2006a

simplify

Simplify representation of uncertain object

Syntax

```
B = simplify(A)
```

```
B = simplify(A,'full')
```

```
B = simplify(A,'basic')
```

```
B = simplify(A,'class')
```

Description

`B = simplify(A)` performs model-reduction-like techniques to detect and eliminate redundant copies of uncertain elements. Depending on the result, the class of `B` may be lower than `A`. The `AutoSimplify` property of each uncertain element in `A` governs what reduction methods are used. After reduction, any uncertain element which does not actually affect the result is deleted from the representation.

`B = simplify(A,'full')` overrides all uncertain element's `AutoSimplify` property, and uses `'full'` reduction techniques.

`B = simplify(A,'basic')` overrides all uncertain element's `AutoSimplify` property, and uses `'basic'` reduction techniques.

`B = simplify(A,'class')` does not perform reduction. However, any uncertain elements in `A` with zero occurrences are eliminated, and the class of `B` may be lower than the class of `A`.

Examples

Create a simple `umat` with a single uncertain real parameter. Select specific elements, note that result remains in class `umat`. Simplify those same elements, and note that class changes.

```
p1 = ureal('p1',3,'Range',[2 5]);
L = [2 p1];
L(1)
UMAT: 1 Rows, 1 Columns
L(2)
UMAT: 1 Rows, 1 Columns
  p1: real, nominal = 3, range = [2  5], 1 occurrence
simplify(L(1))
ans =
     2
simplify(L(2))
Uncertain Real Parameter: Name p1, NominalValue 3, Range [2  5]
```

Create four uncertain real parameters, with a default value of `AutoSimplify('basic')`, and define a high order polynomial [1].

```
m = ureal('m',125000,'Range',[100000 150000]);
xcg = ureal('xcg',.23,'Range',[.15 .31]);
zcg = ureal('zcg',.105,'Range',[0 .21]);
va = ureal('va',80,'Range',[70 90]);
cw = simplify(m/(va*va)*va,'full')
UMAT: 1 Rows, 1 Columns
   m: real, nominal = 1.25e+005, range = [100000  150000],
1 occurrence
  va: real, nominal = 80, range = [70  90], 1 occurrence
cw = m/va;
fac2 = .16726*xcg*cw*cw*zcg - .17230*xcg*xcg*cw ...
      -3.9*xcg*cw*zcg - .28*xcg*xcg*cw*cw*zcg ...
      -.07*xcg*xcg*zcg + .29*xcg*xcg*cw*zcg ...
      + 4.9*xcg*cw - 2.7*xcg*cw*cw ...
      +.58*cw*cw - 0.25*xcg*xcg - 1.34*cw ...
      +100.1*xcg -14.1*zcg - 1.91*cw*cw*zcg ...
      +1.12*xcg*zcg + 24.6*cw*zcg ...
      +.45*xcg*xcg*cw*cw - 46.85
UMAT: 1 Rows, 1 Columns
    m: real, nominal = 1.25e+005, range = [100000  150000],
18 occurrences
   va: real, nominal = 80, range = [70  90], 8 occurrences
  xcg: real, nominal = 0.23, range = [0.15  0.31], 18 occurrences
  zcg: real, nominal = 0.105, range = [0  0.21], 1 occurrence
```

The result of the high-order polynomial is an inefficient representation involving 18 copies of `m`, 8 copies of `va`, 18 copies of `xcg` and 1 copy of `zcg`. Simplify the expression, using the `'full'` simplification algorithm

```
fac2s = simplify(fac2,'full')
UMAT: 1 Rows, 1 Columns
    m: real, nominal = 1.25e+005, range = [100000  150000],
4 occurrences
   va: real, nominal = 80, range = [70  90], 4 occurrences
  xcg: real, nominal = 0.23, range = [0.15  0.31], 2 occurrences
  zcg: real, nominal = 0.105, range = [0  0.21], 1 occurrence
```

which results in a much more economical representation.

Alternatively, change the `AutoSimplify` property of each parameter to `'full'` before forming the polynomial.

```
m.AutoSimplify = 'full';
xcg.AutoSimplify = 'full';
zcg.AutoSimplify = 'full';
va.AutoSimplify = 'full';
```

You can form the polynomial, which immediately gives a low order representation.

```
cw = m/va;
fac2f = .16726*xcg*cw*cw*zcg - .17230*xcg*xcg*cw ...
      -3.9*xcg*cw*zcg - .28*xcg*xcg*cw*cw*zcg ...
      -.07*xcg*xcg*zcg + .29*xcg*xcg*cw*zcg ...
      + 4.9*xcg*cw - 2.7*xcg*cw*cw ...
      +.58*cw*cw - 0.25*xcg*xcg - 1.34*cw ...
      +100.1*xcg -14.1*zcg - 1.91*cw*cw*zcg ...
      +1.12*xcg*zcg + 24.6*cw*zcg ...
      +.45*xcg*xcg*cw*cw - 46.85
UMAT: 1 Rows, 1 Columns
    m: real, nominal = 1.25e+005, range = [100000  150000],
4 occurrences
   va: real, nominal = 80, range = [70  90], 4 occurrences
  xcg: real, nominal = 0.23, range = [0.15  0.31], 2 occurrences
  zcg: real, nominal = 0.105, range = [0  0.21], 1 occurrence
```

Create two real parameters, `da` and `dx`, and a 2-by-3 matrix, `ABmat`, involving polynomial expressions in the two real parameters .

```
da = ureal('da',0,'Range',[-1 1]);
dx = ureal('dx',0,'Range',[-1 1]);
a11 = -.32 + da*(.8089 + da*(-.987 + 3.39*da)) + .15*dx;
a12 = .934 + da*(.0474 - .302*da);
a21 = -1.15 + da*(4.39 + da*(21.97 - 561*da*da)) ...
     + dx*(9.65 - da*(55.7 + da*177));
```

```
a22 = -.66 + da*(1.2 - da*2.27) + dx*(2.66 - 5.1*da);
b1 = -0.00071 + da*(0.00175 - da*.00308) + .0011*dx;
b2 = -0.031 + da*(.078 + da*(-.464 + 1.37*da)) + .0072*dx;
ABmat = [a11 a12 b1;a21 a22 b2]
UMAT: 2 Rows, 3 Columns
  da: real, nominal = 0, range = [-1  1], 19 occurrences
  dx: real, nominal = 0, range = [-1  1], 2 occurrences
```

Use `'full'` simplification to reduce the complexity of the description.

```
ABmatsimp = simplify(ABmat,'full')
UMAT: 2 Rows, 3 Columns
  da: real, nominal = 0, range = [-1  1], 7 occurrences
  dx: real, nominal = 0, range = [-1  1], 2 occurrences
```

Alternatively, you can set the parameter's `AutoSimplify` property to `'full'`.

```
da.AutoSimplify = 'full';
dx.AutoSimplify = 'full';
```

Now you can rebuild the matrix

```
a11 = -.32 + da*(.8089 + da*(-.987 + 3.39*da)) + .15*dx;
a12 = .934 + da*(.0474 - .302*da);
a21 = -1.15 + da*(4.39 + da*(21.97 - 561*da*da)) ...
     + dx*(9.65 - da*(55.7 + da*177));
a22 = -.66 + da*(1.2 - da*2.27) + dx*(2.66 - 5.1*da);
b1 = -0.00071 + da*(0.00175 - da*.00308) + .0011*dx;
b2 = -0.031 + da*(.078 + da*(-.464 + 1.37*da)) + .0072*dx;
ABmatFull = [a11 a12 b1;a21 a22 b2]
UMAT: 2 Rows, 3 Columns
  da: real, nominal = 0, range = [-1  1], 7 occurrences
  dx: real, nominal = 0, range = [-1  1], 2 occurrences
```

Limitations

Multidimensional model reduction and realization theory are only partially complete theories. The heuristics used by `simplify` are that - heuristics. The order in which expressions involving uncertain elements are built up, eg., distributing across addition and multiplication, can affect the details of the representation (i.e., the number of occurrences of a `ureal` in an uncertain matrix). It is possible that `simplify`'s naive methods cannot completely resolve these differences, so one may be forced to work with “nonminimal” representations of uncertain systems.

Algorithms

`simplify` uses heuristics along with one-dimensional model reduction algorithms to partially reduce the dimensionality of the representation of an uncertain matrix or system.

References

[1] Varga, A. and G. Looye, “Symbolic and numerical software tools for LFT-based low order uncertainty modeling,” *IEEE International Symposium on Computer Aided Control System Design,* 1999, pp. 5-11.

[2] Belcastro, C.M., K.B. Lim and E.A. Morelli, “Computer aided uncertainty modeling for nonlinear parameter-dependent systems Part II: F-16 example,” *IEEE International Symposium on Computer Aided Control System Design,* 1999, pp. 17-23.

See Also

`ucomplex` | `umat` | `ureal` | `uss` | `uss`

Introduced before R2006a

skewdec

Form skew-symmetric matrix

Syntax

```
X = skewdec(m,n)
```

Description

`X = skewdec(m,n)` forms the m-by-m skew-symmetric matrix

$$\begin{bmatrix} 0 & -(n+1) & -(n+2) & \dots \\ (n+1) & 0 & -(n+3) & \dots \\ (n+2) & (n+3) & 0 & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

This function is useful to define skew-symmetric matrix variables. In this case, set `n` to the number of decision variables already used.

Examples

Skew-Symmetric Matrix

Create a 3-by-3 skew-symmetric matrix for an LMI problem in which n = 2. Display the matrix to verify its form.

```
X = skewdec(3,2)
```

```
X = 3×3

     0    -3    -4
     3     0    -5
     4     5     0
```

See Also

`decinfo` | `lmivar`

Introduced before R2006a

slowfast

Slow and fast modes decomposition

Syntax

```
[G1,G2] = slowfast(G,ns)
```

Description

`slowfast` computes the slow and fast modes decompositions of a system $G(s)$ such that

$$G(s) = [G_1(s)] + [G_2(s)]$$

`G(s)` contains the `N` slowest modes (modes with the smallest absolute value) of `G`.

$[G_1(s)] := \left(\widehat{A}_{11}, \widehat{B}_1, \widehat{C}_1, \widehat{D}_1\right)$ denotes the slow part of $G(s)$. The slow poles have low frequency and magnitude values.

$[G_2(s)] := \left(\widehat{A}_{22}, \widehat{B}_2, \widehat{C}_2, \widehat{D}_2\right)$ denotes the fast part. The fast poles have high frequency and magnitude values.

The variable `ns` denotes the index where the modes will be split.

Use `freqsep` to separate slow and fast modes at a specified cutoff frequency instead of a specified number of modes.

References

M.G. Safonov, E.A. Jonckheere, M. Verma and D.J.N. Limebeer, “Synthesis of Positive Real Multivariable Feedback Systems”, *Int. J. Control*, vol. 45, no. 3, pp. 817-842, 1987.

See Also

`freqsep` | `modreal` | `schur`

Introduced before R2006a

squeeze

Remove singleton dimensions for `umat` objects

Syntax

```
B = squeeze(A)
```

Description

`B = squeeze(A)` returns an array `B` with the same elements as `A` but with all the singleton dimensions removed. A singleton is a dimension such that `size(A,dim)==1`. 2-D arrays are unaffected by squeeze so that row vectors remain rows.

See Also

`permute` | `reshape`

Introduced before R2006a

uss/ssbal

Scale state/uncertainty while preserving uncertain input/output map of uncertain system

Syntax

```
usysout = ssbal(usys)
```

```
usysout = ssbal(usys,wc)
```

```
usysout = ssbal(usys,wc,FSflag)
```

```
usysout = ssbal(usys,wc,FSflag,BLTflag)
```

Description

`usysout = ssbal(usys)` yields a system whose input/output and uncertain properties are the same as `usys`, a `uss` object. The numerical conditioning of `usysout` is usually better than that of `usys`, improving the accuracy of additional computations performed with `usysout`. `usysout` is a `uss` object. The balancing algorithm uses `mussv` to balance the constant uncertain state-space matrices in discrete time. If `usys` is a continuous-time uncertain system, the uncertain state-space is mapped by using a bilinear transformation into discrete time for balancing.

`usysout = ssbal(usys,wc)` defines the critical frequency `wc` for the bilinear prewarp transformation from continuous time to discrete time. The default value of `wc` is 1 when the nominal uncertain system is stable and `1.25*mxeig` when it is unstable. `mxeig` corresponds to the value of the real, most positive pole of `usys`.

`usysout = ssbal(usys,wc,FSflag)` sets the scaling flag `FSflag` to handle repeated uncertain parameters. Setting `FSflag=1` uses full matrix scalings to balance the repeated uncertain parameter blocks. `FSflag=0`, the default, uses a single, positive scalar to balance the repeated uncertain parameter blocks.

`usysout = ssbal(usys,wc,FSflag,BLTflag)` sets the bilinear transformation flag, `BLTflag`. By default, `BLTflag=1` transforms the continuous-time system `usys` to a discrete-time system for balancing. `BLTflag=0` results in balancing the continuous-time state-space data from `usys`. Note that if `usys` is a discrete-time system, no bilinear transformation is performed.

`ssbal` does not work on an array of uncertain systems. An error message is generated to alert you to this.

Examples

Consider a two-input, two-output, two-state uncertain system with two real parameter uncertainties, `p1` and `p2`.

```
p2=ureal('p2',-17,'Range',[-19 -11]);
p1=ureal('p1',3.2,'Percentage',0.43);
A = [-12 p1;.001 p2];
B = [120 -809;503 24];
C = [.034 .0076; .00019 2];
usys = ss(A,B,C,zeros(2,2))
USS: 2 States, 2 Outputs, 2 Inputs, Continuous System
  p1: real, nominal = 3.2, variability = [-0.43  0.43]%, 1 occurrence
  p2: real, nominal = -17, range = [-19  -11], 1 occurrence
usys.NominalValue
a =
         x1     x2
   x1    -12    3.2
   x2  0.001    -17

b =
        u1    u2
   x1   120  -809
   x2   503    24

c =
           x1       x2
   y1    0.034   0.0076
   y2  0.00019        2

d =
      u1  u2
   y1   0   0
   y2   0   0

Continuous-time model.
ssbal is used to balance the uncertain system usys.

usysout = ssbal(usys)
USS: 2 States, 2 Outputs, 2 Inputs, Continuous System
  p1: real, nominal = 3.2, variability = [-0.43  0.43]%,
1 occurrence
  p2: real, nominal = -17, range = [-19  -11], 1 occurrence

usysout.NominalValue
```

```
a = 
              x1        x2
   x1        -12    0.3302
   x2   0.009692       -17

b = 
            u1      u2
   x1   0.7802   -5.26
   x2     31.7   1.512

c = 
             x1       x2
   y1     5.229   0.1206
   y2   0.02922    31.74

d = 
        u1  u2
   y1    0   0
   y2    0   0

Continuous-time model.
```

See Also

c2d | canon | d2c | mussv | mussvextract | ss2ss

Introduced before R2006a

stack

Construct array by stacking uncertain matrices, models, or arrays

Syntax

```
umatout = stack(arraydim,umat1,umat2,...)
```

```
usysout = stack(arraydim,usys1,usys2,...)
```

Description

`stack` constructs an uncertain array by stacking uncertain matrices, models, or arrays along array dimensions of an uncertain array.

`umatout = stack(arraydim,umat1,umat2,...)` produces an array of uncertain matrices, `umatout`, by stacking (concatenating) the `umat` matrices (or `umat` arrays) `umat1`, `umat2`,... along the array dimension `arraydim`. All models must have the same number of columns and rows. The column/row dimensions are not counted in the array dimensions.

`umatout = stack(arraydim,usys1,usys2,...)` produces an array of uncertain models, `ufrd` or `uss`, or `usysout`, by stacking (concatenating) the `ufrd` or `uss` matrices (or `ufrd` or `uss` arrays) `usys1`, `usys2`,... along the array dimension `arraydim`. All models must have the same number of columns and rows (the same input/output dimensions). Note that the input/output dimensions are not considered for arrays.

Examples

Consider `usys1` and `usys2`, two single-input/single-output `uss` models:

```
zeta = ureal('zeta',1,'Range',[0.4 4]);
wn = ureal('wn',0.5,'Range',[0.3 0.7]);
P1 = tf(1,[1 2*zeta*wn wn^2]);
P2 = tf(zeta,[1 10]);
```

You can stack along the first dimension to produce a 2-by-1 `uss` array.

```
stack(1,P1,P1)
USS: 2 States, 1 Output, 1 Input, Continuous System [array, 2 x 1]
    wn: real, nominal = 0.5, range = [0.3  0.7], 3 occurrences
  zeta: real, nominal = 1, range = [0.4  4], 1 occurrence
```

You can stack along the second dimension to produce a 1-by-2 `uss` array.

```
stack(2,P1,P2)   % produces a 1-by-2 USS array.
USS: 2 States, 1 Output, 1 Input, Continuous System [array, 1 x 2]
    wn: real, nominal = 0.5, range = [0.3  0.7], 3 occurrences
  zeta: real, nominal = 1, range = [0.4  4], 1 occurrence
```

You can stack along the third dimension to produce a 1-by-1-by-2 `uss` array.

```
stack(3,P1,P2)   % produces a 1-by-1-by-2 USS array.
USS: 2 States, 1 Output, 1 Input, Continuous System
[array, 1 x 1 x 2]
    wn: real, nominal = 0.5, range = [0.3  0.7], 3 occurrences
  zeta: real, nominal = 1, range = [0.4  4], 1 occurrence
```

See Also

`append` | `blkdiag` | `horzcat` | `vertcat`

Introduced before R2006a

symdec

Form symmetric matrix

Syntax

```
x = symdec(m,n)
```

Description

`symdec(m,n)` forms an m-by-m symmetric matrix of the form

$$\begin{bmatrix} (n+1) & (n+2) & (n+4) & \dots \\ (n+2) & (n+3) & (n+5) & \dots \\ (n+4) & (n+5) & (n+6) & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

This function is useful to define symmetric matrix variables. `n` is the number of decision variables.

Examples

Symmetric Matrix

Create a 4-by-4 symmetric matrix for an LMI problem in which n = 2. Display the matrix to verify its form.

```
X = symdec(4,2)
```

```
X = 4×4

     3     4     6     9
     4     5     7    10
     6     7     8    11
     9    10    11    12
```

See Also
`decinfo` | `skewdec`

Introduced before R2006a

sysic

Build interconnections of certain and uncertain matrices and systems

Syntax

```
sysout = sysic
```

Description

`sysic` requires that 3 variables with fixed names be present in the calling workspace: `systemnames`, `inputvar` and `outputvar`.

`systemnames` is a `char` containing the names of the subsystems (`double, tf, zpk, ss, uss, frd, ufrd`, etc) that make up the interconnection. The names must be separated by spaces with no additional punctuation. Each named variable must exist in the calling workspace.

`inputvar` is a `char`, defining the names of the external inputs to the interconnection. The names are separated by semicolons, and the entire list is enclosed in square brackets `[ ]`. Inputs can be scalar or multivariate. For instance, a 3-component (`x,y,z`) force input can be specified with 3 separate names, `Fx, Fy, Fz`. Alternatively, a single name with a defined integer dimension can be specified, as in `F{3}`. The order of names in `inputvar` determines the order of inputs in the interconnection.

`outputvar` is a `char`, describing the outputs of the interconnection. Outputs do not have names-they are simply linear combinations of individual subsystem's outputs and external inputs. Semicolons delineate separate components of the interconnections outputs. Between semicolons, signals can be added and subtracted, and multiplied by scalars. For multivariable subsystems, arguments within parentheses specify which subsystem outputs are to be used and in what order. For instance, `plant(2:4,1,9:11)` specifies outputs `2,3,4,1,9,10,11` from the subsystem `plant`. If a subsystem is listed in `outputvar` without arguments, then all outputs from that subsystem are used.

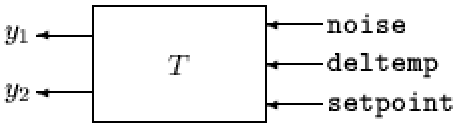
`sysic` also requires that for every subsystem name listed in `systemnames`, a corresponding variable, `input_to_ListedSubSystemName` must exist in the calling workspace. This variable is similar to `outputvar` – it defines the input signals to this particular subsystem as linear combinations of individual subsystem's outputs and external inputs.

`sysout = sysic` will perform the interconnection described by the variables above, using the subsystem data in the names found in `systemnames`. The resulting interconnection is returned in the output argument, listed above as `sysout`.

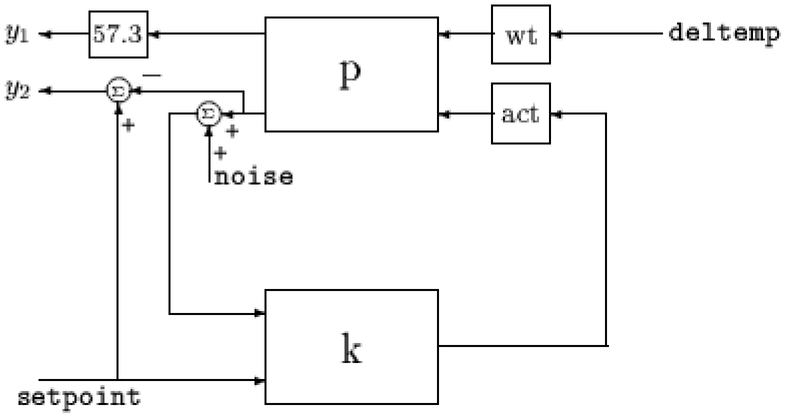
After running `sysic` the variables `systemnames, inputvar, outputvar` and all of the `input_to_ListedSubSystemName` will exist in the workspace. Setting the optional variable `cleanupsysic` to `'yes'` will cause these variables to be removed from the workspace after `sysic` has formed the interconnection.

Examples

A simple system interconnection, identical to the system illustrated in the `iconnect` description. Consider a three-input, two-output LTI matrix *T*,



which has internal structure



```
P = rss(3,2,2);
K = rss(1,1,2);
```

```
A = rss(1,1,1);
W = rss(1,1,1);
systemnames = 'W A K P';
inputvar = '[noise;deltemp;setpoint]';
outputvar = '[57.3*P(1);setpoint-P(2)]';
input_to_W = '[deltemp]';
input_to_A = '[K]';
input_to_K = '[P(2)+noise;setpoint]';
input_to_P = '[W;A]';
cleanupsysic = `yes';
T = sysic;
exist(`inputvar')
```

Limitations

The syntax of `sysic` is limited, and for the most part is restricted to what is shown here. The `iconnect` interconnection object can also be used to define complex interconnections, and has a more flexible syntax.

Within `sysic`, error-checking routines monitor the consistency and availability of the subsystems and their inputs. These routines provide a basic level of error detection to aid the user in debugging.

See Also

`iconnect`

Introduced before R2006a

ucomplex

Create uncertain complex parameter

Syntax

```
A = ucomplex('NAME',nominalvalue)
```

```
A = ucomplex('NAME',nominalvalue,'Property1',Value1,...
         'Property2',Value2,...)
```

Description

An uncertain complex parameter is used to represent a complex number whose value is uncertain. Uncertain complex parameters have a name (the `Name` property), and a nominal value (the `NominalValue` property).

The uncertainty (potential deviation from the nominal value) is described in two different manners:

- `Radius` (radius of disc centered at `NominalValue`)
- `Percentage` (disc size is percentage of magnitude of `NominalValue`)

The `Mode` property determines which description remains invariant if the `NominalValue` is changed (the other is derived). The default `Mode` is `'Radius'` and the default radius is 1.

Property/Value pairs can also be specified at creation. For instance,

```
B = ucomplex('B',6-j,'Percentage',25)
```

sets the nominal value to `6-j`, the percentage uncertainty to `25` and, implicitly, the `Mode` to `'Percentage'`.

Examples

Sample Uncertain Complex Parameter

Compute 400 random samples of an uncertain complex parameter and visualize them in a plot.

Create an uncertain complex parameter with internal name A.

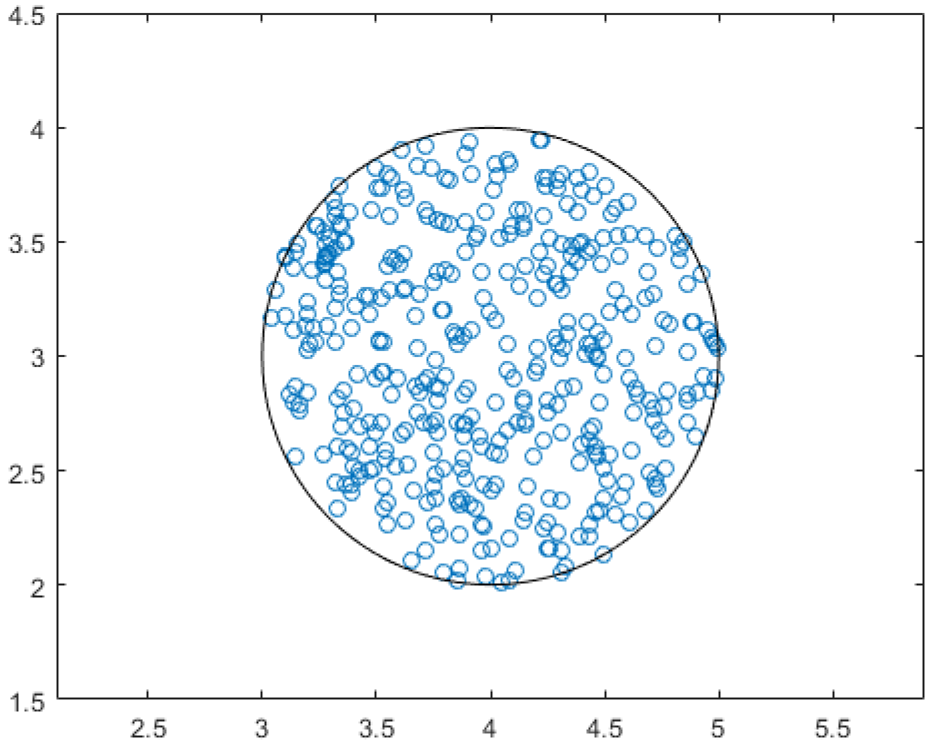
```
A = ucomplex('A',4+3*j)
```

```
A =

  Uncertain complex parameter "A" with nominal value 4+3i and radius 1.
```

The uncertain parameter's possible values are a complex disc of radius 1, centered at 4 + 3_j_. The value of `A.percentage` is 20 (radius is 1/5 of the magnitude of the nominal value).

You can visualize the uncertain complex parameter by sampling and plotting the data.

```
sa = usample(A,400);
w = linspace(0,2*pi,200);
circ = sin(w) + j*cos(w);
rc = real(A.NominalValue+circ);
ic = imag(A.NominalValue+circ);
plot(real(sa(:)),imag(sa(:)),'o',rc,ic,'k-')
xlim([2.5 5.5])
ylim([1.5 4.5])
axis equal
```



See Also

`get` | `ucomplexm` | `ultidyn` | `umat` | `ureal`

Introduced before R2006a

ucomplexm

Create uncertain complex matrix

Syntax

```
M = ucomplexm('Name',NominalValue)
```

```
M = ucomplexm('Name',NominalValue,'WL',WLvalue,'WR',WRvalue)
```

```
M = ucomplexm('Name',NominalValue,'Property',Value)
```

Description

`M = ucomplexm('Name',NominalValue)` creates an uncertain complex matrix representing a ball of complex-valued matrices, centered at a `NominalValue` and named `Name`.

`M = ucomplexm('Name',NominalValue,'WL',WLvalue,'WR',WRvalue)` creates an uncertain complex matrix with weights `WL` and `WR`. Specifically, the values represented by `M` are all matrices `H` that satisfy `norm(inv(M.WL)*(H - M.NominalValue)*inv(M.WR)) <= 1.` `WL` and `WR` are square, invertible, and weighting matrices that quantify the size and shape of the ball of matrices represented by this object. The default values for `WL` and `WR` are identity matrices of appropriate dimensions.

Trailing Property/Value pairs are allowed, as in

```
M = ucomplexm('NAME',nominalvalue,'P1',V1,'P2',V2,...)
```

The property `AutoSimplify` controls how expressions involving the uncertain matrix are simplified. Its default value is `'basic'`, which means elementary methods of simplification are applied as operations are completed. Other values for `AutoSimplify` are `'off''`, no simplification performed, and `'full'` which applies model-reduction-like techniques to the uncertain object.

Examples

Sample an Uncertain Complex Matrix

Create a `ucomplexm` with the name `F`, nominal value `[1 2 3; 4 5 6]`, and weighting matrices `WL = diag([.1.3])`, `WR = diag([.4 .8 1.2])`.

```
F = ucomplexm('F',[1 2 3;4 5 6],'WL',diag([.1 .3]),...
   'WR',diag([.4 .8 1.2]))
```

```
F =

  Uncertain complex matrix "F" with 2 rows and 3 columns.
```

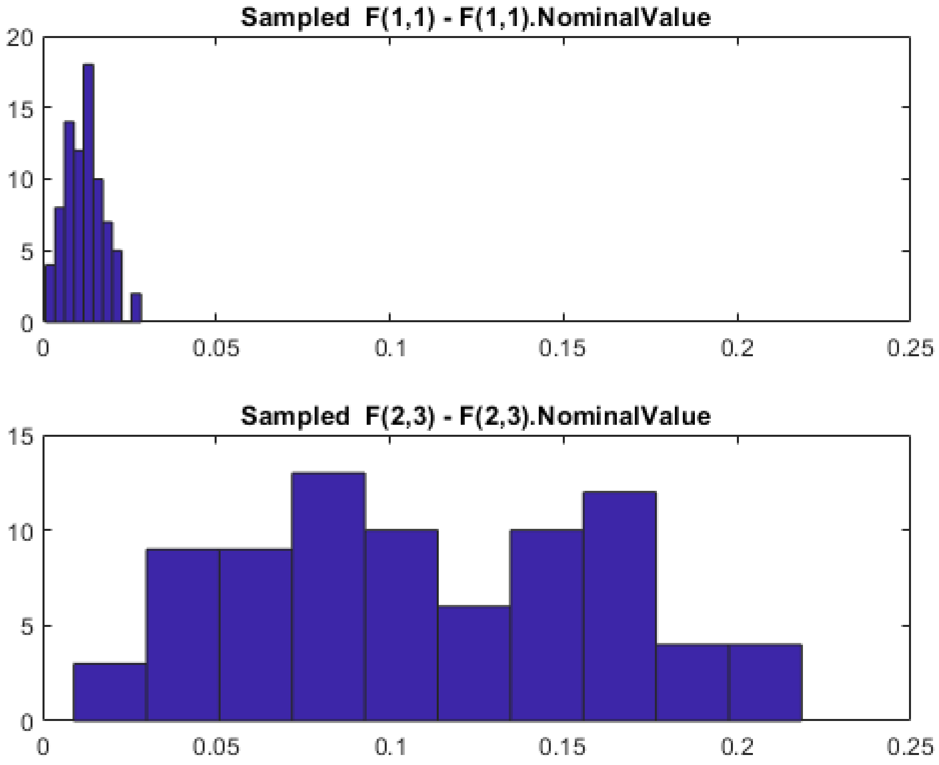
Sample the difference between the uncertain matrix and its nominal value at 80 points, yielding a 2-by-3-by-80 matrix `typicaldev`.

```
typicaldev = usample(F - F.NominalValue,80);
```

Plot histograms of the deviations in the (1,1) entry and the (2,3) entry of the complex matrix.

The absolute values of the (1,1) entry and the (2,3) entry are shown by histogram plots. Typical deviations in the (1,1) entry should be about 10 times smaller than the typical deviations in the (2,3) entry.

```
subplot(2,1,1);
td11 = squeeze(typicaldev(1,1,:));
hist(abs(td11));
xlim([0 .25])
title('Sampled  F(1,1) - F(1,1).NominalValue')
subplot(2,1,2);
td23 = squeeze(typicaldev(2,3,:));
hist(abs(td23));
title('Sampled  F(2,3) - F(2,3).NominalValue')
```



See Also

`get` | `ucomplex` | `ultidyn` | `umat` | `ureal`

Introduced before R2006a

ucover

Fit an uncertain model to set of LTI responses

Syntax

```
usys = ucover(Parray,Pnom,ord)
usys = ucover(Parray,Pnom,ord1,ord2,utype)
[usys,info] = ucover(Parray,...)
[usys_new,info_new] = ucover(Pnom,info,ord1_new,ord2_new)
```

Description

`usys = ucover(Parray,Pnom,ord)` returns an uncertain model `usys` with nominal value `Pnom` and whose range of behaviors includes all responses in the LTI array `Parray`. The uncertain model structure is of the form $usys = Pnom(1 + W(s)\Delta(s))$, where

- Δ is an `ultidyn` object that represents uncertain dynamics with unit peak gain.
- W is a stable, minimum-phase shaping filter that adjusts the amount of uncertainty at each frequency.

`ord` is the number of states (order) of W. `Pnom` and `Parray` can be `ss`, `tf`, `zpk`, or `zpk` models. `usys` is of class `ufrd` when `Pnom` is an `frd` model and is an `uss` model otherwise.

`usys = ucover(Parray,Pnom,ord1,ord2,utype)` specifies the order `ord1` and `ord2` of each diagonal entry of $W1$ and $W2$, where $W1$ and $W2$ are diagonal, stable, minimum-phase shaping filters. `utype` specifies the uncertain model structure, as described in "Uncertain Model Structures" on page 1-566, and can be '`InputMult`' (default), '`OutputMult`' or '`Additive`'. `ord1` and `ord2` can be:

- `[]`, which implies that the corresponding filter is 1.
- Scalar, which implies that the corresponding filter is scalar-valued.
- Vectors with as many entries as diagonal entries in $W1$ and $W2$.

`[usys,info] = ucover(Parray,...)` returns a structure `info` that contains optimization information. `info.W1opt` and `Info.W2opt` contain the values of $W1$ and

*W*2 computed on a frequency grid and `info.W1` and `info.W2` contain the fitted shaping filters W1 and W2.

`[usys_new,info_new] = ucover(Pnom,info,ord1_new,ord2_new)` improves the fit using initial filter values in `info and` new orders `ord1_new` and `ord2_new` of *W*1 and *W*2. This syntax speeds up command execution by reusing previously computed information. Use this syntax when you are changing filter orders in an iterative workflow.

Examples

Fit Uncertain Model to Array of LTI Responses

Fit an uncertain model to an array of LTI responses. The responses might be, for example, the results of multiple runs of acquisition of frequency response data from a physical system.

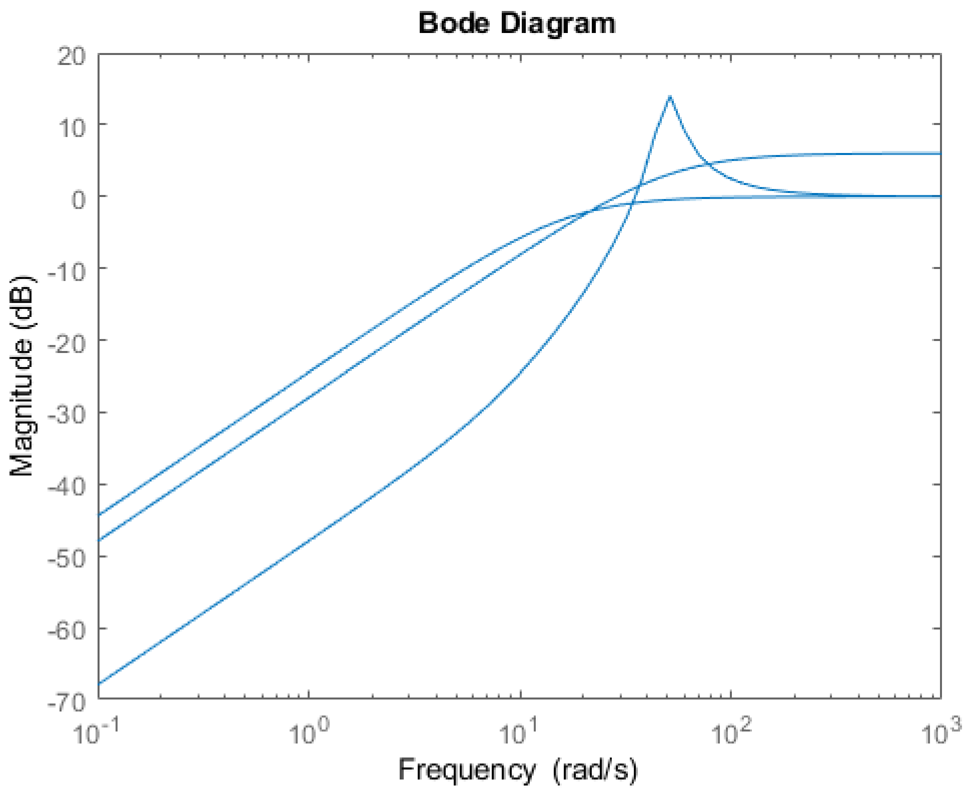
For the purpose of this example, generate the frequency response data by creating an array of LTI models and sampling the frequency response of those models.

```
Pnom = tf(2,[1 -2]);
p1 = Pnom*tf(1,[.06 1]);
p2 = Pnom*tf([-.02 1],[.02 1]);
p3 = Pnom*tf(50^2,[1 2*.1*50 50^2]);
array = stack(1,p1,p2,p3);
Parray = frd(array,logspace(-1,3,60));
```

The frequency response data in `Parray` represents three separate data acquisition experiments on the system.

Plot relative errors between the nominal plant response and the three models in the LTI array.

```
relerr = (Pnom-Parray)/Pnom;
bodemag(relerr)
```



If you use a multiplicative uncertainty model structure, the magnitude of the shaping filter should fit the maximum relative errors at each frequency.

Try a 1st-order shaping filter to fit the maximum relative errors.

```
[P,Info] = ucover(Parray,Pnom,1);
```

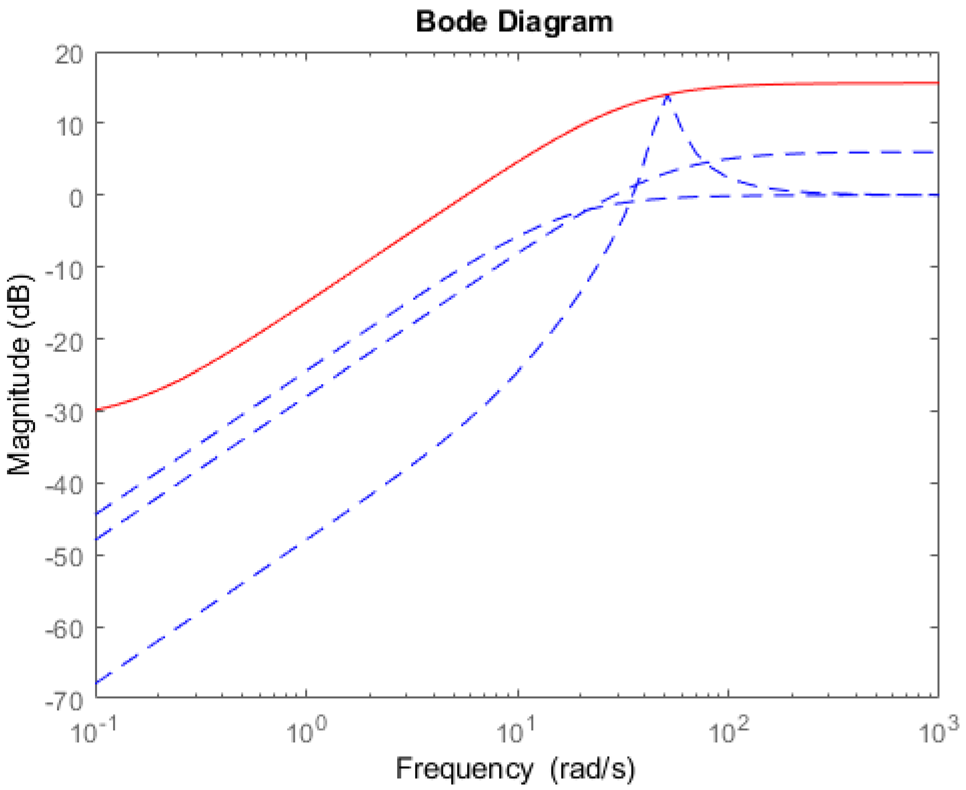
`P` is an uncertain state-space (`uss`) model that captures the uncertainty as a `ultidyn` uncertain dynamics block.

```
P.Uncertainty
```

```
ans = struct with fields:
    Parray_InputMultDelta: [1x1 ultidyn]
```

The `Info` structure contains other information about the fit, including the shaping filter. Plot the response to see how well the shaping filter fits the relative errors.

```
bodemag(relerr,'b--',Info.W1,'r',{0.1,1000});
```

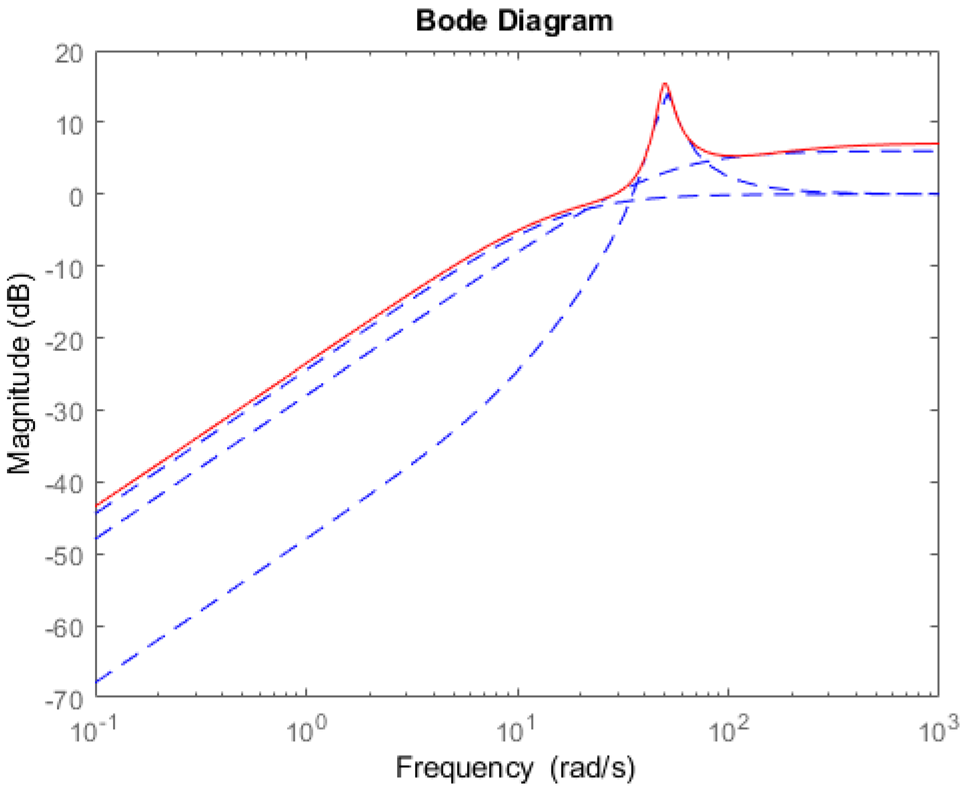


The plot shows that the filter `W1` is too conservative and exceeds the maximum relative error at most frequencies. To obtain a tighter fit, rerun the function using a 4th-order filter.

```
[P,Info] = ucover(Parray,Pnom,4);
```

Evaluate the fit by plotting the Bode magnitude plot.

```
bodemag(relerr,'b--',Info.W1,'r',{0.1,1000});
```

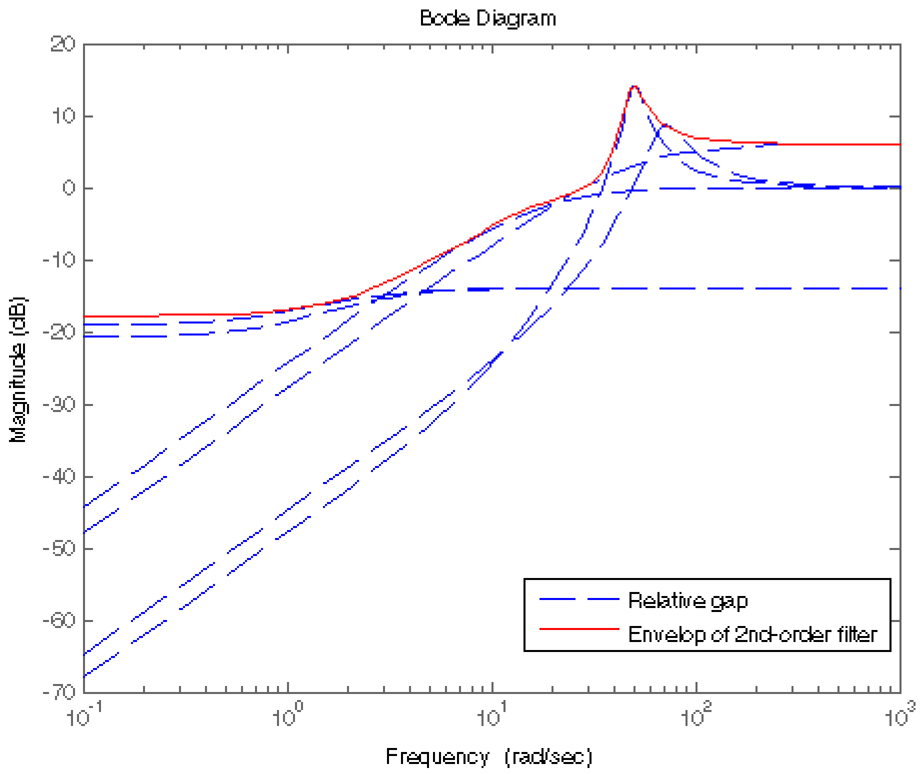


This plot shows that for the 4th-order filter, the magnitude of `W1` closely matches the minimum uncertainty amount.

More About

Uncertain Model Structures

When fitting the responses of LTI models in `Parray`, the gaps between `Parray` and the nominal response `Pnom` of the uncertain model are modeled as uncertainty on the system dynamics. To model the frequency distribution of these unmodeled dynamics, `ucover` measures the gap between `Pnom` and `Parray` at each frequency and selects a shaping filter W whose magnitude approximates the maximum gap between `Pnom` and `Parray`. The following figure shows the relative gap between the nominal response and six LTI responses, enveloped using a second-order shaping filter.



The software then sets the uncertainty to $W \cdot \Delta$, where Δ is an `ultidyn` object that represents unit-gain uncertain dynamics. This ensures that the amount of uncertainty at each frequency is specified by the magnitude of W and therefore closely tracks the gap between `Pnom` and `Parray`.

There are three possible uncertainty model structures:

- Input Multiplicative of the form $usys = Pnom \times (I + W_1 \times \Delta \times W_2)$.
- Output Multiplicative of the form $usys = (I + W1 \times \Delta \times W2) \times Pnom$.
- Additive of the form $usys = Pnom + W1 \times \Delta \times W_2$.

Use additive uncertainty to model the absolute gaps between `Pnom` and `Parray`, and multiplicative uncertainty to model relative gaps.

Note For SISO models, input and output multiplicative uncertainty are equivalent. For MIMO systems with more outputs than inputs, the input multiplicative structure may be too restrictive and not adequately cover the range of models.

The model structure $usys = Pnom \times (I + W \times \Delta)$ that you obtain using `usys = ucover(Parray,Pnom,ord)`, corresponds to $W_1 = W \times I$ and $W_1 = 1$.

Algorithms

The `ucover` command designs the minimum-phase shaping filters W_1 and W_2 in two steps:

1. Computes the optimal values of W_1 and W_2 on a frequency grid.
2. Fits W_1 and W_2 values with the dynamic filters of the specified orders using the `fitmagfrd` command.

See Also

`frd` | `ss` | `tf` | `usample` | `zpk`

Topics

“Modeling a Family of Responses as an Uncertain System”
“Simultaneous Stabilization Using Robust Control”
“First-Cut Robust Design”

Introduced in R2009b

udyn

Create unstructured uncertain dynamic system object

Syntax

```
n = udyn('name',iosize);
```

Description

`n = udyn('name',iosize)` creates an unstructured uncertain dynamic system class, with input/output dimension specified by `iosize`. This object represents the class of completely unknown multivariable, time-varying nonlinear systems.

For practical purposes, these uncertain elements represent noncommuting symbolic variables (placeholders). All algebraic operations, such as addition, subtraction, and multiplication (i.e., cascade) operate properly, and substitution (with `usubs`) is allowed.

The analysis tools (e.g., `robstab`) do not currently handle these types of uncertain elements. Therefore, these elements do not provide a significant amount of usability, and their role in the toolbox is small.

Examples

You can create a `2-by-3 udyn` element and check its size and properties.

```
N = udyn('N',[2 3])
Uncertain Dynamic System: Name N, size 2x3
size(N)
ans =
     2     3
get(N)
            Name: 'N'
    NominalValue: [2x3 double]
    AutoSimplify: 'basic'
```

See Also

`ucomplex` | `ucomplexm` | `ultidyn` | `ureal`

Introduced before R2006a

ufind

Find uncertain variables in Simulink model

Syntax

```
uvars = ufind('mdl')
```

```
[uvars,pathinfo] = ufind('mdl')
```

```
uvars = ufind(usys_1,usys_2,...)
```

Description

`uvars = ufind ('mdl')` finds Uncertain State Space blocks in the Simulink model `mdl`. It returns a structure `uvars` that contains all uncertain variables associated with the Uncertain State Space blocks. Each uncertain variable is a `ureal` or `ultidyn` object and is listed by name in `uvars`.

`[uvars,pathinfo] = ufind('mdl')` returns a cell array `pathinfo`that contains paths to the Uncertain State Space blocks and the corresponding uncertain variables in the block. The first column of `pathinfo` lists the block paths through the model hierarchy and the second column lists the uncertain variables associated with the block. Use `pathinfo` to verify that all Uncertain State Space blocks in the model `mdl` have been identified.

`uvars = ufind(usys_1,usys_2,...)` collects all uncertain variables referenced by the uncertain model `usys_n`. `usys_n` can be `uss` or `ufrd` models. Use this syntax as an alternative to querying the model itself, when you know the uncertain models that the Uncertain State Space blocks use.

`ufind` can find Uncertain State Space blocks inside Masked Subsystems, Library Links, and Model References but not inside Accelerated Model References. `ufind` errors out if the same uncertain variable name has different definitions in the model. For example, if your model contains two Uncertain State Space blocks where the uncertain system variables define the same uncertain variable `'unc_par"` as `ultidyn('unc_par',[1 1])` and `ureal('unc_par',5)`, such an error occurs.

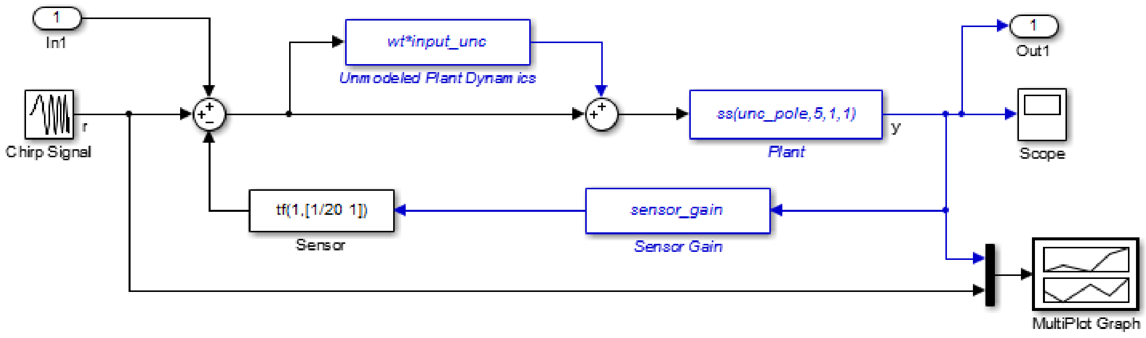
Examples

Find all Uncertain State Space blocks and uncertain variables in a Simulink model:

1. Open the Simulink model.

   ```
   open_system('usim_model')
   ```

   The model, as shown in the following figure, contains three Uncertain State Space blocks named Unmodeled Plant Dynamics, Plant, and Sensor Gain. These blocks depend on three uncertain variables named `input_unc`, `unc_pole` and `sensor_gain`.



2. Use `ufind` to find all Uncertain State Space blocks and uncertain variables in the model.

   ```
   [uvars,pathinfo] = ufind('usim_model')
   ```

3. Type `uvars` to view the structure `uvars`. MATLAB returns the following result:

   ```
   uvars =

         input_unc: [1x1 ultidyn]
       sensor_gain: [1x1 ureal]
          unc_pole: [1x1 ureal]
   ```

   Each uncertain variable is a `ureal` or `ultidyn` object and is listed by name in `uvars`.

4. View the Uncertain State Space block paths and uncertain variables.

   a. Type `pathinfo(:,1)` to view paths of the Uncertain State Space blocks in the model. MATLAB returns the following result:

```
ans =

    'usim_model/Plant'
    'usim_model/Sensor Gain'
    'usim_model/Unmodeled Plant Dynamics'
```

b Type `pathinfo(:,2)` to view the uncertain variables referenced by each Uncertain State Space block. MATLAB returns the following results:

```
ans =

    'unc_pole'
    'sensor_gain'
    'input_unc'
```

Tutorials

"Vary Uncertainty Values Using Individual Uncertain State Space Blocks"

"Vary Uncertainty Values Across Multiple Uncertain State Space Blocks"

Robustness Analysis in Simulink

How To

"Simulate Uncertainty Effects"

See Also

Uncertain State Space | `usample`

Introduced in R2009b

ufrd

Uncertain frequency response data model

Syntax

```
ufrd_sys = ufrd(M,freqs)
ufrd_sys = ufrd(M,freqs,frequnits)
ufrd_sys = ufrd(M,freqs,frequnits,timeunits)
```

Description

Uncertain frequency response data models (`ufrd`) arise when combining numeric `frd` models with uncertain models such as `ureal`, `ultidyn`, or `uss`. A `ufrd` model keeps track of how the uncertain elements affect the frequency response. Use `ufrd` for robust stability and worst-case performance analysis.

There are three ways to construct a `ufrd` model:

1. Combine numeric `frd` models with uncertain models using model arithmetic. For example:

   ```
   sys = frd(rand(100,1),logspace(-2,2,100));
   k = ureal('k',1);
   D = ultidyn('Delta',[1 1]);
   ufrd_sys = k*sys*(1+0.1*D)
   ```

   `ufrd_sys` is a `ufrd` model with uncertain elements `k` and `D`.
2. `ufrd_sys = ufrd(M,freqs)` converts the dynamic system model (Control System Toolbox) or static model (Control System Toolbox) `M` to `ufrd`. If `M` contains Control Design Blocks that do not represent uncertainty, these blocks are replaced by their current value. (To preserve both tunable and uncertain Control Design Blocks, use `genfrd` instead.)

   Use `ufrd_sys = ufrd(M,freqs,frequnits)` to specify the frequency units of the frequencies in `freqs`. The argument `frequnits` can take the following values:

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

Use `ufrd_sys = ufrd(M,freqs,frequnits,timeunits)` to specify the time unit of `ufrd_sys` when `M` is a static model. `timeunits` can take the following values:

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

3. Use `frd` to construct a `ufrd` model from an uncertain matrix (`umat`) representing uncertain frequency response data. For example:

```
a = ureal('delta',1,'percent',50);
freq = logspace(-2,2,100);
RespData = rand(1,1,100) * a;
usys = frd(RespData,freq,0.1)
```

Examples

Compute the uncertain frequency response of an uncertain system (`uss` model) with both parametric uncertainty (`ureal`) and unmodeled dynamics uncertainty (`ultidyn`).

```
p1 = ureal('p1',5,'Range',[2 6]);
p2 = ureal('p2',3,'Plusminus',0.4);
p3 = ultidyn('p3',[1 1]);
Wt = makeweight(.15,30,10);
A = [-p1 0;p2 -p1];
B = [0;p2];
C = [1 1];
usys = uss(A,B,C,0)*(1+Wt*p3);

usysfrd = ufrd(usys,logspace(-2,2,60));
```

Plot 20 random samples and the nominal value of the uncertain frequency response.

```
bode(usysfrd,'r',usysfrd.NominalValue,'b+')
```

See Also

`frd` | `genfrd` | `ss` | `uss`

Topics

"Control Design Blocks" (Control System Toolbox)

Introduced before R2006a

ulinearize

Linearize Simulink model with Uncertain State Space block

Syntax

```
ulin = ulinearize('sys',io)
```

```
ulin = ulinearize('sys',op,io)
```

```
ulin = ulinearize('sys',op,io,options)
```

```
ulin = ulinearize('sys',op)
```

```
ulin_block = ulinearize('sys',op,'blockname')
```

```
[ulin,op] = ulinearize('sys',snapshottimes,...);
```

```
ulin = ulinearize('sys','StateOrder',stateorder)
```

Description

`ulin = ulinearize('sys',io)` linearizes the Simulink model `sys` that contains Uncertain State Space blocks and returns a linear time-invariant uncertain system `ulin.` `ulin` is an `uss` object. `io` is an I/O object that specifies linearization I/O points in the model. Use `getlinio` or `linio` to create `io`. The linearization occurs at the operating point specified in the model.

`ulin=ulinearize('sys',io,op)` linearizes the model at the operating point specified in the operating point object `op`. Use `operpoint` or `findop` to create `op`. Both `op` and `io` are associated with the same model `sys`.

`ulin=ulinearize('sys',io,op,options)` takes a linearization options object `options` that contains several options for linearization and returns linear time-invariant uncertain system `ulin`. Use `linearizeOptions` to create `options`.

`ulin=ulinearize('sys',op)` linearizes the model `sys` at the operating point specified in the operating point object `op`. The software uses root-level inport and outport blocks in `sys` as I/O points for linearization.

`ulin_block=ulinearize('sys',op,'blockname',...)` takes the name of a block `blockname` in the model `sys` and returns a linear time-invariant uncertain system `ulin_block`. You can also specify a fourth argument `options` to provide options for the linearization.

`[ulin,op] = ulinearize('sys',snapshottimes,...)` creates operating points for linearization by simulating the model and taking snapshots of the system's states and inputs at times specified in the vector `snapshottimes`. `ulin` is a set of linear time-invariant uncertain systems and `op` is the set of operating point objects used in linearization. You can also specify I/O object for linearization, or a block name. If you do not specify an I/O object or block name, the linearization uses root-level inport and outport blocks in the model. You can also supply an additional argument, `options`, to provide options for linearization.

`ulin = ulinearize('sys','StateOrder',stateorder)` creates a linear-time-invariant uncertain system `ulin`, whose states are in a specified order. Specify the state order in the cell array `stateorder` by entering the names of the blocks containing states in the model. For all blocks, you can enter block names as the full block path. For continuous blocks, you can alternatively enter block names as the user-defined unique state name.

Examples

Compute uncertain linearization of a Simulink model containing Uncertain State Space blocks:

```
% Define uncertain variables and uncertain system variables
% to use in Uncertain State Space blocks.
unc_pole = ureal('unc_pole',-5,'Range',[-10 -4]);
plant = ss(unc_pole,5,1,0);
wt = makeweight(0.25,130,2.5);
input_unc = ultidyn('input_unc',[1 1]);
sensor_pole = ureal('sensor_pole',-20,'Range',[-30 -10]);
sensor = tf(1,[1/(-sensor_pole) 1]);

% Open Simulink model. The model contains three Uncertain State
% Space blocks named Unmodeled Plant Dynamics, Uncertain Plant and
% Uncertain Sensor, and linearization I/O points.
open_system('rct_ulinearize_uss')

% Obtain linearization I/O points.
```

```
mdl = 'rct_ulinearize_uss';
io = getlinio(mdl);

% Compute the uncertain linearization of the model.
ulin = ulinearize(mdl,io)
% MATLAB returns an uss object with 5 states.
```

Tutorials

“Linearize Block to Uncertain Model”

Linearization of Simulink Models with Uncertainty

How To

“Obtain Uncertain State-Space Model from Simulink Model”

See Also

`udyn` | `ultidyn` | `ureal` | `uss`

Introduced in R2009b

ultidyn

Create uncertain linear time-invariant object

Syntax

```
H = ultidyn('Name',iosize)
```

```
H = ultidyn('Name',iosize,'Property1',Value1,'Property2',Value2,...)
```

Description

`H = ultidyn('Name',iosize)` creates an uncertain linear, time-invariant objects are used to represent unknown dynamic objects whose only known attributes are bounds on their frequency response. Uncertain linear, time-invariant objects have a name (the `Name` property), and an input/output size (`ioSize` property).

Trailing Property/Value pairs are allowed in the construction.

```
H = ultidyn('name',iosize,'Property1',Value1,'Property2',Value2,...)
```

The property `Type` is `'GainBounded'` (default) or `'PositiveReal'`, and describes in what form the knowledge about the object's frequency response is specified.

- If `Type` is `'GainBounded'`, then the knowledge is an upper bound on the magnitude (i.e., absolute value), namely `abs(H)<= Bound` at all frequencies. The matrix generalization of this is ‖H‖`<= Bound`.
- If `Type` is `'PositiveReal'` then the knowledge is a lower bound on the real part, namely `Real(H) >= Bound` at all frequencies. The matrix generalization of this is `H +H' >= 2*Bound`

The property `Bound` is a real, scalar that quantifies the bound on the frequency response of the uncertain object as described above.

The property `SampleStateDimension` is a positive integer, defining the state dimension of random samples of the uncertain object when sampled with `usample`. The default value is 1.

The property AutoSimplify controls how expressions involving the uncertain matrix are simplified. Its default value is 'basic', which means elementary methods of simplification are applied as operations are completed. Other values for AutoSimplify are 'off', no simplification performed, and 'full' which applies model-reduction-like techniques to the uncertain object.

Use the property SampleMaxFrequency to limit the natural frequency for sampling. Randomly sampled uncertain dynamics are no faster than the specified value. The default value is Inf (no limit).

To model frequency-dependent uncertainty levels, multiply the ultidyn object by a suitable shaping filter. For example, for a ultidyn object dH, the following commands specify an uncertainty bound that increases from 0.1 at low frequencies to 10 at high frequencies.

```
W = tf([1 .1],[.1 1]);
dH = W*dH;
```

Examples

MIMO Uncertain Dynamics

Create an ultidyn object with internal name 'H', dimensions 2-by-3, norm bounded by 7.

```
H = ultidyn('H',[2 3],'Bound',7)
```

```
Uncertain GainBounded LTI Dynamics: Name H, 2x3, Gain Bound = 7
```

Nyquist Plot of Uncertain Dynamics

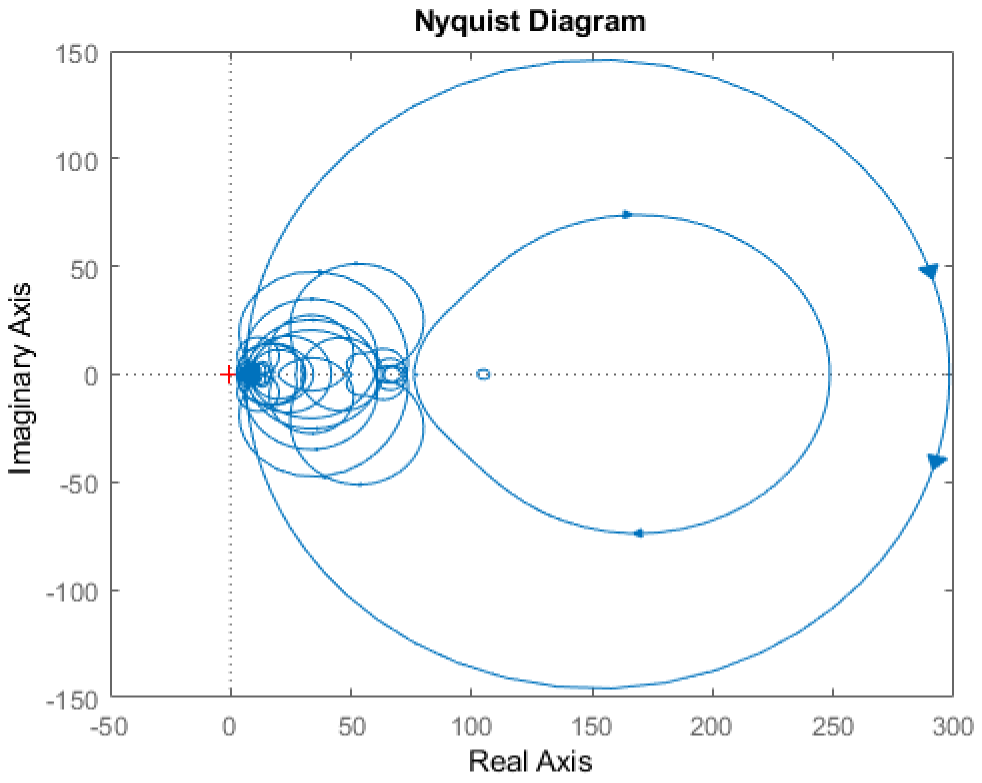
Create a scalar ultidyn object with an internal name 'B', whose frequency response has a real part greater than 2.5.

```
B = ultidyn('B',[1 1],'Type','PositiveReal','Bound',2.5)
```

```
B =

  Uncertain LTI dynamics "B" with 1 outputs, 1 inputs, and positive real bound of 2.5.
```

Change the `SampleStateDimension` to 5, and plot the Nyquist plot of 30 random samples.

```
B.SampleStateDimension = 5;
nyquist(usample(B,30))
```



See Also

get | ureal | uss

Introduced before R2006a

umat

Create uncertain matrix

Syntax

```
M = umat(A)
```

Description

Uncertain matrices are rational expressions involving uncertain elements of type `ureal`, `ucomplex`, or `ucomplexm`. Use uncertain matrices for worst-case gain analysis and for building uncertain state-space (`uss`) models.

Create uncertain matrices by creating uncertain elements and combining them using arithmetic and matrix operations. For example:

```
p = ureal('p',1);
M = [0 p; 1 p^2]
```

creates a 2-by-2 uncertain matrix (a `umat` object) with the uncertain parameter `p`.

The syntax `M = umat(A)` converts the double array `A` to a `umat` object with no uncertainty.

Most standard matrix manipulations are valid on uncertain matrices, including addition, multiplication, inverse, horizontal and vertical concatenation. Specific rows/columns of an uncertain matrix can be referenced and assigned also.

If `M` is a `umat`, then `M.NominalValue` is the result obtained by replacing each uncertain element in `M` with its own nominal value.

If `M` is a `umat`, then `M.Uncertainty` is an object describing all the uncertain elements in `M`. All element can be referenced and their properties modified with this `Uncertainty` gateway. For instance, if `B` is an uncertain real parameter in `M`, then `M.Uncertainty.B` accesses the uncertain element `B` in `M`.

Examples

Create 3 uncertain elements and then a `3-by-2 umat`.

```
a = ureal('a',5,'Range',[2 6]);
b = ucomplex('b',1+j,'Radius',0.5);
c = ureal('c',3,'Plusminus',0.4);
M = [a b;b*a 7;c-a b^2]
```

`M` is an uncertain matrix (`umat` object) with the uncertain parameters `a`, `b`, and `c`.

View the properties of `M` with `get`

```
get(M)
```

The nominal value of `M` is the result when all atoms are replaced by their nominal values.

```
M.NominalValue
ans =
   5.0000             1.0000 + 1.0000i
   5.0000 + 5.0000i   7.0000
  -2.0000                  0 + 2.0000i
```

Change the nominal value of `a` within `M` to 4. The nominal value of `M` reflects this change.

```
M.Uncertainty.a.NominalValue = 4;
M.NominalValue
ans =
   4.0000             1.0000 + 1.0000i
   4.0000 + 4.0000i   7.0000
  -1.0000                  0 + 2.0000i
```

Get a random sample of `M`, obtained by taking random samples of the uncertain atoms within `M`.

```
usample(M)
ans =
   2.0072             0.8647 + 1.3854i
   1.7358 + 2.7808i   7.0000
   1.3829            -1.1715 + 2.3960i
```

Select the 1st and 3rd rows, and the 2nd column of `M`. The result is a 2-by-1 `umat`, whose dependence is only on `b`.

```
M([1 3],2)
```

See Also

`ucomplex` | `ucomplexm` | `ultidyn` | `ureal` | `usample`

Introduced before R2006a

uplot

Plot multiple frequency response objects and `double`s on same graph

Syntax

```
uplot(G1)
uplot(G1,G2)
uplot(G1,Xdata,Ydata)
uplot(G1,Xdata,Ydata,...)
uplot(G1,linetype)
uplot(G1,linetype,G2,...)
uplot(G1,linetype,Xdata,Ydata,linetype)
uplot(type,G1,linetype,Xdata,Ydata,linetype)
H = uplot(G1)
H = uplot(G1,G2)
H = uplot(G1,Xdata,Ydata)
H = uplot(G1,Xdata,Ydata,...)
H = uplot(G1,linetype)
H = uplot(G1,linetype,G2,...)
H = uplot(G1,linetype,Xdata,Ydata,linetype)
```

Description

`uplot` plots `double` and `frd` objects. The syntax is the same as the MATLAB `plot` command except that all data is contained in `frd` objects, and the axes are specified by `type`.

The (optional) `type` argument must be one of

Type	Description
`'iv,d'`	Data versus independent variable (default)
`'iv,m'`	Magnitude versus independent variable
`'iv,lm'`	`log`(magnitude) versus independent variable
`'iv,p'`	Phase versus independent variable
`'liv,m'`	Magnitude versus `log`(independent variable)
`'liv,d'`	Data versus `log`(independent variable)
`'liv,m'`	Magnitude versus `log`(independent variable)
`'liv,lm'`	`log`(magnitude) versus `log`(independent variable)
`'liv,p'`	Phase versus `log`(independent variable)
`'r,i'`	Real versus imaginary (parametrize by independent variable)
`'nyq'`	Real versus imaginary (parametrize by independent variable)
`'nic'`	Nicholas plot
`'bode'`	Bode magnitude and phase plot

The remaining arguments of `uplot` take the same form as the MATLAB `plot` command. Line types (for example,`'+', 'g-.', or '*r'`) can be optionally specified after any frequency response argument.

There is a subtle distinction between constants and `frd` objects with only one independent variable. A constant is treated as such across all frequencies, and consequently shows up as a line on any graph with the independent variable as an axis. A `frd` object with only one frequency point always shows up as a point. You might need to specify one of the more obvious point types in order to `see` it (e.g., `'+'`, `'x'`, etc.).

Examples

Plot Multiple Frequency Responses

Create two SISO second-order systems, and calculate their frequency responses over different frequency ranges.

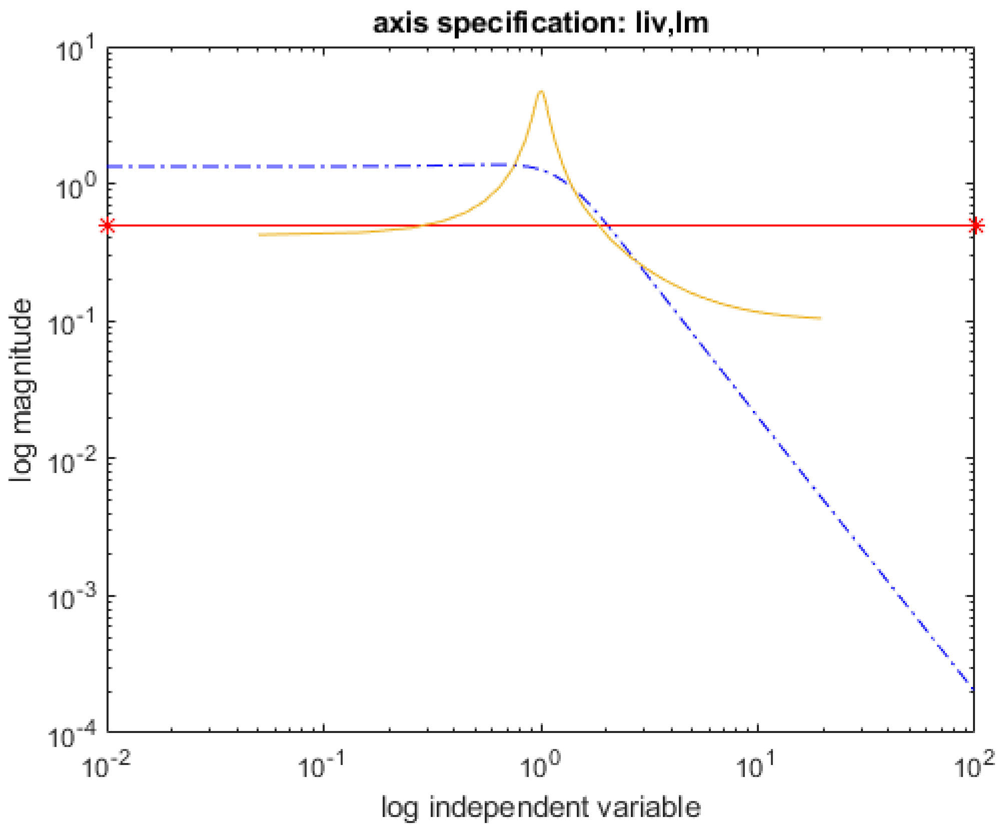
```
a1 = [-1,1;-1,-0.5];
b1 = [0;2]; c1 = [1,0]; d1 = 0;
sys1 = ss(a1,b1,c1,d1);
a2 = [-.1,1;-1,-0.05];
b2 = [1;1]; c2 = [-0.5,0]; d2 = 0.1;
sys2 = ss(a2,b2,c2,d2);
omega = logspace(-2,2,100);
sys1g = frd(sys1,omega);
omega2 = [ [0.05:0.1:1.5] [1.6:.5:20] [0.9:0.01:1.1] ];
omega2 = sort(omega2);
sys2g = frd(sys2,omega2);
```

Create an `frd` object with a single frequency.

```
sys3 = rss(1,1,1);
rspot = frd(sys3,2);
```

The following plot uses the `plot_type` specification `'liv,lm'`.

```
uplot('liv,lm',sys1g,'b-.',rspot,'r*-',sys2g);
xlabel('log independent variable')
ylabel('log magnitude')
title('axis specification: liv,lm')
```



See Also

`bode` | `nichols` | `nyquist` | `plot` | `semilogx` | `semilogy` | `sigma`

Introduced before R2006a

ureal

Create uncertain real parameter

Syntax

```
p = ureal('name',nominalvalue)

p = ureal('name',nominalvalue,'Property1',Value1,...
'Property2',Value2,...)
```

Description

An uncertain real parameter is used to represent a real number whose value is uncertain. Uncertain real parameters have a name (the `Name` property), and a nominal value (`NominalValue` property).

The uncertainty (potential deviation from `NominalValue`) is described (equivalently) in 3 different properties:

- `PlusMinus`: the additive deviation from `NominalValue`
- `Range`: the interval containing `NominalValue`
- `Percentage`: the percentage deviation from `NominalValue`

The `Mode` property specifies which one of these three descriptions remains unchanged if the `NominalValue` is changed (the other two descriptions are derived). The possible values for the `Mode` property are `'Range', 'Percentage'` and `'PlusMinus'`.

The default `Mode` is `'PlusMinus'`, and `[-1 1]` is the default value for the `'PlusMinus'` property. The range of uncertainty need not be symmetric about `NominalValue`.

The property `AutoSimplify` controls how expressions involving the uncertain matrix are simplified. Its default value is `'basic'`, which means elementary methods of simplification are applied as operations are completed. Other values for `AutoSimplify` are `'off''`, no simplification performed, and `'full'`, which applies model-reduction-like techniques to the uncertain object.

Examples

Example 1

Create an uncertain real parameter and use `get` to display the properties and their values. Create uncertain real parameter object `a` with the internal name `'a'` and nominal value `5`.

```
a = ureal('a',5)
Uncertain Real Parameter: Name a, NominalValue 5, variability = [-1  1]
get(a)
            Name: 'a'
    NominalValue: 5
            Mode: 'PlusMinus'
           Range: [4 6]
       PlusMinus: [-1 1]
      Percentage: [-20 20]
    AutoSimplify: 'basic'
```

Note that the `Mode` is `'PlusMinus'`, and that the value of `PlusMinus` is indeed `[-1 1]`. As expected, the range description of uncertainty is `[4 6]`, while the percentage description of uncertainty is `[-20 20]`.

Set the range to `[3 9]`. This leaves `Mode` and `NominalValue` unchanged, but all three descriptions of uncertainty have been modified.

```
a.Range = [3 9];
get(a)
            Name: 'a'
    NominalValue: 5
            Mode: 'PlusMinus'
           Range: [3 9]
       PlusMinus: [-2 4]
      Percentage: [-40 80]
    AutoSimplify: 'basic'
```

Example 2

Property/Value pairs can also be specified at creation.

```
b = ureal('b',6,'Percentage',[-30 40],'AutoSimplify','full');
get(b)
            Name: 'b'
    NominalValue: 6
```

```
            Mode: 'Percentage'
           Range: [4.2000 8.4000]
       PlusMinus: [-1.8000 2.4000]
      Percentage: [-30.0000 40.0000]
    AutoSimplify: 'full'
```

Note that `Mode` is automatically set to `'Percentage'`.

Example 3

Specify the uncertainty in terms of percentage, but force `Mode` to `'Range'`.

```
c = ureal('c',4,'Mode','Range','Percentage',25);
get(c)
            Name: 'c'
    NominalValue: 4
            Mode: 'Range'
           Range: [3 5]
       PlusMinus: [-1 1]
      Percentage: [-25 25]
    AutoSimplify: 'basic'
```

See Also

`getLimits` | `ucomplex` | `umat` | `uss`

Introduced before R2006a

uss/usample

Generate random samples of uncertain or generalized model

Syntax

```
B = usample(A);
B = usample(A,N)
[B,SampleValues] = usample(A,N)
[B,SampleValues] = usample(A,Names,N)
[B,SampleValues] = usample(A,Names1,N1,Names2,N2,...)
[B,SampleValues] = usample(A,N,Wmax)
[B,SampleValues] = usample(A,Names,N,Wmax)
```

Description

`B = usample(A)` substitutes a random sample of the uncertain objects in `A`, returning a certain (i.e., not uncertain) array of size `[size(A)]`. The input `A` can be any uncertain element, matrix, or system, such as `ureal`, `umat`, `uss`, or `ufrd`. `A` can also be any generalized matrix or system, such as `genss` or `genmat`, that contains uncertain blocks and other types of Control Design Blocks (Control System Toolbox). If `A` contains non-uncertain control design blocks, these are unchanged in `B`. Thus, for example, `usample` applied to a `genss` with both tunable and uncertain blocks, the result is a `genss` array with only tunable blocks.

`B = usample(A,N)` substitutes `N` random samples of the uncertain objects in `A`, returning a certain (i.e., not uncertain) array of size `[size(A) N]`.

`[B,SampleValues] = usample(A,N)` additionally returns the specific sampled values (as a `Struct` whose field names are the names of `A`'s uncertain elements) of the uncertain elements. Hence, `B` is the same as `usubs(A,SampleValues)`.

`[B,SampleValues] = usample(A,Names,N)` samples only the uncertain elements listed in the `Names` variable (cell, or char array). If `Names` does not include all the uncertain objects in A, then B will be an uncertain object. Any entries of Names that are not elements of A are simply ignored. Note that usample(A,fieldnames(A.Uncertainty),N) is the same as usample(A,N).

[B,SampleValues] = usample(A,Names1,N1,Names2,N2,...) takes N1 samples of the uncertain elements listed in Names1, and N2 samples of the uncertain elements listed in Names2, and so on. size(B) will equal [size(A) N1 N2 ...].

The scalar parameter Wmax in

```
[B,SampleValues] = usample(A,N,Wmax)
[B,SampleValues] = usample(A,Names,N,Wmax)
[B,SampleValues] = usample(A,Names,N,Wmax)
```

affects how ultidyn elements within A are sampled, restricting the poles of the samples. If A is a continuous-time uss or ufrd, then the poles of sampled GainBounded ultidyn elements in SampleValues will each have magnitude <= BW. If A is a discrete-time, then sampled GainBounded ultidyn elements are obtained by Tustin transformation, using BW/(2*TS) as the (continuous) pole magnitude bound. In this case, BW should be < 1. If the ultidyn type is PositiveReal, then the samples are obtained by bilinearly transforming (see “Normalizing Functions for Uncertain Elements”) the GainBounded elements described above.

Examples

Sample Real Parameter

Create a real uncertain parameter, sample it, and plot a histogram of the sampled values.

```
A = ureal('A',5);
Asample = usample(A,500);
```

Examine the size of the parameter and the sample array.

```
size(A)
```

```
Uncertain real scalar.
```

```
size(Asample)
```

```
ans = 1×3
```

```
     1     1   500
```

A is a scalar parameter. The dimensions of Asample reflect that A is a 1-by-1 parameter. Examine the data type of Asample.

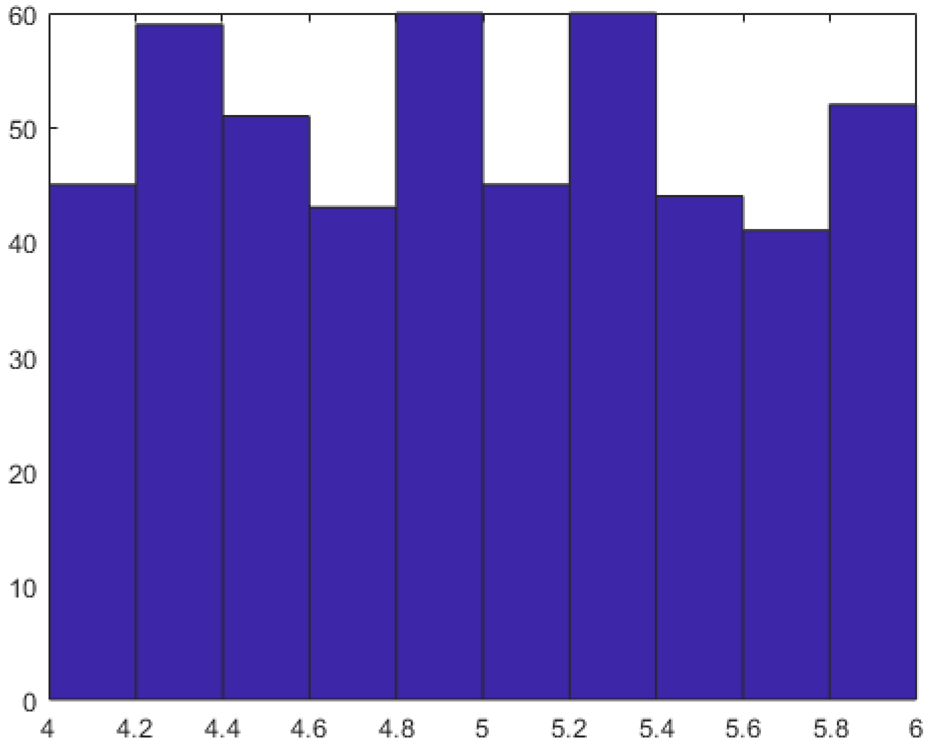
```
class(Asample)
```

```
ans = 
'double'
```

The samples of the scalar parameter are numerical values.

Plot the histogram of sampled values.

```
hist(Asample(:))
```



Sample Responses of Uncertain Control System Model

This example illustrates how to sample the open and closed-loop response of an uncertain plant model for Monte Carlo analysis.

Create two uncertain real parameters and an uncertain plant.

```
gamma = ureal('gamma',4);
tau = ureal('tau',.5,'Percentage',30);
P = tf(gamma,[tau 1]);
```

Create an integral controller based on the nominal values of plant uncertainties.

```
KI = 1/(2*tau.Nominal*gamma.Nominal);
C = tf(KI,[1 0]);
```

Now create an uncertain closed-loop system.

```
CLP = feedback(P*C,1);
```

Sample the plant at 20 values, distributed uniformly about the `tau` and `gamma` parameter cube.

```
[Psample1D,Values1D] = usample(P,20);
size(Psample1D)
```

```
20x1 array of state-space models.
Each model has 1 outputs, 1 inputs, and 1 states.
```

This sampling returns an array of 20 state-space models, each representing the closed-loop system within the uncertainty.

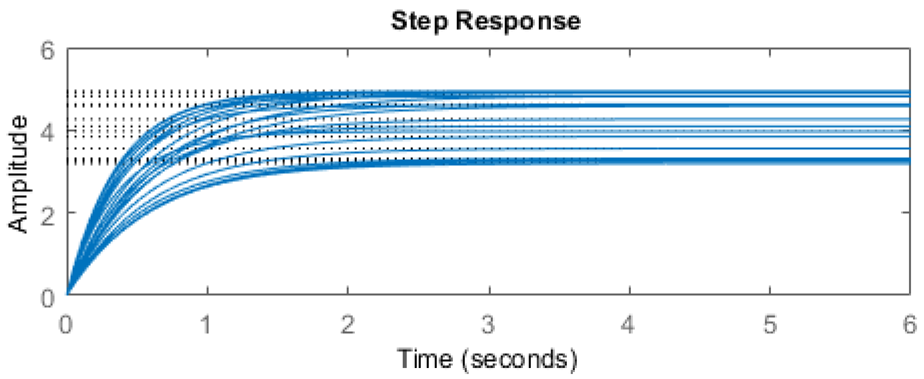
Now sample the plant at 10 values of `tau` and 15 values of `gamma`.

```
[Psample2D,Values2D] = usample(P,'tau',10,'gamma',15);
size(Psample2D)
```

```
10x15 array of state-space models.
Each model has 1 outputs, 1 inputs, and 1 states.
```

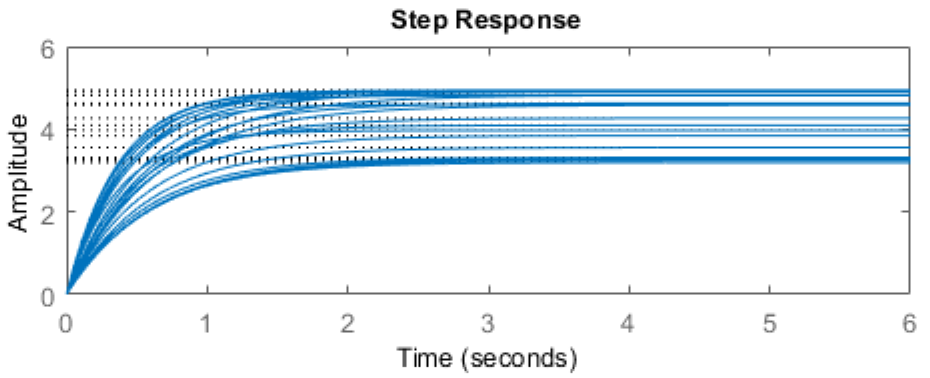
Plot the step responses of the 1-D sampled plant.

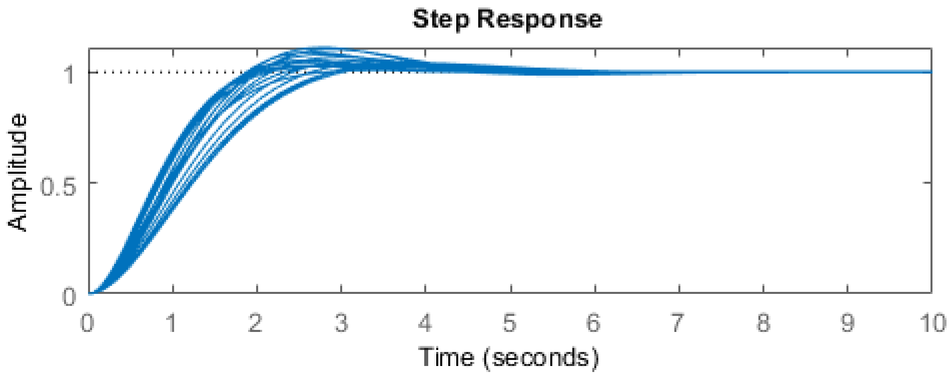
```
subplot(2,1,1);
step(Psample1D)
```



Evaluate the uncertain closed-loop model at the same values using `usubs`, and plot the step response.

```
subplot(2,1,2);
step(usubs(CLP,Values1D))
```





Restrict Pole Locations in Sampled Uncertain Dynamics

To see the effect of limiting the bandwidth of sampled models with `Wmax`, create two `ultidyn` objects.

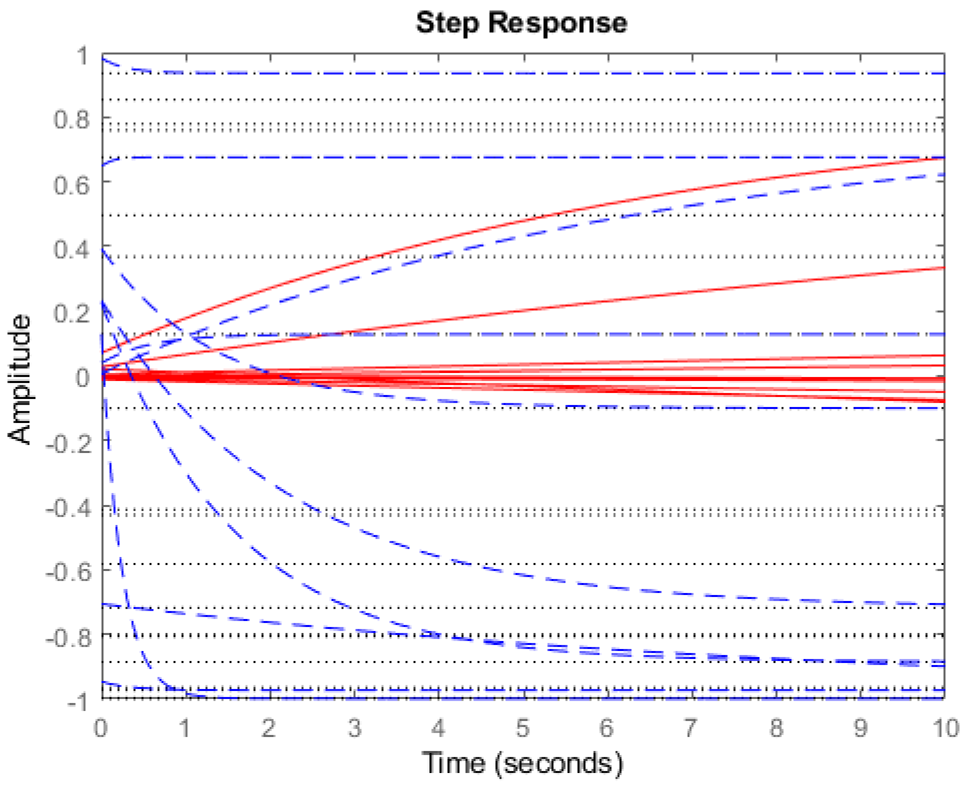
```
A = ultidyn('A',[1 1]);
B = ultidyn('B',[1 1]);
```

Sample 10 instances of each, using a bandwidth limit of 1 rad/sec on `A`, and 20 rad/sec on `B`.

```
Npts = 10;
As = usample(A,Npts,1);
Bs = usample(B,Npts,20);
```

Plot 10-second step responses, for the two sample sets.

```
step(As,'r',Bs,'b--',10)
```



The lower bandwidth limit on the samples of A results in generally slower step responses for those samples.

See Also

`rsampleBlock` | `ucomplex` | `ufind` | `ufrd` | `ultidyn` | `umat` | `ureal` | `usample` | `uss` | `usubs`

Introduced in R2009b

usample

Generate random samples of uncertain variables in a Simulink model

Syntax

```
samples = usample(uvars,N)
```

```
samples = usample(uvars)
```

```
samples = usample(uvars,N,Wmax)
```

Description

This function is for generating random samples of uncertain variables stored in a data structure you obtain from a Simulink model, using `ufind`. To generate random samples from uncertain models or generalized state-space models, use `usample`.

`samples = usample(uvars,N)` generates `N` random samples of the uncertain variables in `uvars`. `uvars` is a structure that lists uncertain variables (`ureal`, `ucomplex` or `ultidyn`) by name. You can automatically obtain `uvars` for a Simulink model that contains Uncertain State Space blocks using `ufind`. `samples` is an *N*-by-1 structure array whose field names and values are the names and sample values of the uncertain variables. Use this syntax, together with `ufind`, to generate random samples for uncertain variables in Simulink models.

`samples = usample(uvars)` is equivalent to `usample(uvars,1)`.

`samples = usample(uvars,N,Wmax)` specifies constraints, as described in `uss/usample`, for sampling uncertain variables of type `ultidyn` in `uvars`.

Examples

Generate random samples of uncertain variables

```
% Create a structure that contains uncertain variables a and % b.
uvars = struct('a',ureal('a',5),'b',ultidyn('b',[2 3],'Bound',7))
```

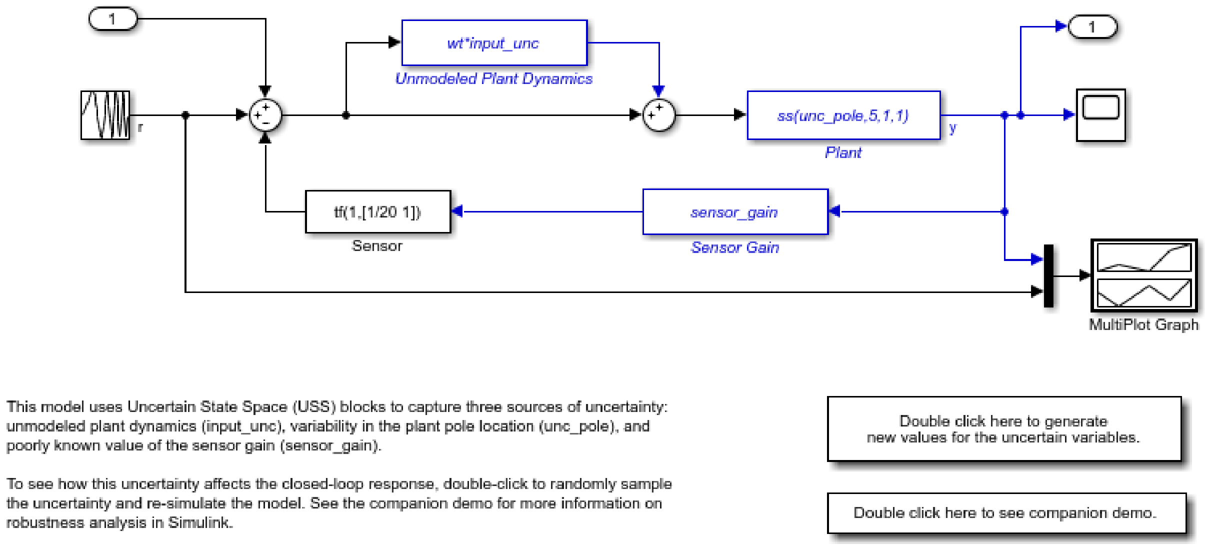
```
% Use usample to generate random values of a and b.
samples = usample(uvars)
```

Sample Uncertain Variables in a Simulink® Model

Generate random samples of uncertain variables in a Simulink® model.

Open the model.

```
open_system('usim_model')
```



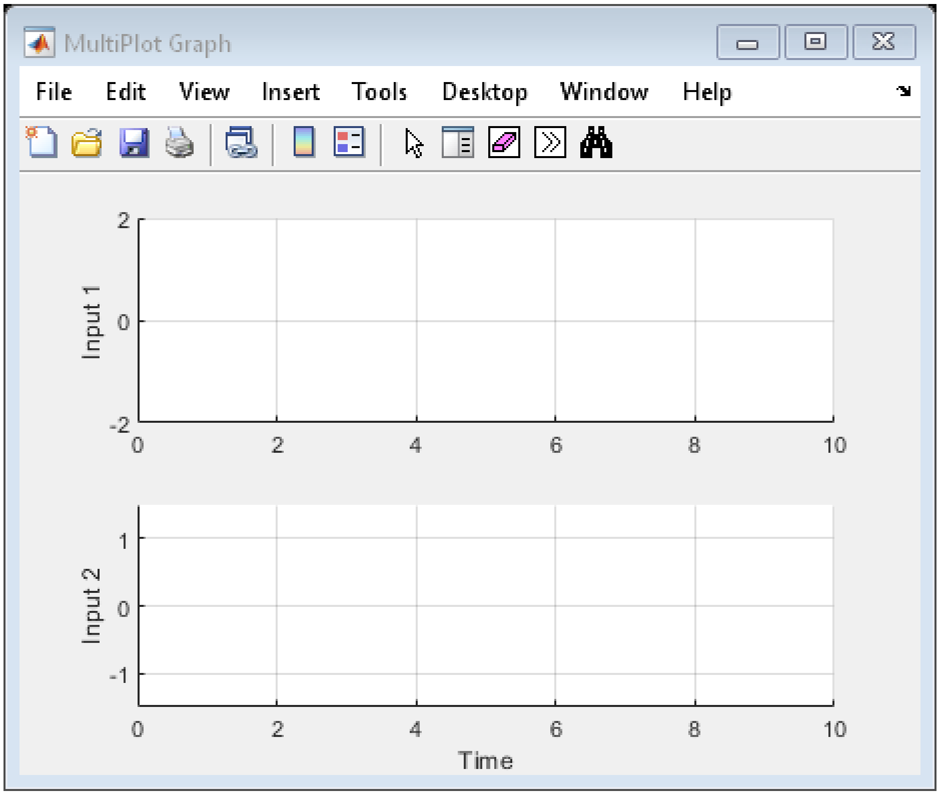
This model uses Uncertain State Space (USS) blocks to capture three sources of uncertainty: unmodeled plant dynamics (input_unc), variability in the plant pole location (unc_pole), and poorly known value of the sensor gain (sensor_gain).

To see how this uncertainty affects the closed-loop response, double-click to randomly sample the uncertainty and re-simulate the model. See the companion demo for more information on robustness analysis in Simulink.

The model contains three Uncertain State Space blocks named Unmodeled Plant Dynamics, Plant, and Sensor Gain. These blocks depend on three uncertain variables named `input_unc`, `unc_pole`, and `sensor_gain`.

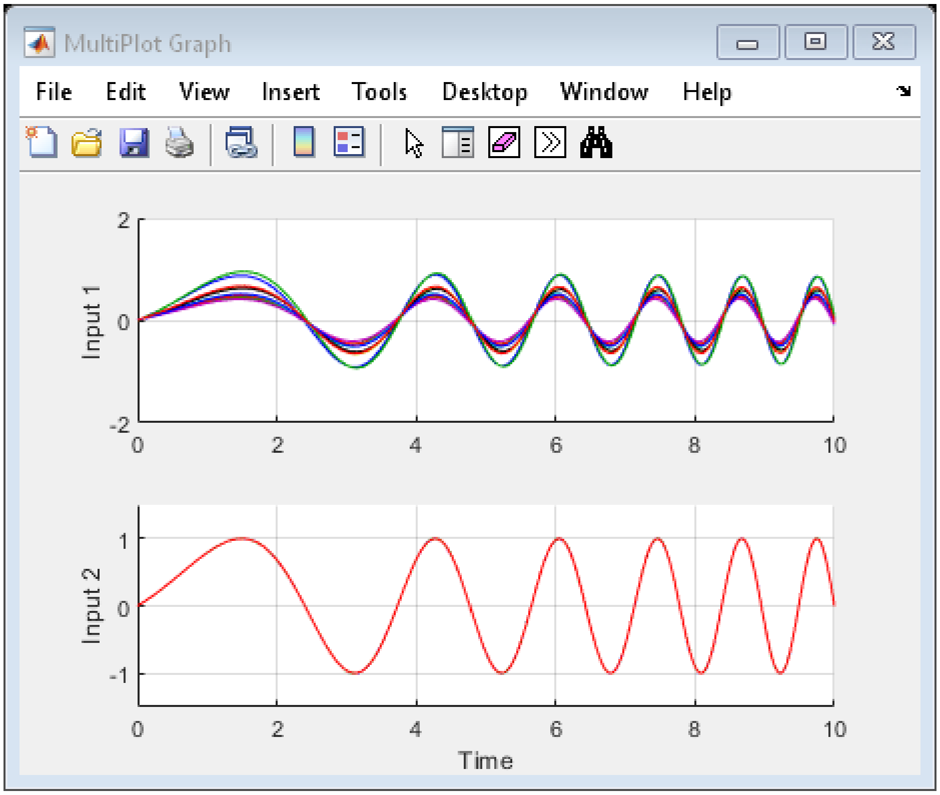
Use `ufind` to find all Uncertain State Space blocks and uncertain variables in the model.

```
uvars = ufind('usim_model');
```



Use `usample` to generate random samples of `input_unc`, `unc_pole`, and `sensor_gain`. Simulate the closed-loop response for each of these random samples.

```
for i=1:10;
   uval = usample(uvars);
    sim('usim_model',10);
end
```



The MultiPlot Graph block displays the simulated responses.

Tutorials

“Vary Uncertainty Values Using Individual Uncertain State Space Blocks”

“Vary Uncertainty Values Across Multiple Uncertain State Space Blocks”

Robustness Analysis in Simulink

How To

“Simulate Uncertainty Effects”

See Also

`ucomplex` | `ufind` | `ufrd` | `ultidyn` | `umat` | `ureal` | `uss` | `usubs`

Introduced before R2006a

usimfill

(Not recommended) Helper function for USS System blocks to set "User-defined Uncertainty" field or state of "Uncertainty value" menu

Compatibility

Note `usimfill` is not recommended. Use `ufind` instead.

Syntax

```
usimfill(ModelName,val)
```

```
usimfill(ModelName,'Uncertainty value','Nominal')
```

```
usimfill(ModelName,'Uncertainty value','User defined')
```

Description

The command `usimfill` allows simple control of some parameters of all USS System blocks in a Simulink model.

`usimfill(ModelName,val)` pushes the character vector `val` into the `Uncertainty value` name field of all USS System blocks in the Simulink model specified by `ModelName`.

`usimfill(ModelName,'Uncertainty value','Nominal')` sets the `Uncertainty value` pulldown menu to `Nominal` for all USS System blocks in the Simulink model specified by `ModelName`. Only a limited number of characters are needed to make this specification, so `usimfill(ModelName,'U','N')` accomplishes the same effect.

`usimfill(ModelName,'Uncertainty value','User defined')` sets the `Uncertainty value` pulldown menu to `User defined` for all USS System blocks in the Simulink model specified by `ModelName`. Only a limited number of characters are needed to make this specification, so `usimfill(ModelName,'U','U')` accomplishes the same effect.

Examples

Open the model file associated with the example.

```
open_system('usim_model');
unc_pole = ureal('unc_pole',-5,'Range',[-10 -4]);
plant = ss(unc_pole,5,1,1);
input_unc = ultidyn('input_unc',[1 1]);
wt = makeweight(0.25,130,2.5);
sensor_gain = ureal('sensor_gain',1,'Range',[0.1 2]);
```

This has three USS System blocks. They are plant with a `ureal` atom named `unc_pole`; `input_unc` which is a `ultidyn` object, and `sensor_gain` which is a `ureal` atom.

Run `usimfill` on the model, filling in the field with the label `'newData'`.

```
usimfill('usim_model','newData');
```

View all of the dialog boxes, and see that `'newData'` has been entered.

Run `usimfill` on the model, changing the `Uncertainty Selection` to `Nominal`.

```
usimfill('usim_model','Uncertainty value','Nominal');
```

Similarly run `usimfill` on the model, changing the `Uncertainty Selection` to `User Specified Uncertainty`.

```
usimfill('usim_model','Uncertainty value','User defined');
```

Now generate a random sample of the uncertain atoms, and run the simulation

```
newData = usimsamp('usim_model',120);
sim('usim_model');
```

See Also

`usample` | `usiminfo` | `usimsamp` | `usubs`

Introduced in R2007a

usiminfo

(Not recommended) Find USS System blocks within specified Simulink model and check for consistency

Compatibility

Note `usiminfo` is not recommended. Use `ufind` instead.

Syntax

```
[cflags,allupaths,allunames,upaths,unames,csumchar]
= usiminfo(sname, silent)
```

Description

The command `usiminfo` returns information regarding the locations of all USS System blocks within a Simulink model and determines if these compatibility conditions are satisfied. It is possible to have uncertain objects of the same name throughout a Simulink model. The helper functions `usimsamp` and `usimfill` assume that these are the same uncertainty. Hence uncertain objects of the same name should have the same object properties and `Uncertainty value` in the USS System pull-down menu. `usiminfo` provides information about the uncertainty in the Simulink diagram `sname`.

The following describes the input and outputs arguments of `usiminfo:`

Input Arguments	Description
`sname`	Simulink diagram name
`silent`	Display inconsistencies between uncertain atoms, when not empty. Default is empty.

Output Arguments	Description
`cflag`	Compatibility flag set to 1 if all uncertainties are consistent, set to 0 if an uncertainty definition(s) is consistent and set to –1 if common uncertainties in different blocks have different `Uncertainty value`.
`allupaths`	Path names of USS System blocks in the model (cell).
`allunames`	Uncertainties names in Simulink model (cell).
`upaths`	Path names associated with each `allunames` entry (cell).
`unames`	Uncertainty names associated with each `allupaths` entry (cell).
`csumchar`	Character array with description of uncertainties and their associated block path names. Empty if there is a conflict with `unames`.

See Also
`usample` | `usimfill` | `usimsamp` | `usubs`

Introduced in R2007a

usimsamp

(Not recommended) Generate random instance of all uncertain atoms present in all USS System blocks of Simulink model

Compatibility

Note `usimsamp` is not recommended. Use `usample` instead.

Syntax

```
sample = usimsamp(ModelName)
```

```
sample = usimsamp(ModelName,BW)
```

Description

The command `usimsamp` samples a Simulink model. Note that if the model contains any USS System blocks, then the model can be interpreted as an uncertain Simulink model. The sample generated by `usimsamp` is a scalar structure, with `fieldnames` corresponding to the uncertain atoms within all of the USS System blocks, and the values are specific random samples of the atoms.

For `ultidyn` atoms, the magnitude of the sampled poles can be limited using an optional second bandwidth argument, `BW`. See `usample` for more information on this parameter.

Examples

Open the model file associated with the example.

```
open_system('usim_model');
```

This has 3 USS System blocks. They are `plant` with a `ureal` atom named `unc_pole`; `input_unc` which is a `ultidyn` object, and `sensor_gain` which is a `ureal` atom.

Run `usimsamp` on the model, yielding a structure as described above.

```
unc_pole = ureal('unc_pole',-5,'Range',[-10 -4]);
plant = ss(unc_pole,5,1,1);
input_unc = ultidyn('input_unc',[1 1]);
wt = makeweight(0.25,130,2.5);
sensor_gain = ureal('sensor_gain',1,'Range',[0.1 2]);
data = usimsamp('usim_model')
data =
      input_unc: [1x1 ss]
    sensor_gain: 0.9935
       unc_pole: -4.1308
```

See Also

`usample` | `usimfill` | `usiminfo` | `usubs`

Introduced in R2007a

uss

Uncertain state-space model

Description

Use `uss` model objects to represent uncertain dynamic systems.

The two dominant forms of model uncertainty are:

- Uncertainty in parameters of the underlying differential equation models (uncertain state-space matrices)
- Frequency-domain uncertainty, which often quantifies model uncertainty by describing absolute or relative uncertainty in the frequency response (uncertain or unmodeled linear dynamics)

`uss` model objects can represent dynamic systems with either or both forms of uncertainty. You can use `uss` to perform robust stability and performance analysis and to test the robustness of controller designs.

Creation

There are several ways to create a `uss` model object, including:

- Use `tf` with one or more uncertain real parameters (`ureal`). For example:

  ```
  p = ureal('p',1);
  usys = tf(p,[1 p]);
  ```

  For another example, see "Transfer Function with Uncertain Coefficients" on page 1-625.
- Use `ss` with uncertain state-space matrices (`umat`). For example:

  ```
  p = ureal('p',1);
  A = [0 3*p; -p p^2];
  B = [0; p];
  C = ones(2);
  ```

```
D = zeros(2,1);
usys = ss(A,B,C,D);
```

  For another example, see “Uncertain State-Space Model” on page 1-627.

- Combine numeric LTI models with uncertain elements using model interconnection commands such as `connect`, `series`, or `parallel`, or model arithmetic operators such as *, +, or -. For example:

  ```
  sys = tf(1,[1 1]);
  p = ureal('p',1);
  D = ultidyn('Delta',[1 1]);
  usys = p*sys*(1 + 0.1*D);
  ```

  For another example, see “System with Uncertain Dynamics” on page 1-628.
- Convert a double array or a numeric LTI model to `uss` form using `usys = uss(sys)`. In this case, the resulting `uss` model object has no uncertain elements. For example:

  ```
  M = tf(1,[1 1 1]);
  usys = uss(M);
  ```
- Use `ucover` to create a `uss` model whose range of possible frequency responses includes all responses in an array of numeric LTI models. The resulting model expresses the range of behaviors as dynamic uncertainty (`ultidyn`).

Properties

`NominalValue` — Nominal value of uncertain model

`ss` model object

Nominal value of the uncertain model, specified as a state-space (`ss`) model object. The state-space model is obtained by setting all the uncertain control design blocks of the uncertain model to their nominal values.

`Uncertainty` — Uncertain elements

structure

Uncertain elements of the model, specified as a structure whose fields are the names of the uncertain blocks, and whose values are the control design blocks themselves. Thus, the values stored in the structure can be `ureal`, `umat`, `ultidyn`, or other uncertain control design blocks. For instance, the following commands create an uncertain model `usys` with two uncertain parameters, `p1` and `p2`.

```
p1 = ureal('p1',1);
p2 = ureal('p2',3);
A = [0 3*p1; -p1 p1^2];
B = [0; p2];
C = ones(2);
D = zeros(2,1);
usys = ss(A,B,C,D);
```

The `Uncertainty` property of `usys` is a structure with two fields, `p1` and `p2`, whose values are the corresponding `ureal` uncertain parameters.

```
usys.Uncertainty
```

```
ans = 

  struct with fields:

    p1: [1×1 ureal]
    p2: [1×1 ureal]
```

You can access or examine each uncertain parameter individually. For example:

```
get(usys.Uncertainty.p1)
```

```
    NominalValue: 1
            Mode: 'PlusMinus'
           Range: [0 2]
       PlusMinus: [-1 1]
      Percentage: [-100 100]
    AutoSimplify: 'basic'
            Name: 'p1'
```

`A, B, C, D, E` — State-space matrices
numeric matrix | uncertain matrix

This property is read-only.

State-space matrices, specified as numeric matrices or uncertain matrices (`umat`). The state-space matrices are evaluated by fixing all dynamic uncertainty blocks (`udyn`, `ultidyn`) to their nominal values.

- `A` — State matrix A, specified as a square matrix or `umat` with as many rows and columns as there are system states.
- `B` — Input-to-state matrix B, specified as a matrix or `umat` with as many rows as there are system states and as many columns as there are system inputs.

- `C` — State-to-output matrix *C*, specified as a matrix or `umat` with as many rows as there are system outputs and as many columns as there are system states.
- `D` — Feedthrough matrix *D*, specified as a matrix or `umat` with as many rows as there are system outputs and as many columns as there are system inputs.
- `E` — *E* matrix for implicit (descriptor) state-space models, specified as a matrix or `umat` of the same dimensions as `A`. By default `E = []`, meaning that the state equation is explicit. To specify an implicit state equation $E\ dx/dt = Ax + Bu$, set this property to a square matrix of the same size as `A`. See `dss` for more information about descriptor state-space models.

`StateName` — State names

`{''}` (default) | character vector | cell array of character vectors

State names, specified as one of these values:

- Character vector — For first-order models
- Cell array of character vectors — For models with two or more states
- `''` — For unnamed states

You can specify `StateName` using a string, such as `"velocity"`, but the state name is stored as a character vector, `'velocity'`.

Example: `'velocity'`

Example: `{'x1','x2'}`

`StateUnit` — State units

`{''}` (default) | character vector | cell array of character vectors

State units, specified as one of these values:

- Character vector — For first-order models
- Cell array of character vectors — For models with two or more states
- `''` — For states without specified units

Use `StateUnit` to keep track of the units each state is expressed in. `StateUnit` has no effect on system behavior.

You can specify `StateUnit` using a string, such as `"mph"`, but the state units are stored as a character vector, `'mph'`.

Example: `'mph'`

Example: `{'rpm','rad/s'}`

`InternalDelay` — Internal delays
scalar | vector

Internal delays, specified as a scalar or vector. For continuous-time models, internal delays are expressed in the time unit specified by the `TimeUnit` property of the model object. For discrete-time models, internal delays are expressed as integer multiples of the sample time `Ts`. For example, `InternalDelay = 3` means a delay of three sampling periods.

You can modify the values of internal delays. However, the number of entries in `InternalDelay` cannot change, because it is a structural property of the model.

Internal delays arise, for example, when closing feedback loops on systems with delays, or when connecting delayed systems in series or parallel. For more information about internal delays, see “Closing Feedback Loops with Time Delays” (Control System Toolbox).

`InputDelay` — Delay at inputs
0 (default) | scalar | vector

Delay at each input, specified as a scalar or a vector. For a system with `Nu` inputs, set `InputDelay` to an `Nu`-by-1 vector. Each entry of this vector is a numerical value that represents the input delay for the corresponding input channel. For continuous-time models, specify input delays in the time unit stored in the `TimeUnit` property of the model object. For discrete-time models, specify input delays in integer multiples of the sample time `Ts`. For example, `InputDelay = 3` means a delay of three sample times.

Set `InputDelay` to a scalar value to apply the same delay to all channels.

`OutputDelay` — Delay at outputs
0 (default) | scalar | vector

Delay at each output, specified as a scalar or a vector. For a system with `Ny` outputs, set `OutputDelay` to an `Ny`-by-1 vector. Each entry of this vector is a numerical value that represents the output delay for the corresponding output channel. For continuous-time models, specify output delays in the time unit stored in the `TimeUnit` property of the model object. For discrete-time models, specify output delays in integer multiples of the sample time `Ts`. For example, `OutputDelay = 3` means a delay of three sample times.

Set `OutputDelay` to a scalar value to apply the same delay to all channels.

`Ts` — Sample time

0 (default) | –1 | positive scalar

Sample time, specified as:

- 0 — For continuous-time models.
- Positive scalar value — For discrete-time models. Specify the sample time in the units given in the `TimeUnit` property of the model.
- –1 — For discrete-time models with unspecified sample time.

Changing this property does not discretize or resample the model. Use `c2d` and `d2c` to convert between continuous-time and discrete-time representations. Use `d2d` to change the sample time of a discrete-time system.

`TimeUnit` — Model time units

`'seconds'` (default) | `'minutes'` | `'milliseconds'` | ...

Model time units, specified as one of these values:

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

You can specify `TimeUnit` using a string, such as `"hours"`, but the time units are stored as a character vector, `'hours'`.

Model properties such as sample time `Ts`, `InputDelay`, `OutputDelay`, and other time delays are expressed in the units specified by `TimeUnit`. Changing this property has no effect on other properties, and therefore changes the overall system behavior. Use `chgTimeUnit` to convert between time units without modifying system behavior.

InputName — Names of input channels
{''} (default) | character vector | cell array of character vectors

Names of input channels, specified as one of these values:

- Character vector — For single-input models
- Cell array of character vectors — For models with two or more inputs
- '' — For inputs without specified names

You can use automatic vector expansion to assign input names for multi-input models. For example, if sys is a two-input model, enter:

```
sys.InputName = 'controls';
```

The input names automatically expand to {'controls(1)';'controls(2)'}.

You can use the shorthand notation u to refer to the InputName property. For example, sys.u is equivalent to sys.InputName.

Input channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

You can specify InputName using a string, such as "voltage", but the input name is stored as a character vector, 'voltage'.

InputUnit — Units of input signals
{''} (default) | character vector | cell array of character vectors

Units of input signals, specified as one of these values:

- Character vector — For single-input models
- Cell array of character vectors — For models with two or more inputs
- '' — For inputs without specified units

Use InputUnit to keep track of the units each input signal is expressed in. InputUnit has no effect on system behavior.

You can specify `InputUnit` using a string, such as `"voltage"`, but the input units are stored as a character vector, `'voltage'`.

Example: `'voltage'`

Example: `{'voltage','rpm'}`

`InputGroup` — Input channel groups
structure with no fields (default) | structure

Input channel groups, specified as a structure where the fields are the group names and the values are the indices of the input channels belonging to the corresponding group. When you use `InputGroup` to assign the input channels of MIMO systems to groups, you can refer to each group by name when you need to access it. For example, suppose you have a five-input model `sys`, where the first three inputs are control inputs and the remaining two inputs represent noise. Assign the control and noise inputs of `sys` to separate groups.

```
sys.InputGroup.controls = [1:3];
sys.InputGroup.noise = [4 5];
```

Use the group name to extract the subsystem from the control inputs to all outputs.

```
sys(:,'controls')
```

Example: `struct('controls',[1:3],'noise',[4 5])`

`OutputName` — Names of output channels
`{''}` (default) | character vector | cell array of character vectors

Names of output channels, specified as one of these values:

- Character vector — For single-output models
- Cell array of character vectors — For models with two or more outputs
- `''` — For outputs without specified names

You can use automatic vector expansion to assign output names for multi-output models. For example, if `sys` is a two-output model, enter:

```
sys.OutputName = 'measurements';
```

The output names automatically expand to
`{'measurements(1)';'measurements(2)'}`.

You can use the shorthand notation `y` to refer to the `OutputName` property. For example, `sys.y` is equivalent to `sys.OutputName`.

Output channel names have several uses, including:

- Identifying channels on model display and plots
- Extracting subsystems of MIMO systems
- Specifying connection points when interconnecting models

You can specify `OutputName` using a string, such as `"rpm"`, but the output name is stored as a character vector, `'rpm'`.

OutputUnit — Units of output signals
`{''}` (default) | character vector | cell array of character vectors

Units of output signals, specified as one of these values:

- Character vector — For single-output models
- Cell array of character vectors — For models with two or more outputs
- `''` — For outputs without specified units

Use `OutputUnit` to keep track of the units each output signal is expressed in. `OutputUnit` has no effect on system behavior.

You can specify `OutputUnit` using a string, such as `"voltage"`, but the output units are stored as a character vector, `'voltage'`.

Example: `'voltage'`

Example: `{'voltage','rpm'}`

OutputGroup — Output channel groups
structure with no fields (default) | structure

Output channel groups, specified as a structure where the fields are the group names and the values are the indices of the output channels belonging to the corresponding group. When you use `OutputGroup` to assign the output channels of MIMO systems to groups, you can refer to each group by name when you need to access it. For example, suppose you have a four-output model `sys`, where the second output is a temperature, and the rest are state measurements. Assign these outputs to separate groups.

```
sys.OutputGroup.temperature = [2];
sys.InputGroup.measurements = [1 3 4];
```

Use the group name to extract the subsystem from all inputs to the measurement outputs.

```
sys('measurements',:)
```

Example: `struct('temperature',[2],'measurement',[1 3 4])`

`Notes` — Text notes about model
`[0×1 string]` (default) | string | cell array of character vector

Text notes about the model, stored as a string or a cell array of character vectors. The property stores whichever of these two data types you provide. For instance, suppose that `sys1` and `sys2` are dynamic system models, and set their `Notes` properties to a string and a character vector, respectively.

```
sys1.Notes = "sys1 has a string.";
sys2.Notes = 'sys2 has a character vector.';
sys1.Notes
sys2.Notes
```

```
ans = 

    "sys1 has a string."


ans = 

    'sys2 has a character vector.'
```

`UserData` — Data associated with model
`[]` (default) | any data type

Data of any kind that you want to associate and store with the model, specified as any MATLAB data type.

`Name` — Model name
`''` (default) | character vector

Model name, stored as a character vector. You can specify `Name` using a string, such as `"DCmotor"`, but the output units are stored as a character vector, `'DCmotor'`.

Example: `'system_1'`

`SamplingGrid` — Sampling grid for model arrays
structure with no fields (default) | structure

Sampling grid for model arrays, specified as a structure. For model arrays that are derived by sampling one or more independent variables, this property tracks the variable values associated with each model in the array. This information appears when you display or plot the model array. Use this information to trace results back to the independent variables.

Set the field names of the data structure to the names of the sampling variables. Set the field values to the sampled variable values associated with each model in the array. All sampling variables should be numeric and scalar valued, and all arrays of sampled values should match the dimensions of the model array.

For example, suppose you create a 11-by-1 array of linear models, `sysarr`, by taking snapshots of a linear time-varying system at times `t = 0:10`. The following code stores the time samples with the linear models.

```
 sysarr.SamplingGrid = struct('time',0:10)
```

Similarly, suppose you create a 6-by-9 model array, `M`, by independently sampling two variables, `zeta` and `w`. The following code attaches the `(zeta,w)` values to `M`.

```
[zeta,w] = ndgrid(<6 values of zeta>,<9 values of w>)
M.SamplingGrid = struct('zeta',zeta,'w',w)
```

When you display `M`, each entry in the array includes the corresponding `zeta` and `w` values.

```
M
```

```
M(:,:,1,1) [zeta=0.3, w=5] =

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


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

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

...
```

For model arrays generated by linearizing a Simulink model at multiple parameter values or operating points, the software populates `SamplingGrid` automatically with the variable values that correspond to each entry in the array. For example, the Simulink Control Design commands `linearize` and `slLinearizer` populate `SamplingGrid` in this way.

Object Functions

Most functions that work on numeric LTI models also work on `uss` models. These include model interconnection functions such as `connect` and `feedback`, and linear analysis functions such as `bode` and `stepinfo`. Some functions that generate plots, such as `bode` and `step`, plot random samples of the uncertain model to give you a sense of the distribution of uncertain dynamics. When you use these commands to return data, however, they operate on the nominal value of the system only.

In addition, you can use functions such as `robstab` and `wcgain` to perform robustness and worst-case analysis of uncertain systems represented by `uss` models. You can also use tuning functions such as `systune` for robust controller tuning.

The following lists contain a representative subset of the functions you can use with `uss` models.

Model Interconnection

feedback	Feedback connection of multiple models
connect	Block diagram interconnections of dynamic systems
series	Series connection of two models
parallel	Parallel connection of two models

Linear Analysis

step	Step response plot of dynamic system; step response data
bode	Bode plot of frequency response, or magnitude and phase data
sigma	Singular values plot of dynamic system
margin	Gain margin, phase margin, and crossover frequencies
diskmargin	Disk-based stability margins of feedback loops

Robustness and Worst-Case Analysis

uss/usample	Generate random samples of uncertain or generalized model
robstab	Robust stability of uncertain system
robgain	Robust performance of uncertain system

wcgain	Worst-case gain of uncertain system
wcsigma	Plot worst-case gain of uncertain system

Control System Design and Tuning

musyn	Robust controller design using mu synthesis
systune	Tune fixed-structure control systems modeled in MATLAB

Examples

Transfer Function with Uncertain Coefficients

Create a second-order transfer function with uncertain natural frequency and damping coefficient.

```
w0 = ureal('w0',10);
zeta = ureal('zeta',0.7,'Range',[0.6,0.8]);

usys = tf(w0^2,[1 2*zeta*w0 w0^2])
```

```
usys =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    w0: Uncertain real, nominal = 10, variability = [-1,1], 5 occurrences
    zeta: Uncertain real, nominal = 0.7, range = [0.6,0.8], 1 occurrences

Type "usys.NominalValue" to see the nominal value, "get(usys)" to see all properties, a
```

`usys` is an uncertain state-space (`uss`) model with two Control Design Blocks. The uncertain real parameter `w0` occurs five times in the transfer function, twice in the numerator and three times in the denominator. To reduce the number of occurrences, you can rewrite the transfer function by dividing numerator and denominator by `w0^2`.

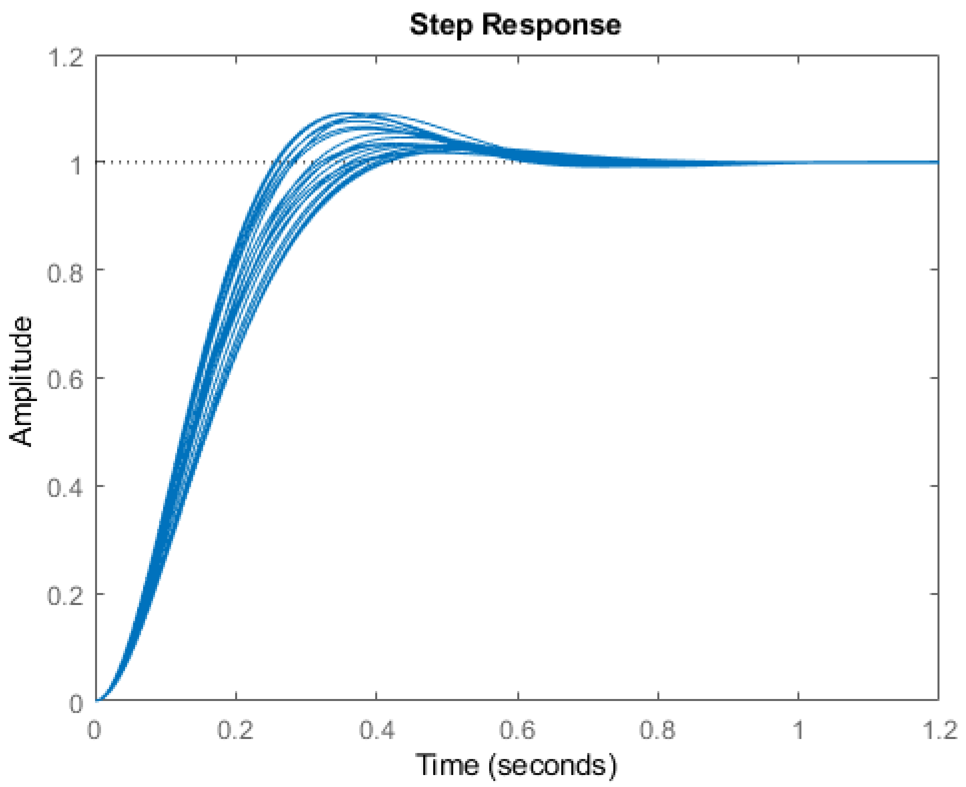
```
usys = tf(1,[1/w0^2 2*zeta/w0 1])
```

```
usys =

  Uncertain continuous-time state-space model with 1 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    w0: Uncertain real, nominal = 10, variability = [-1,1], 3 occurrences
    zeta: Uncertain real, nominal = 0.7, range = [0.6,0.8], 1 occurrences

Type "usys.NominalValue" to see the nominal value, "get(usys)" to see all properties, a
```

In the new formulation, there are only three occurrences of the uncertain parameter `w0`. Reducing the number of occurrences of a Control Design Block in a model can improve the performance of calculations involving the model.

Examine the step response of the system to get a sense of the range of responses that the uncertainty represents.

```
step(usys)
```



When you use linear analysis commands like `step` and `bode` to create response plots of uncertain systems, they automatically plot random samples of the system. While these samples give you a sense of the range of responses that fall within the uncertainty, they do not necessarily include the worst-case response. To analyze worst-case responses of uncertain systems, use `wcgain` or `wcsigma`.

Uncertain State-Space Model

To create an uncertain state-space model, you first use Control Design Blocks to create uncertain elements. Then, use the elements to specify the state-space matrices of the system.

For instance, create three uncertain real parameters and build state-spaces matrices from them.

```
p1 = ureal('p1',10,'Percentage',50);
p2 = ureal('p2',3,'PlusMinus',[-.5 1.2]);
p3 = ureal('p3',0);

A = [-p1 p2; 0 -p1];
B = [-p2; p2+p3];
C = [1 0; 1 1-p3];
D = [0; 0];
```

The matrices constructed with uncertain parameters, `A`, `B`, and `C`, are uncertain matrix (`umat`) objects. Using them as inputs to `ss` results in a 2-output, 1-input, 2-state uncertain system.

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

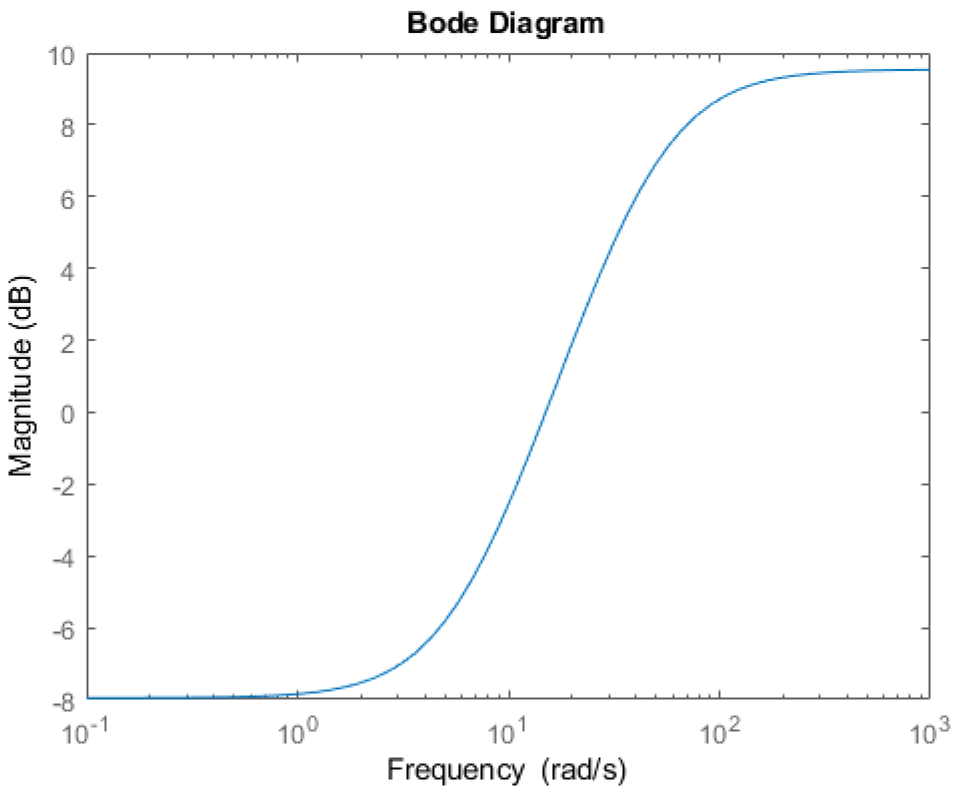
```
sys =

  Uncertain continuous-time state-space model with 2 outputs, 1 inputs, 2 states.
  The model uncertainty consists of the following blocks:
    p1: Uncertain real, nominal = 10, variability = [-50,50]%, 2 occurrences
    p2: Uncertain real, nominal = 3, variability = [-0.5,1.2], 2 occurrences
    p3: Uncertain real, nominal = 0, variability = [-1,1], 2 occurrences

Type "sys.NominalValue" to see the nominal value, "get(sys)" to see all properties, and
```

The display shows that the system includes the three uncertain parameters.

System with Uncertain Dynamics

Create an uncertain system comprising a nominal model with a frequency-dependent amount of uncertainty. You can model such uncertainty using `ultidyn` and a weighting function that represents the frequency profile of the uncertainty. Suppose that at low frequency, below 3 rad/s, the model can vary up to 40% from its nominal value. Around 3 rad/s, the percentage variation starts to increase. The uncertainty crosses 100% at 15 rad/s and reaches 2000% at approximately 1000 rad/s. Create a transfer function with an appropriate frequency profile, `Wunc`, to use as a weighting function that modulates the amount of uncertainty with frequency.

```
Wunc = makeweight(0.40,15,3);
bodemag(Wunc)
```

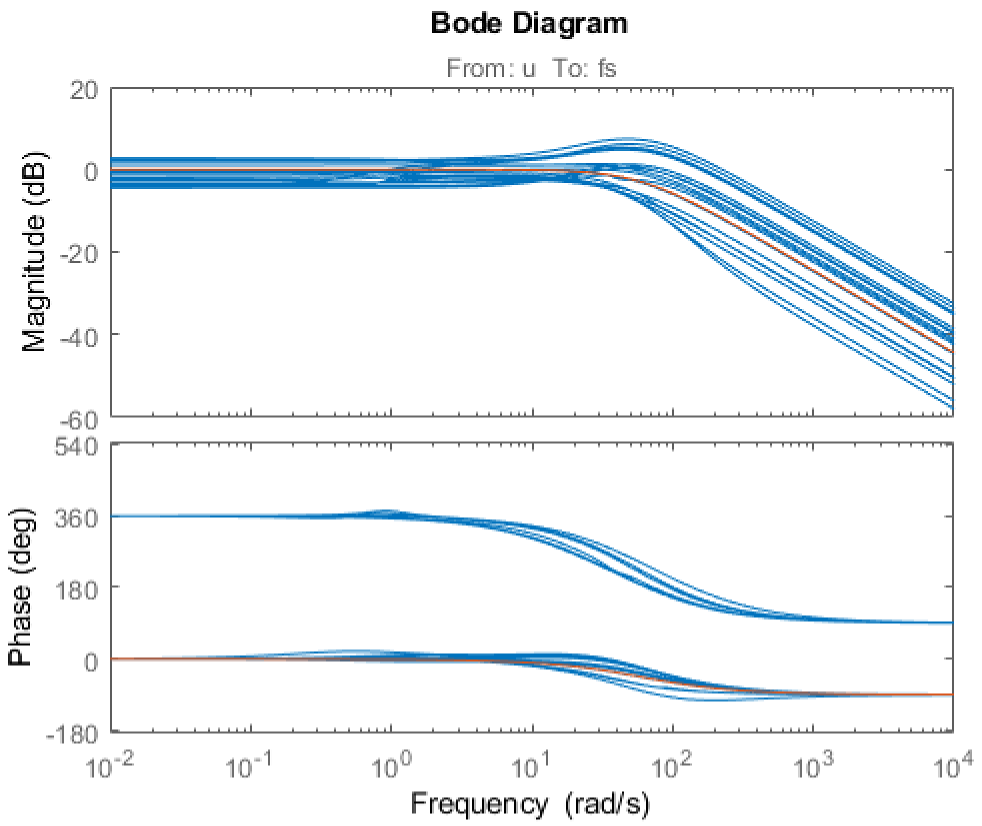


Next, create a transfer function representing the nominal value of the system. For this example, use a transfer function with a single pole at *s* = –60 rad/s. Then, create a `ultidyn` model to represent 1-input, 1-output uncertain dynamics, and add the weighted uncertainty to the nominal transfer function.

```
sysNom = tf(1,[1/60 1]);
unc = ultidyn('unc',[1 1],'SampleStateDim',3); % samples of uncertain dynamics have thr

usys = sysNom*(1 + Wunc*unc);

% Set properties of usys
usys.InputName = 'u';
usys.OutputName = 'fs';
```

Examine random samples of usys to see the effect of the uncertain dynamics.

```
bode(usys,usys.Nominal)
```



Properties of uss Objects

`uss` models, like all model objects, include properties that store dynamics and model metadata. View the properties of an uncertain state-space model.

```
p1 = ureal('p1',10,'Percentage',50);
p2 = ureal('p2',3,'PlusMinus',[-.5 1.2]);
p3 = ureal('p3',0);
```

```
A = [-p1 p2; 0 -p1];
B = [-p2; p2+p3];
C = [1 0; 1 1-p3];
D = [0; 0];
sys = ss(A,B,C,D);     % create uss model

get(sys)
```

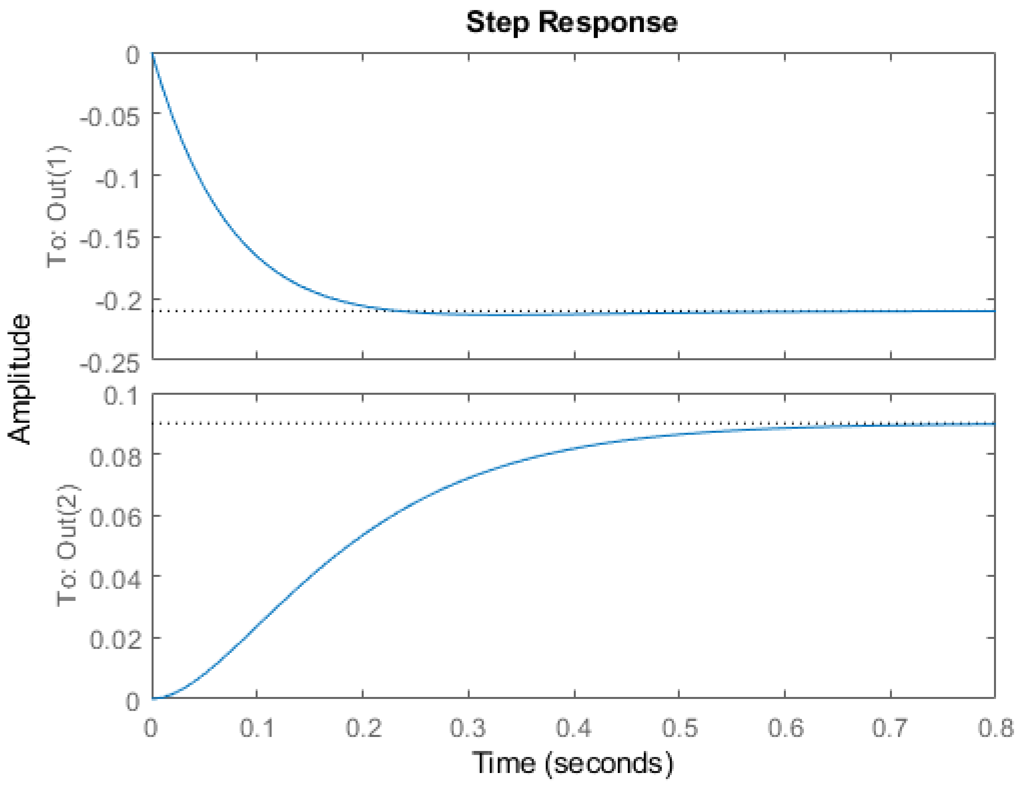
```
     NominalValue: [2x1 ss]
      Uncertainty: [1x1 struct]
                A: [2x2 umat]
                B: [2x1 umat]
                C: [2x2 umat]
                D: [2x1 double]
                E: []
        StateName: {2x1 cell}
        StateUnit: {2x1 cell}
    InternalDelay: [0x1 double]
       InputDelay: 0
      OutputDelay: [2x1 double]
               Ts: 0
         TimeUnit: 'seconds'
        InputName: {''}
        InputUnit: {''}
       InputGroup: [1x1 struct]
       OutputName: {2x1 cell}
       OutputUnit: {2x1 cell}
      OutputGroup: [1x1 struct]
            Notes: [0x1 string]
         UserData: []
             Name: ''
     SamplingGrid: [1x1 struct]
```

Most of the properties behave similarly to how they behave for `ss` model objects. The `NominalValue` property is itself an `ss` model object. You can therefore analyze the nominal value as you would any state-space model. For instance, compute the poles and step response of the nominal system.

```
pole(sys.NominalValue)
```

```
ans = 2×1

   -10
   -10
```

```
step(sys.NominalValue)
```



As with the uncertain matrices (`umat`), the `Uncertainty` property is a structure containing the uncertain elements. You can use this property for direct access to the uncertain elements. For instance, check the `Range` of the uncertain element named `p2` within `sys`.

```
sys.Uncertainty.p2.Range
```

```
ans = 1×2

    2.5000    4.2000
```

Change the uncertainty range of p2 within sys.

```
sys.Uncertainty.p2.Range = [2 4];
```

This command changes only the range of the parameter called p2 in sys. It does not change the variable p2 in the MATLAB workspace.

```
p2.Range
```

```
ans = 1×2

    2.5000    4.2000
```

See Also

ucomplex | ultidyn | umat | ureal | uss/usample

Topics

"Uncertain State-Space Models"

Introduced before R2006a

usubs

Substitute given values for uncertain elements of uncertain objects

Syntax

```
B = usubs(M,ElementName1,value1,ElementName2,value2,...)
B = usubs(M,S)
B = usubs(M,...,'-once')
B = usubs(M,...,'-batch')
```

Description

Use `usubs` to substitute a specific value for an uncertain element of an uncertain model object. The value can itself be uncertain. It needs to be the correct size, but otherwise can be of any class, and can be an array. Hence, the result can be of any class. In this manner, uncertain elements act as symbolic placeholders, for which specific values (which can also contain other placeholders too) can be substituted.

`B = usubs(M,ElementName1,value1,ElementName2,value2,...)` sets the elements in `M`, identified by `ElementName1`, `ElementName2`, etc., to the values in `value1`, `value2`, etc. respectively.

You can also use the character vectors `'NominalValue'` or `'Random'` as any `value` argument. If you do so, the nominal value or a random instance of the uncertain element is used. You can partially specify these character vectors, instead of typing the full expression. For example, you can use `'Nom'` or `'Rand'`.

`B = usubs(M,S)` instantiates the uncertain elements of `M` to the values specified in the structure `S`. The field names of `S` are the names of the uncertain elements to replace. The values are the corresponding replacement values. To provide several replacement values, make `S` a struct array, where each struct contains one set of replacement values. A structure such as `S` typically comes from robustness analysis commands such as `robstab`, `usample`, or `wcgain`.

`B = usubs(M,...,'-once')` performs vectorized substitution in the uncertain model array `M`. Each uncertain element is replaced by a single value, but this value may change across the model array. To specify different substitute values for each model in the array `M`, use:

- A cell array for each `valueN` that causes the uncertain element `ElementNameN` in `M(:,:,k)` to be replaced by `valueN(k)`. For example, if `M` is a 2-by-3 array, then a 2-by-3 cell array `value1` replaces `ElementName1` of the model `M(:,:,k)` with the corresponding `value1(k)`.
- A struct array `S` that specifies one set of substitute values `S(k)` for each model `M(:,:,k)`.

Numeric array formats are also accepted for `value1,value2,...`. For example, `value1` can be a 2-by-3 array of LTI models, a numeric array of size `[size(name1) 2 3]`, or a 2-by-3 matrix when the uncertain element `name1` is scalar-valued. The array sizes of `M`, `S`, `value1,value2,...` must agree along non-singleton dimensions. Scalar expansion takes place along singleton dimensions.

Vectorized substitution (`'-once'`) is the default for model arrays when no substitution method is specified.

`B = usubs(M,...,'-batch')` performs batch substitution in the uncertain model array `M`. Each uncertain element is replaced by an array of values, and the same values are used for all models in `M`. In batch substitution, `B` is a model array of size `[size(M) VS]`, where `VS` is the size of the array of substitute values.

Examples

Evaluate Uncertain Matrix for Multiple Values of Uncertain Parameters

Evaluate an uncertain matrix at several different values of the uncertain parameters of the matrix.

Create an uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b];
```

Evaluate the matrix at four different combinations of values for the uncertain parameters `a` and `b`.

```
B = usubs(M,'a',[1;2;3;4],'b',[10;11;12;13]);
```

This command evaluates M for the four different (a, b) combinations (1,10), (2,11), and so on. Therefore, B is a 1-by-2-by-4 array of numeric values containing the four evaluated values of M.

Evaluate Uncertain Matrix over Grid of Uncertain Parameters

Evaluate an uncertain matrix over a 3-by-4 grid of values of the uncertain parameters of the matrix.

Create a 2-by-2 uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b;0 a*b];
```

Build arrays of values for the uncertain parameters.

```
aval = [1;2;3;4];
bval = [10;20;30];
[as,bs] = ndgrid(aval,bval);
```

This command builds two 4-by-3 grids of values.

Evaluate M over the parameter grids A and B.

```
B = usubs(M,'a',as,'b',bs);
```

This command evaluates M for each four different combination of values (A(k),B(k)). B is a 2-by-2-by-4-by-3 array of numeric values, which is a 4-by-3 array of values of M, i.e., M evaluated over the parameter grids.

Instantiate Uncertain Parameter by Batch Substitution of Parameter for Array of Values

Evaluate an array of uncertain models, substituting an array of values for an uncertain parameter.

Create a 1-by-2 uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b];
```

Replace a by each of the values 1, 2, 3, and 4.

```
Ma = usubs(M,'a',[1;2;3;4]);
```

This command returns a 4-by-1 array of 1-by-2 uncertain matrices that contain one uncertain parameter b.

For each model in the array Ma, evaluate b at 10, 20, and 30.

```
B = usubs(Ma,'b',[10;20;30],'-batch');
```

The '-batch' flag causes usubs to evaluate each model in the array at all three values of b. Thus B is a 4-by-3 array of M values.

The '-batch' syntax here yields the same result as the parameter grid approach used in the previous example:

```
aval = [1;2;3;4];
bval = [10;20;30];
[as,bs] = ndgrid(aval,bval);
B = usubs(M,'a',as,'b',bs);
```

Instantiate Uncertain Parameter Using Different Value for Each Entry in Array

Evaluate an array of uncertain models, substituting a different value for the uncertain parameter in each entry in the array.

Create a 1-by-2 uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b];
```

Replace a by each of the values 1, 2, 3, and 4.

```
Ma = usubs(M,'a',[1;2;3;4]);
```

This command returns a 4-by-1 array of 1-by-2 uncertain matrices that contain one uncertain parameter `b`.

For each model in the array `Ma`, evaluate `b`. Use `b = 10` for the first entry in the array, `b = 20` for the second entry, and so on.

```
B = usubs(Ma,'b',{10;20;30;40},'-once');
```

The `'-once'` flag causes `usubs` to evaluate the first model in the array using the first specified value for `b`, the second model for the second specified value, etc.

Replace Uncertain Parameters with Values Returned by usample

Replace the uncertain parameters in an uncertain models by values specified in struct array form, as returned by `usample`.

This is useful, for example, when you have multiple uncertain models that use the same set of parameters, and you want to evaluate all models at the same parameter values.

Create two uncertain matrices that have the same uncertain parameters, `a` and `b`.

```
a = ureal('a',5);
b = ureal('b',-3);
M1 = [a b];
M2 = [a b;0 a*b];
```

Generate some random samples of `M1`.

```
[M1rand,samples] = usample(M1,5);
```

`M1rand` is an array of five values of `M1`, evaluated at randomly generated values of `a` and `b`. These `a` and `b` values are returned in the struct array `samples`.

Examine the struct array `samples`.

```
samples
```

```
samples=5×2 struct
    a
    b
```

The field names of `samples` correspond to the uncertain parameters of `M1`. The values are the parameter values used to generate `M1rand`. Because `M2` has the same parameters, you can use this structure to evaluate `M2` at the same set of values.

```
M2rand = usubs(M2,samples);
```

This command returns a 1-by-5 array of instantiations of `M2`.

See Also

`gridureal` | `simplify` | `usample`

Introduced before R2006a

wcdiskmargin

Worst-case disk-based stability margins of uncertain feedback loops

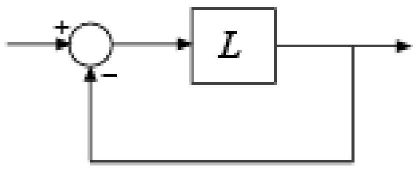
Syntax

```
[wcDM,wcu] = wcdiskmargin(L,'siso')
[wcMM,wcu] = wcdiskmargin(L,'mimo')
[wcMMIO,wcu] = wcdiskmargin(P,C)
___ = wcdiskmargin(___,E)
___ = wcdiskmargin(___,opts)
[___,info] = wcdiskmargin(___)
```

Description

The worst-case disk margin is the smallest disk margin that occurs within a specified uncertainty range. It is also the minimum guaranteed margin over the uncertainty range. `wcdiskmargin` estimates the worst-case disk margins and corresponding worst-case gain and phase margins for both loop-at-a-time and multiloop variations. The function also returns the worst-case perturbation, the combination of uncertain elements that yields the weakest margins.

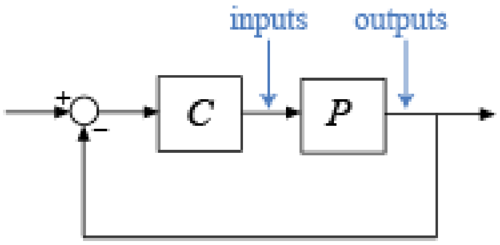
`[wcDM,wcu] = wcdiskmargin(L,'siso')` estimates the worst-case loop-at-a-time disk-based stability margins for the uncertain negative feedback loop `feedback(L,eye(N))`, where `N` is the number of inputs and outputs in `L`.



While `diskmargin` computes stability margins for a nominal model, `wcdiskmargin` computes the worst-case (smallest) disk margin over the modeled uncertainty in `L`. Disk-based margin analysis provides a stronger guarantee of robust stability than the classical gain and phase margins. For general information about disk margins, see “Stability Analysis Using Disk Margins”.

`[wcMM,wcu] = wcdiskmargin(L,'mimo')` estimates the worst-case multiloop disk margins.

`[wcMMIO,wcu] = wcdiskmargin(P,C)` computes the worst-case stability margins when considering independent, concurrent variations at both the plant inputs and plant outputs the negative feedback loop of the following diagram.



`___ = wcdiskmargin(___,E)` specifies an additional eccentricity parameter that varies the shape of the disk region used to compute the stability margins. You can use the eccentricity argument with any of the previous syntaxes.

`___ = wcdiskmargin(___,opts)` specifies additional options for the computation. Use `wcOptions` to create `opts`. You can use `opts` with any of the previous syntaxes.

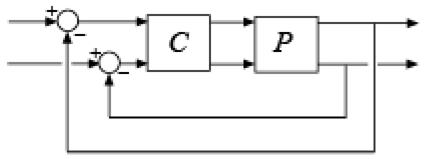
`[___,info] = wcdiskmargin(___)` returns a structure with additional information about the worst-case margins and the perturbations that generate them. You can use this output argument with any of the previous syntaxes.

Examples

Worst-Case Disk Margins for Uncertain MIMO Feedback Loop

Use `wcdiskmargin` to compute worst-case loop-at-a-time and multiloop disk margins. This example illustrates that loop-at-a-time margins can give an overly optimistic assessment of the true robustness of MIMO feedback loops. Margins of individual loops can be sensitive to small perturbations within other loops.

Consider the closed-loop system of the following illustration.



P is a two-input, two-output second-order plant and *C* is a 2x2 static gain. Construct *P* in state-space form, assuming that it has an uncertain parameter and some dynamic uncertainty. Compute the worst-case disk margins at the plant output.

```
alpha = ureal('alpha',10);
a = [-0.2 alpha;-alpha -0.2];
b = eye(2);
c = [1 8;-10 1];
d = zeros(2,2);
P = ss(a,b,c,0);
DEL = ultidyn('DEL',[2 2],'Bound',1e-2);
Pu = P*(eye(2)+DEL);

C = [1 -2;0 1];
L = Pu*C;

[wcDM,wcu] = wcdiskmargin(L,'siso');
```

(To compute the margins at the plant input, use `L = C*Pu`.) Examine the worst-case loop-at-a-time disk margins, returned in the structure array `wcDM`. Each entry in this structure array contains the worst-case stability margins of the corresponding channel.

```
wcDM(1)
```

```
ans = struct with fields:
           GainMargin: [0.2057 4.8617]
          PhaseMargin: [-66.7537 66.7537]
           DiskMargin: 1.3176
           LowerBound: 1.3176
           UpperBound: 1.3205
    CriticalFrequency: 0
```

The result in `wcDM(1)` gives guaranteed stability margins for the specified uncertainty range. As long as the open-loop gain of the first channel changes by a factor no less than about 0.20 and no more than about 4.86, the closed loop remains stable for all `(alpha,DEL)` values within the specified range. Similarly, the closed loop remains stable as long as the phase variation does not exceed about 66.75° in absolute value.

Similarly, `wcDM(2)` shows that in the second feedback channel, the gain can vary by any factor between about 0.16 and about 6.3 or the phase can vary by up to about 72°, and the system remains stable for all uncertainties.

```
wcDM(2)
```

```
ans = struct with fields:
           GainMargin: [0.1583 6.3187]
          PhaseMargin: [-72.0140 72.0140]
           DiskMargin: 1.4535
           LowerBound: 1.4535
           UpperBound: 1.4567
    CriticalFrequency: 0
```

The lower bound returned by `wcdiskmargin` is a theoretical minimum guaranteed worst-case disk margin. The upper bound corresponds to an actual perturbation in the specified uncertainty range that approaches the lower-bound prediction. The output `wcu` contains the values of that perturbation for each feedback channel. `wcu(1)` is the worst combination of `(alpha,DEL)` for the first channel. For `L` evaluated at this combination, the actual disk margins are close to the values in `wcDM(1)`, and the upper bound matches `wcDM(1).UpperBound`.

```
Lwc = usubs(L,wcu(1));
DM = diskmargin(Lwc);
DM(1)
```

```
ans = struct with fields:
     GainMargin: [0.2047 4.8863]
    PhaseMargin: [-66.8678 66.8678]
     DiskMargin: 1.3205
     LowerBound: 1.3205
     UpperBound: 1.3205
      Frequency: 0
```

In practice, uncertainties as well as gain and phase variations affect both channels simultaneously. To estimate the stability margins with respect to such independent and concurrent uncertainty, examine the worst-case multiloop disk margins.

```
[wcMM,wcu] = wcdiskmargin(L,'mimo');
wcMM
```

```
wcMM = struct with fields:
           GainMargin: [0.5981 1.6720]
          PhaseMargin: [-28.2347 28.2347]
           DiskMargin: 0.5030
           LowerBound: 0.5030
           UpperBound: 0.5043
    CriticalFrequency: 0
```

That the guaranteed margins are significantly smaller than when considering one loop at a time. This result occurs because it takes a smaller amount of gain (or phase) variation to destabilize the feedback loop when both channels are subject to concurrent and independent variations. As with the loop-at-a-time margins, the upper bound on the worst-case margin corresponds to the worst-case `(alpha,DEL)` found by `wcdiskmargin` and returned in `wcu`. For the multiloop margin, there is only one set of worst-case uncertainty values.

Finally, compute the multiloop margin against simultaneous variations in gain (or phase) at both the plant inputs and plant outputs. When you allow the gain (or phase) to vary in more places, it becomes easier to destabilize the feedback loop, so the margins get smaller. Thus, the multiloop margin provides the most conservative guarantee of closed-loop stability for any `(alpha,DEL)` in the specified ranges.

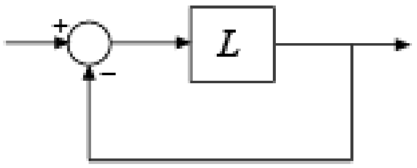
```
[wcMMIO,wcu] = wcdiskmargin(Pu,C);
wcMMIO
```

```
wcMMIO = struct with fields:
           GainMargin: [0.8198 1.2198]
          PhaseMargin: [-11.3106 11.3106]
           DiskMargin: 0.1981
           LowerBound: 0.1981
           UpperBound: 0.1985
    CriticalFrequency: 0
```

Input Arguments

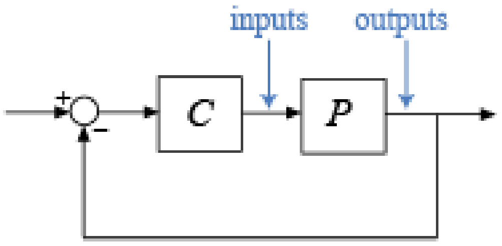
L — Uncertain open-loop response
uncertain model | model array

Uncertain open-loop response, specified as an uncertain model such as a `uss` or `ufrd` model. `L` can be SISO or MIMO, as long as it has the same number of inputs and outputs. `wcdiskmargin` computes the worst-case disk-based stability margins for the negative-feedback closed-loop system `feedback(L,eye(N))`.



To compute the worst-case disk margins of the positive feedback system `feedback(L,eye(N),+1)`, use `wcdiskmargin(-L)`.

When you have a controller `P` and a plant `C`, you can compute the worst-case disk margins for gain (or phase) variations at the plant inputs or outputs, as in the following diagram.



- To compute margins at the plant outputs, set `L = P*C`.
- To compute margins at the plant inputs, set `L = C*P`.
- To consider variations at both the plant inputs and the plant output, use the syntax `[wcMMIO,wcu] = wcdiskmargin(P,C)` instead.

L can be continuous time or discrete time. If L is a generalized state-space model (genss) then wcdiskmargin uses the current value of the tunable control design blocks in L.

If L is a frequency-response data model (such as ufrd), then wcdiskmargin computes the margins at each frequency represented in the model. The function returns the worst-case margins at the frequency with the smallest disk margin.

If L is a model array, then wcdiskmargin computes margins for each model in the array.

P — Plant
uncertain model

Plant, specified as an uncertain model such as a uss or ufrd model. P can be SISO or MIMO, as long as P*C has the same number of inputs and outputs. wcdiskmargin computes the worst-case disk margins for a negative-feedback closed-loop system. To compute the disk margins of the system with positive feedback, use wcdiskmargin(P,-C).

P can be continuous time or discrete time. If P is a generalized state-space model (genss) then wcdiskmargin uses the current value of the tunable control design blocks in P.

If P is a frequency-response data model (such as frd), then wcdiskmargin computes the margins at each frequency represented in the model. The function returns the worst-case margins at the frequency with the smallest disk margin.

C — Controller
dynamic system model

Controller, specified as a dynamic system model. C can be SISO or MIMO, as long as P*C has the same number of inputs and outputs. wcdiskmargin computes the disk-based stability margins for a negative-feedback closed-loop system. To compute the disk margins of the system with positive feedback, use wcdiskmargin(-C,P).

C can be continuous time or discrete time. If C is a generalized state-space model (genss) then wcdiskmargin uses the current value of the tunable control design blocks in C.

If C is a frequency-response data model (such as frd), then wcdiskmargin computes the margins at each frequency represented in the model. The function returns the worst-case margins at the frequency with the smallest disk margin.

E — Eccentricity
0 (default) | real scalar

Eccentricity of uncertainty region used to compute the stability margins, specified as a real scalar value. Use this parameter to vary the shape of the disk region used to model gain and phase variations. Varying the eccentricity parameter yields lower estimates of the true stability margins, letting you infer a larger region of guaranteed stability than that obtained using the default `E` = 0. Some special values of `E` include:

- 0 — Margins based on balanced sensitivity function
- 1 — Margins based on sensitivity function
- –1 — Margins based on complementary sensitivity function

For more detailed information about how the choice of `E` affects the margin computation, see "Stability Analysis Using Disk Margins".

`opts` — Options for margin computation
`wcOptions` object

Options for worst-case computation, specified as an object you create with `wcOptions`. The available options include settings that let you:

- Extract frequency-dependent worst-case margins.
- Examine the sensitivity of the worst-case margins to each uncertain element.
- Improve the results of the worst-case margin calculation by setting certain options for the underlying `mussv` calculation.

For more information about all available options, see `wcOptions`.

Example: `wcOptions('Sensitivity','on','MussvOptions','m3')`

Output Arguments

`wcDM` — Worst-cast disk margins for each feedback channel
structure | structure array

Worst-case disk margins for each feedback channel with all other loops closed, returned as a structure for SISO feedback loops, or an *N*-by-1 structure array for a MIMO loop with *N* feedback channels. The fields of `wcDM(i)` are:

- `GainMargin` — Minimum guaranteed gain margin of the corresponding feedback channel, returned as a vector of the form `[gmin,gmax]`. These values mean that as long as the open-loop gain of the *i*th channel changes by a factor no less than `gmin` and no more than gmax, the closed loop remains stable for all uncertainty values within the ranges specified in L. If the closed-loop system goes unstable for some combination of uncertainty values, then DM(i).GainMargin = [1 1].

- PhaseMargin — Minimum guaranteed phase margin of the corresponding feedback channel, returned as a vector of the form [-pm,pm] in degrees. If the closed-loop system goes unstable for some combination of uncertainty values, then wcDM(i).PhaseMargin = [0 0].
- DiskMargin — Minimum guaranteed disk margin (see “Stability Analysis Using Disk Margins” for the definition and interpretation of the disk margin). If the closed-loop system is unstable for some combination of uncertain-element values, then wcDM(i).DiskMargin = 0.
- LowerBound — Lower bound on worst-case disk margin. This value is the same as DiskMargin.
- UpperBound — Upper bound on worst-case disk margin. This value is the disk margin obtained for the worst perturbation found by wcdiskmargin, returned as wcu(i). The actual worst-case disk margin is no better than this value.
- CriticalFrequency — Frequency at which the disk margin for the worst perturbation wcu(i) is weakest, as a function of frequency. This value is in rad/TimeUnit, where TimeUnit is the TimeUnit property of L.

When L = P*C is the open-loop response of a system comprising a controller and plant with unit negative feedback in each channel, wcDM contains the stability margins for variations at the plant outputs. To compute the stability margins for variations at the plant inputs, use L = C*P. To compute the stability margins for simultaneous, independent variations at both the plant inputs and outputs, use wcMMIO = wcdiskmargin(P,C).

When L is a model array, wcDM has additional dimensions corresponding to the array dimensions of L. For instance, if L is a 1-by-3 array of two-input, two-output models, then wcDM is a 2-by-3 structure array. wcDM(j,k) contains the margins for the jth feedback channel of the kth model in the array.

wcu — Worst-case perturbation
structure array | structure

Worst case perturbation of uncertain elements, returned as:

- A structure array of dimensions *N*-by-1 for loop-at-a-time margins, where *N* is the number of feedback channels

- A scalar structure for multiloop margins

The lower bound returned by `wcdiskmargin` is a theoretical minimum guaranteed worst-case disk margin. The upper bound corresponds to an actual perturbation in the specified uncertainty range that approaches the lower-bound prediction. `wcu` contains the values of that perturbation. For example, if the input system includes uncertain elements `M` and `delta`, then `wcu.M` and `wcu.delta` contain the worst perturbations found by `wcdiskmargin`. It is possible that a worse perturbation exists, but no perturbation can yield a worse margin than the lower bound returned by `wcdiskmargin`.

Use `usubs` to substitute these values for the uncertain elements in the input system, to obtain the dynamic system that has the worst-case disk margin.

`wcMM` — Worst-case multiloop disk margins

structure

Worst-case multiloop disk margins, returned as a structure. The gain (or phase) margins quantify how much gain variation (or phase variation) the system can tolerate in all feedback channels at once while remaining stable. Thus, `MM` is a single structure regardless of the number of feedback channels in the system. (For SISO systems, `MM = DM`.) The fields of `MM` are:

- `GainMargin` — Minimum guaranteed multiloop gain margin, returned as a vector of the form `[gmin,gmax]`. These values mean that as long as the gain in all loop channels changes by a factor no less than `gmin` and no more than `gmax`, the closed loop remains stable for all uncertainty values within the ranges specified in `L`. If the closed-loop system goes unstable for some combination of uncertainty values, then `wcMM.GainMargin = [1 1]`.
- `PhaseMargin` — Minimum guaranteed multiloop phase margin, returned as a vector of the form `[-pm,pm]` in degrees. If the closed-loop system goes unstable for some combination of uncertainty values, then `wcMM.PhaseMargin = [0 0]`.
- `DiskMargin` — Minimum guaranteed disk margin (see “Stability Analysis Using Disk Margins” for the definition and interpretation of the disk margin). If the closed-loop system is unstable for some combination of uncertain-element values, then `wcMM.DiskMargin = 0`.
- `LowerBound` — Lower bound on worst-case disk margin. This value is the same as `DiskMargin`.
- `UpperBound` — Upper bound on worst-case disk margin. This value is the disk margin obtained for the worst perturbation found by `wcdiskmargin`, returned as `wcu`. The actual worst-case multiloop disk margin is no better than this value.

- `CriticalFrequency` — Frequency at which the disk margin for the worst perturbation `wcu` is weakest, as a function of frequency. This value is in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `L`.

When `L = P*C` is the open-loop response of a system comprising a controller and plant with unit negative feedback in each channel, `MM` contains the stability margins for variations at the plant outputs. To compute the stability margins for variations at the plant inputs, use `L = C*P`. To compute the stability margins for simultaneous, independent variations at both the plant inputs and outputs, use `wcMMIO = wcdiskmargin(P,C)`.

When `L` is a model array, `MM` is a structure array with one entry for each model in `L`.

`wcMMIO` — Worst-case disk margins for independent variations in all input and output channels

structure

Worst-case disk margins for independent variations in all input and output channels of the plant `P`, returned as a structure having the same fields as `wcMM`.

`info` — Additional information about worst-case values

structure

Additional information about the worst-case values, returned as a structure with the following fields:

Field	Description
`Model`	Index of the model that has the smallest disk margin, when `L` is an array of models.

Field	Description
`Frequency`	Frequency points at which `wcdiskmargin` returns the minimum guaranteed margins, returned as a vector. • If the `'VaryFrequency'` option of `wcOptions` is `'off'`, then `info.Frequency` is the critical frequency, the frequency at which the worst-case disk margin occurs. If the largest lower bound and the smallest upper bound on the worst-case disk margin occur at different frequencies, then `info.Frequency` is a vector containing these two frequencies. • If the `'VaryFrequency'` option of `wcOptions` is `'on'`, then `info.Frequency` contains the frequencies selected by `wcdiskmargin`. These frequencies are guaranteed to include the frequency at which the worst-case disk margin occurs. The `'VaryFrequency'` option is meaningful only for `uss` and `genss` models. `wcdiskmargin` ignores the option for `ufrd` and `genfrd` models.
`Bounds`	Lower and upper bounds on the actual worst-case disk margin of the model, returned as an array. `info.Bounds(:,1)` contains the lower bound at each corresponding frequency in `info.Frequency`, and `info.Bounds(:,2)` contains the corresponding upper bounds.

Field	Description
`WorstPerturbation`	Worst perturbations at each frequency point in `info.Frequency`, returned as a structure array. Here, worst refers to the perturbations that cause the smallest disk margin at a particular frequency. The fields of `info.WorstPerturbation` are the names of the uncertain elements in the input model. Each field contains the worst value of the corresponding element at each frequency. For example, if `L` includes an uncertain parameter `p` and SISO uncertain dynamics `delta`, then `info.WorstPerturbation.p` is a collection of numeric values and `info.WorstPerturbation.delta` is a collection of SISO state-space models.
`Sensitivity`	Sensitivity of the worst-case disk margin to each uncertain element, returned as a structure when the `'Sensitivity'` option of `wcOptions` is `'on'`. The fields of `info.Sensitivity` are the names of the uncertain elements in the input model. Each field contains a percentage that measures how much the uncertainty in the corresponding element affects the worst disk margin. For example, if `info.Sensitivity.p` is 50, then a given fractional change in the uncertainty range of `p` causes half as much fractional change in the worst disk margin. If the `'Sensitivity'` option of `wcOptions` is off (the default setting), then `info.Sensitivity` is `NaN`.

Tips

- `wcdiskmargin` assumes negative feedback. To compute the worst-case disk margins of a positive feedback system, use `wcdiskmargin(-L)` or `wcdiskmargin(P,-C)`.

Algorithms

`wcdiskmargin` models gain (and phase) variation as `ucomplex` uncertainty, combines it with the specified plant uncertainty, and uses `mussv` to compute the worst-case disk margins and perturbation. This generalizes the `diskmargin` algorithm to feedback loops with uncertainty. For more information about disk-margin computation and interpretation, see “Stability Analysis Using Disk Margins”.

See Also

`diskmargin` | `wcOptions` | `wcgain`

Topics

“Stability Analysis Using Disk Margins”

Introduced in R2018b

wcgain

Worst-case gain of uncertain system

Syntax

```
[wcg,wcu] = wcgain(usys)
[wcg,wcu] = wcgain(usys,w)
[wcg,wcu] = wcgain(___,opts)
[wcg,wcu,info] = wcgain(___)
```

Description

`[wcg,wcu] = wcgain(usys)` calculates the worst-case peak gain of the uncertain system `usys`. Peak gain refers to the maximum gain over frequency (H_∞ norm). For multi-input, multi-output (MIMO) systems, gain refers to the largest singular value of the frequency response matrix. (See `sigma` for more information about singular values.) The structure `wcg` contains upper and lower bounds on the worst-case gain and the critical frequency at which the lower bound peaks. (See "Worst-Case Gain" on page 1-667.) The structure `wcu` contains the values of the uncertain elements of `usys` that cause the worst-case peak gain.

`[wcg,wcu] = wcgain(usys,w)` restricts worst-case computation to the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `wcgain` returns the worst-case gain in the interval between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then `wcgain` calculates the worst-case gain at the specified frequencies only, and returns the worst of those gains.

`[wcg,wcu] = wcgain(___,opts)` specifies additional options for the computation. Use `wcOptions` to create `opts`. You can use this syntax with any of the previous input-argument combinations.

`[wcg,wcu,info] = wcgain(___)` returns a structure with additional information about the worst-case gains and the perturbations that generate them. See `info` for details about this structure. You can use this syntax with any of the previous input-argument combinations.

Examples

Worst-Case Peak Gain of Closed-Loop System

Consider a control system whose plant is nominally an integrator with some additive dynamic uncertainty. Create a model of the plant.

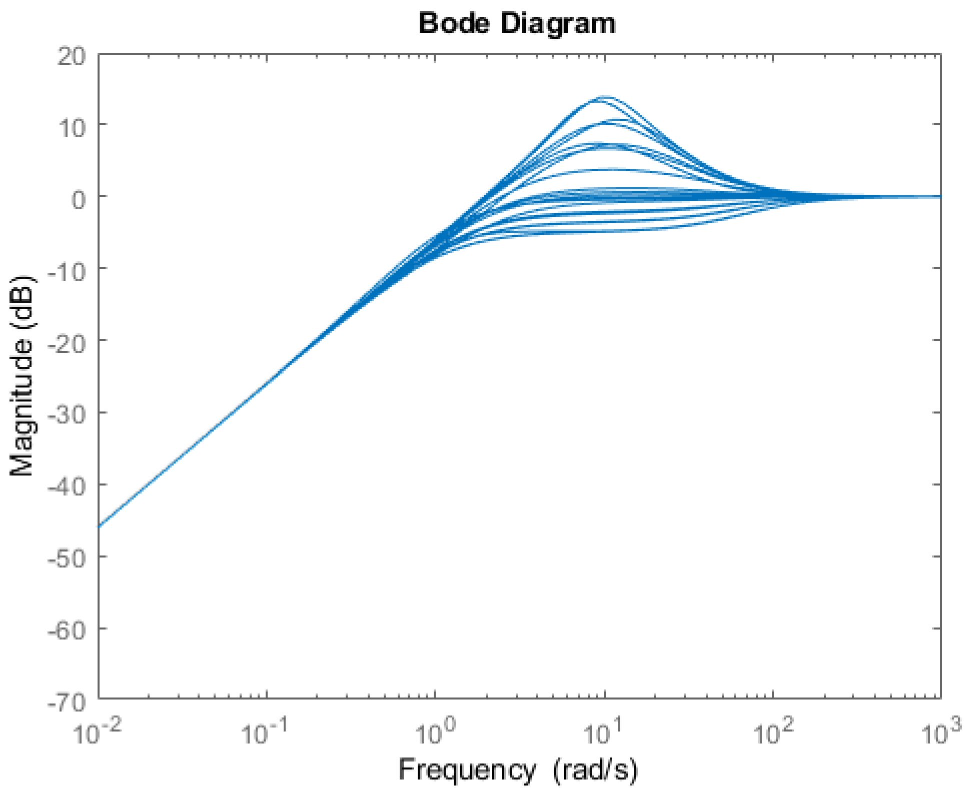
```
delta = ultidyn('delta',[1 1],'bound',0.4);
G = tf(1,[1 0]) + delta;
```

Create a PD controller for the model. Suppose you want to examine the worst-case disturbance rejection performance. Build the closed-loop sensitivity function to examine the worst-case gain of a disturbance at the plant input.

```
C = pid(2,0,-0.04,0.02);
S = feedback(1,G*C);
```

Because of the uncertainty, the frequency response of this transfer function falls within some envelope. The frequency-response magnitude of a few samples of the system gives a sense of that envelope.

```
bodemag(S)
```



Each sample has a different peak gain. Find the highest peak-gain value within the envelope and the corresponding values for the uncertain elements.

```
[wcg,wcu] = wcgain(S);
wcg
```

```
wcg = struct with fields:
           LowerBound: 5.1036
           UpperBound: 5.1140
    CriticalFrequency: 10.7241
```

The `LowerBound` and `UpperBound` fields of `wcg` show that the worst-case peak gain is around 5.1. This gain occurs at the critical frequency around 10.6 rad/s.

The output `wcu` is a structure that contains the perturbation to `delta` that causes the worst-case gain. Confirm the result by substituting this value into the sensitivity function.

```
Swc = usubs(S,wcu);
getPeakGain(Swc)
```

```
ans = 5.1037
```

Because the system has dynamic uncertainty `delta` with gain not exceeding 0.4, the worst-case value of `delta` should be a system with peak gain of 0.4. Confirm this result.

```
getPeakGain(wcu.delta)
```

```
ans = 0.4000
```

Worst-Case Gain at Frequencies in a Range

Consider a model of a control system containing uncertain elements.

```
k = ureal('k',10,'Percent',40);
delta = ultidyn('delta',[1 1]);
G = tf(18,[1 1.8 k]) * (1 + 0.5*delta);
C = pid(2.3,3,0.38,0.001);
CL = feedback(G*C,1);
```

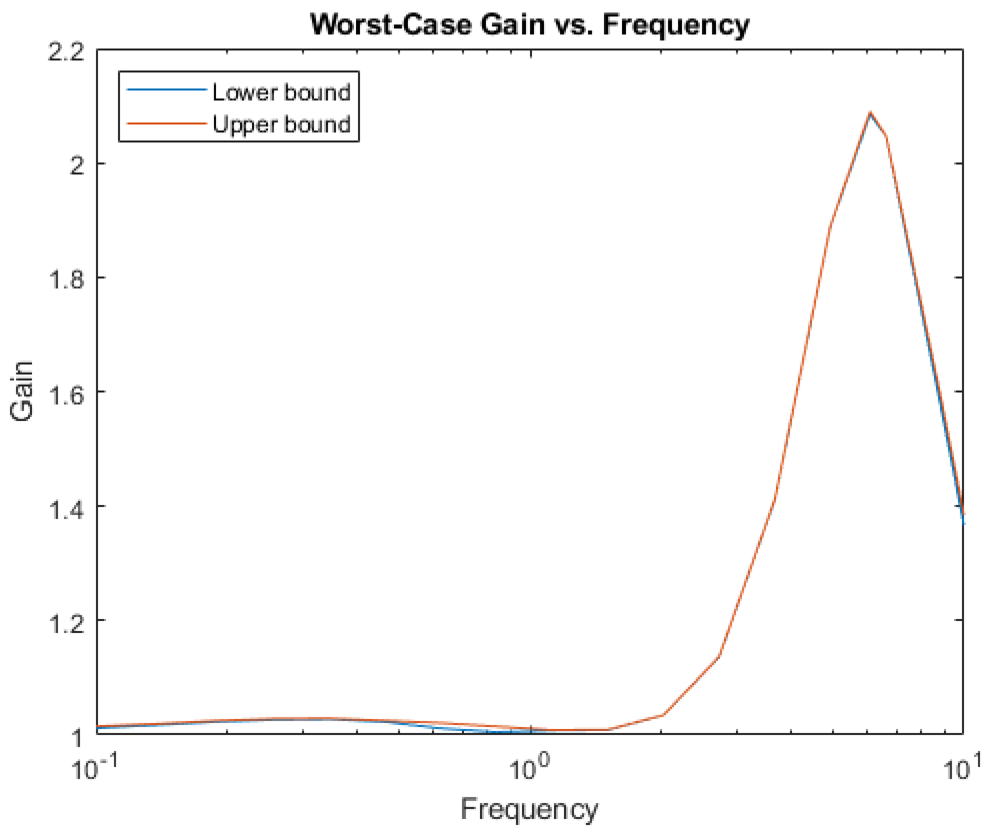
By default, `wcgain` returns only the worst-case peak gain over all frequencies. To obtain worst-case gain values at multiple frequencies, use the `'VaryFrequency'` option of `wcOptions`. For example, compute the highest possible gain of the system at frequency points between 0.1 and 10 rad/s.

```
opts = wcOptions('VaryFrequency','on');
[wcg1,wcu1,info1] = wcgain(CL,{0.1,10},opts);
info1
```

```
info1 = struct with fields:
                 Model: 1
             Frequency: [19x1 double]
                Bounds: [19x2 double]
     WorstPerturbation: [19x1 struct]
           Sensitivity: [1x1 struct]
    BadUncertainValues: [19x1 struct]
            ArrayIndex: 1
```

`wcgain` returns the vector of frequencies in the `info` output, in the `Frequencies` field. `info.Bounds` contains the upper and lower bounds on the worst-case gain at each frequency. Use these values to plot the frequency dependence of the worst-case gain.

```
semilogx(info1.Frequency,info1.Bounds)
title('Worst-Case Gain vs. Frequency')
ylabel('Gain')
xlabel('Frequency')
legend('Lower bound','Upper bound','Location','northwest')
```



The curve shows the high-gain envelope for all systems within the uncertainty ranges of `CL`. You can also use `wcsigma` to plot this envelope along with samples of the system.

When you use the `'VaryFrequency'` option, `wcgain` chooses frequency points automatically. The frequencies it selects are guaranteed to include the frequency at which the worst-case gain is highest (within the specified range). Display the returned frequency values to confirm that they include the critical frequency.

```
info1.Frequency
```

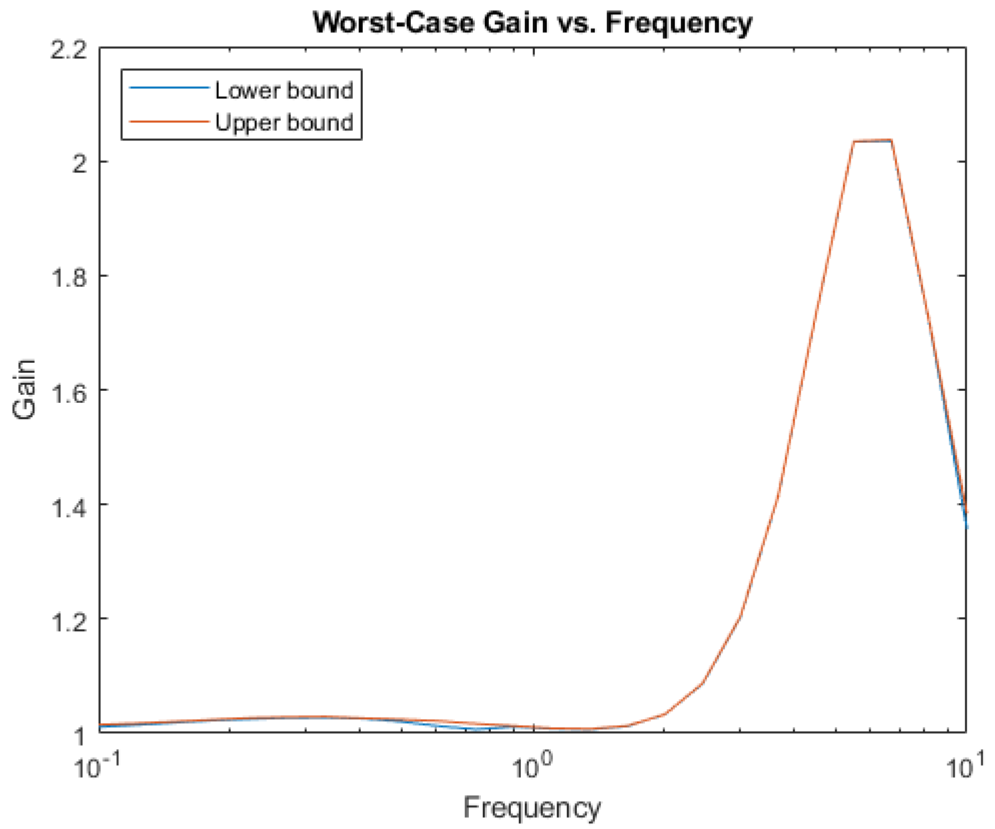
```
ans = 19×1

    0.1000
    0.1061
    0.1425
    0.1914
    0.2572
    0.3455
    0.4642
    0.6236
    0.8377
    1.1253
      ⋮
```

```
wcg1.CriticalFrequency
```

```
ans = 6.0749
```

Alternatively, instead of using `'VaryFrequency'`, you can specify particular frequencies at which to compute the worst-case gains. `info.Bounds` contains the worst-case gains at all specified frequencies.

```
w = logspace(-1,1,24);
[wcg2,wcu2,info2] = wcgain(CL,w);
semilogx(w,info2.Bounds)
title('Worst-Case Gain vs. Frequency')
ylabel('Gain')
xlabel('Frequency')
legend('Lower bound','Upper bound','Location','northwest')
```



When you provide the frequency grid in this way, the results are not guaranteed to include the overall worst-case gain, which might fall between specified frequency points. To see this, examine `wcg1` and `wcg2`, which contain the bounds for the two approaches.

```
wcg1
```

```
wcg1 = struct with fields:
           LowerBound: 2.0848
           UpperBound: 2.0897
    CriticalFrequency: 6.0749
```

```
wcg2
```

```
wcg2 = struct with fields:
           LowerBound: 2.0349
           UpperBound: 2.0370
    CriticalFrequency: 6.7002
```

`wcg1`, computed using `VaryFrequency`, finds a higher peak gain than the specified frequency grid.

Determine Effect of Uncertainty Range on Worst-Case Response

Consider a feedback loop with a first-order plant and a PI controller. The time constant of the plant is uncertain, and the feedback loop accounts for unmodeled dynamic uncertainty. Compute the worst-case gain of the sensitivity function `Si` at the plant inputs. Use the `'Sensitivity'` option of `wcOptions` to compute how sensitive this worst-case gain is to each uncertain element.

```
% Create uncertain system and controller
delta = ultidyn('delta',[1 1]);
tau = ureal('tau',5,'range',[4 6]);
P = tf(1,[tau 1])*(1+0.25*delta);
C = pid(4,4);

opt = wcOptions('Sensitivity','on');
Si = inv(1 + C*P);
[wcg,~,info] = wcgain(Si,opt);
```

The `Sensitivity` field of the `info` output structure includes data that reflects how much the maximum gain of the input sensitivity function changes with each uncertain element.

```
info.Sensitivity
```

```
ans = struct with fields:
    delta: 44
      tau: 9
```

This result tells you that if the uncertainty range of delta increases by 10%, the peak input sensitivity increases by about 4.4%. Similarly, a 10% increase in the uncertainty range of tau causes about a 0.9% increase in the peak input sensitivity.

Improve Worst-Case Perturbation

Specifying certain options for the structured-singular-value computation that underlies the worst-gain computation can yield better results in some cases. For example, consider a sample plant and controller.

```
load(fullfile(matlabroot,'examples','robust','wcgExampleData.mat'))
```

This command loads `gPlant`, a MIMO plant with 10 outputs, 8 inputs, and 11 uncertain elements. It also loads `Kmu`, a state-space controller model. Form a closed-loop system with these models, and examine the worst-case gain.

```
CL = lft(gPlant,Kmu);
[wcg,wcu] = wcgain(CL);
wcg
```

```
wcg = struct with fields:
           LowerBound: 10.8207
           UpperBound: 11.2135
    CriticalFrequency: 6.6782
```

There is a large difference between the lower and upper bounds on the worst-case gain. To get a better estimate of the actual worst-case gain, increase the number of restarts that `wcgain` uses for computing of the worst-case perturbation and associated lower bound. Doing so can result in a tighter lower bound. This option does not affect the upper-bound calculation.

```
opt = wcOptions('MussvOptions','m3');
[wcg,wcu] = wcgain(CL,opt);
wcg
```

```
wcg = struct with fields:
           LowerBound: 10.8207
           UpperBound: 11.2135
    CriticalFrequency: 6.6782
```

The difference between the lower bound and upper bound on the worst-case gain is much smaller. The critical frequency is different as well.

Input Arguments

`usys` — Dynamic system with uncertainty

`uss` | `ufrd` | `genss` | `genfrd`

Dynamic system with uncertainty, specified as a `uss`, `ufrd`, `genss`, or `genfrd` model that contains uncertain elements. For `genss` or `genfrd` models, `wcgain` uses the current value of any tunable blocks and folds them into the known (not uncertain) part of the model.

`usys` can also be an array of uncertain models. In that case, `wcgain` returns the worst-case gain across all models in the array, and the `info` output contains the index of the corresponding model.

`w` — Frequencies

`{wmin,wmax}` | vector

Frequencies at which to compute worst-case gains, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function returns the worst-case gain in the interval between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function computes the worst-case gain at each specified frequency.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

`opts` — Options for margin computation

`wcOptions` object

Options for worst-case computation, specified as an object you create with `wcOptions`. The available options include settings that let you:

- Extract frequency-dependent worst-case gains (see “Worst-Case Gain” on page 1-667).
- Examine the sensitivity of the worst-case gain to each uncertain element.
- Improve the results of the worst-case gain calculation by setting certain options for the underlying `mussv` calculation. For an example, see “Improve Worst-Case Perturbation” on page 1-662.

For more information about all available options, see `wcOptions`.

Example: `wcOptions('Sensitivity','on','MussvOptions','m3')`

Output Arguments

`wcg` — Worst-case peak gain and critical frequency
structure

Worst-case peak gain and critical frequency, returned as a structure containing the following fields:

Field	Description
`LowerBound`	Lower bound on the actual worst-case peak gain of the model, returned as a scalar value. This value is the peak gain corresponding to the worst-case perturbation `wcu`. The exact worst-case peak gain is guaranteed to be no smaller than `LowerBound`.
`UpperBound`	Upper bound on the actual worst-case peak gain of the model, returned as a scalar value. The exact worst-case peak gain is guaranteed to be no larger than `UpperBound`. When you specify a frequency grid as a vector `w`, the guarantee only applies at the specified frequencies.
`CriticalFrequency`	Frequency at which the worst-case peak gain occurs, in rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of `usys`.

`wcu` — Worst-case perturbation
structure

Worst case perturbation of uncertain elements, returned as a structure whose fields are the names of the uncertain elements of `usys`. Each field contains the actual value of the corresponding uncertain element of `usys` when the worst-case peak gain occurs. For example, if `usys` includes an uncertain matrix `M` and SISO uncertain dynamics `delta`, then `wcu.M` is a numeric matrix and `wcu.delta` is a SISO state-space model.

Use `usubs(usys,wcu)` to substitute these values for the uncertain elements in `usys`, to obtain the dynamic system that has the worst-case peak gain.

`info` — Additional information about worst-case values

structure

Additional information about the worst-case values, returned as a structure with the following fields:

Field	Description
`Model`	Index of the model that has the largest worst-case peak gain, when `usys` is an array of models.
`Frequency`	Frequency points at which `wcgain` returns the worst-case gain, returned as a vector. • If the `'VaryFrequency'` option of `wcOptions` is `'off'`, then `info.Frequency` is the critical frequency, the frequency at which the worst-case peak gain occurs. If the largest lower bound and the smallest upper bound on the worst-case gain occur at different frequencies, then `info.Frequency` is a vector containing these two frequencies. • If the `'VaryFrequency'` option of `wcOptions` is `'on'`, then `info.Frequency` contains the frequencies selected by `wcgain`. These frequencies are guaranteed to include the frequency at which the worst-case peak gain occurs. • If you specify a vector of frequencies `w` at which to compute the worst-case gains, then `info.Frequency = w`. When you specify a frequency vector, these frequencies are not guaranteed to include the frequency at which the worst-case peak gain occurs. The `'VaryFrequency'` option is meaningful only for `uss` and `genss` models. `wcgain` ignores the option for `ufrd` and `genfrd` models.

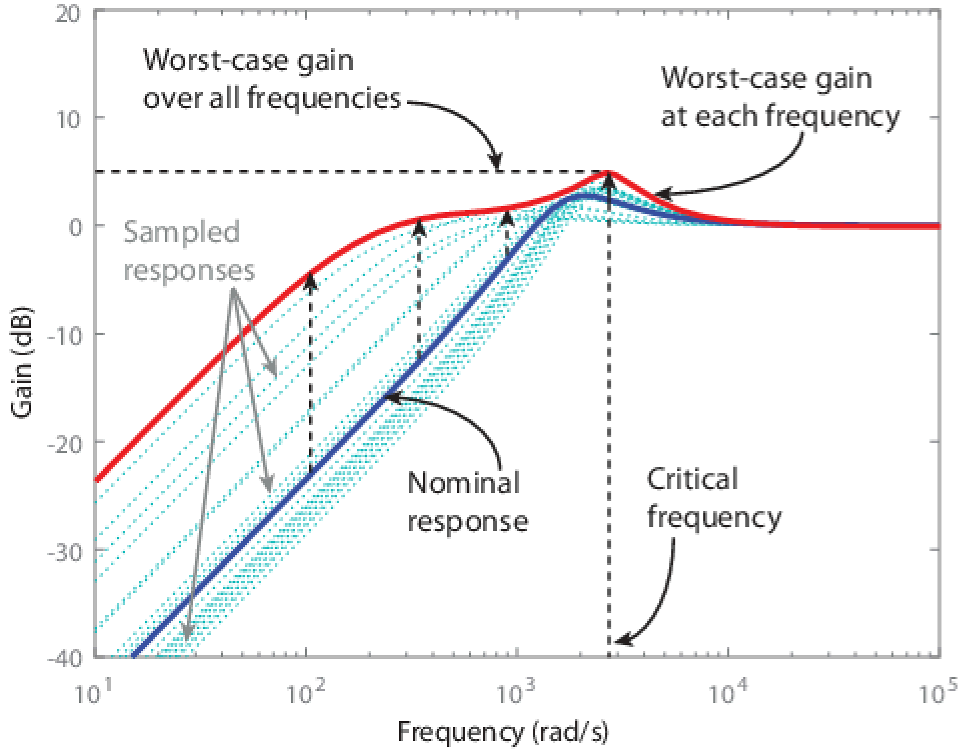
Field	Description
`Bounds`	Lower and upper bounds on the actual worst-case gain of the model, returned as an array. `info.Bounds(:,1)` contains the lower bound at each corresponding frequency in `info.Frequency`, and `info.Bounds(:,2)` contains the corresponding upper bounds.
`WorstPerturbation`	Perturbations that cause the worst-case gain at each frequency point in `info.Frequency`, returned as a structure array. The fields of `info.WorstPerturbation` are the names of the uncertain elements in `usys`, and each field contains the worst-case value of the corresponding element at each frequency. For example, if `usys` includes an uncertain parameter `p` and SISO uncertain dynamics `delta`, then `info.WorstPerturbation.p` is a collection of numeric values and `info.WorstPerturbation.delta` is a collection of SISO state-space models.
`Sensitivity`	Sensitivity of the worst-case gain to each uncertain element, returned as a structure when the `'Sensitivity'` option of `wcOptions` is `'on'`. The fields of `info.Sensitivity` are the names of the uncertain elements in `usys`. Each field contains a percentage that measures how much the uncertainty in the corresponding element affects the worst-case gain. For example, if `info.Sensitivity.p` is 50, then a given fractional change in the uncertainty range of `p` causes half as much fractional change in the worst-case gain. If the `'Sensitivity'` option of `wcOptions` is off (the default setting), then `info.Sensitivity` is `NaN`.
`BadUncertainValues`	Same as `WorstPerturbation`. Included for compatibility with R2016a and earlier.
`ArrayIndex`	Same as `Model`. Included for compatibility with R2016a and earlier.

More About

Worst-Case Gain

By default, `wcgain` returns the peak gain (or peak singular value, for MIMO systems) achievable within the uncertainty range, over all frequencies (or the frequencies specified by `w`). You can obtain the peak gain as a function of frequency using the `VaryFrequency` option of `wcOptions`.

To understand the difference, consider the following illustration, representing the magnitude of the frequency response of an uncertain system.



The dark blue curve is the nominal response of the system. The light blue curves show various sampled responses of the system. The `wcg` output of `wcgain` contains the bounds on the worst-case gain over all frequencies, about 5 dB in the illustration. The frequency at which this gain occurs is the critical frequency, also returned in `wcg`.

If you set the `VaryFrequency` option of `wcOptions` to `'on'`, then `wcgain` also calculates the maximum gain at each frequency point, shown by the red curve. `wcgain` returns these values in `info.Bounds`. See “Worst-Case Gain at Frequencies in a Range” on page 1-657 for an example. You can also use `wcsigma` to visualize the worst-case gain as a function of frequency.

Algorithms

Computing the worst-case gain at a particular frequency is equivalent to computing the structured singular value, μ, for some appropriate block structure (μ-analysis).

For `uss` and `genss` models, `wcgain(usys)` and `wcgain(usys,{wmin,wmax})` use an algorithm that finds the worst-case gain across frequency. This algorithm does not rely on frequency gridding and is not adversely affected by sharp peaks of the μ structured singular value. See “Getting Reliable Estimates of Robustness Margins” for more information.

For `ufrd` and `genfrd` models, `wcgain` computes the μ lower and upper bounds at each frequency point. This computation offers no guarantee between frequency points and can be inaccurate if the uncertainty gives rise to sharp resonances. The syntax `wcgain(uss,w)`, where `w` is a vector of frequency points, is the same as `wcgain(ufrd(uss,w))` and also relies on frequency gridding to compute the worst-case gain.

In general, the algorithm for state-space models is faster and safer than the frequency-gridding approach. In some cases, however, the state-space algorithm requires many μ calculations. In those cases, specifying a frequency grid as a vector `w` can be faster, provided that the worst-case gain varies smoothly with frequency. Such smooth variation is typical for systems with dynamic uncertainty.

See Also

`mussv` | `robstab` | `wcOptions` | `wcdiskmargin` | `wcsigma`

Topics

"Robust Stability and Worst-Case Gain of Uncertain System"
"Robustness and Worst-Case Analysis"

Introduced before R2006a

wcgainOptions

(Not recommended) Option set for `wcsens`

Note `wcgainOptions` is not recommended. Use `wcOptions` instead. For more information, see “Compatibility Considerations”.

Syntax

```
opt = wcgainOptions
opt = wcgainOptions(Name,Value,...)
```

Description

`opt = wcgainOptions` returns the default option set for a `wcsens` calculation. Use dot notation to set the values of the options listed in “Input Arguments” on page 1-670.

`opt = wcgainOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Note To create options sets for `wcgain`, `wcdiskmargin`, or `wcsigma`, use `wcOptions`.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Sensitivity`

Determines whether to compute the sensitivity of worst-case responses with respect to each individual uncertain element. This quantity determines how sensitive each system response (such as sensitivity or complementary sensitivity) is to variations in the uncertain parameters.

`Sensitivity` takes the following values:

- `'on'` — `wcsens` computes the sensitivity of the worst-case responses with respect to each individual uncertain element. This provides an indication of which elements are most problematic.
- `'off'` — `wcsens` does not compute the sensitivity of the worst-case responses with respect to each individual uncertain element.

Default: `'on'`

`VaryUncertainty`

Percentage variation of uncertainty for calculations of sensitivity to uncertainty. The sensitivity estimate uses a finite difference calculation.

Default: 25

`LowerBoundOnly`

Determines whether only the lower bound is computed.

`LowerBoundOnly` takes the following values:

- `'on'` — `wcsens` only computes a lower bound on the worst-case response
- `'off'` — `wcgain` computes lower and upper bounds on the worst-case response

Default: `'off'`

`MaxOverFrequency`

`MaxOverFrequency` takes the following values:

- `'on'` — `wcsens` computes the worst-case response function
- `'off'` — `wcsens` computes the worst possible response at each frequency point

Default: `'on'`

`MaxOverArray`

For uncertain model arrays, determines if worst-case response is calculated over entire array or individually for all models in array.

`MaxOverArray` takes the following values:

- `'on'` — `wcsens` computes the worst-case response over all models
- `'off'` — `wcsens` computes the worst-case response for each model individually

Default: `'on'`

`AbsTol`

Absolute tolerance on computed bound.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`.

Relaxing tolerance speeds up the computation.

Default: 0.02

`RelTol`

Relative tolerance on computed bound.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`.

Default: 0.05

`AbsMax`

Absolute threshold for lower bound.

The algorithm terminates if `LowerBound >= AbsMax + RelMax * NominalGain`.

Specify `AbsMax` and `RelMax` to terminate when the lower bound is large enough compared to the nominal gain.

Default: 5

`RelMax`

Relative threshold for lower bound.

The algorithm terminates if `LowerBound >= AbsMax + RelMax * NominalGain`.

Specify `AbsMax` and `RelMax` to terminate when the lower bound is large enough compared to the nominal gain.

Default: 20

`NSearch`

Number of lower bound searches at each frequency

Default: 2

Output Arguments

`opt`

Option set containing the specified options for `wcsens`.

Examples

Determine Effect of Uncertainty Range on Worst-Case Response

Consider a feedback loop with a first-order plant and a PI controller. The time constant of the plant is uncertain, and the feedback loop accounts for unmodeled dynamic uncertainty. Compute the worst-case gain of the sensitivity function `Si` at the plant inputs. Use the `'Sensitivity'` option of `wcgainOptions` to compute how sensitive this worst-case gain is to each uncertain element.

```
% Create uncertain system and controller
delta = ultidyn('delta',[1 1]);
tau = ureal('tau',5,'range',[4 6]);
P = tf(1,[tau 1])*(1+0.25*delta);
C = pid(4,4);
```

```
opt = wcgainOptions('Sensitivity','on');
wcst = wcsens(P,C,'Si',opt);
```

The `Sensitivity` field of the `Si` entry in the output structure includes data that reflects how much the maximum gain of the input sensitivity function changes with each uncertain element.

```
wcst.Si.Sensitivity
```

```
ans = struct with fields:
    delta: 44
      tau: 9
```

This result tells you that if the uncertainty range of delta increases by 10%, the peak input sensitivity increases by about 4.4%. Similarly, a 10% increase in the uncertainty range of tau causes about a 0.9% increase in the peak input sensitivity.

Compatibility Considerations

`wcgainoptions` is not recommended

Not recommended starting in R2019a

Use `wcOptions` to create option sets for `wcgain`, `wcdiskmargin`, and other worst-case computation functions.

There are no plans to remove `wcgainoptions` at this time.

See Also

`wcOptions`

Introduced in R2011b

wcgainplot

(Not recommended) Graphical worst-case gain analysis

Note `wcgainplot` is not recommended. Use `wcsigma` instead.

Syntax

```
wcgainplot(sys)
wcgainplot(sys,w)
wcgainplot(sys,...,options)
```

Description

`wcgainplot(sys)` plots the nominal and worst-case gains of the uncertain system `sys` as a function of frequency. For multi-input, multi-output (MIMO) systems, gain refers to the largest singular value of the frequency response matrix. (See `sigma` for more information about singular values.) The plot includes:

- Nominal — nominal gain of `sys`
- Worst — the response falling within the uncertainty of `sys` that has the highest peak gain
- Worst-case gain (lower bound) — the lowest worst-case gain at each frequency
- Worst-case gain (upper bound) — the highest gain within the uncertainty at each frequency
- Sampled Uncertainty — 20 responses randomly sampled from `sys`

`wcgainplot(sys,w)` focuses the plot on the frequencies specified by `w`.

- If `w` is a cell array `{wmin,wmax}`, `wcgainplot` plots the worst-case gains in the range `{wmin,wmax}`.
- If `w` is an array of frequencies, `wcgainplot` plots the worst-case gains at each frequency in the array.

`wcgainplot(sys,...,options)` uses the options set `options` to specify additional options for the computation of the worst-case gains. Use `wcOptions` to create the options set.

Input Arguments

`sys`

Uncertain dynamic system (Control System Toolbox).

`w`

Frequencies of worst-case gain plots. Specify frequencies in radians/`TimeUnit`, where `TimeUnit` is the time unit of `sys`.

- If `w` is a cell array `{wmin,wmax}`, `wcgainplot` plots the worst-case gains in the range `{wmin,wmax}`.
- If `w` is an array of frequencies, `wcgainplot` plots the worst-case gains at each frequency in the array.

`options`

Options set specifying additional options for the computation of the worst-case gains. Use `wcOptions` to create the options set.

Algorithms

`wcgainplot` uses `wcgain` to compute the worst-case gains. Use the `options` argument to `wcgainplot` to set options for the `wcgain` algorithm.

`wcgainplot` uses `usample` to compute the `Sampled Uncertainty` curves.

See Also

`sigma` | `usample` | `wcOptions` | `wcgain` | `wcsigma`

Introduced in R2011b

wcmargin

(Not recommended) Worst-case disk stability margins of uncertain feedback loops

Note `wcmargin` is not recommended. Use `wcdiskmargin` instead. For more information, see “Compatibility Considerations”.

Syntax

```
wcmarg = wcmargin(L)
```

```
wcmargi = wcmargin(p,c)
```

```
[wcmargi,wcmargo] = wcmargin(p,c)
```

```
wcmargi = wcmargin(p,c,opt)
```

```
[wcmargi,wcmargo] = wcmargin(p,c,opt)
```

Description

Classical gain and phase margins define the allowable loop-at-a-time variations in the nominal system gain and phase for which the feedback loop retains stability.

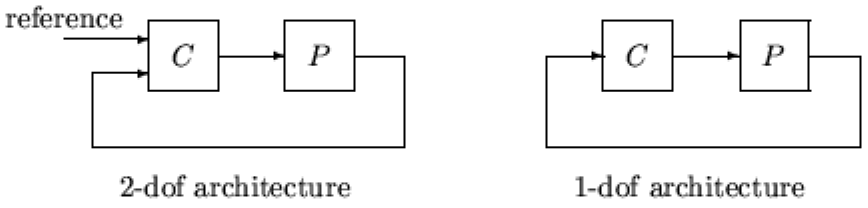
An alternative to classical gain and phase margins is the disk margin. The disk margin is the largest region for each channel such that for all gain and phase variations inside the region the nominal closed-loop system is stable. See “Stability Analysis Using Disk Margins” to learn more about the algorithm.

Consider a system with uncertain elements. It is of interest to determine the margin of each individual channel in the presence of uncertainty. These margins are called worst-case margins. Worst-case margin, `wcmargin` calculates the largest disk margin such that for values of the uncertainty and all gain and phase variations inside the disk, the closed-loop system is stable. The worst-case gain and phase margin bounds are defined based on the balanced sensitivity function. Hence, results from the worst-case margin calculation imply that the closed-loop system is stable for a given uncertainty set and would remain stable in the presence of an additional gain and phase margin variation in the specified input/output channel.

`wcmargL = wcmargin(L)` calculates the combined worst-case input and output loop-at-a-time gain/phase margins of the feedback loop consisting of the loop transfer matrix `L` in negative feedback with an identity matrix. `L` must be an uncertain system, `uss` or `ufrd` object. If `L` is a `uss` object, the frequency range and number of points used to calculate `wcmargL` are chosen automatically. Note that in this case, the worst-case margins at the input and output are equal because an identity matrix is used in feedback. `wcmarg` is a `NU`-by-`1` structure with the following fields:

Field	**Description**
`GainMargin`	Guaranteed bound on worst-case, single-loop gain margin at plant inputs. Loop-at-a-time analysis.
`PhaseMargin`	Loop-at-a-time worst-case phase margin at plant inputs. Units are degrees.
`Frequency`	Frequency associated with the worst-case margin (rad/s).
`WCUnc`	Structure of the worst-case uncertainty values associated with the worst-case disk gain and phase margins for the `i`-th loop `L(i,i)`.
`Sensitivity`	`Struct` with `M` fields. Field names are names of uncertain elements of `P` and `C`. Values of fields are positive numbers, which each entry indicating the local sensitivity of the worst-case margins to all the individual uncertain element's uncertainty ranges. For instance, a value of 50 indicates that if the uncertainty range is enlarged by 8%, then the worst-case gain should increase by about 4%. If the `Sensitivity` property of the `wcOptions` object is `'off'`, the values are `NaN`.

`[wcmargi,wcmargo] = wcmargin(P,C)` calculates the combined worst-case input and output loop-at-a-time gain/phase margins of the feedback loop consisting of `C` in negative feedback with `P`. `C` should only be the compensator in the feedback path, without reference channels, if it is a 2-Dof architecture. That is, if the closed-loop system has a 2-Dof architecture the reference channel of the controller should be eliminated resulting in a 1-Dof architecture as shown in the following figure. Either `P` or `C` must be an uncertain system, `uss` or `ufrd`, or an uncertain matrix, `umat`. If `P` and `C` are `ss/tf/zpk` or `uss` objects, the frequency range and number of points used to calculate `wcmargi` and `wcmargo` are chosen automatically.



Basic Syntax

```
[wcmargi,wcmargo] = wcmargin(L)
[wcmargi,wcmargo] = wcmargin(P,C)
```

`wcmargi` and `wcmargo` are structures corresponding to the loop-at-a-time worst-case, single-loop gain and phase margin of the channel. For the single-loop transfer matrix L of size N-by-N, `wcmargi` is a N-by-1 structure. For the case with two input arguments, the plant model `P` will have N_Y outputs and N_U inputs and hence the controller `C` must have N_U outputs and N_Y inputs. `wcmargi` is a `NU`-by-`1` structure with the following fields:

Field	Description
`GainMargin`	Guaranteed bound on worst-case, single-loop gain margin at plant inputs. Loop-at-a-time analysis.
`PhaseMargin`	Loop-at-a-time worst-case phase margin at plant inputs. Units are degrees.
`Frequency`	Frequency associated with the worst-case margin (rad/s).
`WCUnc`	Structure of the worst-case uncertainty values associated with the worst-case disk gain and phase margins for the `i`-th loop `L(i,i)`.
`Sensitivity`	`Struct` with M fields. Field names are names of uncertain elements of P and C. Values of fields are positive numbers, which each entry indicating the local sensitivity of the worst-case margins to all the individual uncertain element's uncertainty ranges. For instance, a value of 50 indicates that if the uncertainty range is enlarged by 8%, then the worst-case gain should increase by about 4%. If the `Sensitivity` property of the `wcOptions` object is `'off'`, the values are `NaN`.

`wcmargo` is an N-by-1 structure for the single loop transfer matrix input and `wcmargo` is an N_Y-by-1 structure when the plant and controller are input. In both these cases, `wcmargo` has the same fields as `wcmargi`. The worst-case bound on the gain and phase margins are calculated based on a balanced sensitivity function.

`[wcmargi,wcmargo] = wcmargin(L,opt)` and

`[wcmargi,wcmargo] = wcmargin(p,c,opt)` specify options described in `opt`. (See `wcOptions` for more details on the options for `wcmargin`.)

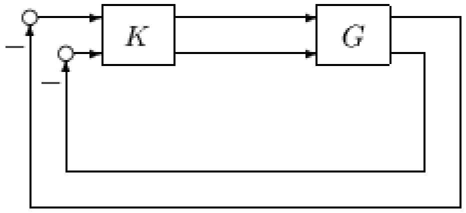
The sensitivity of the worst-case margin calculations to the individual uncertain elements is selected using the options object `opt`. To compute sensitivities, create a `wcOptions` options object, and set the `Sensitivity` property to `'on'`.

Examples

MIMO Loop-at-a-Time Margins

This example is designed to illustrate that loop-at-a-time margins (gain, phase, and/or distance to –1) can be inaccurate measures of multivariable robustness margins. Margins of the individual loops can be very sensitive to small perturbations within other loops.

The nominal closed-loop system considered here is shown as follows.



G and K are 2-by-2 multi-input/multi-output (MIMO) systems, defined as

$$G:=\frac{1}{s^2+\alpha^2}\begin{bmatrix} s-\alpha^2 & \alpha(s+1) \\ -\alpha(s+1) & s-\alpha^2 \end{bmatrix}, K = I_2$$

Set α := 10, construct the nominal model G in state-space form, and compute its frequency response.

```
a = [0 10;-10 0];
b = eye(2);
```

```
c = [1 8;-10 1];
d = zeros(2,2);
G = ss(a,b,c,d);
K = [1 -2;0 1];
```

The nominal plant was analyzed previously using the command. Based on experimental data, the gain of the first input channel, `b(1,1)`, is found to vary between 0.97 and 1.06. The following statement generates the updated uncertain model.

```
ingain1 = ureal('ingain1',1,'Range',[0.97 1.06]);
b = [ingain1 0;0 1];
Gunc = ss(a,b,c,d);
```

Because of differences between measured data and the plant model an 8% unmodeled dynamic uncertainty is added to the plant outputs.

```
unmod = ultidyn('unmod',[2 2],'Bound',0.08);
Gmod = (eye(2)+unmod)*Gunc;
Gmodg = ufrd(Gmod,logspace(-1,3,60));
```

You can use the command `wcmargin` to determine the worst-case gain and phase margins in the presences of the uncertainty.

```
[wcmi,wcmo] = wcmargin(Gmodg,K);
```

The worst-case analysis corresponds to maximum allowable disk margin for all possible defined uncertainty ranges. The worst-case single-loop margin analysis performed using `wcmargin` results in a maximum allowable gain margin variation of 1.31 and phase margin variations of ± 15.6 degs in the second input channel in the presence of the uncertainties `'unmod'` and `'ingain1'`. `wcmi(1)`

```
ans =
     GainMargin: [0.3613 2.7681]
    PhaseMargin: [-50.2745 50.2745]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
wcmi(2)
ans =
     GainMargin: [0.7585 1.3185]
    PhaseMargin: [-15.6426 15.6426]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
```

Hence even though the second channel had infinite gain margin and 90 degrees of phase margin, allowing variation in both uncertainties, `'unmod'` and `'ingain1'` leads to a dramatic reduction in the gain and phase margin.

You can display the sensitivity of the worst-case margin in the second input channel to `'unmod'` and `'ingain1'` as follows:

```
wcmi(2).Sensitivity
ans =
    ingain1: 12.1865
      unmod: 290.4557
```

The results indicate that the worst-case margins are not very sensitive to the gain variation in the first input channel, `'ingain1'`, but very sensitive to the LTI dynamic uncertainty at the output of the plant.

The worst-case single-loop margin at the output results in a maximum allowable gain margin variation of 1.46 and phase margin variation of ± 21.3 degs in the second output channel in the presence of the uncertainties `'unmod'` and `'ingain1'`.

```
wcmo(1)
ans =
     GainMargin: [0.2521 3.9664]
    PhaseMargin: [-61.6995 61.6995]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
wcmo(2)
ans =
     GainMargin: [0.6835 1.4632]
    PhaseMargin: [-21.2984 21.2984]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
```

You can display the sensitivity of the worst-case margin in the second output channel to `'unmod'` and `'ingain1'` as follows:

```
wcmo(2).Sensitivity
ans =
    ingain1: 16.3435
      unmod: 392.1320
```

The results are similar to the worst-case margins at the input. However, the worst-case margins at the second output channel are even more sensitive to the LTI dynamic uncertainty than the input channel margins.

Compatibility Considerations

`wcmargin` is not recommended

Not recommended starting in R2019a

For worst-case stability margins, use the `wcdiskmargin` command. `wcdiskmargin` can compute both loop-at-a-time and multiloop margins, while `wcmargin` only computes loop-at-a-time margins. `wcdiskmargin` can also compute stability margins with respect to independent, concurrent variations at both the plant inputs and plant outputs. Further, `wcdiskmargin` includes an optional eccentricity parameter, *E*, that lets you vary the shape of the uncertainty region used to compute the disk margin. Varying the eccentricity can improve the gain and phase margin estimates.

The following table shows some typical uses of `wcmargin` and how to update your code to use `wcdiskmargin` instead.

Not Recommended	Recommended
`wcmarg = wcmargin(L)`	`[wcDM,wcu] = wcdiskmargin(L,'siso')`
`[wcmargI,wcmargO] = wcmargin(P,C)`	`[wcmargI,wcuI] = wcdiskmargin(C*P,'siso')`, for margins at plant input `[wcmargO,wcuO] = wcdiskmargin(P*C,'siso')`, for margins at plant output

There are no plans to remove `wcmargin` at this time.

See Also

`dmplot` | `loopsens` | `robstab` | `usubs` | `wcOptions` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

wcmarginOptions

(Not recommended) Option set for `wcmargin`

Note `wcmarginOptions` is not recommended. Use `wcOptions` instead.

Syntax

```
opt = wcmarginOptions
opt = wcmarginOptions(Name,Value,...)
```

Description

`opt = wcmarginOptions` returns the default option set for `wcmargin`.

`opt = wcmarginOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`Sensitivity`

Determines whether to compute the sensitivity of worst-case gain with respect to each individual uncertain element.

`Sensitivity` takes the following values:

- `'on'` — Sensitivity of the worst-case gain is computed with respect to each individual uncertain element. This provides an indication of which elements are most problematic.
- `'off'` — `wcmargin` does not compute the sensitivity of the worst-case gain with respect to each individual uncertain element.

Default: `'off'`

`AbsTol`

Absolute tolerance on computed worst-case margin bounds.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`

Default: 0.02

`RelTol`

Relative tolerance on computed worst-case margin bounds.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`

Default: 0.05

Output Arguments

`opt`

Option set containing the specified options for `wcmargin`.

Examples

Create an options set for `wcmargin` with an 0.01 and 0.03 as the absolute and relative tolerances on the worst-case margin bounds, respectively.

```
opt = wcmarginOptions('AbsTol',0.01,'RelTol',0.03);
```

Alternatively, use dot notation to set the values of `opt`.

```
opt = wcmarginOptions;
opt.AbsTol = 0.01;
opt.RelTol = 0.03;
```

See Also

`wcOptions`

Introduced in R2011b

wcnorm

Worst-case norm of uncertain matrix

Syntax

```
maxnorm = wcnorm(m)
```

```
[maxnorm,wcu] = wcnorm(m)
```

```
[maxnorm,wcu] = wcnorm(m,opts)
```

```
[maxnorm,wcu,info] = wcnorm(m)
```

```
[maxnorm,wcu,info] = wcnorm(m,opts)
```

Description

The norm of an uncertain matrix generally depends on the values of its uncertain elements. Determining the maximum norm over all allowable values of the uncertain elements is referred to as a *worst-case norm* analysis. The maximum norm is called the *worst-case norm*.

As with other *uncertain-system* analysis tools, only bounds on the worst-case norm are computed. The exact value of the worst-case norm is guaranteed to lie between these upper and lower bounds.

Basic syntax

Suppose `mat` is a `umat` or a `uss` with *M* uncertain elements. The results of

```
[maxnorm,maxnormunc] = wcnorm(mat)
```

`maxnorm` is a structure with the following fields.

Field	Description
`LowerBound`	Lower bound on worst-case norm, positive scalar.

Field	Description
`UpperBound`	Upper bound on worst-case norm, positive scalar.

`maxnormunc` is a structure that includes values of uncertain elements and maximizes the matrix norm. There are *M* field names, which are the names of uncertain elements of `mat`. The value of each field is the corresponding value of the uncertain element, such that when jointly combined, lead to the norm value in `maxnorm.LowerBound`. The following command shows the norm:

```
norm(usubs(mat,maxnormunc))
```

Basic syntax with third output argument

A third output argument provides information about sensitivities of the worst-case norm to the uncertain elements ranges.

```
[maxnorm,maxnormunc,info] = wcnorm(mat)
```

The third output argument `info` is a structure with the following fields:

Field	Description
`Model`	Index of model with largest gain (when `mat` is an array of uncertain matrices)
`WorstPerturbation`	Structure of worst-case uncertainty values. The fields of `info.WorstPerturbation` are the names of the uncertain elements in `mat`, and each field contains the worst-case value of the corresponding element.
`Sensitivity`	A `struct` with *M* fields. Fieldnames are names of uncertain elements of `sys`. Field values are positive numbers, each entry indicating the local sensitivity of the worst-case norm in `maxnorm.LowerBound` to all of the individual uncertain elements' uncertainty ranges. For instance, a value of 25 indicates that if the uncertainty range is increased by 8%, then the worst-case norm should increase by about 2%. If the `Sensitivity` property of the `wcOptions` object is `'off'`, the values are `NaN`.
`BadUncertainValues`	Same as `WorstPerturbation`. Included for compatibility with R2016a and earlier.
`ArrayIndex`	Same as `Model`. Included for compatibility with R2016a and earlier.

Examples

Worst-Case Norm and Condition Number of an Uncertain Matrix

Construct an uncertain matrix and compute the worst-case norm of the matrix and of its inverse. These computations let you accurately estimate the worst-case, or the largest, value of the condition number of the matrix.

```
a = ureal('a',5,'Range',[4 6]);
b = ureal('b',3,'Range',[2 10]);
c = ureal('c',9,'Range',[8 11]);
d = ureal('d',1,'Range',[0 2]);

M = [a b;c d];
Mi = inv(M);

maxnormM = wcnorm(M)
```

```
maxnormM = struct with fields:
    LowerBound: 14.7199
    UpperBound: 14.7227
```

```
maxnormMi = wcnorm(Mi)
```

```
maxnormMi = struct with fields:
    LowerBound: 2.5963
    UpperBound: 2.5968
```

The condition number of `M` must be less than the product of the two upper bounds for all values of the uncertain elements of `M`. Conversely, the condition number of the largest value of `M` must be at least equal to the condition number of the nominal value of `M`. Compute these bounds on the worst-case value of the condition number.

```
condUpperBound = maxnormM.UpperBound*maxnormMi.UpperBound;
condLowerBound = cond(M.NominalValue);
[condLowerBound condUpperBound]
```

```
ans = 1×2

    5.0757   38.2312
```

The range between these lower and upper bounds is fairly large. You can get a more accurate estimate. Recall that the condition number of an n-by-m matrix `M` can be expressed as an optimization, where a free norm-bounded matrix Δ tries to align the gains of `M` and `inv(M)`:

$$\kappa(M) = \max_{\substack{\Delta \in C^{m \times m} \\ \sigma_{\max}(\Delta) \le 1}} \left(\sigma_{\max}\left(M\Delta M^{-1}\right)\right)$$

If `M` is uncertain, then the worst-case condition number involves further maximization over the possible values of `M`. Therefore, you can compute the worst-case condition number of an uncertain matrix by using a `ucomplexm` uncertain element and using `wcnorm` to carry out the maximization.

Create a 2-by-2 `ucomplexm` element with nominal value 0.

```
Delta = ucomplexm('Delta',zeros(2,2));
```

The range of values represented by `Delta` includes 2-by-2 matrices with the maximum singular value less than or equal to 1.

Create the expression involving `M`, `Delta`, and `inv(M)`.

```
H = M*Delta*Mi;

opt = wcOptions('MussvOptions','m5');
[maxKappa,wcu,info] = wcnorm(H,opt);
maxKappa
```

```
maxKappa = struct with fields:
    LowerBound: 26.8406
    UpperBound: 38.2349
```

Verify that the values in `wcu` make the condition number as large as `maxKappa.LowerBound`.

```
cond(usubs(M,wcu))
```

```
ans = 26.9629
```

Algorithms

See `wcgain`.

See Also

`norm` | `wcOptions` | `wcgain`

Introduced before R2006a

wcOptions

Option set for worst-case analysis

Syntax

```
opts = wcOptions
opts = wcOptions(Name,Value,...)
```

Description

`opts = wcOptions` returns the default option set for worst-case analysis commands such as `wcgain`, `wcdiskmargin`, or `wcsigma`.

`opts = wcOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Examples

Options for Worst-Case Gain Calculation

Create an options set to calculate the worst-case gain while allowing the uncertain parameters to vary by 20% more than the range specified in the model. Also, configure the options to include the element-by-element sensitivity in the calculation.

```
opts = wcOptions('ULevel',1.2,'Sensitivity','on');
```

Alternatively, create a default option set, and use dot notation to set the values of particular options.

```
opts = wcOptions;
opts.ULevel = 1.2;
opts.Sensitivity = 'on';
```

Use `opts` as an input argument to a worst-case analysis command such as `wcgain`.

Improve Worst-Case Perturbation

Specifying certain options for the structured-singular-value computation that underlies the worst-gain computation can yield better results in some cases. For example, consider a sample plant and controller.

```
load(fullfile(matlabroot,'examples','robust','wcgExampleData.mat'))
```

This command loads `gPlant`, a MIMO plant with 10 outputs, 8 inputs, and 11 uncertain elements. It also loads `Kmu`, a state-space controller model. Form a closed-loop system with these models, and examine the worst-case gain.

```
CL = lft(gPlant,Kmu);
[wcg,wcu] = wcgain(CL);
wcg
```

```
wcg = struct with fields:
           LowerBound: 10.8207
           UpperBound: 11.2135
    CriticalFrequency: 6.6782
```

There is a large difference between the lower and upper bounds on the worst-case gain. To get a better estimate of the actual worst-case gain, increase the number of restarts that `wcgain` uses for computing of the worst-case perturbation and associated lower bound. Doing so can result in a tighter lower bound. This option does not affect the upper-bound calculation.

```
opt = wcOptions('MussvOptions','m3');
[wcg,wcu] = wcgain(CL,opt);
wcg
```

```
wcg = struct with fields:
           LowerBound: 10.8207
           UpperBound: 11.2135
    CriticalFrequency: 6.6782
```

The difference between the lower bound and upper bound on the worst-case gain is much smaller. The critical frequency is different as well.

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ULevel','1.5','Sensitivity','on'`

`ULevel` — Uncertainty level

1 (default) | positive scalar

Uncertainty level to use for the worst-case computation, specified as the comma-separated pair consisting of `'ULevel'` and a positive scalar value. This option scales the normalized uncertainty by the factor you specify. Such scaling lets you examine the effect of smaller or larger uncertainty range without changing the uncertainty levels in your model. For instance, to see the effect of doubling the ranges of all uncertain parameters, set `'ULevel'` to 2. To see the effect of shrinking the ranges, set `'ULevel'` to 0.5. The default value, 1, corresponds to the amount of uncertainty specified in the model.

`Display` — Display progress of computation and summary report

`'off'` (default) | `'on'`

Display progress and summary report of the worst-case gain computation, specified as the comma-separated pair consisting of `'Display'` and one of these values:

- `'off'` — Do not display progress and report.
- `'on'` — Display progress and report. When you use this option, a progress indicator and summary of results is displayed in the command window, similar to the following.

```
The worst-case gain is at most 11.2.
 -- There is a bad perturbation amounting to 100% of the modeled uncertainty.
 -- This perturbation causes a gain of 9.03 at the frequency 5.5 rad/seconds.
```

This setting overrides the silent (`'s'`) option in the `MussvOptions` option.

`VaryFrequency` — Compute worst-case gain as function of frequency

`'off'` (default) | `'on'`

Return worst-case gain as a function of frequency, specified as the comma-separated pair consisting of `'VaryFrequency'` and one of these values:

- `'off'` — Only return worst-case gains at frequencies where the worst values occur.
- `'on'` — Compute worst-case gains over a frequency grid suitable for plotting. The frequency grid is chosen automatically based on system dynamics. This calculation is done in addition to identifying the critical frequency where the gain peaks. Access the frequency values and corresponding gains in the `info` output of `wcgain` or other worst-case analysis command.

This option is ignored for `ufrd` and `genfrd` models.

`Sensitivity` — Calculate sensitivity of worst-case gains
`'off'` (default) | `'on'`

Calculate the sensitivity of the worst-case gain to each uncertain element in the model, specified as the comma-separated pair consisting of `'Sensitivity'` and either `'off'` or `'on'`.

Each uncertain element contributes to the overall worst case in a coupled manner. Set this option to `'on'` to estimate the sensitivity of the margin to each element. This element-by-element sensitivity provides an indication of which elements are most problematic. Access the sensitivity estimates in the `info` output of the worst-case computation command.

`SensitivityPercent` — Percentage variation of uncertainty for computing sensitivity
25 (default) | positive scalar value

Percentage variation of uncertainty level for computing sensitivity, specified as the comma-separated pair consisting of `'SensitivityPercent'` and a positive scalar value. The sensitivity to a particular uncertain element is estimated using a finite difference calculation. This calculation increases the (normalized) amount of uncertainty on this element by some percentage, computes the resulting worst-case gain, and computes the ratio of percent variations. This option specifies the percentage increase in uncertainty level applied to each element. The default value is 25%.

`MussvOptions` — Options for `mussv` calculation
`''` (default) | character vector

Options for the `mussv` calculation that underlies the worst-case calculations, specified as the comma-separated pair consisting of `'MussvOptions'` and a character vector such as `'sm3'` or `'ad'`.

Some `MussvOptions` values that are especially useful for improving worst-case calculations include:

- `'a'` — Force the use of LMI optimization to compute the μ upper bound, which yields better results in general but can be expensive when some `ureal` elements are repeated multiple times.
- `'mN'` — Use multiple restarts when computing the μ lower bound, which corresponds to the lower bound for the worst-case gain. This option can reduce the gap between the lower bound and upper bound on the worst-case gains. `N` is the number of restarts. For example, setting `'MussvOptions'` to `'m3'` causes three restarts. See “Improve Worst-Case Perturbation” on page 1-693 for an example.

See `mussv` for the remaining available options and corresponding characters. The default, `''`, uses the default options for `mussv`.

Output Arguments

`opts` — Options for worst-case calculations
`wcOptions` object

Options for worst-case calculations, returned as a `wcOptions` object. Use the options as an input argument to any of the worst-case analysis functions, such as `wcgain` and `wcsigma`. For example:

```
[wcgain,wcu,info] = wcgain(usys,opts)
```

See Also

`wcdiskmargin` | `wcgain` | `wcsigma`

Topics

“Robustness and Worst-Case Analysis”

Introduced in R2016b

wcsens

(Not recommended) Calculate worst-case sensitivity and complementary sensitivity functions of plant-controller feedback loop

---

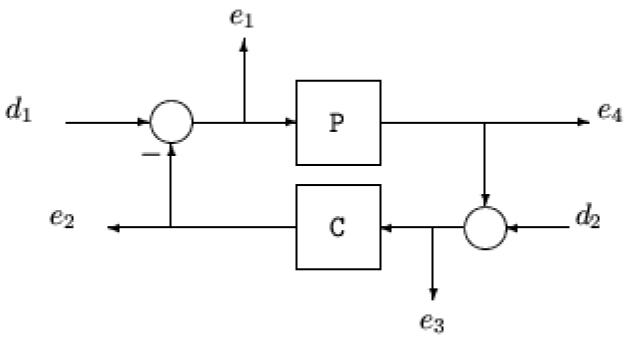
Note `wcsens` is not recommended. Use `wcgain` instead. For more information, see "Compatibility Considerations".

Syntax

```
wcst = wcsens(L)

wcst = wcsens(L,type)

wcst = wcsens(L,opt)

wcst = wcsens(L,type,scaling)

wcst = wcsens(L,type,scaling,opt)

wcst = wcsens(P,C)

wcst = wcsens(P,C,type)

wcst = wcsens(P,C,opt)

wcst = wcsens(P,C,type,scaling)

wcst = wcsens(P,C,type,scaling,opt)
```

Description

The sensitivity function, $S = (I + L)^{-1}$, and the complementary sensitivity function, $T = L(I + L)^{-1}$, where L is the loop gain matrix associated with the input, CP, or output, PC, are two transfer functions related to the robustness and performance of the closed-loop system. The multivariable closed-loop interconnection structure, shown below, defines the input/output sensitivity, complementary sensitivity and loop transfer functions.



The following table gives the values of the input and output sensitivity functions for this control structure.

Description	Equation
Input sensitivity S_i (closed-loop transfer function from d_1 to e_1)	$S_i = (I + CP)^{-1}$
Input complementary sensitivity T_i (closed-loop transfer function from d_1 to e_2)	$T_i = CP(I + CP)^{-1}$
Output sensitivity S_o (closed-loop transfer function from d_2 to e_2)	$S_o = (I + PC)^{-1}$
Output complementary sensitivity T_o (closed-loop transfer function from d_2 to e_4)	$T_o = PC(I + PC)^{-1}$
Input loop transfer function L_i	$L_i = CP$
Output loop transfer function L_o	$L_o = PC$

`wcst = wcsens(L)` calculates the worst-case sensitivity and complementary sensitivity functions for the loop transfer matrix `L` in feedback in negative feedback with an identity matrix. If `L` is a `uss` object, the frequency range and number of points are chosen automatically.

`wcst = wcsens(P,C)` calculates the worst-case sensitivity and complementary sensitivity functions for the feedback loop `C` in negative feedback with `P. C` should only be the compensator in the feedback path, not any reference channels, if it is a 2-dof architecture (see `loopsens`). If `P` and `C` are `ss/tf/zpk` or `uss` objects, the frequency range and number of points are chosen automatically. `wcst` is a structure with the following substructures:

Fields of `wcst`

Field	Description
`Si`	Worst-case input-to-plant sensitivity function
`Ti`	Worst-case input-to-plant complementary sensitivity function
`So`	Worst-case output-to-plant sensitivity function
`To`	Worst-case output-to-plant complementary sensitivity function
`PSi`	Worst-case plant times input-to-plant sensitivity function
`CSo`	Worst-case compensator times output-to-plant sensitivity function
`Stable`	1 if nominal closed loop is stable, 0 otherwise. `NaN` for `frd/ufrd` objects.

Each sensitivity substructure is a structures with five fields `MaximumGain, BadUncertainValues, System, BadSystem, Sensitivity` derived from the outputs of `wcgain`.

Fields of Si, So, Ti, To, PSi, CSo

Field	Description
MaximumGain	struct with fields LowerBound, UpperBound and CriticalFrequency. LowerBound and UpperBound are bounds on the unweighted maximum gain of the uncertain sensitivity function. CriticalFrequency is the frequency at which the maximum gain occurs.
BadUncertainValues	Struct, containing values of uncertain elements which maximize the sensitivity gain. There are *M* fluidness, which are the names of uncertain elements of sensitivity function. The value of each field is the corresponding value of the uncertain element, such that when jointly combined, lead to the gain value in MaximumGain.LowerBound.
System	Uncertain sensitivity function (ufrd or uss).
BadSystem	Worst-case system based on the uncertain object values in BadUncertainValues. BadSystem is defined as BadSystem=usubs(System, BadUncertainValues).
Sensitivity	Struct with M fields, fieldnames are names of uncertain elements of system. Values of fields are positive numbers, each entry indicating the local sensitivity of the maximum gain to all of the individual uncertain elements uncertainty ranges. For instance, a value of 50 indicates that if the uncertainty range is enlarged by 8%, then the maximum gain should increase by about 4%. If the 'Sensitivity' property of the wcgainOptions object is 'off', the values are NaN.

wcst = wcsens(L,type) and wcst = wcsens(P,C,type) allow selection of individual Sensitivity and Complementary Sensitivity functions, type, as 'Si','Ti','So','To','PSi','CSo' corresponding to the sensitivity and complementary sensitivity functions. Setting type to 'S' or 'T' selects all sensitivity functions ('Si','So','PSi','CSo') or all complementary sensitivity functions ('Ti','To'). Similarly, setting type to 'Input' or 'Output' selects all input Sensitivity functions ('Si','Ti','PSi') or all output sensitivity functions ('So,'To','CSo'). 'All' selects all six Sensitivity functions for analysis (default). type may also be a cell array containing multiple function types, such as {'Si','To'}.

wcst = wcsens(L,type,scaling) and wcst = wcsens(P,C,type,scaling) add a scaling to the worst-case sensitivity analysis. scaling is one of the following:

- `'Absolute'` (default) — Calculates bounds on the maximum gain of the uncertain sensitivity function.
- `'Relative'` — Finds bounds on the maximum relative gain of the uncertain sensitivity function. That is, the maximum relative gain is the largest ratio of the worst-case gain and the nominal gain evaluated at each frequency point in the analysis.
- LTI model (`ss`, `tf`, `zpk`, or `frd`) — Calculates bounds on the maximum scaled gain of the uncertain sensitivity function. The input/output dimensions of the LTI model must be either 1-by-1, or compatible with `P` and `C`.

You can also combine `type` and `scaling` in a cell array, e.g.

```
wcst = wcsens(P,C,{'Ti','So'},'Abs','Si','Rel','PSi',wt)
```

`wcst = wcsens(P,C,opt)` or `wcst = wcsens(P,C,type,scaling,opt)` specifies options for the worst-case gain calculation as defined by `opt`. (See `wcgainOptions` for more details on the options for `wcsens`.)

The sensitivity of the worst-case sensitivity calculations to the individual uncertain components can be determined using the options object `opt`. To compute the sensitivities to the individual uncertain components, create a `wcgainOptions` options object, and set the `Sensitivity` property to `'on'`.

```
opt = wcgainOptions('Sensitivity','on');
wcst = wcsens(P,C,opt)
```

Examples

The following constructs a feedback loop with a first order plant and a proportional-integral controller. The time constant is uncertain and the model also includes an multiplicative uncertainty. The nominal (input) sensitivity function has a peak of 1.09 at omega = 1.55 rad/sec. Since the plant and controller are single-input / single-output, the input/output sensitivity functions are the same.

```
  delta = ultidyn('delta',[1 1]);
  tau = ureal('tau',5,'range',[4 6]);
  P = tf(1,[tau 1])*(1+0.25*delta);
  C=tf([4 4],[1 0]);
  looptransfer = loopsens(P,C);
  Snom = looptransfer.Si.NominalValue;
  norm(Snom,inf)
  ans =
    1.0864
```

`wcsens` is then used to compute the worst-case sensitivity function as the uncertainty ranges over its possible values. More information about the fields in `wcst.Si` can be found in the `wcgain` help. The `badsystem` field of `wcst.Si` contains the worst case sensitivity function. This worst case sensitivity has a peak of 1.52 at omega = 1.02 rad/sec. The `maxgainunc` field of `wcst.Si` contains the perturbation that corresponds to this worst case sensitivity function.

```
wcst = wcsens(P,C)
wcst =
        Si: [1x1 struct]
        Ti: [1x1 struct]
        So: [1x1 struct]
        To: [1x1 struct]
       PSi: [1x1 struct]
       CSo: [1x1 struct]
    Stable: 1
Swc = wcst.Si.BadSystem;
omega = logspace(-1,1,50);
bodemag(Snom,'-',Swc,'-.',omega);
legend('Nominal Sensitivity','Worst-Case Sensitivity',...
  'Location','SouthEast')
norm(Swc,inf)
ans =
    1.5075
```

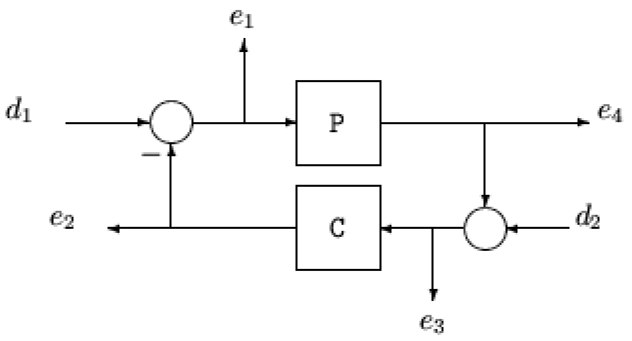
For multi-input/multi-output systems the various input/output sensitivity functions will, in general, be different.

Compatibility Considerations

wcsens is not recommended

Not recommended starting in R2019a

Use of `wcsens` is not recommended. Instead, form the transfer function you want to analyze, and use `wcgain` to obtain the worst-case sensitivity. This approach has improved numeric stability and more reliable results relative to `wcsens`. To form the transfer function, assuming the following control structure, refer to the following diagram and table.



The following table gives the values of the input and output sensitivity functions for this control structure.

Description	**Equation**
Input sensitivity S_i (closed-loop transfer function from d_1 to e_1)	$S_i = (I + CP)^{-1}$
Input complementary sensitivity T_i (closed-loop transfer function from d_1 to e_2)	$T_i = CP(I + CP)^{-1}$
Output sensitivity S_o (closed-loop transfer function from d_2 to e_2)	$S_o = (I + PC)^{-1}$
Output complementary sensitivity T_o (closed-loop transfer function from d_2 to e_4)	$T_o = PC(I + PC)^{-1}$
Input loop transfer function L_i	$L_i = CP$
Output loop transfer function L_o	$L_o = PC$

For an example, see “Worst-Case Sensitivity Functions of Feedback Loops”.

There are no plans to remove `wcsens` at this time.

References

J. Shin, G.J. Balas, and A.K. Packard, “Worst case analysis of the X-38 crew return vehicle flight control system,” *AIAA Journal of Guidance, Dynamics and Control,* vol. 24, no. 2, March-April 2001, pp. 261-269.

See Also

`loopsens` | `robstab` | `usubs` | `wcdiskmargin` | `wcgain`

Introduced before R2006a

wcsigma

Plot worst-case gain of uncertain system

Syntax

```
wcsigma(usys)
wcsigma(usys,w)
wcsigma( ___ ,opts)
```

Description

`wcsigma(usys)` plots the nominal and worst-case gains of the uncertain system `usys` as a function of frequency. For multi-input, multi-output (MIMO) systems, gain refers to the largest singular value of the frequency response matrix. (See `sigma` for more information about singular values.) The plot includes:

- Nominal — Nominal gain of `usys`.
- Worst — The response falling within the uncertainty of `usys` that has the highest peak gain.
- Worst-case gain (lower bound) — The lowest possible worst-case gain at each frequency.
- Worst-case gain (upper bound) — The highest possible gain within the uncertainty at each frequency. This curve represents the envelope produced by finding the highest possible gain at each frequency.
- Sampled Uncertainty — 20 responses randomly sampled from `usys`.

`wcsigma(usys,w)` focuses the plot on the frequencies specified by `w`.

- If `w` is a cell array of the form `{wmin,wmax}`, then `wcsigma` plots the worst-case gains in the range `{wmin,wmax}`.
- If `w` is an array of frequencies, then `wcsigma` plots the worst-case gains at each frequency in the array.

`wcsigma( ___ ,opts)` specifies additional options for the computation. Use `wcOptions` to create `opts`.

Examples

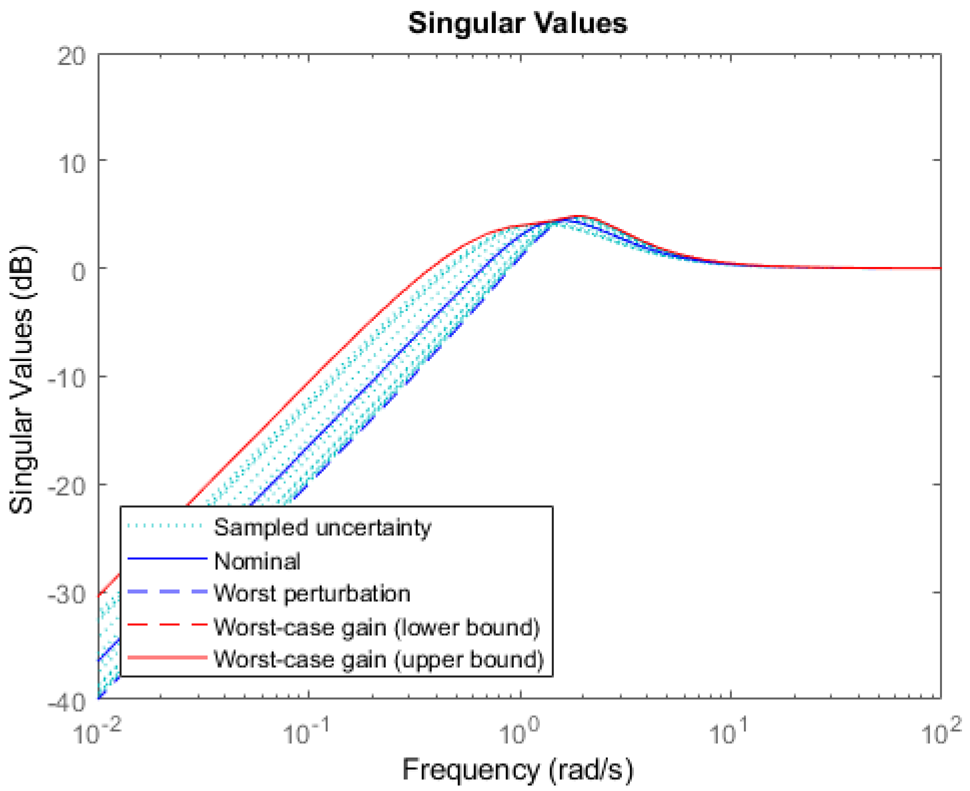
Plot Worst-Case Gain of Uncertain System

Plot the worst-case gain of the following system:

$$sys = \frac{s^2 + 3s}{s^2 + 2s + a}.$$

The uncertain parameter $a = 2 \pm 1$.

```
a = ureal('a',2);
usys = tf([1 3 0],[1 2 a]);
wcsigma(usys)
```

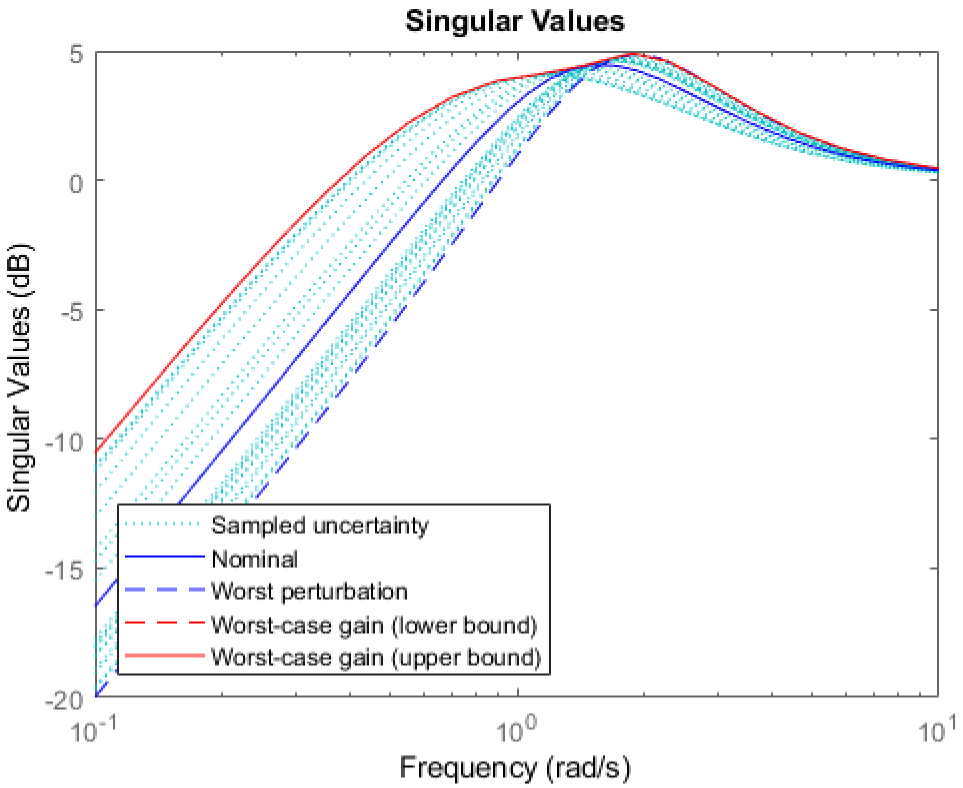


The `Worst` curve identifies the single response within the uncertainty that yields the highest gain at any frequency. The `Worst-case gain (upper bound)` curve is the envelope produced by finding the highest gain within the uncertainty at each frequency.

The `Worst perturbation` curve identifies the combination of uncertain elements within the specified range that yields the highest overall gain. This perturbation corresponds to the `wcu` output of `wcgain`. The `Worst-case gain` curves show the lower and upper bounds on the worst-case gain at each frequency. For any perturbation within the specified uncertainty range, the principal gains (singular values) of the perturbed system lie below the `Worst-case gain (upper bound)` curve.

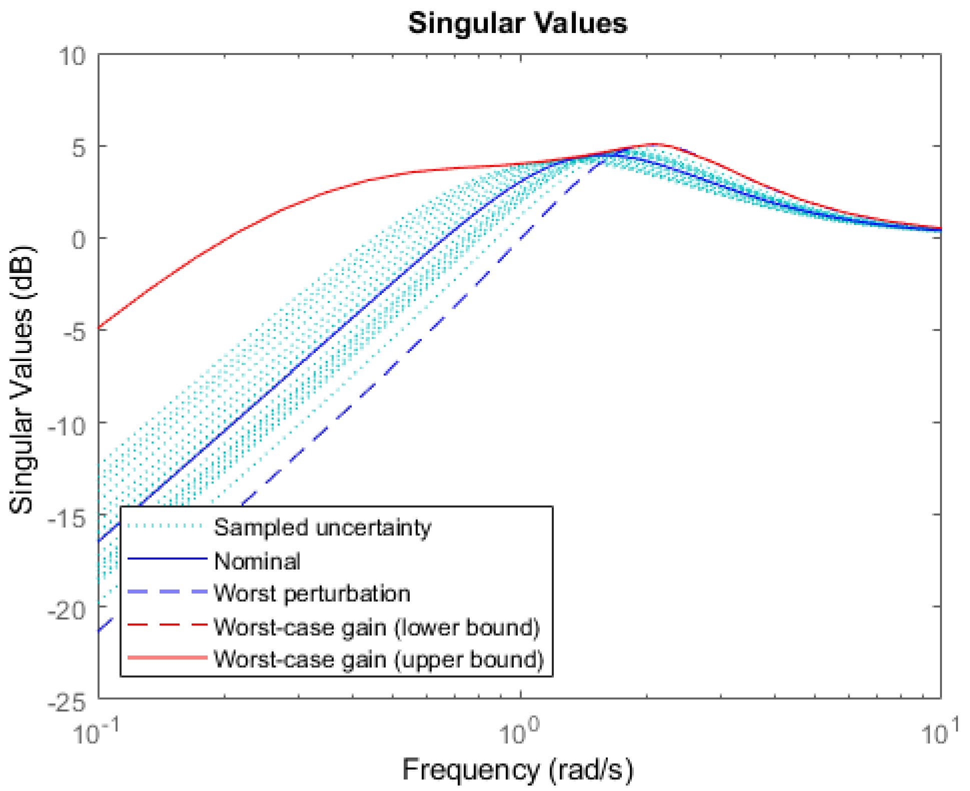
Focus the plot on the region between 0.1 and 10 rad/s.

```
w = {0.1 10};
wcsigma(usys,w)
```



Examine the effect on the worst-case response of increasing the uncertainty range. To do this without changing the uncertainty specified in `usys`, use the `ULevel` option of `wcOptions`. This option scales the normalized uncertainty by the factor you specify. For example, examine the worst-case response a 50% greater uncertainty range.

```
opts = wcOptions('ULevel',1.5);
wcsigma(usys,w,opts)
```



The plot shows that increasing the uncertainty range substantially increases the worst-case gain at low frequencies.

Input Arguments

usys — Dynamic system with uncertainty

uss | ufrd | genss | genfrd

Dynamic system with uncertainty, specified as a `uss`, `ufrd`, `genss`, or `genfrd` model that contains uncertain elements.

For `genss` or `genfrd` models, `wcsigma` uses the current value of any tunable blocks and folds them into the known (not uncertain) part of the model.

`w` — Frequencies

`{wmin,wmax}` | vector

Frequencies at which to plot worst-case gains, specified as the cell array `{wmin,wmax}` or as a vector of frequency values.

- If `w` is a cell array of the form `{wmin,wmax}`, then the function plots the worst-case gains at frequencies ranging between `wmin` and `wmax`.
- If `w` is a vector of frequencies, then the function plots the worst-case gains at each specified frequency. For example, use `logspace` to generate a row vector with logarithmically spaced frequency values.

Specify frequencies in units of rad/`TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model.

`opts` — Options for worst-case gain computation

`wcOptions` object

Options for worst-case computation, specified as an object you create with `wcOptions`. Setting certain options for `mussv` can improve the results of the worst-case calculation. See `wcOptions` for more information.

Example: `wcOptions('ULevel',2,'MussvOptions','m3')`

Algorithms

`wcsigma` uses `wcgain` to compute the worst-case gains. Use the `opts` argument to set options for the `wcgain` algorithm.

`wcsigma` uses `usample` to compute the `Sampled Uncertainty` curves.

See Also

`sigma` | `uss` | `wcOptions` | `wcgain`

Topics

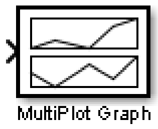
"Robustness and Worst-Case Analysis"

Introduced in R2016b

Block Reference

MultiPlot Graph

Plot multiple signals



Description

The MultiPlot Graph block displays signals in a MATLAB figure.

If the input signal is a vector, then each component of the vector is plotted in a separate axes. Lines are added to the axes in subsequent simulations. The most recent data is plotted in red. Older plots cycle through seven different colors. The block acts as a “hold-on, subplotter.”

There are two buttons in the toolbar menu. The eraser button clears the data from all axes. The export button saves all the visible plot data to the MATLAB workspace in a variable named by the dialog box entry **Variable for Export to Workspace**. The format is a struct array, following the behavior of a `To Workspace` block, using the “Structure, With Time” save format.

The MultiPlot Graph block can be used in conjunction with the Uncertain State Space block to visualize Monte Carlo and worst-case simulation time responses.

Parameters

t-min, t-max

The parameter entries t-min and t-max are the minimum and maximum x-axis limits. t-min and t-max may be vectors corresponding to each subplot.

y-min, y-max

The parameter entries y-min and y-max are the minimum and maximum y-axis limits and similarly may be vector quantities.

Sample time

Sample time corresponds to the sample time at which to collect points.

Title

Specifies the title of the multiplot figure.

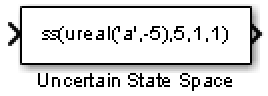
Variable for Export to Workspace

Variable name of the MATLAB object to contain all the visible plot data exported to the MATLAB workspace. The format is a struct array, following the behavior of a To Workspace block, using the "Structure, With Time" save format.

Introduced in R2007a

Uncertain State Space

Specify uncertain system in Simulink



Description

The Uncertain State Space block lets you model parametric and dynamic uncertainty in Simulink. The block accepts uncertain state space (`uss`) models or any model that can be converted to `uss`, such as `umat`, `ureal` and `ultidyn` objects.

Parameters

Uncertain system variable (uss)

Linear state-space model with uncertainty (`uss` object). Specify an `uss` object using one of the following:

- Function or expression that evaluates to an `uss` object. For example:
  - `ss(ureal('a',-5),5,1,1)`
  - `wt*input_unc`, where input_unc is an `ultidyn` object and `wt` and `input_unc` are defined in the MATLAB workspace.
- Variable name, defined in the MATLAB workspace. For example, `unc_sys`, where you define `unc_sys` = `ss(ureal('a',-5),5,1,1)` in the workspace. This returns an `uss` object.
- Model type that can be converted to an `uss` object. For example:
  - LTI models (`tf`, `zpk` and `ss`)
  - Uncertain matrix (`umat`)

- Uncertain real parameters (`ureal`)
- Uncertain dynamics (`ultidyn`).

When the block is in a model with synchronous state control (see the State Control block), you must specify a discrete-time model.

Uncertainty value (struct or [] to use nominal value)

Values of uncertain variables. The `uss` object that you enter in the **Uncertain system variable (uss)** field depends on uncertain variables (`ureal` or `ultidyn` object). Use this field to specify the values of these uncertain variables to use for simulation or linearization. Specify the value as one of the following:

Value	Description
`[]`	Use nominal values.
Structure	Use user-defined values. For example, `struct('a',1)` specifies a value of 1 for the uncertain variable `a`. Use `ufind` and `usample` to generate randomized values of uncertain variables for Monte Carlo simulation. For more information, see “Vary Uncertainty Values Using Individual Uncertain State Space Blocks” and “Vary Uncertainty Values Across Multiple Uncertain State Space Blocks” in the *Robust Control Toolbox User's Guide*.

Initial states (nominal dynamics)

If the nominal value of the uncertain state variable, `unc_sys.NominalValue where unc_sys` is the uncertain system variable specified in the **Uncertain system variable** field, has states, specify the initial condition for these states. The value defaults to zero.

Initial states (uncertain dynamics)

If the uncertain system contains some dynamic uncertainty (`ultidyn`), specify the initial state of these dynamics. The value defaults to zero.

See Also

`ufind` | `ulinearize` | `ultidyn` | `umat` | `ureal` | `usample` | `uss`

Topics

“Simulate Uncertainty Effects”
“Compute Uncertain State-Space Models from Simulink Models”
“Robustness Analysis in Simulink”
“Linearization of Simulink Models with Uncertainty”

Introduced in R2009b

USS System

Import uncertain systems into Simulink

Compatibility

Note USS System block is not recommended. Use Uncertain State Space block instead.

Description

The USS System block accepts USS and UMAT containing `ureal` and `ultidyn` uncertain objects, as well as `ureal` and `ultidyn` objects. An instance of the uncertain system is used in the simulation or linearization. Internally, USS models are converted to their state space equivalent for evaluation.

Parameters

USS system variable

The uncertain object (USS, UMAT, `ureal`, or `ultidyn`) is entered in the USS system variable.

Initial states (nominal dynamics)

If the nominal value for the USS system variable has states, then the initial condition for these states is entered in `Initial states (nominal dynamics).`

Uncertainty value

The values for the uncertain elements are controlled by the `Uncertainty value` menu. If `Nominal` is selected, then the nominal value of the uncertain object is used. If you select `User defined`, then you must enter a MATLAB structure in the `User-defined` `uncertainty (struct)` dialog box. The field names of the structure should correspond to the names of the uncertain atoms within the USS system variable, while the values of the fields are the values used for the uncertain objects (using the command `usubs`). If some of these values are SS objects, then these states are referred to as uncertainty states.

The order of the uncertainty states is determined by the order of atoms in the Uncertainty property of the USS system variable. The state dimension is determined by the actual data in the `User-defined uncertainty` structure. Any extra fields in the `User-defined uncertainty` structure are ignored.

User-defined uncertainty (struc)

If `User defined` is selected from the `Uncertainty value` pop-up menu, then the structure data entered in `User-defined uncertainty (struct)` must contain fields corresponding to every uncertain atom of the USS system variable. Extra fields are ignored. `usimsamp` generates a random instance of each atom in a Simulink model. It returns a structure, suitable for entry in `User-defined uncertainty (struct)`.

Initial states (uncertain dynamics)

The initial condition for the uncertainty states is entered in `Initial states (uncertain dynamics)`.

Introduced in R2007a