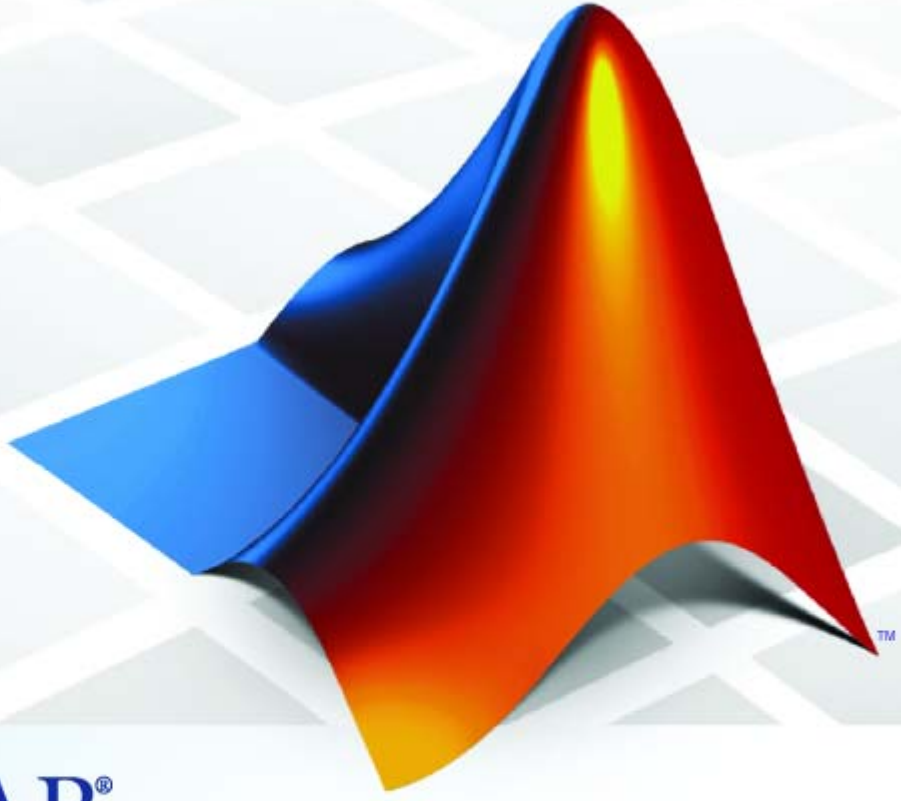
Robust Control Toolbox™ 3
Reference

Gary Balas
Richard Chiang
Andy Packard
Michael Safonov



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Robust Control Toolbox™ Reference

Trademarks

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

Patents

The MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

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)

Contents

1 Function Reference

Function Reference

Frequency-Response Data (FRD) Models (p. 1-10)	Functions for operating on FRD models
Supporting Utilities (p. 1-11)	Additional functions for working with systems containing uncertain elements
LMIs (p. 1-11)	Functions for building and solving systems of Linear Matrix Inequalities
Simulink (p. 1-13)	Functions for using with Simulink models

Uncertain Elements

`ucomplex`	Create uncertain complex parameter
`ucomplexm`	Create uncertain complex matrix
`udyn`	Create unstructured uncertain dynamic system object
`ultidyn`	Create uncertain linear time-invariant object
`ureal`	Create uncertain real parameter

Uncertain Matrices and Systems

`diag`	Diagonalize vector of uncertain matrices and systems
`randatom`	Generate random uncertain `atom` objects
`randumat`	Generate random uncertain `umat` objects
`randuss`	Generate stable, random `uss` objects
`ufrd`	Create uncertain frequency response data (`ufrd`) object, or convert another model type to `ufrd` model
`umat`	Create uncertain matrix
`uss`	Specify uncertain state space models or convert LTI model to uncertain state space model

Manipulation of Uncertain Models

`actual2normalized`	Calculate normalized distance between uncertain atom nominal value and specified value
`gridureal`	Grid `ureal` parameters uniformly over their range
`isuncertain`	Check whether argument is uncertain class type
`lftdata`	Decompose uncertain objects into fixed normalized and fixed uncertain parts
`normalized2actual`	Convert value for atom in normalized coordinates to corresponding actual value
`repmat`	Replicate and tile array
`simplify`	Simplify representation of uncertain object
`squeeze`	Remove singleton dimensions for `umat` objects
`usample`	Generate random samples of uncertain variables
`uss/ssbal`	Scale state/uncertainty while preserving uncertain input/output map of uncertain system
`usubs`	Substitute given values for uncertain elements of uncertain objects

Interconnection of Uncertain Models

`iconnect`	Create empty `iconnect` (interconnection) objects
`icsignal`	Create `icsignal` object of specified dimension
`imp2exp`	Convert implicit linear relationship to explicit input-output relation
`stack`	Construct array by stacking uncertain matrices, models, or arrays
`sysic`	Build interconnections of certain and uncertain matrices and systems

Model Order Reduction

`balancmr`	Balanced model truncation via square root method
`bstmr`	Balanced stochastic model truncation (BST) via Schur method
`hankelmr`	Hankel minimum degree approximation (MDA) without balancing
`hankelsv`	Compute Hankel singular values for stable/unstable or continuous/discrete system
`imp2ss`	System realization via Hankel singular value decomposition
`modreal`	Modal form realization and projection
`ncfmr`	Balanced model truncation for normalized coprime factors

`reduce`	Simplified access to Hankel singular value based model reduction functions
`schurmr`	Balanced model truncation via Schur method
`slowfast`	Slow and fast modes decomposition

Robustness and Worst-Case Analysis

`cpmargin`	Coprime stability margin of plant-controller feedback loop
`gapmetric`	Compute upper bounds on Vinnicombe `gap` and `nugap` distances between two systems
`loopmargin`	Stability margin analysis of LTI and Simulink® feedback loops
`loopsens`	Sensitivity functions of plant-controller feedback loop
`mussv`	Compute bounds on structured singular value (μ)
`mussvextract`	Extract `muinfo` structure returned by `mussv`
`ncfmargin`	Calculate normalized coprime stability margin of plant-controller feedback loop
`popov`	Perform Popov robust stability test
`robopt`	Create options object for use with `robuststab` and `robustperf`
`robustperf`	Robust performance margin of uncertain multivariable system

`robuststab`	Calculate robust stability margins of uncertain multivariable system
`wcgain`	Calculate bounds on worst-case gain of uncertain system
`wcgopt`	Create options object for use with `wcgain`, `wcsens`, and `wcmargin`
`wcmargin`	Worst-case disk stability margins of uncertain feedback loops
`wcnorm`	Worst-case norm of uncertain matrix
`wcsens`	Calculate worst-case sensitivity and complementary sensitivity functions of plant-controller feedback loop

Robustness Analysis for Parameter-Dependent Systems (P-Systems)

`aff2pol`	Convert affine parameter-dependent models to polytopic models
`decay`	Quadratic decay rate of polytopic or affine P-systems
`ispsys`	True for parameter-dependent systems
`pdlstab`	Assess robust stability of polytopic or parameter-dependent system
`pdsimul`	Time response of parameter-dependent system along given parameter trajectory
`polydec`	Compute polytopic coordinates with respect to box corners

`psinfo`	Inquire about polytopic or parameter-dependent systems created with `psys`
`pvec`	Specify range and rate of variation of uncertain or time-varying parameters
`pvinfo`	Describe parameter vector specified with `pvec`
`quadperf`	Compute quadratic H performance of polytopic or parameter-dependent system
`quadstab`	Quadratic stability of polytopic or affine parameter-dependent systems

Controller Synthesis

`augw`	State-space or transfer function plant augmentation for use in weighted mixed-sensitivity H and H loopshaping design
`h2hinfsyn`	Mixed H/H synthesis with pole placement constraints
`h2syn`	H control synthesis for LTI plant
`hinfsyn`	Compute H optimal controller for LTI plant
`loopsyn`	H optimal controller synthesis for LTI plant
`ltrsyn`	LQG loop transfer-function recovery (LTR) control synthesis
`mixsyn`	H mixed-sensitivity synthesis method for robust control loopshaping design

`mkfilter`	Generate Bessel, Butterworth, Chebyshev, or RC filter
`ncfsyn`	Loop shaping design using Glover-McFarlane method

μ-Synthesis

`cmsclsyn`	Approximately solve constant-matrix, upper bound μ-synthesis problem
`dkitopt`	Create options object for use with `dksyn`
`dksyn`	Robust controller design using μ-synthesis
`drawmag`	Mouse-based tool for sketching and fitting
`fitfrd`	Fit frequency response data with state-space model
`fitmagfrd`	Fit frequency response magnitude data with minimum-phase state-space model using log-Chebychev magnitude design
`genphase`	Fit single-input/single-output magnitude data with real, rational, minimum-phase transfer function

Sampled-Data Systems

`sdhinfnorm`	Compute L norm of continuous-time system in feedback with discrete-time system
`sdhinfsyn`	Compute H controller for sampled-data system
`sdlsim`	Time response of sampled-data feedback system

Gain Scheduling

`hinfgs`	Synthesis of gain-scheduled H controllers

Frequency-Response Data (FRD) Models

`frd/loglog`	Log-log scale plot of `frd` objects
`frd/rcond`	LAPACK reciprocal condition estimator of `frd` object
`frd/schur`	Schur decomposition of `frd` object
`frd/semilogx`	Semilog scale plot of `frd` object
`frd/svd`	Singular value decomposition of `frd` object

Supporting Utilities

`bilin`	Multivariable bilinear transform of frequency (*s* or *z*)
`dmplot`	Interpret disk gain and phase margins
`mktito`	Partition LTI system into two-input/two-output system
`sectf`	State-space sector bilinear transformation
`skewdec`	Form skew-symmetric matrix
`symdec`	Form symmetric matrix

LMIs

LMI Systems

`getlmis`	Internal description of LMI system
`lmiedit`	Specify or display systems of LMIs as MATLAB® expressions
`lmiterm`	Specify term content of LMIs
`lmivar`	Specify matrix variables in LMI problem

`newlmi`	Attach identifying tag to LMIs
`setlmis`	Initialize description of LMI system

LMI Characteristics

`dec2mat`	Given values of decision variables, derive corresponding values of matrix variables
`decinfo`	Describe how entries of matrix variable X relate to decision variables
`decnbr`	Total number of decision variables in system of LMIs
`lmiinfo`	Information about variables and term content of LMIs
`lminbr`	Return number of LMIs in LMI system
`mat2dec`	Extract vector of decision variables from matrix variable values
`matnbr`	Number of matrix variables in system of LMIs

LMI Solvers

`defcx`	Help specify c^Tx objectives for mincx solver
`feasp`	Compute solution to given system of LMIs
`gevp`	Generalized eigenvalue minimization under LMI constraints
`mincx`	Minimize linear objective under LMI constraints

Validation of Results

`evallmi`	Given particular instance of decision variables, evaluate all variable terms in system of LMIs
`showlmi`	Return left- and right-hand sides of LMI after evaluation of all variable terms

Modification of Systems of LMIs

`bilin`	Multivariable bilinear transform of frequency (*s* or *z*)
`dmplot`	Interpret disk gain and phase margins
`mktito`	Partition LTI system into two-input/two-output system
`sectf`	State-space sector bilinear transformation
`skewdec`	Form skew-symmetric matrix
`symdec`	Form symmetric matrix

Simulink

`ufind`	Find uncertain variables in Simulink model
`ulinearize`	Linearize Simulink model with Uncertain State Space block

2

Functions — Alphabetical List

actual2normalized

Purpose

Calculate normalized distance between uncertain atom nominal value and specified value

Syntax

```
NDIST = actual2normalized(A,V)
```

Description

`NDIST = actual2normalized(A,V)` is the normalized distance between the nominal value of the uncertain atom `A` and the given value `V`. If `A` is a `ureal`, then `NDIST` may be positive or negative, reflecting that `V` is greater than, or less than the nominal value. If `A` is any other class of uncertain atom, then `ndist` is nonnegative.

If `V` is an array of values, then `NDIST` is an array of normalized distances.

The robustness margins computed in `robuststab` and `robustperf` 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 when the normalized distance of the uncertain element values 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.

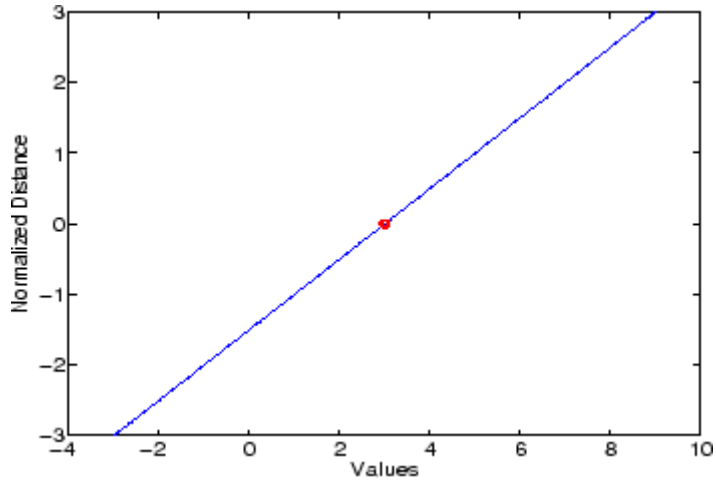
Create uncertain real parameters with a range that is symmetric about the nominal value, where each end point is 1 unit from the nominal. Points that lie inside the range are less than 1 unit from the nominal, while points that lie outside the range are greater than 1 unit from the nominal.

```
a = ureal('a',3,'range',[1 5]);
actual2normalized(a,[1 3 5])
ans =
   -1.0000   -0.0000    1.0000
actual2normalized(a,[2 4])
ans =
```

```
   -0.5000    0.5000
actual2normalized(a,[0 6])
ans =
   -1.5000    1.5000
```

Graph the normalized distance for several values. The nominal point is shown as a red circle. Note that the relationship between a normalized distance and a numerical difference is linear.

```
values = linspace(-3,9,250);
ndist = actual2normalized(a,values);
plot(values,ndist)
```



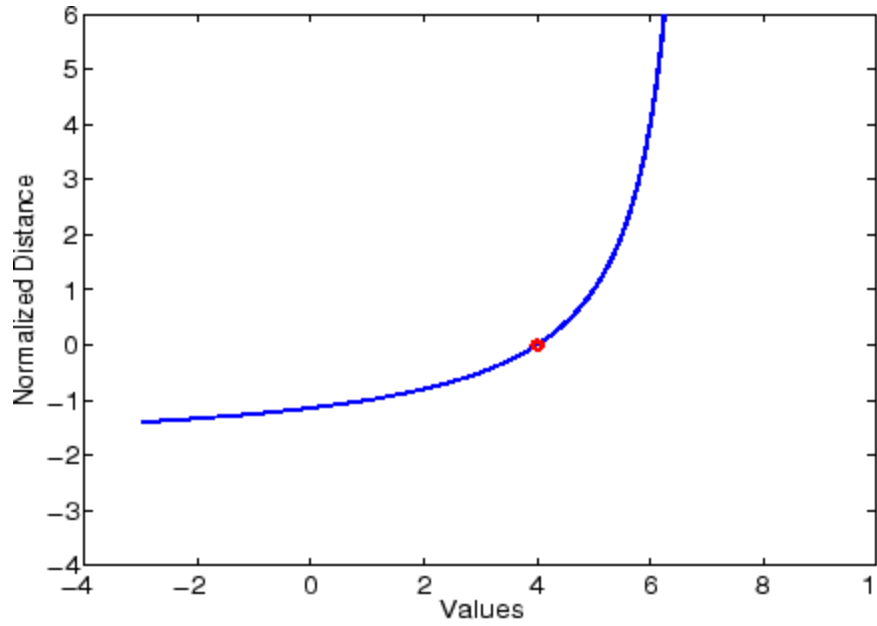
Uncertain Real Parameter with Nonsymmetric Range

Next, create an unasymmetric parameter. It still is true that 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('a',4,'range',[1 5]);
actual2normalized(a,[1 4 5])
ans =
   -1.0000    0.5000    1.0000
actual2normalized(a,[2 4.5])
ans =
   -0.5000    0.7500
actual2normalized(a,[0 6])
ans =
   -1.5000    1.5000
```

Graph the normalized distance for several values. Note that the relationship between normalized distance and numerical difference is very nonlinear.

```
ndistu = actual2normalized(au,values);
plot(values,ndistu,au.NominalValue,0,'ro')
```



Algorithm

For details on the normalize distance, see "Normalizing Functions for Uncertain Atoms" in the *Robust Control Toolbox™ User's Guide*.

See Also

`normalized2actual`

`robuststab`

`robustperf`

Purpose

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

$$E(p)\dot{x} = A(p)x + B(p)u \quad \textbf{(2-1)}$$

$$y = C(p)x + D(p)u \quad \textbf{(2-2)}$$

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 2-1 and Equation 2-2 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`

Purpose

State-space or transfer function plant augmentation for use in weighted mixed-sensitivity H_∞ and H_2 loopshaping design

Syntax

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

Description

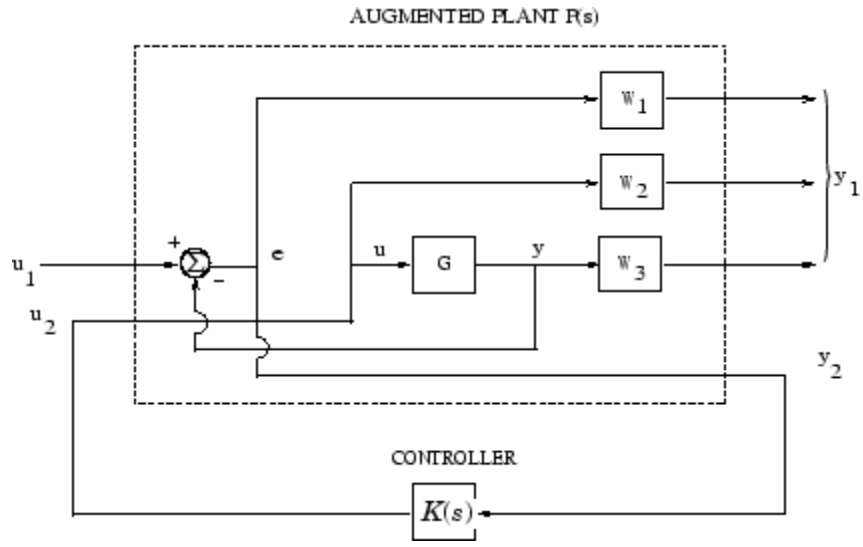
`P = AUGW(G,W1,W2,W3)` computes a state-space model of an augmented LTI plant $P(s)$ with weighting functions $W_1(s)$, $W_2(s)$, and $W_3(s)$ penalizing the error signal, control signal and output signal respectively (see block diagram) so that the closed-loop transfer function matrix is the weighted mixed sensitivity

$$T_{y_1u_1} \triangleq \begin{bmatrix} W_1S \\ W_2R \\ W_3T \end{bmatrix}$$

where S, R and T are given by

$$S = (I + GK)^{-1}$$
$$R = K(I + GK)^{-1}$$
$$T = GK(I + GK)^{-1}$$

The LTI systems S and T are called the *sensitivity* and *complementary sensitivity,* respectively.



Plant Augmentation

For dimensional compatibility, each of the three weights W_1, W_2 and W_3 must be either empty, a scalar (SISO) or have respective input dimensions N_Y, N_U, and N_Y where G is N_Y-by-N_U. If one of the weights is not needed, you may simply assign an empty matrix [ ]; e.g., `P = AUGW(G,W1,[],W3)` is $P(s)$ as in the "Algorithm" on page 2-8 section below, but without the second row (without the row containing `W2`).

Algorithm

The augmented plant $P(s)$ produced by is

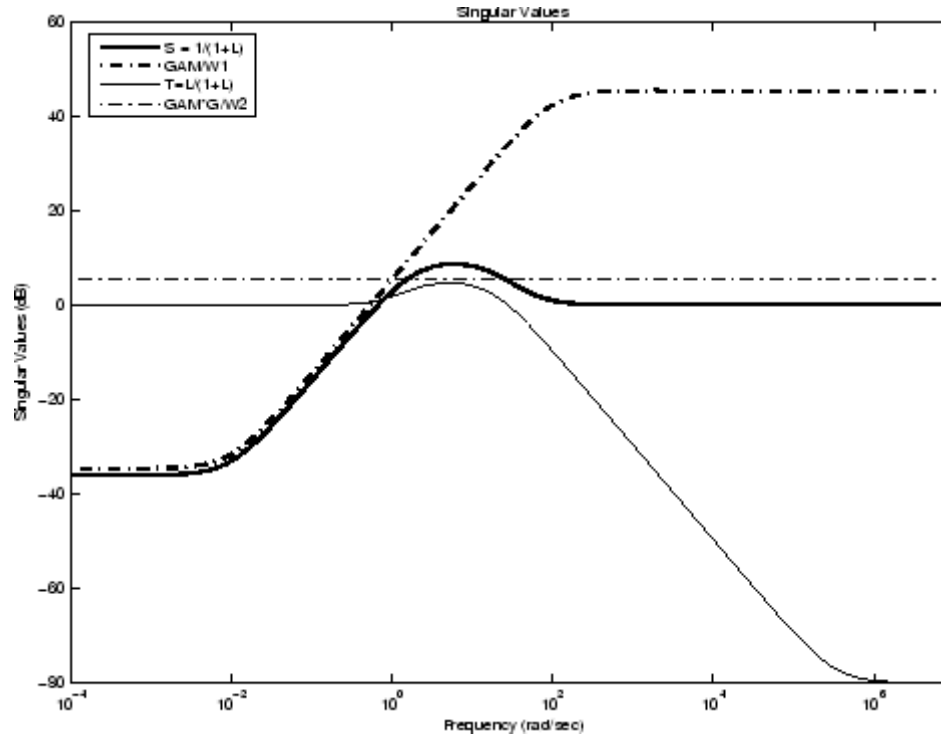
$$P(s) = \left[\begin{array}{c|c} W_1 & -W_1 G \\ 0 & W_2 \\ 0 & W_3 G \\ \hline I & -G \end{array}\right]$$

Partitioning is embedded via `P=mktito(P,NY,NU)`, which sets the InputGroup and OutputGroup properties of P 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);
```

Examples

```
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);
[K,CL,GAM]=hinfsyn(P); [K2,CL2,GAM2]=h2syn(P);
L=G*K; S=inv(1+L); T=1-S; sigma(S,'k',GAM/W1,'k-.',T,'r',GAM*G/W2,'r-.')
legend('S = 1/(1+L)','GAM/W1', 'T=L/(1+L)','GAM*G/W2',2)
```



Limitations

The transfer functions G, W_1, W_2 and W_3 must be proper, i.e., bounded as $s \to \infty$ or, in the discrete-time case, as $z \to \infty$. Additionally, W_1,

W_2 and W_3 should be stable. The plant G should be stabilizable and detectable; else, `P` will not be stabilizable by any `K`.

See Also

`h2syn`

`hinfsyn`

`mixsyn`

`mktito`

Purpose

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, σ_i.

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* $\| G-GRED \|_\infty$ for well-conditioned model reduced problems [1]:

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

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'`overides `ORDER` input.
'Weights'	`{Wout,Win}` cell array	Optimal 1-by-2 cell array of LTI weights `Wout` (output) and `Win` (input). Defaults are 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 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 $\|\infty$) • 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.

Algorithm

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)

$$L_p = U_p \, \Sigma_p^{1/2}$$

$$L_o = U_q \, \Sigma_q^{1/2}$$

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

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

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

5 Finally,

$$\left[\begin{array}{c|c} \hat{A} & \hat{B} \\ \hline \hat{C} & \hat{D} \end{array}\right] = \left[\begin{array}{c|c} S^T{}_{L,BIG} A S_{R,BIG} & S^T{}_{L,BIG} B \\ \hline C S_{R,BIG} & D \end{array}\right]$$

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

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:

```
rand('state',1234); randn('state',5678);G = rss(30,5,4);
[g1, redinfo1] = balancmr(G); % display Hankel SV plot
                              % and prompt for order (try 15:20)
[g2, redinfo2] = balancmr(G,20);
[g3, redinfo3] = balancmr(G,[10:2:18]);
[g4, redinfo4] = balancmr(G,'MaxError',[0.01, 0.05]);
rand('state',12345); randn('state',6789);
wt1 = rss(6,5,5); wt1.d = eye(5)*2;
wt2 = rss(6,4,4); wt2.d = 2*eye(4);
[g5, redinfo5] = balancmr(G, [10:2:18], 'weight',{wt1,wt2});
for i = 1:5
    figure(i); eval(['sigma(G,g' num2str(i) ');']);
end
```

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

`reduce`

`schurmr`

`hankelmr`

`bstmr`

`ncfmr`

`hankelsv`

Purpose

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 $(s \to z)$ or $(s \to \tilde{s})$.

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

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 sampling period.
`'FwdRec'`	forward rectangular: $s = \frac{z-1}{T}$ `AUG` = T, the sampling period.

Bilinear Transform Types (Continued)

Method	Type of bilinear transform
`'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_Bilin'`	`METHOD = 'G_Bilin'`, general bilinear, continuous-time to continuos-time: $s = \frac{\alpha\tilde{s}+\delta}{\gamma\tilde{s}+\beta}$ `AUG` = $[\alpha\ \beta\ \gamma\ \delta]$

Examples

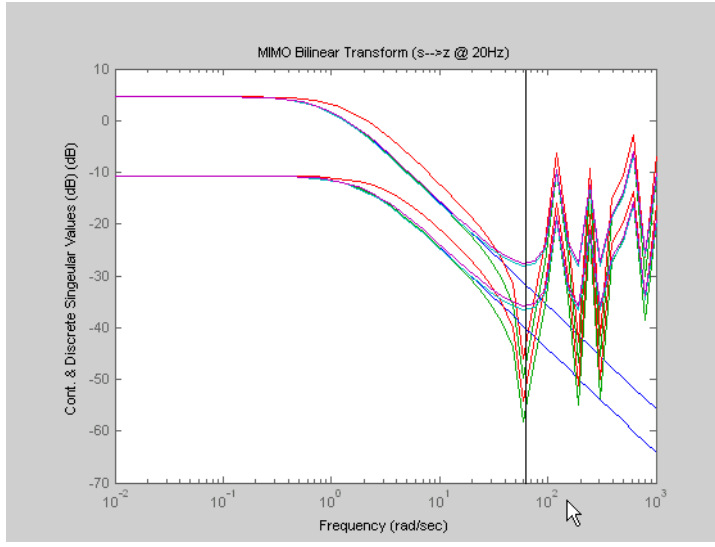
Example 1. Tustin continuous s-plane to discrete z-plane transforms

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



Comparison of Four Bilinear Transforms from Example 1

Example 2. Bilinear continuous to continuous pole-shifting 'S_ftjw'

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

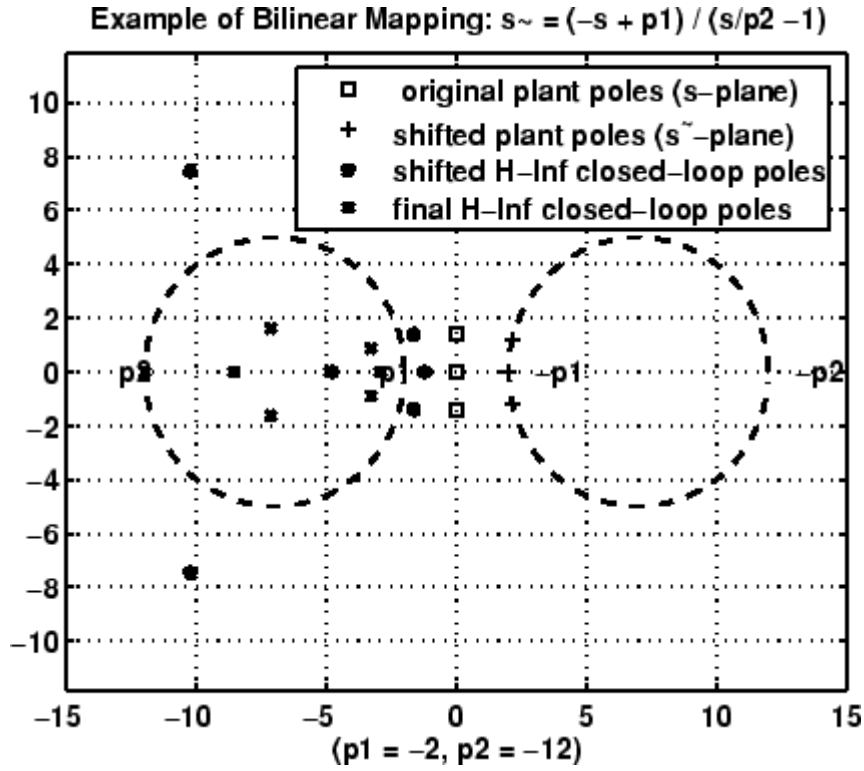
$$G(s) = \frac{1}{s^2(s^2+2)}$$

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**

Algorithm

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

$$\left[\begin{array}{c|c} A_b & B_b \\ \hline C_b & D_b \end{array}\right] = \left[\begin{array}{c|c} (\beta A - \delta I)(\alpha I + \gamma A)^{-1} & (\alpha\beta - \gamma\delta)(\alpha I - \gamma A)^{-1} B \\ \hline C(\alpha I - \gamma A)^{-1} & D + \gamma C(\alpha I - \gamma A)^{-1} B \end{array}\right]$$

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`

Purpose

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* $\| \texttt{GRED-1(G-GRED)} \|_\infty$ or *relative error* $\| \texttt{G^-1(G-GRED)} \|_\infty$ for well-conditioned model reduction problems [1]:

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

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*`'`overides `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)`

Argument	Description
	• `REDINFO.UnstabSV` (Hankel SV of unstable part of G)

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

Algorithm

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,

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

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

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* $(W^*(s))^{-1}G(s)$.

$$V_A = [V_{R,SMALL}, \overbrace{V_{L,BIG}}^{k}]$$

$$V_D = [\overbrace{V_{R,BIG}}^{k}, V_{L,SMALL}]$$

4 Find the SVD of $(V^T_{L,BIG} V_{R,BIG}) = U \Sigma$ ςT

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,

$$\left[\begin{array}{c|c} \hat{A} & \hat{B} \\ \hline \hat{C} & \hat{D} \end{array}\right] = \left[\begin{array}{c|c} S^T_{L,BIG} A S_{R,BIG} & S^T_{L,BIG} B \\ \hline C S_{R,BIG} & D \end{array}\right]$$

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*ω-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.

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:

```
rand('state',1234); randn('state',5678);
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
```

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

reduce

balancmr

hankelmr

schurmr

ncfmr

hankelsv

complexify

Purpose

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 (`robuststab`) 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'`.

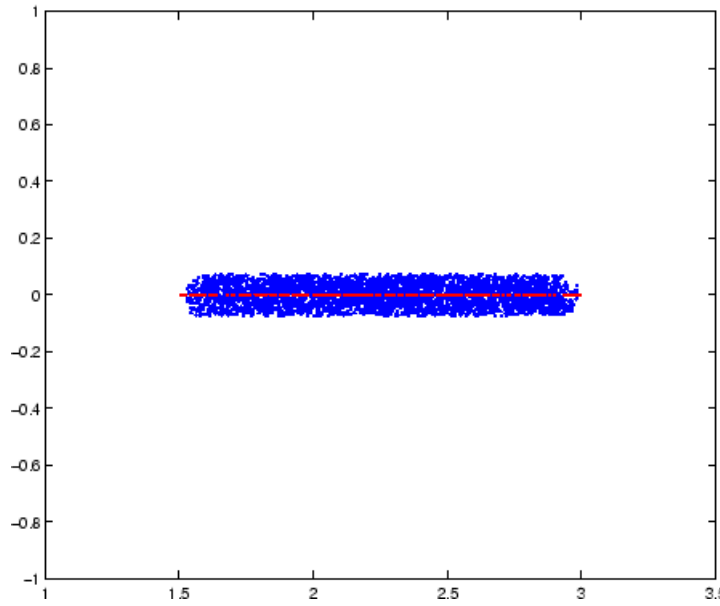
`MC = complexify(M,alpha,'ultidyn')` is the same, except that gain-bounded `ultidyn` atoms are used instead of `ucomplex` atoms. The `ultidyn` atom has its `Bound` equal to `alpha*diff(Range)`.

Examples

See Robust Control Toolbox demo “Getting Reliable Estimates of Robustness Margins” for an example of how complexify is used in robustness analysis.

For illustrative purposes only, create a uncertain real parameter, cast it to a uncertain matrix, and apply a 10% complexification. Finally, 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.

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



See Also

`icomplexify`

`robuststab`

Purpose

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.

Algorithm

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

`dksyn`

`hinfsyn`

`mussv`

`robuststab`

`robustperf`

cpmargin

Purpose

Coprime stability margin of plant-controller feedback loop

Syntax

```
[MARG,FREQ] = cpmargin(P,C)
[MARG,FREQ] = cpmargin(P,C,TOL)
```

Description

`[MARG,FREQ] = cpmargin(P,C)` calculates the normalized coprime factor/gap metric robust stability of the multivariable feedback loop consisting of `C` in negative feedback with `P`. `C` should only be the compensator in the feedback path, not any reference channels, if it is a two degree-of-freedom (2-DOF) architecture. The output `MARG` contains upper and lower bound for the normalized coprime factor/gap metric robust stability margin. `FREQ` is the frequency associated with the upper bound.

`[MARG,FREQ] = cpmargin(P,C,TOL)` specifies a relative accuracy `TOL` for calculating the normalized coprime factor/gap metric robust stability margin. (`TOL=1e-3` by default).

See Also

Comprehensive analysis of feedback loops

`gapmetric`

`wcmargin`

Purpose

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`

decay

Purpose

Quadratic decay rate of polytopic or affine P-systems

Syntax

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

Description

For affine parameter-dependent systems

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

or polytopic systems

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

`decay` returns the quadratic decay rate drate, i.e., the smallest $\alpha \in \mathbf{R}$ 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

`quadstab`

`pdlstab`

`psys`

Purpose

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.

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

$$X = \begin{pmatrix} x_1 & 0 & 0 \\ 0 & x_1 & 0 \\ 0 & 0 & x_1 \end{pmatrix}, \qquad Y = \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}$$

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

lmivar

mat2dec

dec2mat

decnbr

Purpose

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`

Purpose

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

`mat2dec`

`decnbr`

`decinfo`

defcx

Purpose

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

`mincx`

`decinfo`

Purpose

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

newlmi

lmiedit

lmiinfo

Purpose

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

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

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

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

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

See Also

`lmivar`

`setmvar`

`lmiinfo`

diag

Purpose

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`

Purpose

Create options object for use with `dksyn`

Syntax

```
opt = dkitopt
opt = dkitopt('name1',value1,'name2',value2,...)
```

Description

`opt=dkitopt` creates an options object `opt` of class `dkitopt`, used to define user-specified options in the μ-synthesis command `dksyn`. All properties of `opt` are set to their default values.

`opt = dkitopt('name1',value1,'name2',value2,...)` accepts inputs as one or more Property/Value pairs to set user-specified values of individual properties of `opt`. Property names specification is not case-insensitive, and only enough characters to uniquely specify the property name are required.

This table lists the `dkitopt` object properties.

Object Property	Description
`FrequencyVector`	Frequency vector used for analysis. Default is an empty matrix ([]) which results in the frequency range and number of points chosen automatically.
`InitialController`	Controller used to initiate first iteration. Default is an empty SS object.
`AutoIter`	Automated μ-synthesis mode. Default is `'on'`.
`DisplayWhileAutoIter`	Displays iteration progress in `AutoIter` mode. Default is `'off'`.
`StartingIterationNumber`	Starting iteration number. Default is 1.
`NumberOfAutoIterations`	Number of D-K iterations to perform. Default is 10.
`MixedMU`	Accounts for real-valued uncertain parameters for μ-synthesis. For systems with atleast one real-valued uncertain parameter, closed-loop robust performance may improve when the option is set to `'on'`. Default is `'off'`.

Object Property	Description
`AutoScalingOrder`	State order for fitting D-scaling and G-scaling data for real or complex μ-synthesis. Default is [5 2], 5th order D-scalings and 2nd order G-scalings.
`AutoIterSmartTerminate`	Automatic termination of iteration procedure based on progress of design iteration. Default is `'on'`.
`AutoIterSmartTerminateTol`	Tolerance used by `AutoIterSmartTerminate`. Default is 0.005.
`Default`	Structure of property default values.
`Meaning`	Structure text description of each property.

If the `AutoIter` property is set to `'off'`, the D-K iteration procedure is interactive. You are prompted to fit the D-Scale data and provide input on the control design process.

If the `AutoIterSmartTerminate` property is `on`, and a stopping criteria (described below) is satisfied, the iteration performed by `dksyn` will terminate before reaching the specified number of automated iterations (value of `NumberOfAutoIterations`). The stopping criteria involves the objective value (peak value, across frequency, of the upper bound for μ) in the current iteration, denoted v_0, as well as the previous two iterations, (denoted v_{-1} and v_{-2}) and the value of `AutoIterSmartTerminateTol`. If

$$\left|v_0 - v_{-1}\right| < AutoIterSmartTerminateTol * v_0$$

and

$$\left|v_{-1} - v_{-2}\right| < AutoIterSmartTerminateTol * v_0$$

then the stopping criteria is satisfied (for lack of progress). The stopping criteria is also satisfied if

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

which captures a significant increase (undesirable) in the objective.

Examples

This example creates a `dkitopt` options object called `opt` with default property values.

```
opt = dkitopt
Property Object Values:
             FrequencyVector: []
           InitialController: [0x0 ss]
                    AutoIter: 'on'
         DisplayWhileAutoIter: 'off'
       StartingIterationNumber: 1
        NumberOfAutoIterations: 10
                      MixedMU: 'off'
             AutoScalingOrder: [5 2]
        AutoIterSmartTerminate: 'on'
    AutoIterSmartTerminateTol: 0.0050
                      Default: [1x1 struct]
                      Meaning: [1x1 struct]
```

The properties can be modified directly with assignment statements: here user-specified values for the frequency vector, the number of iterations, and the maximum state dimension of the D-scale fittings are set.

```
opt.FrequencyVector = logspace(-2,3,80);
opt.NumberOfAutoIterations = 16;
opt.AutoScalingOrder = 16;
opt
Property Object Values:
             FrequencyVector: [1x80 double]
           InitialController: [0x0 ss]
                    AutoIter: 'on'
         DisplayWhileAutoIter: 'off'
       StartingIterationNumber: 1
        NumberOfAutoIterations: 16
                      MixedMU: 'off'
             AutoScalingOrder: 16
```

```
   AutoIterSmartTerminate: 'on'
AutoIterSmartTerminateTol: 0.0050
                  Default: [1x1 struct]
                  Meaning: [1x1 struct]
```

The same property/value pairs may be set with a single call to `dkitopt`.

```
opt = dkitopt('FrequencyVector',logspace(-2,3,80),...
              'NumberOfAutoIterations',16,...
              'AutoScalingOrder',9);
```

Algorithm

The `dksyn` command stops iterating before the total number of automated iterations (`'NumberOfAutoIterations'`) if `'AutoIterSmartTerminate'` is set to `'on'` and a stopping criterion is satisfied. The stopping criterion involves the m(*i*) value of the current *i*th iteration, `m(`*i*`-1)` and `m(`*i*`-2)`, the previous two iterations and the options property `'AutoIterSmartTerminateTol'`. The *D-K* iteration procedure automatically terminates if the difference between each of the three μ values is less than the relative tolerance of `AutoIterSmartTerminateTol` xμ(*i*) or the current μ value μ(*i*) has increased relative to the μ value of the previous iteration μ(*i*-1) by 20x`AutoIterSmartTerminateTol`.

When the system contains some real-valued uncertain parameters and `MixedMU` is set to `'on'`, the `dksyn` command takes into account that the uncertain parameters are real and this may result in improved robust performance.

See Also

`dksyn`

`h2syn`

`hinfsyn`

`mussv`

`robopt`

`robuststab`

`robustperf`

`wcgopt`

Tutorials

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

Purpose

Robust controller design 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 = robustperf(clp);
bnd = 1/bnd.LowerBound
```

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

Field	Description
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]);
```

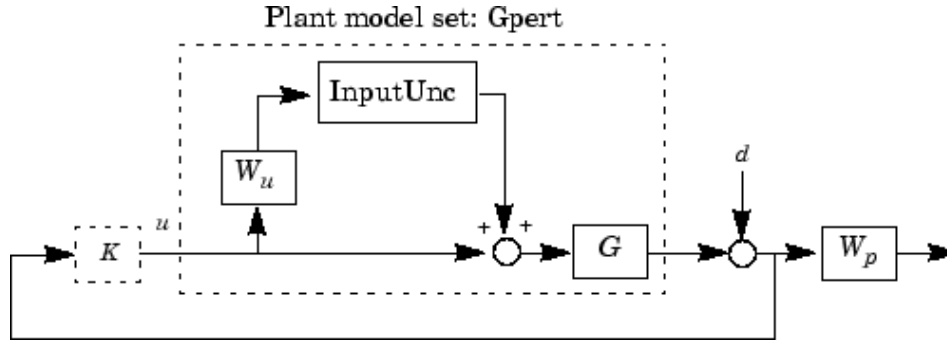
The model itself is uncertain. At low frequency, below 2 rad/s, it can vary up to 25% from its nominal value. Around 2 rad/s the percentage variation starts to increase and reaches 400% at approximately 32 rad/s. The percentage model uncertainty is represented by the weight `Wu` which corresponds to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



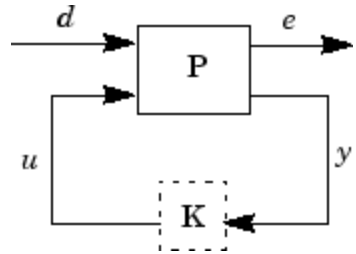
The sensitivity function, S, is defined as $S = \frac{1}{1+PK}$ where `P` is the plant model and `K` is the controller. A weighted sensitivity minimization problem selects a weight `Wp`, which corresponds to the *inverse* of the desired sensitivity function of the closed-loop system as a function of 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 $d \rightarrow e$) 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 =
    0.6819
```

The controller K achieves a robust performance μ value bnd of 0.6819. Therefore you have achieved the robust performance objectives for the given problem.

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

```
[rpnorm,wcf,wcu,report] = robustperf(clp);
disp(report{1})
Uncertain system, clp, achieves robust performance.
The analysis showed clp can tolerate 146% of the model uncertainty
and achieve the performance and stability objectives.
A model uncertainty exists of size 146% that results in
a peak gain performance of 0.686 at 0.569 rad/s.
```

Algorithm

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 dkitopt 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

$$\bar{\sigma}(D_f(j\omega)F_L(P,K)(j\omega)D_f^{-1}(j\omega$$

and

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

are shown for comparison.

(In the first iteration, this step is skipped.) The rational $\hat{D}$ 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 $\hat{D}$, 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 `dkitopt` 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 `dkitopt` 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 `dkitopt` 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 δI) 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.

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.

References

- 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.
- 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.
- 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.
- 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.
- 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

`dkitopt`

`h2syn`

`hinfsyn`

Comprehensive analysis of feedback loop

`mktito`

`mussv`

`robuststab`

`robustperf`

`wcgain`

`wcsens`

`wcmargin`

Tutorials

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

dmplot

Purpose

Interpret disk gain and phase margins

Syntax

```
dmplot
dmplot(diskgm)
[dgm,dpm] = dmplot
```

Description

`dmplot` plots disk gain margin (`dgm`) and disk phase margin (`dpm`). 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.

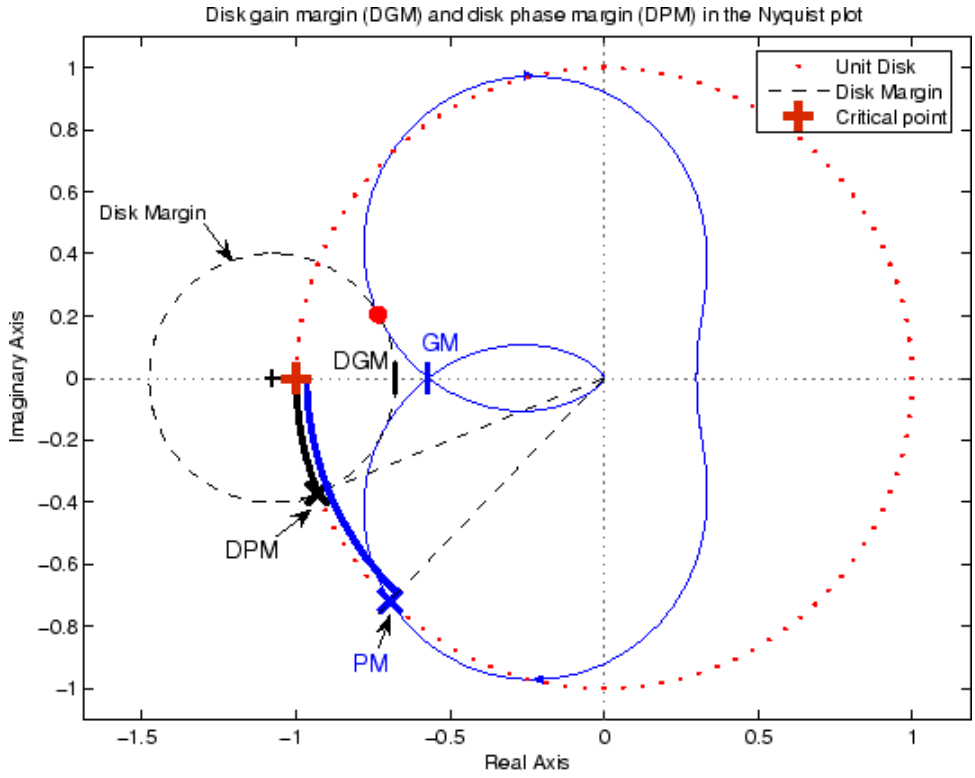
Examples

When you call `dmplot` (without an argument), the resulting plot 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)

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

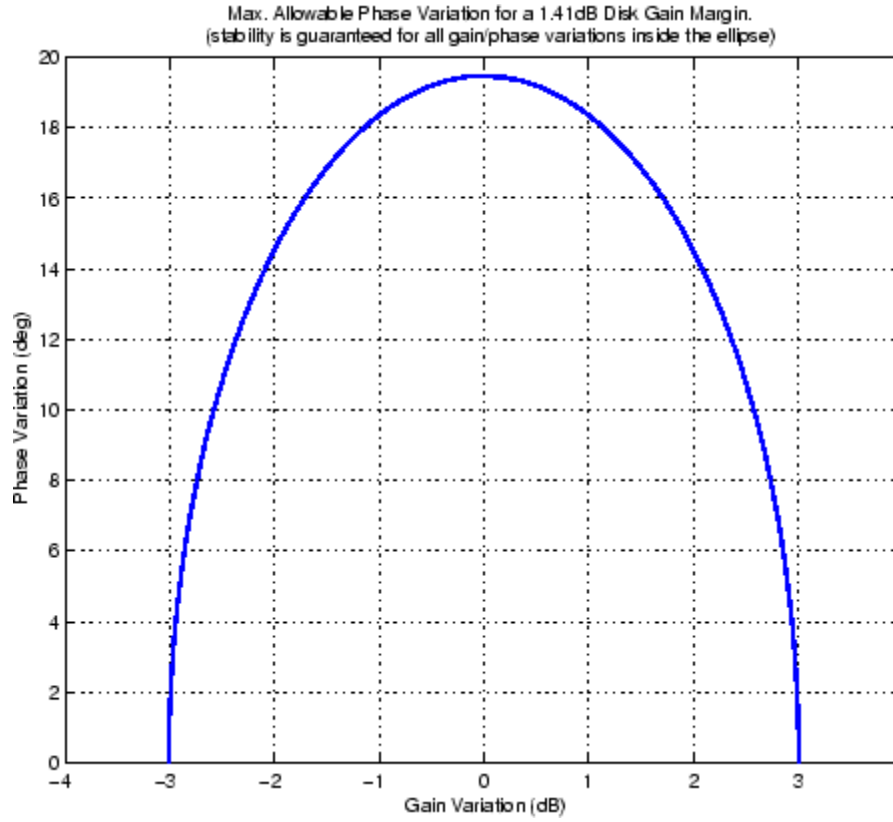
- The Nyquist plot of L corresponds to the blue line.
- The unit disk corresponds to the dotted red line.

- 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, respectively. The disk margins provide a lower bound on classical gain and phase margins.
- The disk margin circle, represented by the dashed black line, corresponds to the largest disk centered at `(DGM + 1/DGM)/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 phase variation allowable, 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.

```
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

Comprehensive stability analysis of feedback loops

`wcmargin`

Purpose

Mouse-based tool for sketching and fitting

Syntax

```
[sysout,pts] = drawmag(data)
[sysout,pts] = drawmag(data,init_pts)
```

Description

`drawmag` interactively uses the mouse in the plot window to create `pts` (the `frd` object) and `sysout` (a stable minimum-phase `ss` object), which approximately fits the frequency response (magnitude) in `pts`.

Input arguments:

`data`	Either a frequency response object that is plotted as a reference, or a constant matrix of the form $[x_{min}\ x_{max}\ y_{min}\ y_{max}]$ specifying the plot window on the data.
`init_pts`	Optional `frd` objects of initial set of points

Output arguments:

`sysout`	Stable, minimum-phase `ss` object that approximately fits, in magnitude, the `pts data`.
`pts`	Frequency response of points.

While `drawmag` is running, all interaction with the program is through the mouse and/or the keyboard. The mouse, if there is one, must be in the plot window. The program recognizes several commands:

- Clicking the mouse button adds a point at the cross-hairs. If the cross-hairs are outside the plotting window, the points are plotted when the fitting, windowing, or replotting mode is invoked. Typing `a` is the same as clicking the mouse button.
- Typing `r` removes the point with frequency nearest that of the cross-hairs.
- Typing any integer between `0` and `9` fits the existing points with a transfer function of that order. The fitting routine approximately

minimizes the maximum error in a log sense. The new fit is displayed along with the points, and the most recent previous fit, if it exists.

- Typing `w` uses the cross-hair location as the initial point in creating a window. Moving the cross-hairs and clicking the mouse or pressing any key then gives a second point at the new cross-hair location. These two points define a new window on the data, which is immediately replotted. This is useful in fine tuning parts of the data. You can call windowing repeatedly.
- Typing `p` simply replots the data using a window that covers all the current data points as well as whatever was specified in `in`. Typically used after windowing to view all the data.
- Typing `k` invokes the keyboard using the `keyboard` command. You should exercise caution when using this option, as it can wreak havoc on the program if variables are changed.

See Also

`ginput`

`loglog`

Purpose

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 $X_1, \ldots, X_K$. The "evaluation" consists of replacing all terms involving $X_1, \ldots, X_K$ 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-hand 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 $X_1, \ldots, X_K$, 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-hand 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

showlmi

setmvar

dec2mat

mat2dec

Purpose

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

$$N^T L x N \le M^T R(x) M, \tag{2-3}$$

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

Minimize t subject to $N^T L(x) N - M^T R(x) M \le tI$

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 $x = (x_1, \ldots, x_N)$ 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 $R = 109$. Setting `options(3)` to a negative value activates the "flexible bound" mode. In this mode, the feasibility radius is initially set to 108, 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 2-210 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

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

$$A_1^T P + P A_1 < 0 \qquad \textbf{(2-4)}$$

$$A_2^T P + P A_2 < 0 \qquad \textbf{(2-5)}$$

$$A_3^T P + P A_3 < 0 \qquad \textbf{(2-6)}$$

with data

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

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

To assess feasibility with `feasp`, first enter the LMIs Equation 2-4 - Equation 2-6:

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

```
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
```

Then call `feasp` to find a feasible decision vector:

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

This returns `tmin = -3.1363`. Hence Equation 2-4 - Equation 2-6 is feasible and 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, type

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

This returns

$$P = \begin{pmatrix} 270.8 & 126.4 \\ 126.4 & 155.1 \end{pmatrix}$$

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

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

The third entry `10` of `options` sets the feasibility radius to 10 while the third argument `-1` sets the target value for `tmin`. This yields `tmin = -1.1745` and a matrix `P` with largest eigenvalue $\lambda_{\max}(P) = 9.6912$.

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

`mincx`

`gevp`

`dec2mat`

Purpose

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.

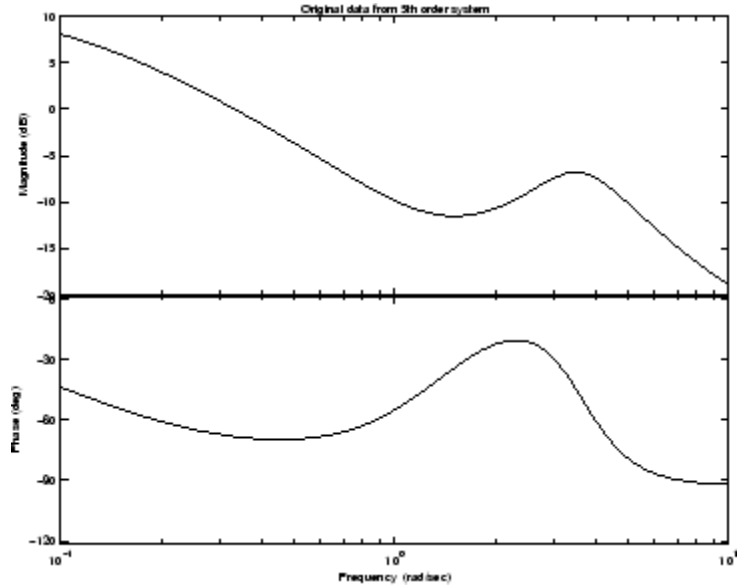
`B = fitfrd(A,N,RD)` forces the relative degree of `B` to be `RD`. `RD` must be a nonnegative integer. The default value for `RD` is 0. If `A` is a row (or column) then `RD` can be a vector of the same size as well, specifying the relative degree of each entry of `B`. If `RD` is a scalar, then it specifies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

You can use the `fitfrd` command to fit D-scale data. For example, 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-');
```

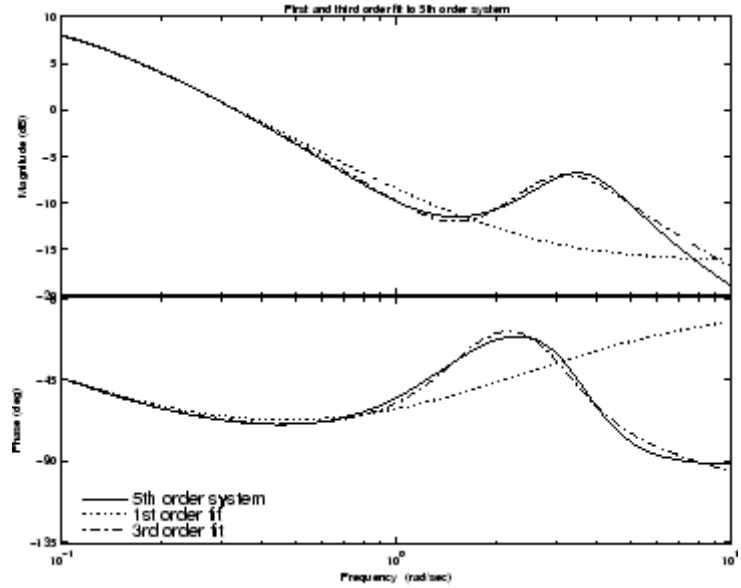


You can try to fit the frequency response D-scale data `sysg` with a first-order system, `b1`. Similarly, you can fit the D-scale data with a third-order system, `b3`.

```
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-.')
```



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`

Purpose

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

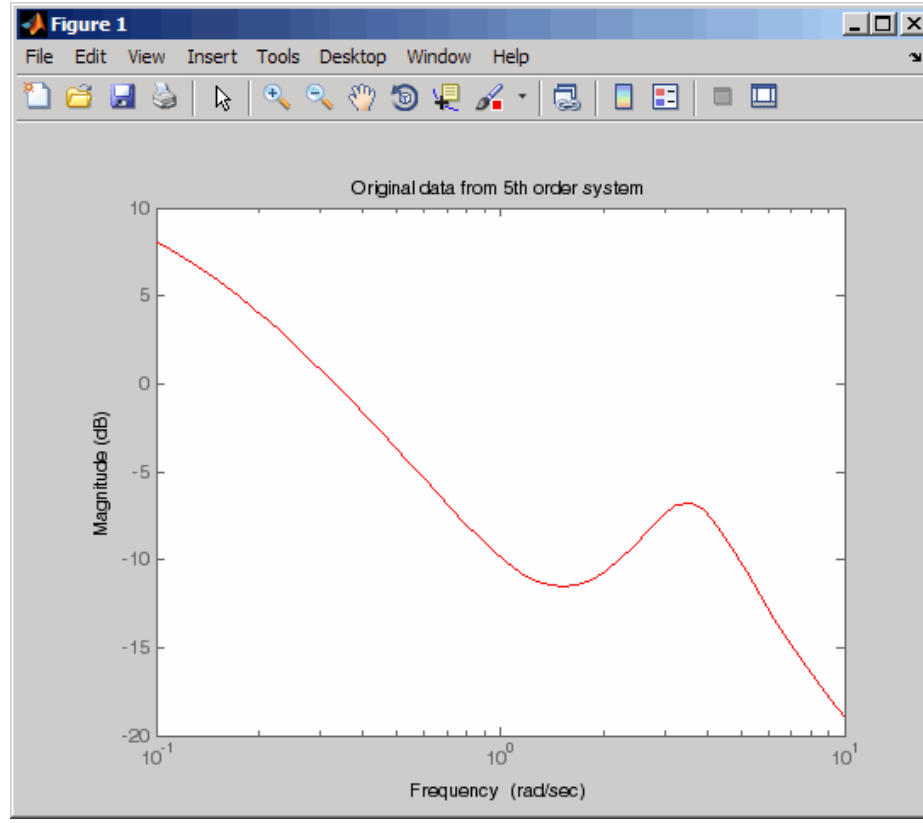
where w denotes the frequency.

Examples

Fit frequency response magnitude data with a stable, minimum-phase statespace model:

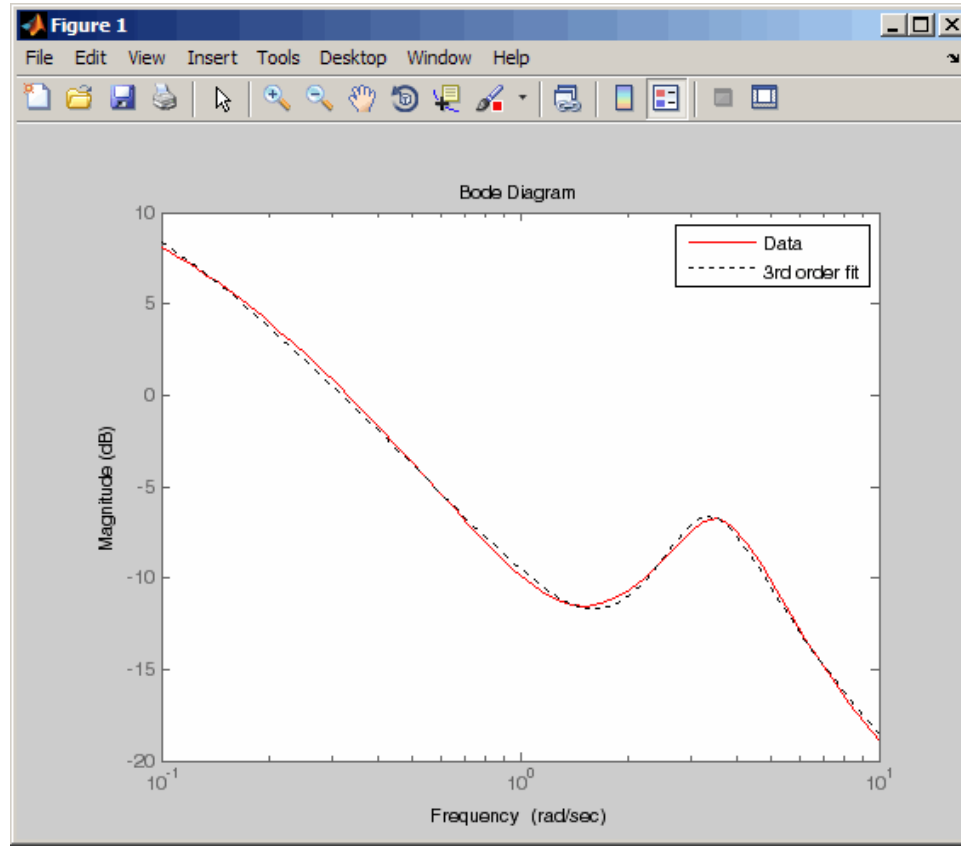
1 Create frequency response magnitude data from a fifth-order system.

```
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');
```



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

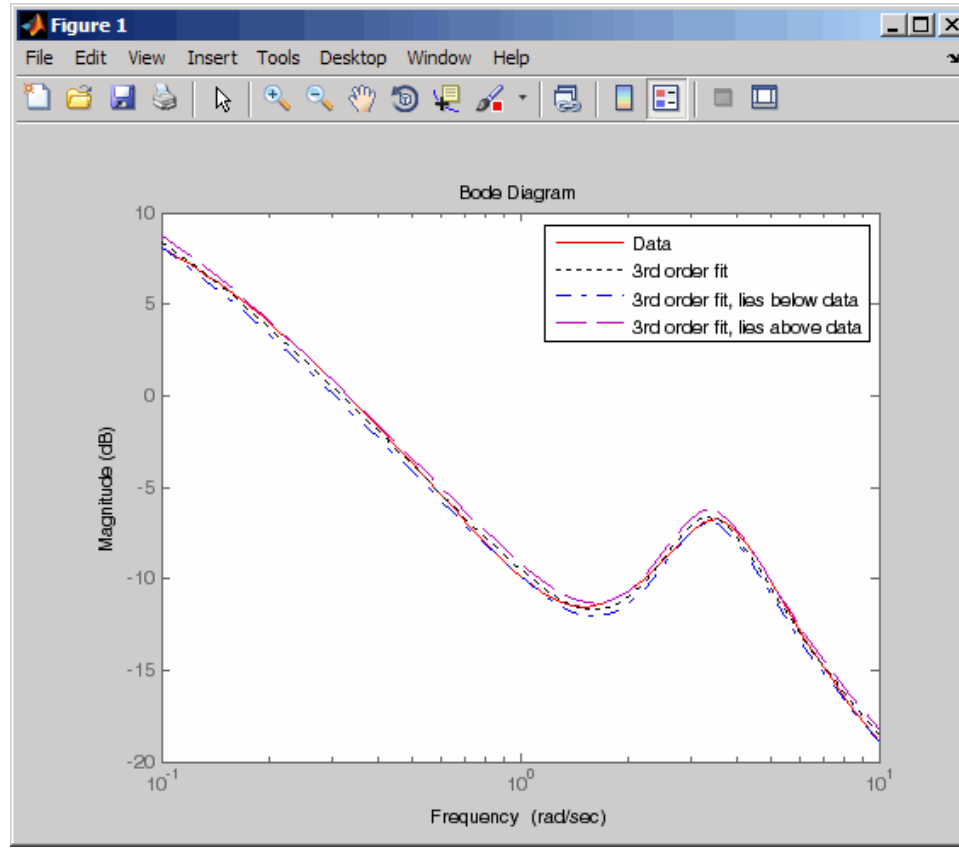
```
ord = 3;
b1 = fitmagfrd(sysg,ord);
b1g = frd(b1,omega);
bodemag(sysg,'r',b1g,'k:');
```



3 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);
```

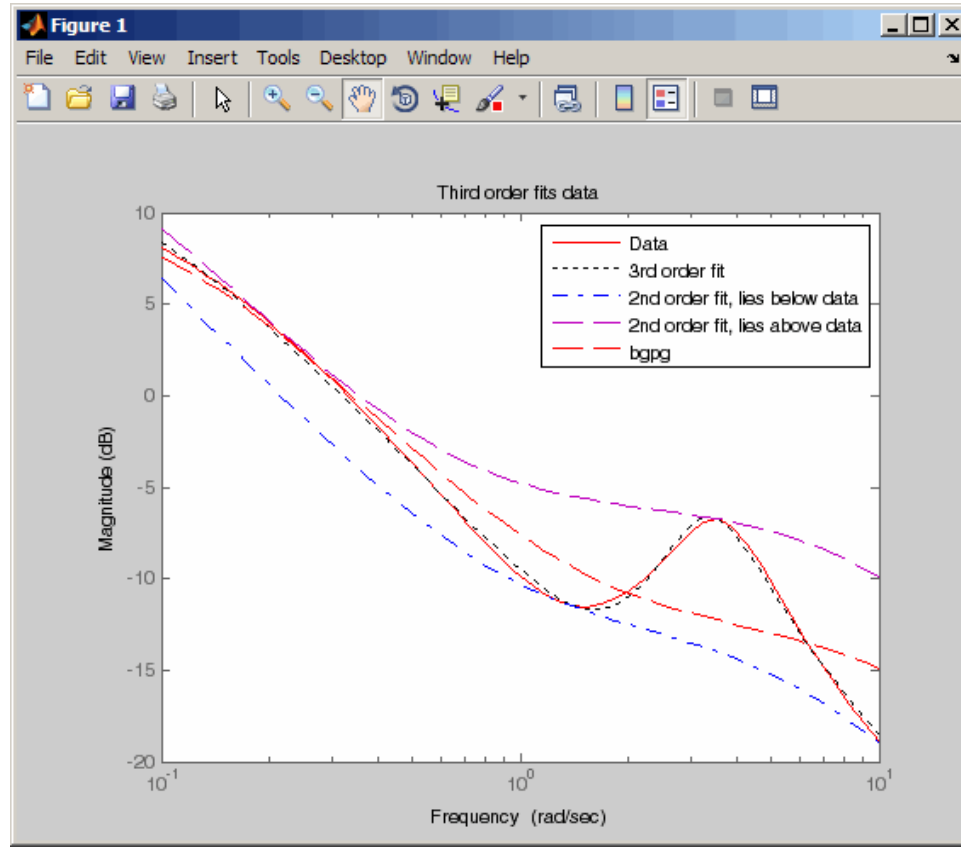
```
b3g = frd(b3,omega);
bodemag(sysg,'r',b1g,'k:',b2g,'b-.',b3g,'m--')
```



4 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--')
```



Algorithm

`fitmagfrd` uses a version of log-Chebychev 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)`.

Limitations

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

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`

gapmetric

Purpose

Compute upper bounds on Vinnicombe gap and nugap distances between two systems

Syntax

```
[gap,nugap] = gapmetric(p0,p1)
[gap,nugap] = gapmetric(p0,p1,tol)
```

Description

`[gap,nugap] = gapmetric(p0,p1)` calculates upper bounds on the gap and nugap (Vinnicombe) metric between systems p0 and p1. The gap and nugap values lie between 0 and 1. A small value (relative to 1) implies that any controller that stabilizes p0 will likely stabilize p1, and, moreover, that the closed-loop gains of the two closed-loop systems will be similar. A gap or nugap of 0 implies that p0 equals p1, and a value of 1 implies that the plants are far apart. The input and output dimensions of p0 and p1 must be the same.

`[gap,nugap] = gapmetric(p0,p1,tol)` specifies a relative accuracy for calculating the gap metric and nugap metric. The default value for tol is 0.001. The computed answers are guaranteed to satisfy

```
gap-tol < gapexact(p0,p1) <=  gap
```

Examples

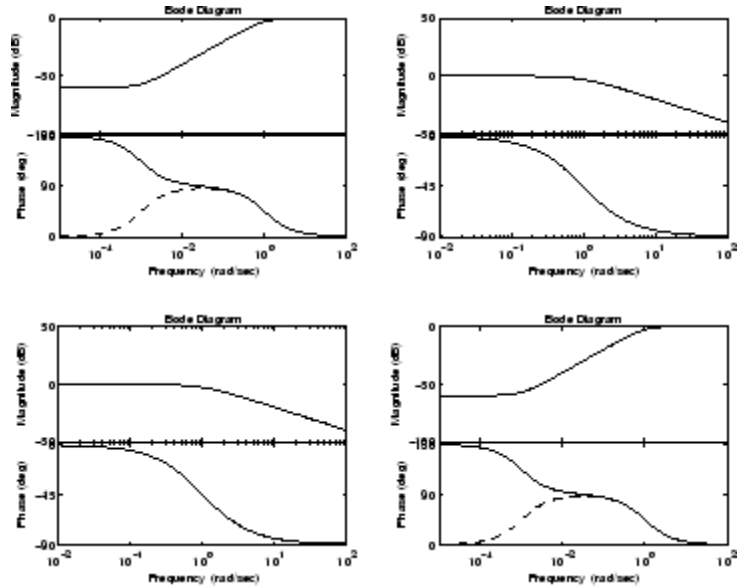
Consider two plant models. One plant is an unstable, first-order, with transfer function 1/(s-0.001) and the other plant is stable first-order with transfer function 1/(s+0.001).

```
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* in the gap and nugap metrics. Intuitively, this is obvious, because, for instance, the feedback controller K=1 stabilizes both plants and renders the closed-loop systems nearly identical.

```
[g,ng] = gapmetric(p1,p2)
g =
    0.0029
ng =
    0.0020
```

```
K = 1;
H1 = loopsens(p1,K);
H2 = loopsens(p2,K);
subplot(2,2,1); bode(H1.Si,'-',H2.Si,'--');
subplot(2,2,2); bode(H1.Ti,'-',H2.Ti,'--');
subplot(2,2,3); bode(H1.PSi,'-',H2.PSi,'--');
subplot(2,2,4); bode(H1.CSo,'-',H2.CSo,'--');
```



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

```
p3 = tf([50],[1 50]);
p4 = tf([8],[1 8])*p3;
```

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

```
[g,ng] = gapmetric(p3,p4)
g =
    0.6156
ng =
    0.6147
```

Algorithm

`gap` and `nugap` compute the gap and ν gap metrics between two LTI objects. Both quantities give a numerical value δ(`p0`,`p1`) between 0 and 1 for the distance between a nominal system `p0` (G_0) and a perturbed system p1 (G_1). The gap metric was introduced into the control literature by Zames and El-Sakkary 1980, and exploited by Georgiou and Smith 1990. The ν gap metric was derived by Vinnicombe 1993. For both of these metrics the following robust performance result holds from Qui and Davidson 1992, and Vinnicombe 1993

$$\arcsin b(G_1,K_1) \geq \arcsin b(G_0,K_0) - \arcsin \delta(G_0,G_1) - \arcsin \delta(K_0,K_1)$$

where

$$b(G,K) = \left\| \begin{bmatrix} I \\ K \end{bmatrix} (I - GK)^{-1} \begin{bmatrix} G & I \end{bmatrix} \right\|_\infty^{-1}$$

The interpretation of this result is that if a nominal plant G_0 is stabilized by controller K_0, with "stability margin" $b(G_0,K_0)$, then the stability margin when G_0 is perturbed to G_1 and K_0 is perturbed to K_1 is degraded by no more than the above formula. Note that $1/b(G,K)$ is also the signal gain from disturbances on the plant input and output to the input and output of the controller. The ν `gap` is always less than or equal to the `gap`, so its predictions using the above robustness result are tighter.

To make use of the gap metrics in robust design, weighting functions need to be introduced. In the above robustness result, G needs to be replaced by W_2GW_1 and K by $W_1^{-1}KW_2^{-1}$ (similarly for G_0, G_1, K_0 and K_1). This makes the weighting functions compatible with the weighting structure in the H_∞ loop shaping control design procedure (see `loopsyn` and `ncfsyn` for more details).

The computation of the gap amounts to solving 2-block H_∞ problems (Georgiou, Smith 1988). The particular method used here for solving the H_∞ problems is based on Green et al., 1990. The computation of the `nugap` uses the method of Vinnicombe, 1993.

References

Georgiou, T.T., "On the computation of the gap metric, " *Systems Control Letters,* Vol. 11, 1988, p. 253-257

Georgiou, T.T., and M. Smith, "Optimal robustness in the gap metric," *IEEE Transactions on Automatic Control,* Vol. 35, 1990, p. 673-686

Green, M., K. Glover, D. Limebeer, and J.C. Doyle, "A J-spectral factorization approach to H_∞ control," *SIAM J. of Control and Opt.,* 28(6), 1990, p. 1350-1371

Qiu, L., and E.J. Davison, "Feedback stability under simultaneous gap metric uncertainties in plant and controller," *Systems Control Letters,* Vol. 18-1, 1992 p. 9-22

Vinnicombe, G., "Measuring Robustness of Feedback Systems," PhD Dissertation, Department of Engineering, University of Cambridge, 1993.

Zames, G., and El-Sakkary, "Unstable systems and feedback: The gap metric," *Proceedings of the Allerton Conference,* October 1980, p. 380-385

See Also

Comprehensive analysis of feedback loop

`loopsyn`

`ncfsyn`

`robuststab`

`wcsens`

`wcmargin`

genphase

Purpose

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`

Purpose

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

`setlmis`

`lmivar`

`lmiterm`

`newlmi`

Purpose

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:

$$C(x) < D(x) \tag{2-7}$$

$$0 < B(x) \tag{2-8}$$

$$A(x) < \lambda B(x) \tag{2-9}$$

where $C(x) < D(x)$ and $A(x) < \lambda B(x)$ denote systems of LMIs. Provided that Equation 2-7 and Equation 2-8 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 2-7 to Equation 2-9 for $\lambda = 1$. The LMIs involving λ are called the *linear-fractional constraints* while Equation 2-7 and Equation 2-8 are regular LMI constraints. The number of linear-fractional constraints Equation 2-9 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 2-9 *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

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

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

$$I < P \tag{2-10}$$

$$A_1^T P + P A_1 < \alpha P \tag{2-11}$$

$$A_2^T P + P A_2 < \alpha P \tag{2-12}$$

$$A_3^T P + P A_3 < \alpha P \tag{2-13}$$

To set up this problem for `gevp`, first specify the LMIs Equation 2-11 to Equation 2-13with 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 2-11 to Equation 2-13, 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:

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

Remark

Generalized eigenvalue minimization problems involve standard LMI constraints Equation 2-7 and linear fractional constraints Equation 2-9. 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 2-8 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-hand 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`

Purpose

Grid ureal parameters uniformly over their range

Syntax

```
B = gridreal(A,N)
[B,SampleValues] = gridreal(A,N)
[B,SampleValues] = gridreal(A,Names,N)
[B,SampleValues] = gridreal(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

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 change by +/- 30 percent.

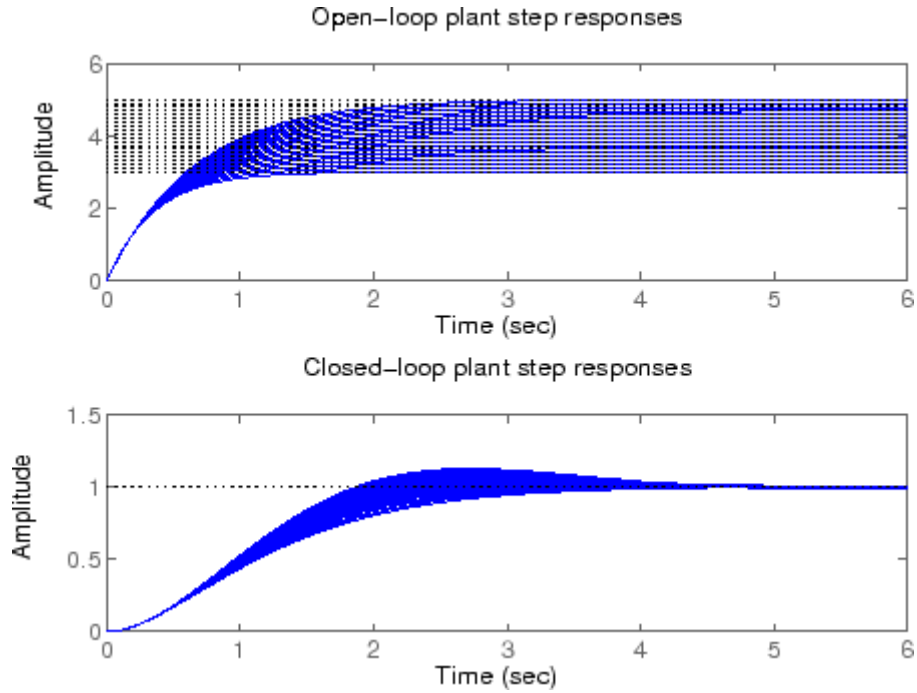
```
gamma = ureal('gamma',4);
tau = ureal('tau',.5,'Percentage',30);
```

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)
```



It clearly illustrates the low-frequency closed-loop insensitivity achieved by the PI control system.

Multi-Parameter Example

The next example illustrates the different options in gridding high-dimensional (e.g., `n` greater than 2) parameter spaces. An uncertain matrix, `m`, is constructed from four uncertain real parameters, `a`, `b`, `c` and `d`, each making up the individual entries.

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

In the first case, the (`a`,`b`) space is gridded at five places, and the (`c`,`d`) space at three places. The uncertain matrix `m` is evaluated at these 15 grid-points resulting in the matrix `m1`.

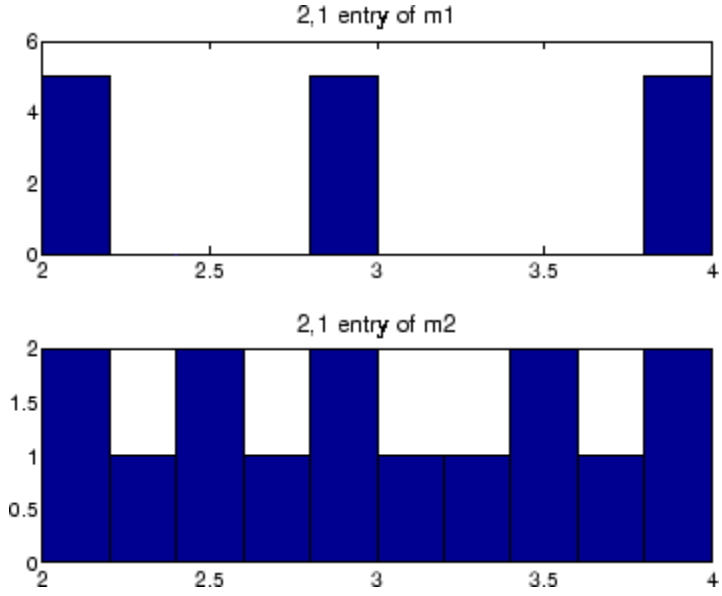
```
m1 = gridureal(m,{'a';'b'},5,{'c';'d'},3);
```

In the second case, the (a,b,c,d) space is gridded at 15 places, and the uncertain matrix m is sampled at these 15 points. The resulting matrix is m2.

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

The (2,1) entry of m is just the uncertain real parameter c. Below, you see the histogram plots 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 (which is also c) takes on 15 distinct values uniformly through its range.

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



See Also

`usample`

`usubs`

Purpose

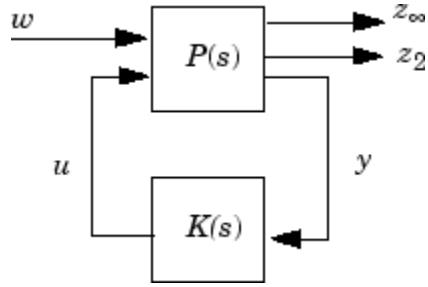
Mixed H_2/H_∞ synthesis with pole placement constraints

Syntax

```
[gopt,h2opt,K,R,S] = hinfmix(P,r,obj,region,dkbnd,tol)
```

Description

`h2hinfyn` performs multi-objective output-feedback synthesis. The control problem is sketched in this figure.



If $T_\infty(s)$ and $T_2(s)$ denote the closed-loop transfer functions from w to z_∞ and z_2, respectively, `hinfmix` computes a suboptimal solution of the following synthesis problem:

Design an LTI controller $K(s)$ that minimizes the mixed H_2/H_∞ criterion

$$\alpha\|T_\infty\|_\infty^2 + \beta\|T_2\|_2^2$$

subject to

- $\|T_\infty\|_{[[\text{BULLET}]]} < \gamma_0$
- $\|T_2\|_2 < \nu_0$
- The closed-loop poles lie in some prescribed LMI region D.

Recall that $\|.\|\infty$ and $\|.\|_2$ denote the H_∞ norm (RMS gain) and H_2 norm of transfer functions.

`P` is any SS, TF, or ZPK LTI representation of the plant $P(s)$, and `r` is a three-entry vector listing the lengths of z_2, y, and u. Note that z_∞ and/or z_2 can be empty. The four-entry vector `obj` = [γ_0, ν_0, α, ß] specifies the

H_2/H_∞ constraints and trade-off criterion, and the remaining input arguments are optional:

- `region` specifies the LMI region for pole placement (the default `region = []` is the open left-half plane). Use `lmireg` to interactively build the LMI region description `region`
- `dkbnd` is a user-specified bound on the norm of the controller feedthrough matrix D_K. The default value is 100. To make the controller $K(s)$ strictly proper, set `dkbnd = 0`.
- `tol` is the required relative accuracy on the optimal value of the trade-off criterion (the default is 10^{-2}).

The function `h2hinfsyn` returns guaranteed H_∞ and H_2 performances `gopt` and `h2opt` as well as the `SYSTEM` matrix `K` of the LMI-optimal controller. You can also access the optimal values of the LMI variables R, S via the extra output arguments `R` and `S`.

A variety of mixed and unmixed problems can be solved with `hinfmix`. In particular, you can use `hinfmix` to perform pure pole placement by setting `obj = [0 0 0 0]`. Note that both z_∞ and z_2 can be empty in such case.

References

Chilali, M., and P. Gahinet, "H_∞ Design with Pole Placement Constraints: An LMI Approach," to appear in *IEEE Trans. Aut. Contr.*, 1995.

Scherer, C., "Mixed H2 H-infinity Control," to appear in *Trends in Control: A European Perspective*, volume of the special contributions to the ECC 1995.

See Also

`lmireg`

`msfsyn`

Purpose

H_2 control synthesis for LTI plant

Syntax

```
[K,CL,GAM,INFO]=H2SYN(P,NMEAS,NCON)
```

Description

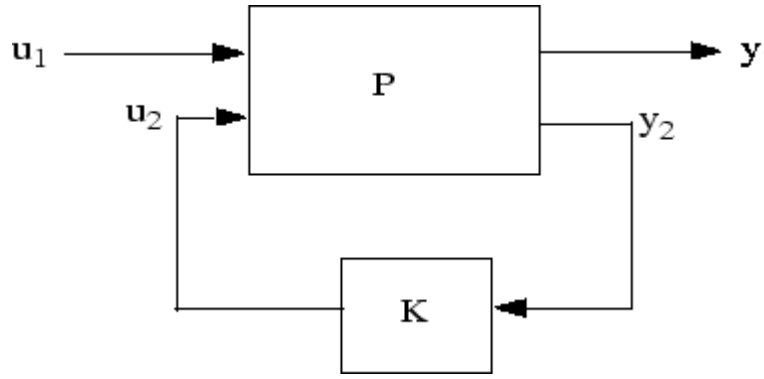
`h2syn` computes a stabilizing H_2 optimal `lti/ss` controller `K` for a partitioned LTI plant `P`. The controller, `K`, stabilizes the plant `P` and has the same number

$$\mathtt{P} = \left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right]$$

of states as `P`. The LTI system `P` is partitioned where inputs to B_1 are the disturbances, inputs to B_2 are the control inputs, output of C_1 are the errors to be kept small, and outputs of C_2 are the output measurements provided to the controller. B_2 has column size (`NCON`) and C_2 has row size (`NMEAS`).

If `P` is constructed with `mktito`, you can omit `NMEAS` and `NCON` from the arguments.

The closed-loop system is returned in `CL` and the achieved H_2 cost γ in `GAM`. `INFO` is a `struct` array that returns additional information about the design.



H_2 control system `CL= lft(P,K)`=$T_{y_1u_1}$.

Output Arguments	Description
`K`	LTI controller
`CL= lft(P,K)`	LTI closed-loop system $T_{y_1u_1}$
`GAM = norm(CL)`	H_2 optimal cost $\gamma = \|T_{y_1u_1}\|_2$
`INFO`	Additional output information

Additional output — structure array `INFO` containing possible additional information depending on `METHOD`)

`INFO.NORMS`	Norms of four different quantities, full information control cost (FI), output estimation cost (OEF), direct feedback cost (DFL) and full control cost (FC). NORMS = [FI OEF DFL FC];
`INFO.KFI`	Full-information gain matrix (constant feedback) $u_2(t) = K_{FI}x(t)$

INFO.GFI	Full-information closed-loop system GFI=ss(A-B2*KFI,B1,C1-D12*KFI,D11)
INFO.HAMX	X Hamiltonian matrix (state-feedback)
INFO.HAMY	Y Hamiltonian matrix (Kalman filter)

Examples

Example 1: Stabilize 4-by-5 unstable plant with three states, NMEAS=2, NCON=2.

```
rand('seed',0);randn('seed',0);
P=rss(3,4,5)';
[K,CL,GAM]=h2syn(P,2,1);
open_loop_poles=pole(P)
closed_loop_poles=pole(CL)
```

Example 2: Mixed-Sensitivity H_2 loop-shaping. Here the goal is to shape the sigma plots of sensitivity $S:=(I+GK)^{-1}$ and complementary sensitivity $T:=GK(I+GK)^{-1}$, by choosing a stabilizing K the minimizes the H_2 norm of

$$T_{y_1 u_1} \stackrel{\Delta}{=} \begin{bmatrix} W_1 S \\ (W_2/G)T \\ W_3 T \end{bmatrix}$$

where $G(s) = \frac{s-1}{s-2}$, $W_1 = \frac{0.1(s+1000)}{100s+1}$, $W_2 = 0.1$, no W_3.

```
s=zpk('s');
G=10*(s-1)/(s+1)^2;
W1=0.1*(s+1000)/(100*s+1); W2=0.1; W3=[];
P=ss(G,W1,W2,W3);
[K,CL,GAM]=h2syn(P);
L=G*K; S=inv(1+L); T=1-S;
sigma(L,'k-.',S,'r',T,'g')
```

Algorithm

The H_2 optimal control theory has its roots in the frequency domain interpretation the cost function associated with time-domain state-space LQG control theory [1]. The equations and corresponding nomenclature used here are taken from the Doyle *et al.*, 1989 [2]-[3].

`h2syn` solves the H_2 optimal control problem by observing that it is equivalent to a conventional Linear-Quadratic Gaussian (LQG) optimal control problem. For simplicity, we shall describe the details of algorithm only for the continuous-time case, in which case the cost function J_{LQG} satisfies

$$J_{LQG} = \lim_{T\to\infty} E\left\{\frac{1}{T}\int_0^T y_1^T y_1 dt\right\}$$

$$= \lim_{T\to\infty} E\left\{\frac{1}{T}\int_0^T [x^T \;\; u_2^T]\begin{bmatrix} Q & N_c \\ N_c^T & R\end{bmatrix}\begin{bmatrix} x \\ u_2\end{bmatrix} dt\right\}$$

$$= \lim_{T\to\infty} E\left\{\frac{1}{T}\int_0^T [x^T \;\; u_2^T]\begin{bmatrix} C_1^T \\ D_{12}^T\end{bmatrix}[C_1 \;\; D_{12}]\begin{bmatrix} x \\ u_2\end{bmatrix} dt\right\}$$

with plant noise u_1 channel of intensity I, passing through the matrix [B1;0;D12] to produce equivalent white correlated with plant ξ and white measurement noise θ having joint correlation function

$$E\left\{\begin{bmatrix}\xi(t) \\ \theta(t)\end{bmatrix}[\xi(\tau) \;\; \theta(\tau)]^T\right\} = \begin{bmatrix} \Xi & N_f \\ N_f^T & \Theta\end{bmatrix}\delta(t-\tau)$$

$$= \begin{bmatrix} B_1 \\ D_{21}\end{bmatrix}\begin{bmatrix} B_1^T & D_{21}^T\end{bmatrix}\delta(t-\tau)$$

The H_2 optimal controller $K(s)$ is thus realizable in the usual LQG manner as a full-state feedback K_{FI} and a Kalman filter with residual gain matrix K_{FC}.

1 Kalman Filter

$$\dot{\hat{x}} = A\hat{x} + B_2 u_2 + K_{FC}(y_2 - C_2\hat{x} - D_{22}u_2)$$

$$K_{FC} = (YC_2^T + N_f)\Theta^{-1} = (YC_2^T + B_1 D_{21}^T)(D_{21}D_{21}^T)^{-1}$$

where $Y = Y^T \geq 0$ solves the Kalman filter Riccati equation

$$YA^T + AY - (YC_2^T + N_f)\Theta^{-1}(C_2 Y + N_f^T) + \Xi = 0$$

2 Full-State Feedback

$$u_2 = K_{FI}\hat{x}$$

$$K_{FI} = R^{-1}(B_2^T X + N_c^T) = (D_{12}^T D_{12})^{-1}(B_2^T X + D_{12}^T C_1)$$

where $X = X^T \geq 0$ solves the state-feedback Riccati equation

$$A^T X + XA - (XB_2 + N_c)R^{-1}(B_2^T X + N_c^T) + Q = 0$$

The final *positive*-feedback H_2 optimal controller $u_2 = K(s)y_2$ has a familiar closed-form

$$K(s) := \left[\begin{array}{c|c} A - K_{FC}C_2 - B_2 K_{FI} + K_{FC}D_{22}K_{FI} & K_f \\ \hline -K_{FI} & 0 \end{array}\right]$$

`h2syn` implements the continuous optimal H_2 control design computations using the formulae described in the Doyle, *et al.* [2]; for discrete-time plants, `h2syn` uses the same controller formula,

except that the corresponding discrete time Riccati solutions (dare) are substituted for X and Y. A Hamiltonian is formed and solved via a Riccati equation. In the continuous-time case, the optimal H_2-norm is infinite when the plant D_{11} matrix associated with the input disturbances and output errors is *non*-zero; in this case, the optimal H_2 controller returned by `h2syn` is computed by first setting D_{11} to zero.

3 Optimal Cost GAM

The full information (FI) cost is given by the equation $(\mathrm{trace}(B_1' X_2 B_1))^{\frac{1}{2}}$. The output estimation cost (OEF) is given by $(\mathrm{trace}(F_2 Y_2 F_2'))^{\frac{1}{2}}$, where $F_2 =: -(B_2' X_2 + D_{12}' C_1)$. The disturbance feedforward cost (DFL) is $(\mathrm{trace}(L_2' X_2 L_2))^{\frac{1}{2}}$, where L_2 is defined by $-(Y_2 C_2' + B_1 D_{21}')$ and the full control cost (FC) is given by $(\mathrm{trace}(C_1 Y_2 C_1'))^{\frac{1}{2}}$. X_2 and Y_2 are the solutions to the X and Y Riccati equations, respectively. For for continuous-time plants with zero feedthrough term (`D11` = 0), and for all discrete-time plants, the optimal H_2 cost $\gamma = \|T_{y_1 u_1}\|_2$ is

```
GAM =sqrt(FI^2 + OEF^2+ trace(D11*D11'));
```

otherwise, `GAM = Inf`.

Limitations

- (A, B_2, C_2) must be *stabilizable and detectable.*
- D_{12} must have full column rank and D_{21} must have full row rank

References

[1] Safonov, M.G., A.J. Laub, and G. Hartmann, "Feedback Properties of Multivariable Systems: The Role and Use of Return Difference Matrix," *IEEE Trans. of Automat. Contr.*, AC-26, pp. 47-65, 1981.

[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, no. 8, pp. 831–847, August 1989.

[3] 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 and Control Letters,* 1988. vol. 11, pp. 167–172, August 1989.

See Also

`augw`

`hinfsyn`

Purpose

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]:

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

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`*'overides `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.

Argument	Value	Description
'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 $\|_\infty$) • 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.

Algorithm

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

$$\left[\begin{array}{c|c} Es - \overline{A} & \overline{B} \\ \hline \overline{C} & \overline{D} \end{array}\right] = \left[\begin{array}{c|c} \rho^2 A^T + QAP & QB \\ \hline CP & D \end{array}\right]$$

$$E = [U_{E1}, U_{E2}] \begin{bmatrix} \Sigma_E & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{E1}^T \\ V_{E2}^T \end{bmatrix}$$

3 Apply the transformation to the descriptor state-space system above we have

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} U_{E1}^T \\ U_{E2}^T \end{bmatrix} (\rho^2 A^T + QAP) \begin{bmatrix} V_{E1} & V_{E2} \end{bmatrix}$$

$$\begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} U_{E1}^T \\ U_{E2}^T \end{bmatrix} \begin{bmatrix} QB & -C^T \end{bmatrix}$$

$$\begin{bmatrix} C_1 & C_2 \end{bmatrix} = \begin{bmatrix} CP \\ -\rho B^T \end{bmatrix} \begin{bmatrix} V_{E1} & V_{E2} \end{bmatrix}$$

$D_1 = D$

4 Form the equivalent state-space model.

$$\begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{C} & \tilde{D} \end{bmatrix} = \begin{bmatrix} \Sigma_E^{-1}(A_{11} - A_{12}A_{22}^{\dagger}A_{21}) & \Sigma_E^{-1}(B_1 - A_{12}A_{22}^{\dagger}B_2) \\ C_1 - C_2A_{22}^{\dagger}A_{21} & D_1 - C_2A_{22}^{\dagger}B_2 \end{bmatrix}$$

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].

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:

```
rand('state',1234); randn('state',5678);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]);
rand('state',12345); randn('state',6789);
wt1 = rss(6,5,5); wt1.d = eye(5)*2;
wt2 = rss(6,4,4); wt2.d = 2*eye(4);
[g5, redinfo5] = hankelmr(G, [10:2:18], 'weight',{wt1,wt2});
for i = 1:5
 figure(i); eval(['sigma(G,g' num2str(i) ');']);
```

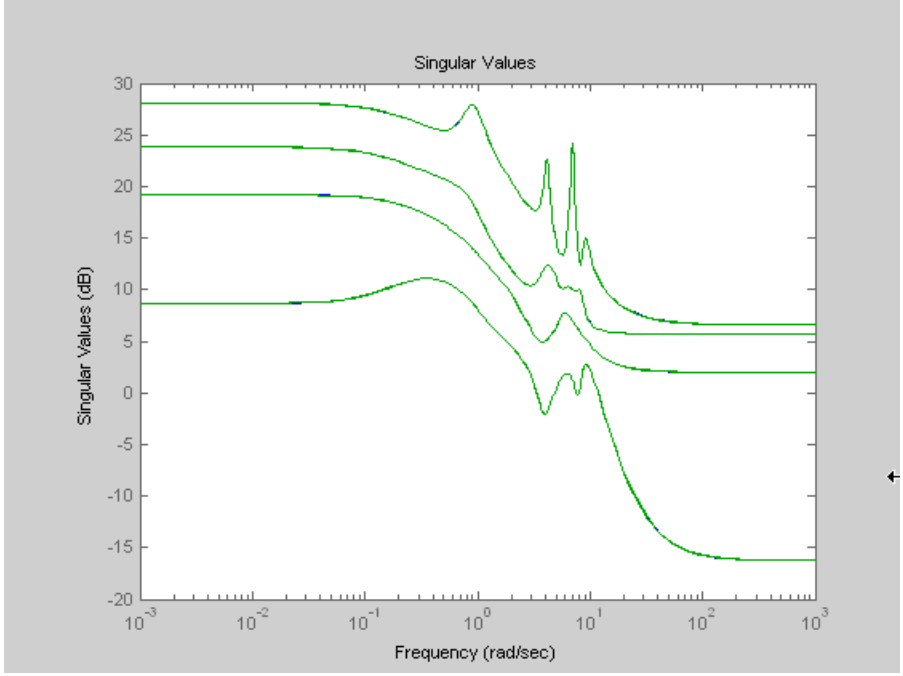
```
end
```

Singular Value Bode Plot of G (30-state, 5 outputs, 4 inputs) on page 2-114 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 2-115.

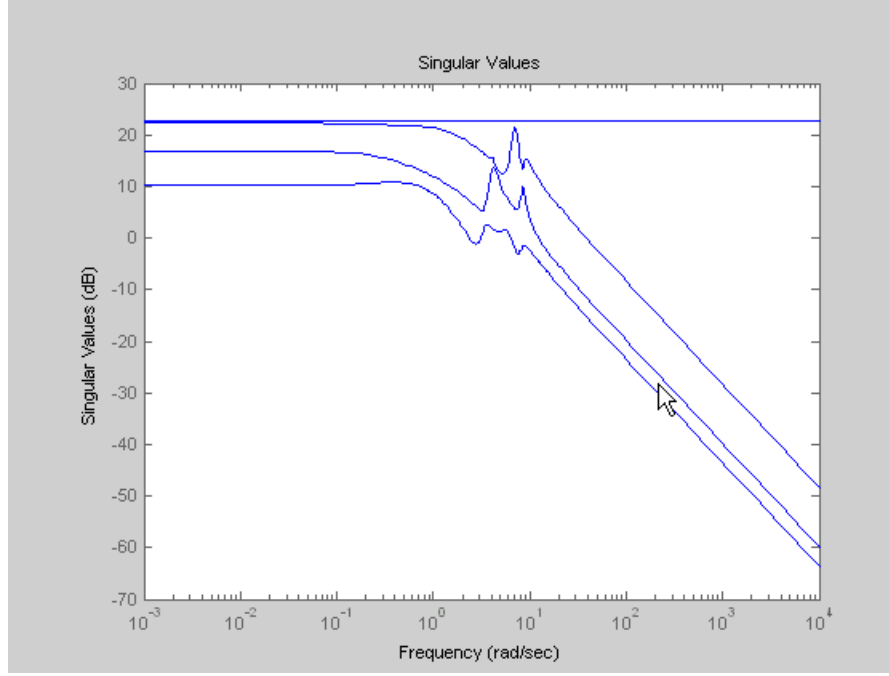
The *Zeroth order Hankel MDA* and its error system sigma plot are obtained via commands

```
[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

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

`reduce`

`balancmr`

`schurmr`

`bstmr`

ncfmr

hankelsv

Purpose

Compute Hankel singular values for stable/unstable or continuous/discrete system

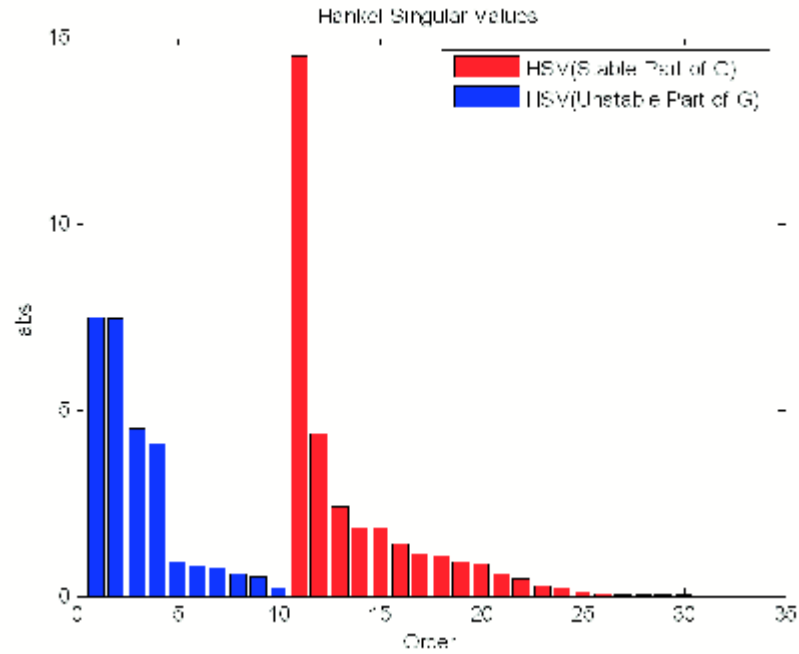
Syntax

```
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

Algorithm

For `ErrorType = 'add'`, `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.

For `ErrorType = 'mult'`, `hankelsv` computes the Hankel singular value of the phase matrix of `G` [2].

For `ErrorType = 'ncf'`, `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

`reduce`

`balancmr`

schurmr

bstmr

ncfmr

hankelmr

Purpose

Synthesis of gain-scheduled H_∞ controllers

Syntax

```
[gopt,pdK,R,S] = hinfgs(pdP,r,gmin,tol,tolred)
```

Description

Given an affine parameter-dependent plant

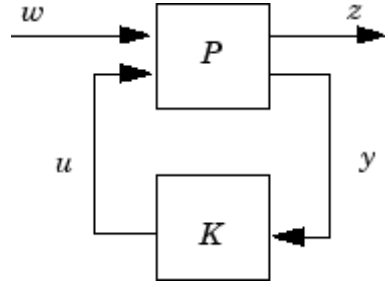
$$P\begin{cases}\dot{x} &= A(p)x + B_1(p)w + B_2u\\ z &= C_1(p)x + D_{11}(p)w + D_{12}u\\ y &= C_2x + D_{21}w + D_{22}u\end{cases}$$

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=[`p_2`,m2]` if $y \in \mathbf{R}^{p2}$ and $u \in \mathbf{R}^{m2}$). 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 ϑ_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 ϑ_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 ϑ_j:

$$p(t) = \alpha_1 \vartheta_1 + \ldots + \alpha_N \vartheta_N, \ \alpha_j \geq 0, \ \sum_{i=1}^{N} \alpha_j = 1$$

   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," submitted to *Automatica*, October 1995.

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

`psys`

`pvec`

`pdsimul`

`polydec`

hinfsyn

Purpose

Compute H_∞ optimal controller for LTI plant

Syntax

```
[K,CL,GAM,INFO] = hinfsyn(P)
[K,CL,GAM,INFO] = hinfsyn(P,NMEAS,NCON)
[K,CL,GAM,INFO] = hinfsyn(P,NMEAS,NCON,KEY1,VALUE1,KEY2,VALUE2,...)
```

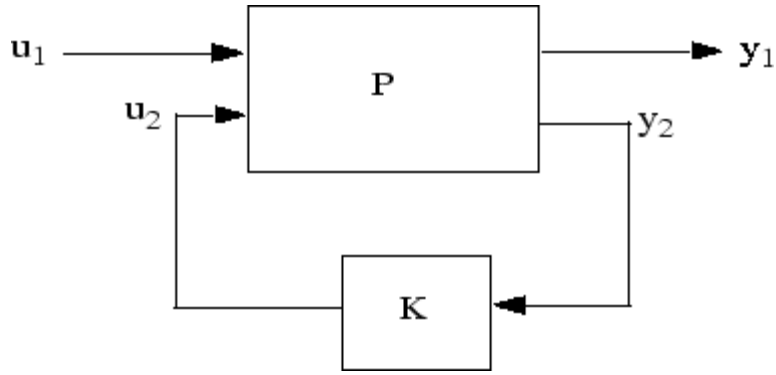
Description

`hinfsyn` computes a stabilizing H_∞ optimal `lti/ss` controller `K` for a partitioned `lti` plant `P`.

$$\mathtt{P} = \left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & D_{11} & D_{12} \\ C_2 & D_{21} & D_{22} \end{array}\right]$$

The controller, `K`, stabilizes the `P` and has the same number of states as `P`. The system `P` is partitioned where inputs to B_1 are the disturbances, inputs to B_2 are the control inputs, output of C_1 are the errors to be kept small, and outputs of C_2 are the output measurements provided to the controller. B_2 has column size (`NCON`) and C_2 has row size (`NMEAS`). The optional `KEY` and `VALUE` inputs determine tolerance, solution method and so forth.

The closed-loop system is returned in `CL` and the achieved H_∞ cost γ in `GAM`. `INFO` is a `struct` array that returns additional information about the design.



H∞ control system `CL= lft(P,K)=` $T_{y_1 u_1}$.

Optional Input Arguments

Property	Value	Description
`'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)
`'SO'`	real	Frequency `SO` at which entropy is evaluated, only applies to `METHOD` '`maxe`' (default=`Inf`)
`'METHOD'`	'*ric*'	Standard 2-Riccati solution (default)
	'*lmi*'	LMI solution
	'*maxe*'	Maximum entropy solution
`'DISPLAY'`	'*off*'	No command window display, or command window displays synthesis progress information (default)
	'*on*'	

When `DISPLAY='on'`, the `hinfsyn` program displays several variables indicating the progress of the algorithm. For each γ value being

tested, the minimum magnitude, real part of the eigenvalues of the X and Y Hamiltonian matrices are displayed along with the minimum eigenvalue of X_∞ and Y_∞, which are the solutions to the X and Y Riccati equations, respectively. The maximum eigenvalue of $X_\infty Y_\infty$, scaled by γ^{-2},is also displayed. A # sign is placed to the right of the condition that failed in the printout.

Output Arguments	Description
`K`	`lti` controller
`CL= lft(P,K)`	`lti` closed-loop system $T_{y_1u_1}$
`GAM = norm(CL,Inf)`	H_∞ cost $\gamma=\|T_{y_1u_1}\|_\infty$
`INFO`	Additional output information

Additional output — structure array `INFO` containing possible additional information depending on `METHOD`)

`INFO.AS`	All solutions controller, `lti` two-port LFT
`INFO.KFI`	Full information gain matrix (constant feedback $u_2(t) = K_{FI}\begin{bmatrix} x(t) \\ u_1(t) \end{bmatrix}$
`INFO.KFC`	Full control gain matrix (constant output-injection; K_{FC} is the dual of K_{FI})
`INFO.GAMFI`	H_∞ cost for full information K_{FI}
`INFO.GAMFC`	H_∞ cost for full control K_{FC}

Algorithm

The default `'ric'` method uses the two-Riccati formulae ([1],[2]) with loopshifting [3]. In the case of the `'lmi'` method, `hinfsyn` employs the LMI technique ([4],[5],[6]). With `'METHOD' 'maxe', K` returns the

max entropy H_∞ controller that minimize an entropy integral relating to the point `s0`; i.e.,

$$\mathbf{Entropy} = -\frac{\gamma^2}{2\pi}\int_{-\infty}^{\infty} \ln\left|\det I - \gamma^{-2} T_{y_1u_1}(j\omega)' \, T_{y_1u_1}(j\omega)\right| \left[\frac{s_0{}^2}{s_0{}^2 + \omega^2}\right] d\omega$$

where $T_{y_1u_1}$ is the closed-loop transfer function `CL`. With all methods, `hinfsyn` uses a standard γ-iteration technique to determine the optimal value of γ. Starting with high and low estimates of γ. The γ-iteration is a b*isection algorithm* that iterates on the value of γ in an effort to approach the optimal H_∞ control design. 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 `TOLGAM` (default=.01)

At each value of γ, the algorithm employed requires tests to determine whether a solution exists for a given γ value. In the case of the `'ric'` method, the conditions checked for the existence of a solution are:

- *H* and *J* Hamiltonian matrices (which are formed from the state-space data of *P* and the γ level) must have no imaginary-axis eigenvalues.
- the stabilizing Riccati solutions X_∞ and Y_∞ associated with the Hamiltonian matrices must exist and be positive, semi-definite.
- spectral radius of (X_∞, Y_∞) must be less than or equal to γ2.

When, `DISPLAY` is 'on', the `hinfsyn` program displays several variables, which indicate which of the above conditions are satisfied for each γ value being tested. In the case of the default `'ric'` method, the display includes the current value of γ being tested, real part of the eigenvalues of the *X* and *Y* Hamiltonian matrices along with the minimum eigenvalue of X_∞ and Y_∞, which are the solutions to the *X* and *Y* Riccati equations, respectively. The maximum eigenvalue of $X_\infty Y_\infty$, scaled by γ–2, is also displayed. A # sign is placed to the right of the condition that failed in the printout. A similar `display` is produced with method `'lmi'`

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 $\omega \in \mathbf{R}$.

$\begin{bmatrix} A - j\omega I & B_1 \\ C_2 & D_{21} \end{bmatrix}$ has full row rank for all $\omega \in \mathbf{R}$.

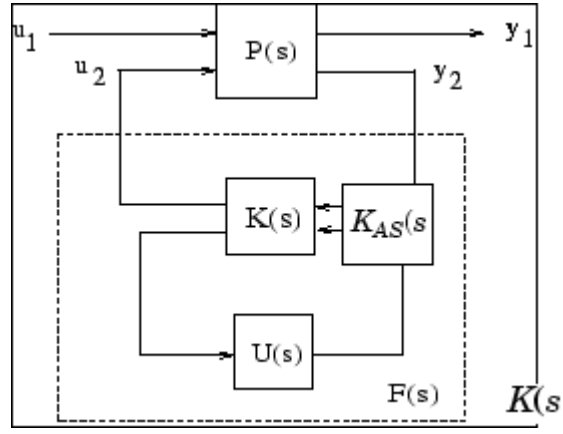
When the above rank conditions do not hold, the controller may have undesirable properties: If D_{12} and D_{21} are not full rank, the H_∞ controller `K` may have large high-frequency gain. If either of the latter two rank conditions does not hold at some frequency ω, the controller may have very lightly damped poles near that frequency ω.

In general, the solution to the infinity-norm optimal control problem is non-unique. Whereas the `K` returned by `hinfsyn` is only a particular $F(s)$, when the `'ric'` method is selected, the `INFO.AS` field of `INFO` give you in addition the all- solution controller parameterization $K_{AS}(s)$ such that all solutions to the infinity-norm control problem are parameterized by a free stable contraction map $U(s)$ constrained by $(\|U(s)\|_\infty < 1)$; that is, every stabilizing controller $K(s)$ that makes

$$\left\|T_{y_1u_1}\right\|_\infty \stackrel{\Delta}{=} \sup_\omega \sigma_{max}(T_{y_1u_1}(j\omega)) < \gamma$$

```
K=lft(INFO.AS,U)
```

where `U` is a stable LTI system satisfying `norm(U,Inf) <1`



All-solution $K_{AS}(s)$ returned by `INFO.AS`

An important use of the infinity-norm control theory is for direct shaping of closed-loop singular value Bode plots of control systems. In such cases, the system $P(s)$ will typically be the plant augmented with suitable loop-shaping filters — see `mixsyn`.

Examples

Following are three simple problems solved via `hinfsyn`.

Example 1: A random 4-by-5 plant with 3-states, `NMEAS=2`, `NCON=2`

```
rand('seed',0);randn('seed',0);
P=rss(3,4,5);
[K,CL,GAM]=hinfsyn(P,2,2);
```

The optimal H_∞ cost in this case is `GAM=0.2641`. You verify that $\|T_{y_1u_1}\|_\infty \triangleq \sup_\omega \sigma_{max}(T_{y_1u_1}(j\omega)) < \gamma$ with a sigma plot

```
sigma(CL,ss(GAM));
```

Example 2: Mixed-Sensitivity $G(s) = \frac{s-1}{s-2}$, $W_1 = \frac{0.1(s+1000)}{100s+1}$, $W_2 = 0.1$, no W_3.

```
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);
[K,CL,GAM]=hinfsyn(P);
sigma(CL,ss(GAM));
```

In this case, `GAM` = 0.1854 = -14.6386 db

Example 3: Mixed sensitivity with W_1 removed.

```
s=zpk('s');
G=(s-1)/(s+1);
W1=[]; W2=0.1; W3=[];
P=augw(G,W1,W2,W3)
[K,CL,GAM]=hinfsyn(P)
```

In this case, `GAM=0`, `K=0`, and `CL=K*(1+G*K)=0`.

Limitation

The plant must be stabilizable from the control inputs u_2 and detectable from the measurement output y_2:

- (A,B_2) must be stabilizable and (C_2,A) must be detectable.

Otherwise, an the `hinfsyn` returns an error.

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 and Control Letters,* vol. 11, pp. 167–172, 1988.

[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, no. 8, pp. 831–847, August 1989

[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, no. 6, pp. 2467-2488, 1989.

[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 and Control Letters*, 19:271–280, 1992.

[5] Gahinet, P., and P. Apkarian, "A linear matrix inequality approach to H_∞-control," *Int J. Robust and Nonlinear Control*, 4(4):421–448, July–August 1994.

[6] Iwasaki, T., and R.E. Skelton, "All controllers for the general H_∞-control problem: LMI existence conditions and state space formulas," *Automatica*, 30(8):1307–1317, August 1994.

See Also

`augw`

`h2syn`

`loopsyn`

`mktito`

`ncfsyn`

icomplexify

Purpose

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`

`robuststab`

Purpose

Create empty `iconnect` (interconnection) objects

Syntax

```
H = iconnect
```

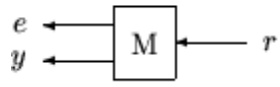
Description

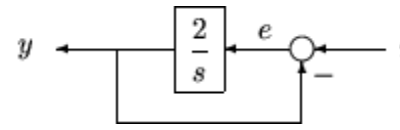
Interconnection objects (class `iconnect`) are an alternative to `sysic`, and are used to build complex interconnections of uncertain matrices and systems.

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.

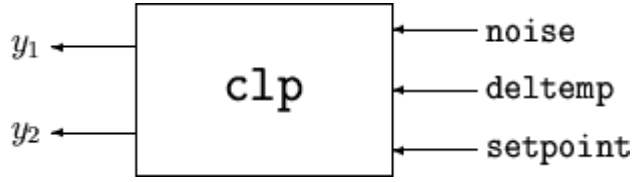
```
        s
 #1:  -----
      s + 2

        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]`.

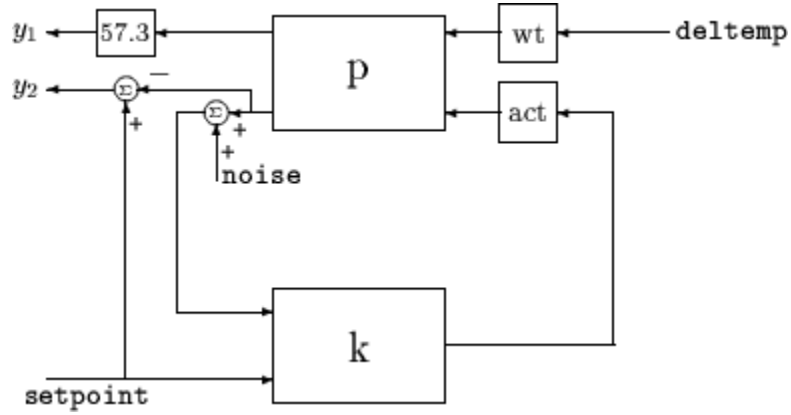
```
R = ureal('R',1,'Percentage',[-10 40]);
```

```
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.
```

Algorithm

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.

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.

See Also

`icsignal`

`sysic`

icsignal

Purpose

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 string argument `name`.

See Also

`iconnect`

`sysic`

Purpose

Convert implicit linear relationship to explicit input-output relation

Syntax

```
B = imp2exp(A,yidx,uidx)
```

Description

`B = imp2exp(A,yidx,uidx)` transforms a linear constraint between variables `Y` and `U` of the form `A(:,[yidx;uidx])*[Y;U] = 0` into an explicit input/output relationship `Y = B*U`. The vectors `yidx` and `uidx` refer to the columns (inputs) of `A` as referenced by the explicit relationship for `B`.

The constraint matrix `A` can be a `double, ss, tf, zpk` and `frd` object as well as an uncertain object, including `umat`, `uss` and `ufrd`. The result `B` will be of the same class.

Examples

Scalar Algebraic Constraint

Consider the constraint `4y + 7u = 0`. Solving for `y` gives `y = 1.75u`. You form the equation using `imp2exp`:

```
A = [4 7];
Yidx = 1;
Uidx = 2;
```

and then

```
B = imp2exp(A,Yidx,Uidx)
B =
   -1.7500
```

yields `B` equal to `-1.75`.

Matrix Algebraic Constraint

Consider two motor/generator constraints among 4 variables `[V;I;T;W]`, namely `[1 -1 0 -2e-3;0 -2e-3 1 0]*[V;I;T;W] = 0`. You can find the 2-by-2 matrix `B` so that `[V;T] = B*[W;I]` using `imp2exp`.

```
A = [1 -1 0 -2e-3;0 -2e-3 1 0];
Yidx = [1 3];
```

```
Uidx = [4 2];
B = imp2exp(A,Yidx,Uidx)
B =
    0.0020    1.0000
         0    0.0020
```

You can find the 2-by-2 matrix `C` so that `[I;W] = C*[T;V]`

```
Yidx = [2 4];
Uidx = [3 1];
C = imp2exp(A,Yidx,Uidx)
C =
         500           0
     -250000         500
```

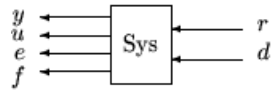
Uncertain Matrix Algebraic Constraint

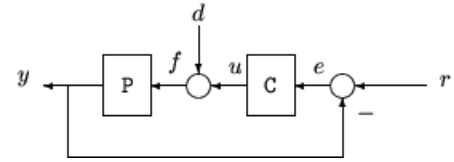
Consider two uncertain motor/generator constraints among 4 variables `[V;I;T;W]`, namely `[1 -R 0 -K;0 -K 1 0]*[V;I;T;W] = 0`. You can find the uncertain 2-by-2 matrix `B` so that `[V;T] = B*[W;I]`.

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

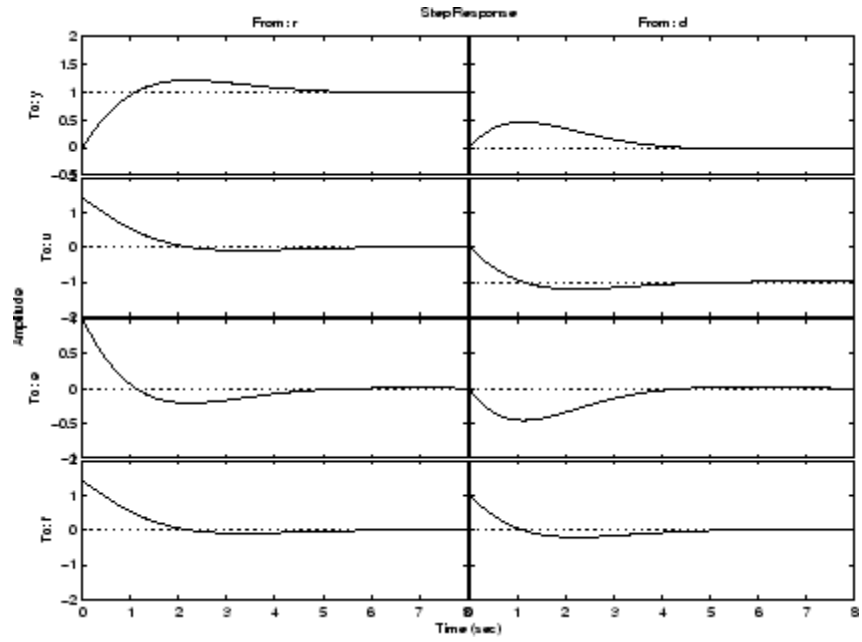
Scalar Dynamic System Constraint

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






```
P = tf([1],[1 0]);
C = tf([2*.707*1 1^2],[1 0]);
A = [1 -1 0 0 0 -1;0 -C 1 0 0 0;0 0 -1 -1 1 0;0 0 0 0 -P 1];
OutputIndex = [6;3;2;5];  % [y;u;e;f]
InputIndex = [1;4];       % [r;d]
Sys = imp2exp(A,OutputIndex,InputIndex);
Sys.InputName = {'r';'d'};
Sys.OutputName = {'y';'u';'e';'f'};
pole(Sys)
ans =
  -0.7070 + 0.7072i
  -0.7070 - 0.7072i
step(Sys)
```



Algorithm

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

See Also

`iconnect`

`inv`

Purpose

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:

$$y = [H_0(:)';H_2(:)';H_3(:)';\ldots;H_N(:)']$$

The variable *tol* bounds the $\mathbf{H}_\infty$ 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 *to*l. The inputs `ts, nu, ny, tol` are optional; if not present they default to the values `ts = 0, nu = 1, ny` = (number of rows of y)/`nu`, $\texttt{tol} = 0.01\bar{\sigma}_1$. The output $\texttt{hsv} = [\bar{\sigma}_1, \bar{\sigma}_2, \ldots]'$ returns the singular values (arranged in descending order of magnitude) of the Hankel matrix:

$$\Gamma = \begin{bmatrix} H_1 & H_2 & H_3 & \cdots & H_N \\ H_2 & H_3 & H_4 & \cdots & 0 \\ H_3 & H_4 & H_5 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ H_N & 0 & \cdots & \cdots & 0s \end{bmatrix}$$

Denoting by G_N a high-order exact realization of y, the low-order approximate model G enjoys the $\mathbf{H}_\infty$ norm bound

$$\left\| G - G_N \right\|_\infty \le totbnd$$

where

$$totbnd = 2 \sum_{i=n+1}^{N} \bar{\sigma}_i$$

Algorithm

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

$$\Gamma = U\Sigma V^* = [U_1 U_2] \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V^*_1 \\ V^*_2 \end{bmatrix} = U_1 \Sigma_1 V^*_1$$

where Σ_1 has dimension n x 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:

$$U_1 = \begin{bmatrix} U_{11} \\ U_{12} \\ U_{13} \end{bmatrix}; \begin{bmatrix} V_{11} \\ V_{12} \\ V_{13} \end{bmatrix}$$

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

$$A = \Sigma_1^{-1/2} \overline{U} \Sigma_1^{1/2}$$

$$B = \Sigma_1^{-1/2} V^*{}_{11}$$

$$C = U_{11} \Sigma_1^{-1/2}$$

$$D = H_0$$

where

$$\overline{U} = \begin{bmatrix} U_{11} \\ U_{12} \end{bmatrix}' \begin{bmatrix} U_{12} \\ U_{13} \end{bmatrix}$$

5 If the sampling 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.Twelth 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.

Purpose

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

`psys`

`psinfo`

isuncertain

Purpose

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
```

Purpose

Decompose uncertain objects into fixed normalized and fixed uncertain parts

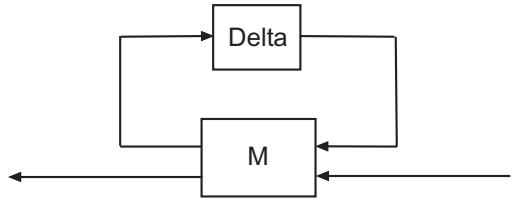
Syntax

```
[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 the string `'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`

lmiedit

Purpose

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 as a MATLAB string (`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

Remark

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

`lmivar`

`lmiterm`

`newlmi`

`lmiinfo`

lmiinfo

Purpose

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-hand 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

$$0 \left(\begin{array}{cc} -2X + A^T Y B + B^T Y^T A + I & XC \\ C^T X & -zI \end{array} \right)$$

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)`.

Remark

`lmiinfo` does not provide access to the numerical value of LMI coefficients.

See Also

`decinfo`

`lminbr`

`matnbr`

`decnbr`

lminbr

Purpose

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`

Purpose

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`). Recall that an LMI region is any convex subset D of the complex plane that can be characterized by an LMI in z and $\bar{z}$, i.e.,

$$D = \{z \in \mathbf{C} : L + Mz + M^T\bar{z} < 0\}$$

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.

See Also

`msfsyn`

lmiterm

Purpose

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 `limterm`, `termID` is a four-entry vector of integers specifying the term location and the matrix variable involved.

$$\text{termID}\,(1) = \begin{cases} +p \\ -p \end{cases}$$

where positive p is for terms on the *left-hand side* of the p-th LMI and negative p i s for terms on the *right-hand side* of the p-th LMI.

Recall that, by convention, the left-hand 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`.

$$\text{termID (2:3)} = \begin{cases} [0,0] & \text{for outer factors} \\ [i,\ j] & \text{for terms in the } (i,\ j)\text{-th} \\ & \quad \text{block of the left or right inner factor} \end{cases}$$

$$\text{termID (4)} = \begin{cases} 0 & \text{for outer factors} \\ x & \text{for variable terms } AXB \\ -x & \text{for variable terms } AX^TB \end{cases}$$

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

$$(AXB) + (AXB^T) = AXB + B^TX^{(T)}A^T$$

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

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 initializing the LMI description with `setlmis` and declaring the matrix variables with `lmivar`, the terms on the left-hand side of this LMI are specified by:

```
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, X3` should be the variable identifiers returned by `lmivar`.

Similarly, the term content of the right-hand side is specified by:

```
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

setlmis

lmivar

getlmis

lmiedit

newlmi

lmivar

Purpose

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 $\pm x_n$ 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 $x_1, \ldots, x_n$.

Example 1

Consider an LMI system with three matrix variables X_1, X_2, X_3 such that

- X_1 is a 3 x 3 symmetric matrix (unstructured),
- X_2 is a 2 x 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 x 5 symmetric matrix, δ_1 and δ_2 are scalars, and I_2 denotes the identity matrix of size 2.

These three variables are defined by

```
setlmis([])
X1 = lmivar(1,[3 1])          % Type 1
X2 = lmivar(2,[2 4])          % Type 2 of dim. 2x4
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.

Example 2

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

$$X = \begin{pmatrix} X_1 & 0 \\ 0 & X_2 \end{pmatrix}$$

where X_1 and X_2 are 2-by-3 and 3-by-2 rectangular matrices, respectively. You can specify this structure as follows:

1 Define the rectangular variables X_1 and X_2 by

```
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 =
    1     2     3
    4     5     6

sX2

sX2 =
    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.

2 Use Type 3 to specify the matrix variable X and define its structure in terms of those of X_1 and X_2:

```
[X,n,sX] = lmivar(3,[sX1,zeros(2);zeros(3),sX2])
```

The resulting variable X has the prescribed structure as confirmed by

```
sX

sX =
     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

setlmis

lmiterm

getlmis

lmiedit

skewdec

delmvar

setmvar

frd/loglog

Purpose

Log-log scale plot of `frd` objects

Syntax

```
loglog(sys)
loglog(sys,linetype)
```

Description

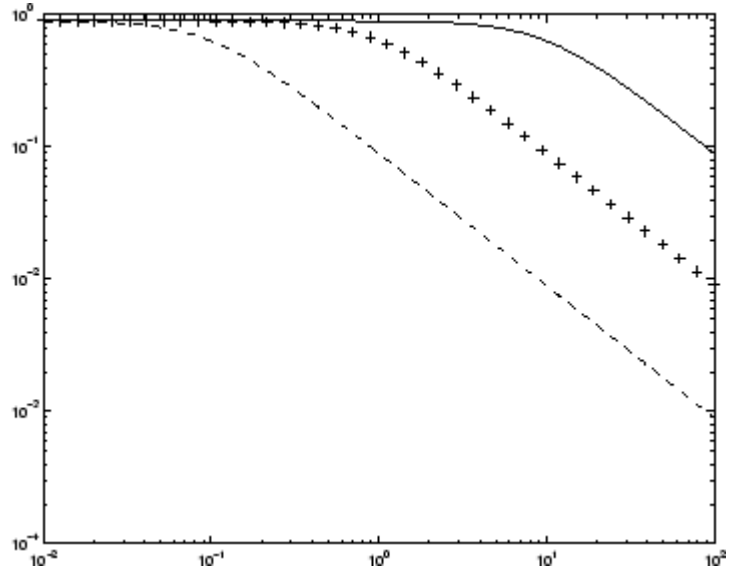
`loglog` is the same as `plot`, except a logarithmic (base 10) scale is used for both the X- and Y-axis.

The argument list consists of (many) *x*-data, *y*-data and line type triples. The *x*-data and *y*-data can be specified individually, as `doubles`, or jointly as an `frd`.

Examples

Generate three frequency response objects, `sys1g`, `sys2g` and `sys3g`. Plot the magnitude of these transfer functions versus frequency on a log-log plot.

```
omega = logspace(-2,2,40);
sys1 = tf(0.9,[.1 1]);
sys1g = frd(sys1,omega);
sys2 = tf(0.9,[1 1]);
sys2g = frd(sys2,omega);
sys3 = tf(0.9,[10 1]);
sys3g = frd(sys3,omega);
loglog(abs(sys1g),abs(sys2g),'r+',sys3g.Freq,...
abs(sys3g.Resp(:)),'--');
```



See Also

`plot`

`semilogx`

`semilogy`

loopmargin

Purpose

Stability margin analysis of LTI and Simulink feedback loops

Syntax

```
[cm,dm,mm] = (L)
[m1,m2] = (L,MFLAG)

[cmi,dmi,mmi,cmo,dmo,mmo,mmio] = (P,C)

[m1,m2,m3] = (P,C,MFLAG)
```

Description

`[cm,dm,mm] = (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 (See `allmargin` for details on the fields of `cm`.)

`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. , (GM_{low}+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.

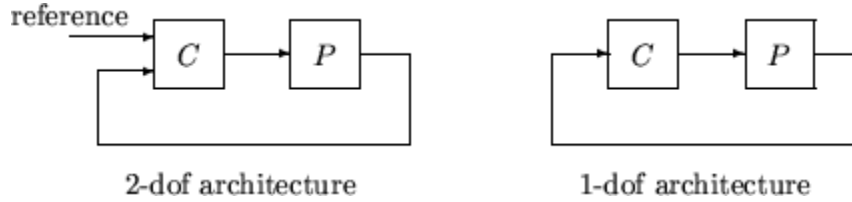
`mm`, or multiloop disk margin, is a structure corresponding to simultaneous, independent, variations in the individual channels of loop transfer matrix `L`. `mm` calculates the largest region such that for all gain and phase variations, occurring independently in each channel, lie inside the region, that the closed-loop system is stable. Note that `mm` is a single structure, independent of because the number of channels, variations in all channels are handled simultaneously. As in the case for disk margin, the guaranteed bounds are calculated based on a balanced sensitivity function.

If `L` is a `ss/tf/zpk` object, the frequency range and number of frequency points used to calculate `dm` and `mm` margins are chosen automatically.

Output arguments can be limited to only those requested using an optional character string argument. `[m1,m2] = (L,'m,c')` returns the

multi-loop diskmargin ('m') in m1, and the classical margins ('c') in m2. Use 'd' to specify the disk margin. This optional second argument may be any combination, in any order, of the 3 characters 'c', 'd' and 'm'.

[cmi,dmi,mmi,cmo,dmo,mmo,mmio] = (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.



2-dof architecture 1-dof architecture

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.

If the closed-loop system is an ss/tf/zpk, the frequency range and number of points used to calculate cm, dm and mm margins are chosen automatically.

Output arguments can be limited to only those requested using an optional character string argument. [m1,m2,m3] = (L,'mo,ci,mm') returns the multi-loop diskmargin at the plant output ('mo') in m1, the classical margins at the plant input ('ci') in m2, and the disk margins for simultaneous, independent variations in all input and output channels ('mm') in m3. This optional third argument may be

any comnination, in any order, of the 7 character pairs 'ci', 'di', 'mi', 'co', 'do, 'mo', and 'mm'.

Usage with Simulink

`[cm,dm,mm] = (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] = (Model,Blocks,Ports,OP)` uses the operating point object `OP` to create linearized systems from the Simulink `Model`.

`[cm,dm,mm,info] = (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.

Margin output arguments can be limited to only those requested using an optional charcter string argument. INFO is always the last output. For example, `[mm,cm,info] = (Model,Blocks,Ports,'m,c')` returns the multi-loop diskmargin (`'m'`) in `mm`, the classical margins (`'c'`) in `cm`, and the `info` structure.

Basic Syntax

`[cm,dm,mm] = (L) cm` is calculated using the `allmargin` command and has the same fields as `allmargin`. The `cm` is a structure with the following fields:

Field	Description
`GMFrequency`	All -180 deg crossover frequencies (in rad/s)
`GainMargin`	Corresponding gain margins (`GM = 1/L` where `L` is the gain at crossover)
`PhaseMargin`	Corresponding phase margins (in degrees)
`PMFrequency`	All 0 dB crossover frequencies (in rad/s)
`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 a structure with the following fields

Field	Description
`GainMargin`	Smallest gain variation (`GM`) such that a disk centered at the point `-(GM(1) + GM(2))/2` would just touch the loop transfer function
`PhaseMargin`	Smallest phase variation, in degrees, corresponding to the disk described in the GainMargin field (degrees)
`Frequency`	Associated with `GainMargin`/`PhaseMargin` fields (rad/s)

`mm` is a structure with the following fields:

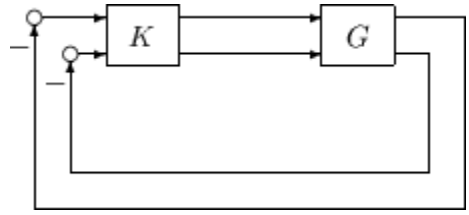
Field	Description
`GainMargin`	Guaranteed bound on simultaneous, independent, gain variations allowed in all plant channels
`PhaseMargin`	Guaranteed bound on simultaneous, independent, phase variations allowed in all plant channels (degrees)
`Frequency`	Associated with `GainMargin`/`PhaseMargin` fields (rad/s)

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. You will see that margins of the individual loops can be very sensitive to small perturbations within other loops.

The nominal closed-loop system considered here is as follows



G and K are `2-by-2` multiinput/multioutput (MIMO) systems, defined as

$$G: = \frac{1}{s^2 + \alpha^2}\begin{bmatrix} s - \alpha^2 & \alpha(s+1) \\ -\alpha(s+1) & s - \alpha^2 \end{bmatrix}, \mathrm{K}=\mathrm{I}_2$$

Set α:= 10, construct 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];
[cmi,dmi,mmi,cmo,dmo,mmo,mmio]=(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 `allmargin` command, `smi(1)`. The disk margin analysis, `dmi`, of the first channel provides similar results.

```
cmi(1)
ans =
    GMFrequency: [1x0 double]
     GainMargin: [1x0 double]
    PMFrequency: 21
    PhaseMargin: 90
    DMFrequency: 21
    DelayMargin: 0.0748
         Stable: 1
dmi(1)
ans =
     GainMargin: [0 Inf]
    PhaseMargin: [-90 90]
      Frequency: 1.1168
```

The second input channel has a gain margin of 2.105 and infinite phase margin based on the single-loop analysis, `cmi(2)`. 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.

```
cmi(2)
ans =
    GMFrequency: 0
     GainMargin: 2.1053
```

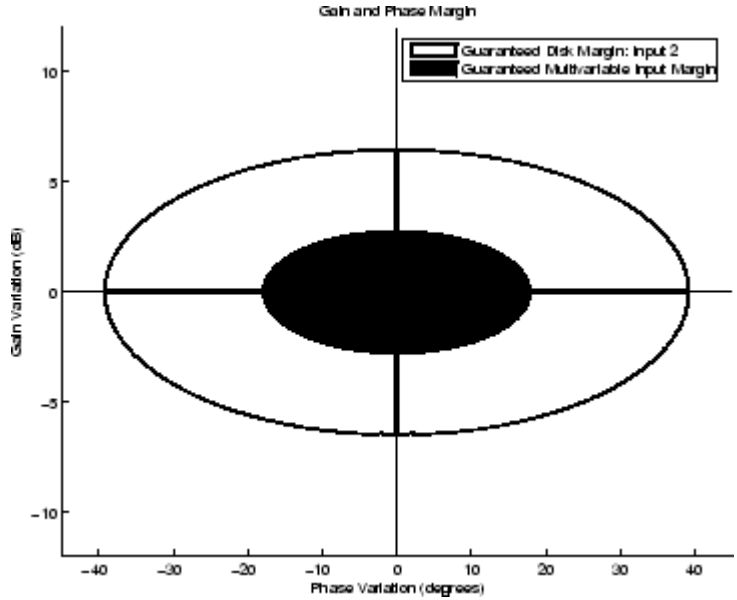
```
    PMFrequency: [1x0 double]
    PhaseMargin: [1x0 double]
    DMFrequency: [1x0 double]
    DelayMargin: [1x0 double]
         Stable: 1
dmi(2)
ans =
     GainMargin: [0.4749 2.1056]
    PhaseMargin: [-39.1912 39.1912]
      Frequency: 0.0200
```

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 =
     GainMargin: [0.7283 1.3730]
    PhaseMargin: [-17.8659 17.8659]
      Frequency: 9.5238e-004
```

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.

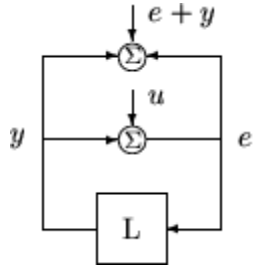
```
mmo
mmo =
     GainMargin: [0.6065 1.6489]
    PhaseMargin: [-27.5293 27.5293]
      Frequency: 0.2287
```

If all the input and output channels are allow to vary independently, `mmio`, the gain margin variation allow are 0.827 and 1.210 and phase margin variations allowed are +/- 10.84 deg.

```
mmio
mmio =
     GainMargin: [0.8267 1.2097]
    PhaseMargin: [-10.8402 10.8402]
      Frequency: 0.2287
```

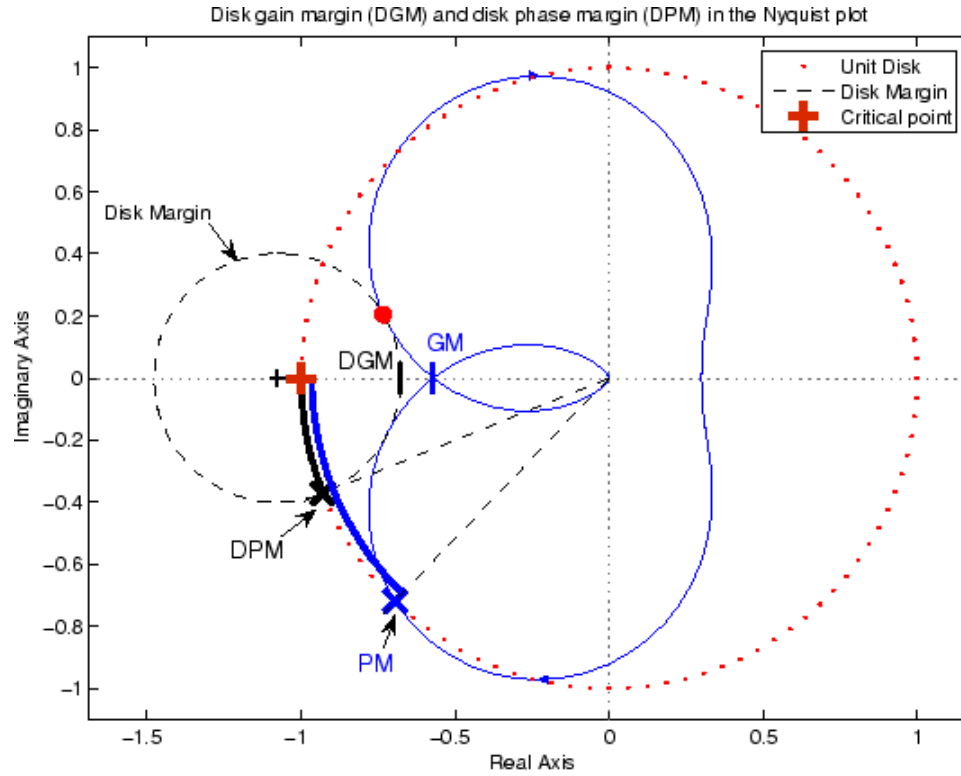
Algorithm

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).



$$\begin{array}{rcccl} e & = & (I-L)^{-1}u & = & Su \\ y & = & L(I-L)^{-1}u & = & Tu \\ e+y & = & (I+L)\cdot(I-L)^{-1}u & = & (S+T)u \end{array}$$

This figure shows a comparison of a disk margin analysis with the 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 $\mu(S+T)$, 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 $\mu(S+T)$ 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 $\mu(S+T)$ 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 $\mu(S+T)$.

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`

`bode`

`loopsens`

`mussv`

`robuststab`

`wcgain`

`wcsens`

`wcmargin`

loopsens

Purpose

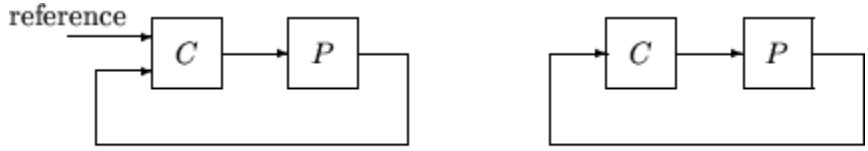
Sensitivity functions of plant-controller feedback loop

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`.



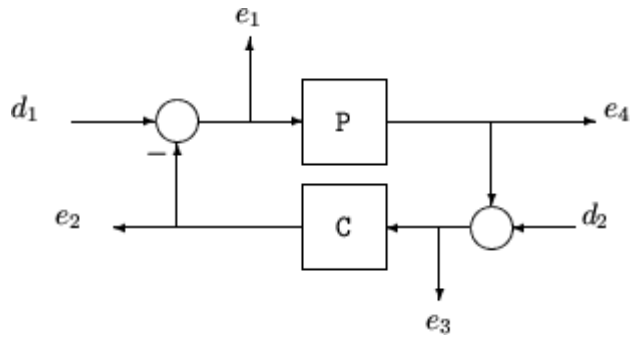
2-dof architecture 1-dof architecture

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
`So`	Output-to-plant sensitivity function
`To`	Output-to-plant complementary sensitivity function
Lo	Output-to-plant loop transfer function

Field	Description
`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.



Description	Equation
Input sensitivity ($TF_{e1 \leftarrow d1}$)	$(I+CP)^{-1}$
Input complementary sensitivity ($TF_{e2 \leftarrow d1}$)	$CP(I+CP)^{-1}$
Output sensitivity ($TF_{e3 \leftarrow d2}$)	$(I+PC)^{-1}$
Output complementary sensitivity ($TF_{e4 \leftarrow d}$)	$PC(I+PC)^{-1}$
Input loop transfer function	CP
Output loop transfer function	PC

Examples

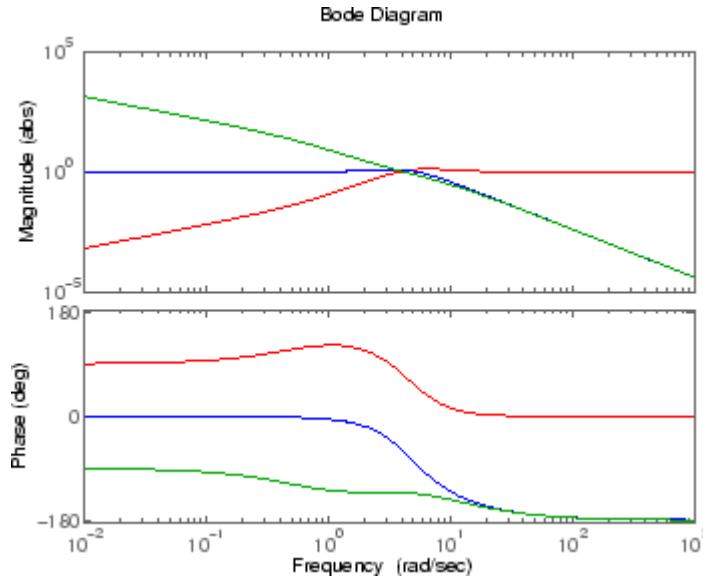
Single Input, Single Output (SISO)

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]);
```

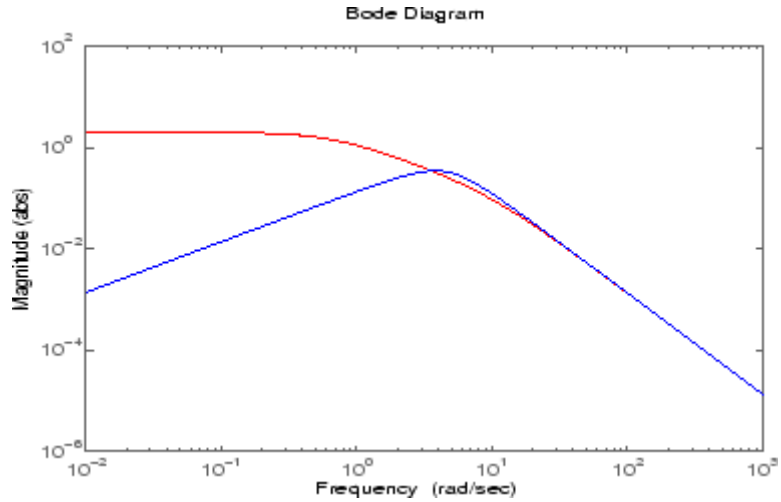
Form the closed-loop (and open-loop) systems with loopsens, and plot Bode plots using the gains at the plant input.

```
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`

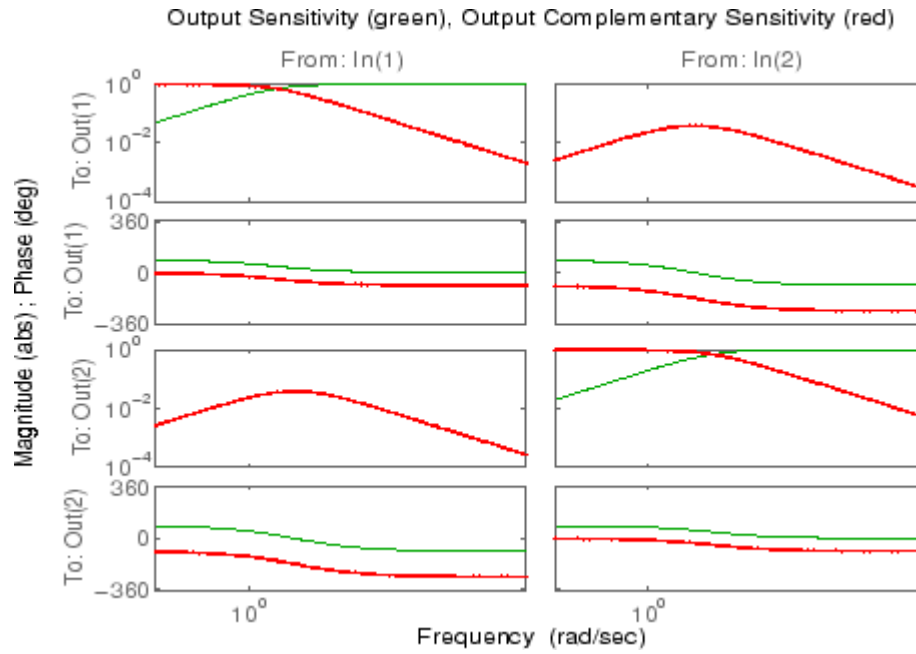
```
bodemag(P,'r',loops.PSi,'b')
```



Multi Input, Multi Output (MIMO)

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))
```



See Also

`loopmargin`

`robuststab`

`wcsens`

`wcmargin`

Purpose

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 0db crossover frequency of the `sigma` plot of $G_d(j\omega)$, then, roughly,

$$\underline{\sigma}(G(j\omega)K(j\omega)) \geq \frac{1}{\gamma}\ \bar{\sigma}(G_d(j\omega)) \quad \text{for all } \omega < \omega_0 \tag{2-14}$$

$$\bar{\sigma}(G(j\omega)K(j\omega)) \leq \gamma\ \underline{\sigma}(G_d(j\omega)) \quad \text{for all } \omega > \omega_0 \tag{2-15}$$

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 $\{\omega_{min},\omega_{max}\}$; ω_{max} should be at least ten times ω_{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`	$\{\omega_{min},\omega_{max}\}$ 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`.

Algorithm

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.,

$$\sigma(G_d) \approx \sigma(G_s) \text{ for all } \omega \in \{\omega_{min},\omega_{max}\}.$$

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 jω-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',[ø_min,ø_max]),
```

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 ω_{max} to be at least 100 times greater than ω_{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`=$\{\omega_{min},\omega_{max}\}$

Examples

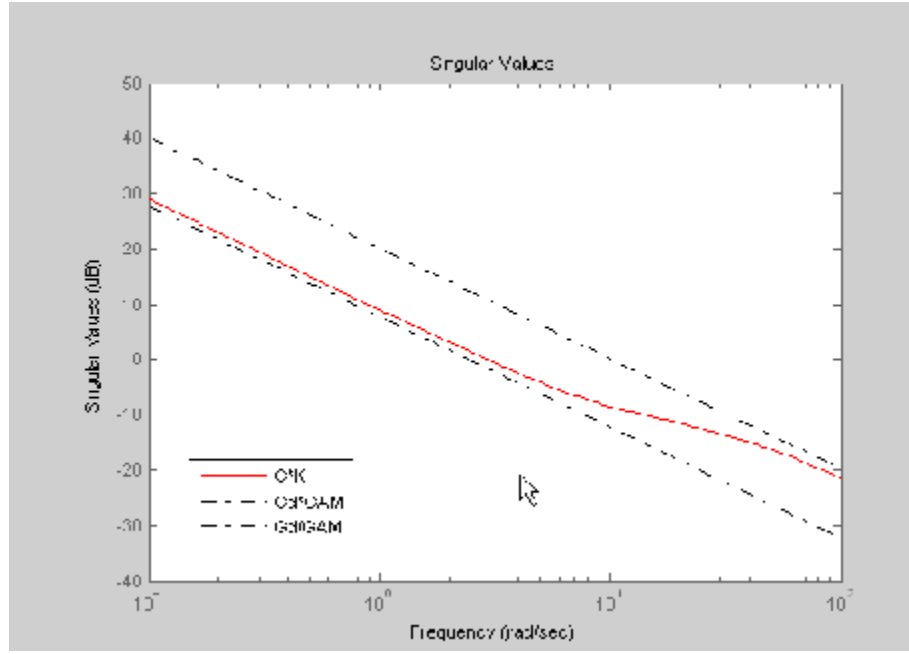
The following code generates the optimal `loopsyn` loopshaping control for the case of a 5-state, 4-output, 5-input plant with a full-rank non-minimum phase zero at s=+10. The result is shown in LOOPSYN controller on page 2-195.

```
rand('seed',0);randn('seed',0);
s=tf('s'); w0=5; Gd=5/s;           % desired bandwith 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
```

This figure shows that the LOOPSYN controller K optimally fits

```
sigma(G*K) = sigma(Gd)—GAM  %  dB
```

In the above example, `GAM` = 2.0423 = 6.2026 dB.



LOOPSYN controller

The `loopsyn` controller `K` optimally fits `sigma(G*K)`. As shown in the preceding figure, it is sandwiched between `sigma(Gd/GAM)`

and `sigma(Gd*GAM)` in accordance with the inequalities in Equation 2-14 and Equation 2-15. 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

$$N_K = N_{Gs} + N_W$$

$$= N_y N_{Gd} + N_{RHP} + N_W$$

$$= N_y N_{Gd} + N_{RHP} + N_G$$

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`.

Bibliography

[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

`loopsyn_demo`

`mixsyn`

`ncfsyn`

Purpose

Compute uncertain system bounding given LTI ss array

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)])).`

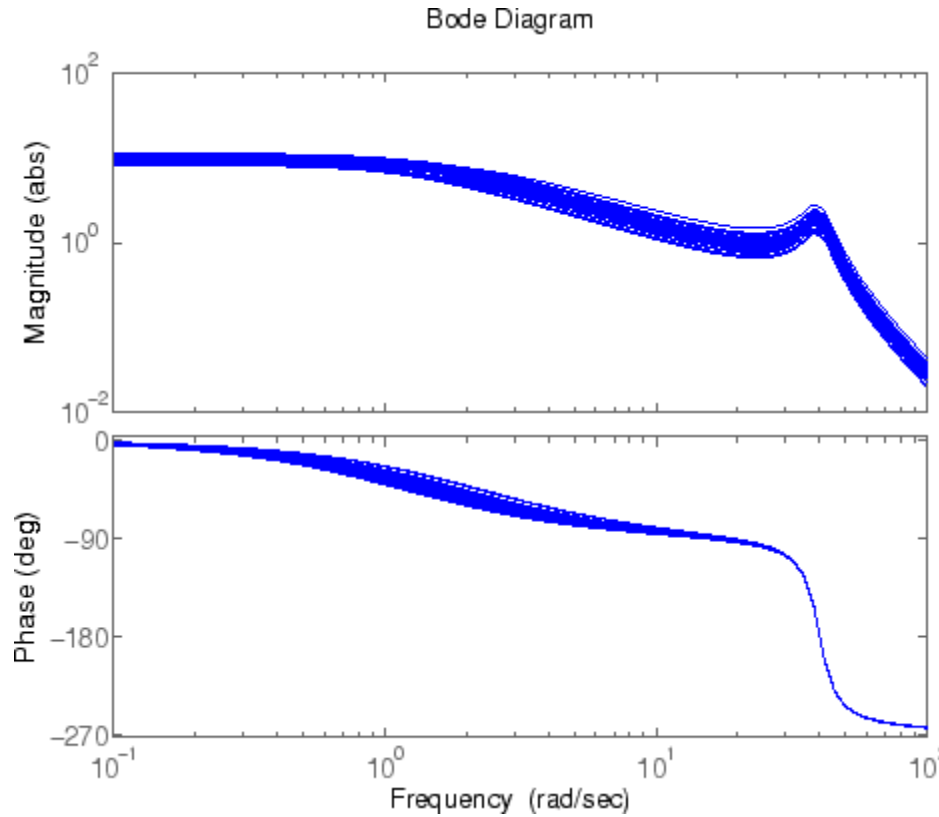
`[usys,wt,diffdata] = ltiarray2uss(P,Parray,ord,type)` returns the norm of the difference (absolute difference for additive, and relative difference for multiplicative uncertainty) between the nominal model `P` and `Parray`. `wt` satisfies `diffdata(w_i) < |wt(w_i)|` at all frequency points.

Examples

See the Robust Control Toolbox demo First-Cut Robust Design for a more detailed example of how to use `ltiarray2uss`.

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 frequency response data is acquired.

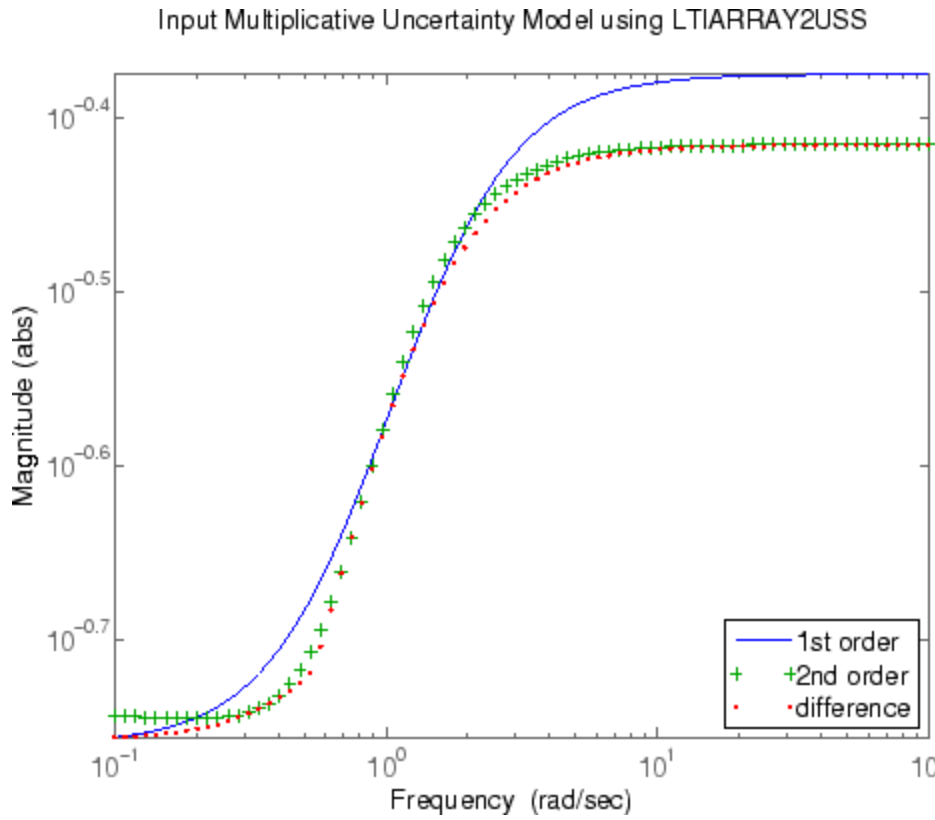
```
gain = ureal('gain',10,'Perc',20);
tau = ureal('tau',.6,'Range',[.42 .9]);
wn = 40;
zeta = 0.1;
usys = tf(gain,[tau 1])*tf(wn^2,[1 2*zeta*wn wn^2]);
sysnom = usys.NominalValue;
parray = usample(usys,30);
om = logspace(-1,2,80);
parrayg = frd(parray,om);
bode(parrayg)
```



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.

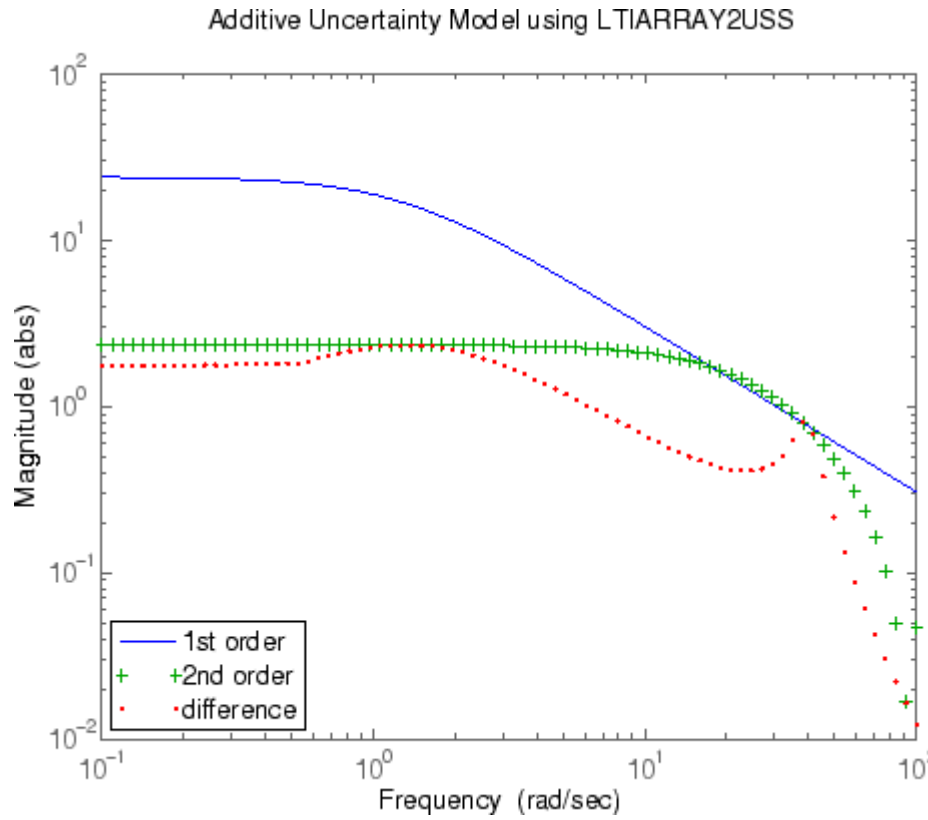
```
[umodIn1,wtIn1,diffdataIn] = ltiarray2uss(sysnom,parrayg,1);
[umodIn2,wtIn2,diffdataIn] = ltiarray2uss(sysnom,parrayg,2);
```

```
bodemag(wtIn1,'b-',wtIn2,'g+',diffdataIn,'r.',om)
```



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)
```



See Also

`fitmagfrd`

`ultidyn`

`uss`

ltrsyn

Purpose

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))`$\rightarrow 0$ as $\rho \rightarrow \infty$, 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))`$\rightarrow 0$ as $\rho \rightarrow \infty$, 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`,
`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

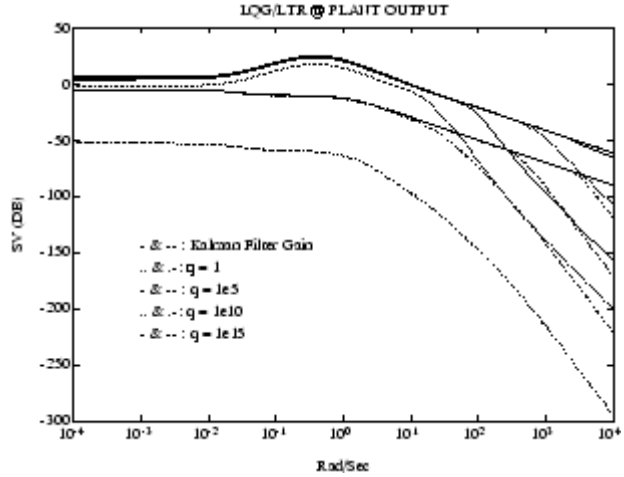
Algorithm

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

$$K(s) = [K_c(Is - A + BK_c + K_fC - K_fDK_c)^{-1}K_f]$$

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 $K_f = \Sigma C^T \Theta^{-1}$ where Σ satisfies the Kalman filter Riccati equation $0 = \Sigma A^T + A\Sigma - \Sigma C^T \Theta^{-1} C\Sigma + \Xi + \rho BB^T$. 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.

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`)

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

lqg

loopsyn

ltrdemo

ncfsyn

matnbr

Purpose

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

`decnbr`

`lmiinfo`

`decinfo`

Purpose

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`

Purpose

Minimize linear objective under LMI constraints

Syntax

```
[copt,xopt] = mincx(lmisys,c,options,xinit,target)
```

Description

The function `mincx` solves the convex program

$$\text{minimize } c^T x \text{ subject to } N^T L(x) N \leq M^T R(x) M \qquad \textbf{(2-16)}$$

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

`defcx`

`mincx`

`dec2mat`

`decnbr`

`feasp`

`gevp`

Purpose

H_∞ mixed-sensitivity synthesis method for robust control loopshaping design

Syntax

```
[K,CL,GAM,INFO]=mixsyn(G,W1,W2,W3)
[K,CL,GAM,INFO]=mixsyn(G,W1,W2,W3,KEY1,VALUE1,KEY2,VALUE2,...)
```

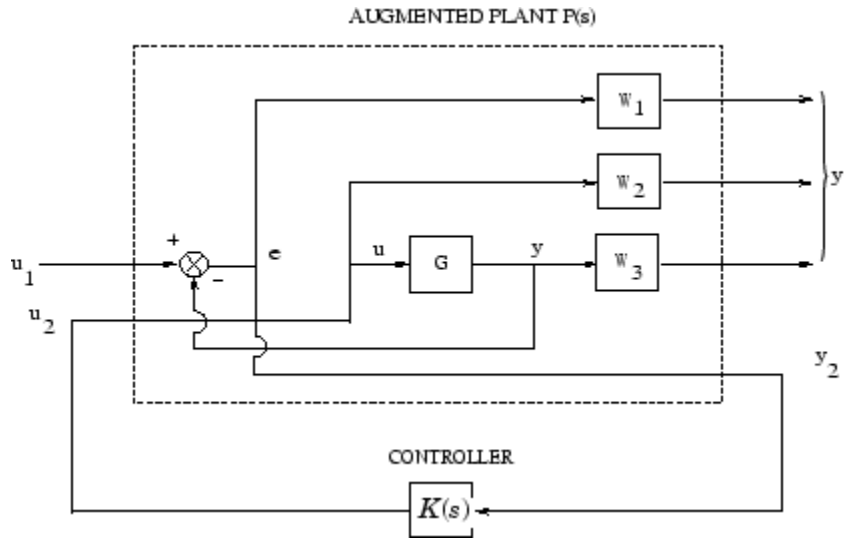
Description

`[K,CL,GAM,INFO]=mixsyn(G,W1,W2,W3)` computes a controller K that minimizes the H_∞ norm of the closed-loop transfer function the weighted mixed sensitivity

$$T_{y_1u_1} \triangleq \begin{bmatrix} W_1S \\ W_2R \\ W_3T \end{bmatrix}$$

where S and T are called the *sensitivity* and *complementary sensitivity,* respectively and S, R and T are given by

$$S = (I + GK)^{-1}$$
$$R = K(I + GK)^{-1}$$
$$T = GK(I + GK)^{-1}$$



Closed-loop transfer function $T_{y_1 u_1}$ for mixed sensitivity `mixsyn`.

The returned values of S, R, and T satisfy the following loop shaping inequalities:

$$\bar{\sigma}(S(j\omega)) \le \gamma \ \underline{\sigma}(W_1^{-1}(j\omega))$$

$$\bar{\sigma}(R(j\omega)) \le \gamma \ \underline{\sigma}(W_2^{-1}(j\omega))$$

$$\bar{\sigma}(T(j\omega)) \le \gamma \ \underline{\sigma}(W_3^{-1}(j\omega))$$

where γ=`GAM`. Thus, W_1, W_3 determine the shapes of sensitivity S and complementary sensitivity T. Typically, you would choose W_1 to be small inside the desired control bandwidth to achieve good disturbance attenuation (i.e., performance), and choose W_3 to be small outside the control bandwidth, which helps to ensure good stability margin (i.e., robustness).

For dimensional compatibility, each of the three weights W_1, W_2 and W_3 must be either empty, scalar (SISO) or have respective input dimensions

N_Y, N_U, and N_Y where G is N_Y-by-N_U. If one of the weights is not needed, you may simply assign an empty matrix []; e.g., `P = AUGW(G,W1,[],W3)` is SYS but without the second row (without the row containing `W2`).

Algorithm

`[K,CL,GAM,INFO]=mixsyn(G,W1,W2,W3,KEY1,VALUE1,KEY2,VALUE2,...)` is equivalent to

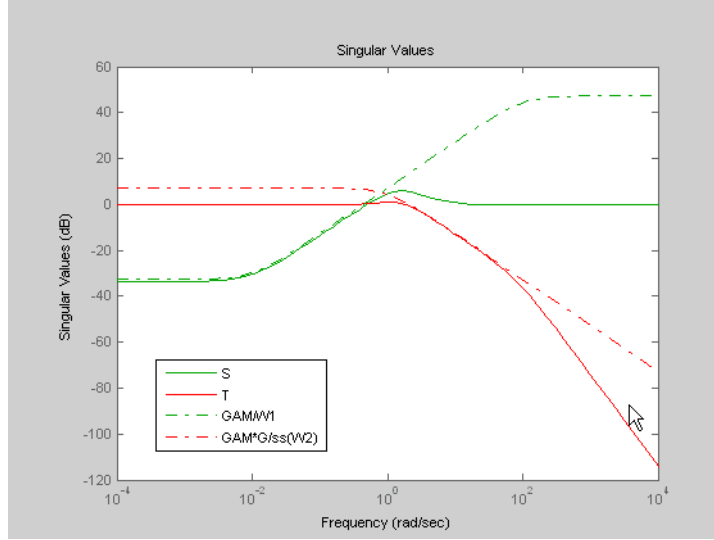
```
[K,CL,GAM,INFO]=...
    hinfsyn(augw(G,W1,W2,W3),KEY1,VALUE1,KEY2,VALUE2,...).
```

`mixsyn` accepts all the same key value pairs as `hinfsyn`.

Examples

The following code illustrates the use of `mixsyn` for sensitivity and complementary sensitivity 'loop-shaping'.

```
s=zpk('s');
G=(s-1)/(s+1)^2;
W1=0.1*(s+100)/(100*s+1); W2=0.1;
[K,CL,GAM]=mixsyn(G,W1,W2,[]);
L=G*K; S=inv(1+L); T=1-S;
sigma(S,'g',T,'r',GAM/W1,'g-.',GAM*G/ss(W2),'r-.')
```



`mixsyn(G,W1,W2,[ ])` shapes sigma plots of *S* and *T* to conform to $/W_1$ and G/W_2, respectively.

Limitations

The transfer functions G, W_1, W_2 and W_3 must be proper, i.e., bounded as $s \to \infty$ or, in the discrete-time case, as $z \to \infty$. Additionally, W_1, W_2 and W_3 should be stable. The plant G should be stabilizable and detectable; else, `P` will not be stabilizable by any `K`.

See Also

`augw`

`hinfsyn`

Purpose

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 string variable `type` specifies the type of filter and can be one of the following

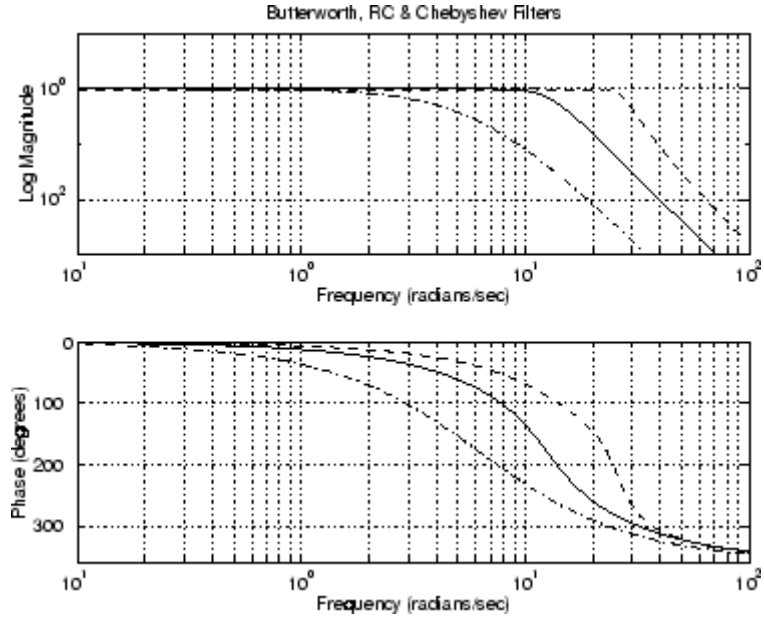
String variable	Description
`'butterw'`	Butterworth filter
`'cheby'`	Chebyshev filter
`'bessel'`	Bessel filter
`'rc'`	Series of resistor/capacitor filters

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

```
butw = mkfilter(2,4,'butterw');
cheb = mkfilter(4,4,'cheby',0.5);
rc = mkfilter(1,4,'rc');
bode(butw_g,'-',cheb_g,'--',rc_g,'-.')
megend('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

Purpose

Partition LTI system into two-input/two-output system

Syntax

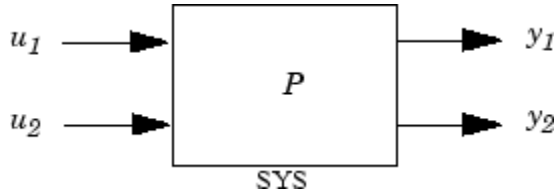
```
SYS=mktito(SYS,NMEAS,NCONT)
```

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`.

Algorithm

```
[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));
```

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]
```

See Also

augw

hinfsyn

h2syn

ltiprops

sdhinfsyn

modreal

Purpose

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.

Algorithm

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.

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`:

```
randn('state',1234); rand('state',5678);
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)
```

See Also

`reduce`

`balancmr`

`schurmr`

`bstmr`

`ncfmr`

`hankelmr`

`hankelsv`

msfsyn

Purpose

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` = $[\gamma_0, \nu_0, \alpha, \beta]$ 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

$$\begin{pmatrix} A(t) & B_1(t) & B_2(t) \\ C_1(t) & D_{11}(t) & D_{12}(t) \\ C_2(t) & 0 & D_{22}(t) \end{pmatrix} \in \mathbf{Co}\left\{ \begin{pmatrix} A_k & B_k & B_{2k} \\ C_{1k} & D_{11k} & D_{12k} \\ C_{2k} & 0 & D_{22k} \end{pmatrix} : k = 1, \ldots, K \right\}$$

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`

Purpose

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`, or `frd` object. If `M` is an N-D array (with $N \geq 3$), then the computation is performed pointwise along the third and higher array dimensions. If `M` is a `frd` object, the computations are performed pointwise in frequency (as well as any 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 $r \times c \times d_1 \times ... \times d_F$. Then `size(bounds)` is $1 \times 2 \times d_1 \times ... \times d_F$. 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 $d_1 \cdot d_2 ... d_F$ (the product of the d_k) would be valid.

`bounds = mussv(M,BlockStructure,Options)` specifies computation options. `Options` is a character string, containing any combination of the following characters:

Option	Meaning
`'a'`	Upper bound to greatest accuracy, using LMI solver
`'an'`	Same as `'a'`, but without automatic prescaling
`'d'`	Display warnings
`'f'`	Fast upper bound (typically not as tight as the default)
`'g6'`	Use gain-based lower bound (note that the default lower bound employs a power iteration) multiple times (in this case `10+6*10` times). Larger numbers typically give better lower bounds. This is an alternative to the default lower bound which uses a power iteration.

Option	Meaning
'i'	Reinitialize lower bound computation at each new matrix (only relevant if M is ND array or frd)
'm7'	Randomly reinitialize lower bound iteration multiple times (in this case 7 times, use 1-9); larger number typically gives better lower bound.
'o'	Run "old" algorithms, from version 3.1.1 and before. Included to allow exact replication of earlier calculations.
's'	Suppress progress information (silent).
'U'	Upper-bound "only" (lower bound uses a fast/cheap algorithm).
'x'	Decrease iterations in lower bound computation (faster but not as tight as default). Use 'U' for an even faster lower bound.

`[bounds,muinfo] = mussv(M,BlockStructure)` returns `muinfo`, a structure containing more detailed information. The information within `muinfo` must be extracted using `mussvextract`. See `mussvextract` for more details.

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.

```
randn(`state',929)
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 =
    1.6280 1.6007
[bounds(1)  bounds(2)]
ans =
    3.4827 3.3011
```

Algorithm

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 mu problem is solved by reformulating the optimization into a semidefinite program (Packard *et al.*, 1991).

References

- 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.
- 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.

- Osborne, E., "On preconditioning of matrices," *Journal of Associated Computer Machines,* Vol. 7, 1960, pp. 338–345.
- 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.
- Safonov, M., "Stability margins for diagonally perturbed multivariable feedback systems," *IEEE Proc.,* Vol. 129, Part D, 1992, pp. 251–256.
- 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.
- 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

Comprehensive analysis of feedback loop

`mussvextract`

`robuststab`

`robustperf`

`wcgain`

`wcsens`

`wcmargin`

Purpose

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 ß and matrices D and G, consistent with `BlockStructure`, such that

$$\bar{\sigma}\left(\left(I+G_L^2\right)^{-\frac{1}{4}}\left(\frac{D_L M D_R^{-1}}{\beta}-jG_M\right)\left(I+G_R^2\right)^{-\frac{1}{4}}\right)\le 1$$

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 ß 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

$$\bar{\sigma}(D_L M D_R^{-1}) \le \beta.$$

The LMI method consists of finding a scalar ß and matrices *D* and *G*, consistent with `BlockStructure`, such that

$$M' D_r M - \beta^2 D_c + j(G_{cr} M - M' G_{rc}) \le 0$$

is negative semidefinite. Again, *D* and *G* have a few different manifestations to match the row and column dimensions of M. Any such ß 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.

```
randn('state',0)
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 =
1.0000e+000            0            0            0
           0  9.9190e-001            0            0
           0            0  1.1255e+000            0
           0            0            0  1.1255e+000
Dr = VSigma.DRight
Dr =
1.0000e+000            0            0            0
           0  9.9190e-001            0            0
           0            0  1.1255e+000            0
           0            0            0  1.1255e+000
[norm(Dl*M/Dr) bounds(1)]
ans =
4.3420e+000  4.3420e+000
```

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 =
-2.0045e-005 +6.1649e-016i
 -1.4688e+001 -2.4975e-016i
 -2.0436e+001 -4.7583e-016i
 -1.9100e+001 +1.4136e-015i
```

Note that `VDelta` matches the structure defined by `BlockStructure`, and the norm of `VDelta` agrees with the lower bound,

```
VDelta
VDelta =
1.0698e-001 -2.0405e-001i            0                        0                        0
           0             1.4920e-001 +1.7556e-001i            0                        0
```

```
            0                         0              -5.4173e-002 -1.0932e-002i -5.7140e-002 +9
            0                         0               2.8071e-002 -8.0807e-002i -1.3608e-001 -1
[norm(VDelta) 1/bounds(2)]
ans =
    0.2304 0.2304
```

and that `M*VDelta` has an eigenvalue exactly at 1.

```
eig(M*VDelta)
ans =
1.0000e+000 -8.3267e-017i
 -6.1108e-002 +2.5748e-001i
  4.1427e-018 -5.8578e-018i
 -1.9637e-001 -5.6540e-002i
```

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 =
    4.342 4.340
    3.470 3.470
```

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 =
-4.4665e-002 -4.4823e-019i
 -5.2486e-004 +1.5623e-018i
 -1.8028e-003 +3.2493e-019i
 -1.2558e-003 +1.2973e-019i
```

`VDelta2` matches the structure defined by `BlockStructure`, and the norm of `VDelta2` agrees with the lower bound,

```
VDelta2
VDelta2 =
   0.2882 0                 0                 0
        0            0.2882 0                 0
        0                 0           -0.152 - 0.2448i        0
        0                 0                 0           -0.0395 -0.2855i
[norm(VDelta2) 1/bounds2(2)]
ans =
```

```
    0.2882 0.2882
```

and that `M*VDelta2` has an eigenvalue exactly at 1.

```
eig(M*VDelta2)
ans =
-3.3623e-001 +2.1885e-001i
 -3.6805e-001 -1.5645e-001i
  1.0000e+000 -1.4169e-016i
  4.5066e-001 -3.4481e-001i
```

See Also

`mussv`

Purpose

Calculate normalized coprime stability margin of plant-controller feedback loop

Syntax

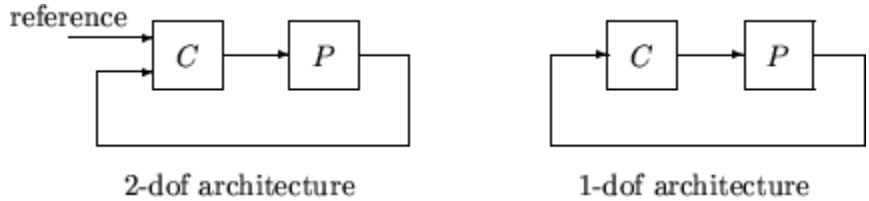
```
[marg,freq] = ncfmargin(P,C)
[marg,freq] = ncfmargin(P,C,tol)
```

Description

`[marg,freq] = ncfmargin(P,C)` calculates the normalized coprime factor/gap metric robust stability margin $b(P, C)$, `marg`, of the multivariable feedback loop consisting of `C` in negative feedback with `P`. The normalized coprime factor $b(P, C)$ is defined as

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

`C` should only be the compensator in the feedback path, such as the *1-dof* architecture shown below (on the right). If the compensator has *2-dof* architecture shown below (on the left), you must eliminate the reference channels before calling `ncfmargin`. `freq` is the frequency associated with the upper bound on `marg`.



2-dof architecture 1-dof architecture

The normalized coprime factor robust stability margin lies between 0 and 1 and is used as an indication of robustness to unstructured perturbations. Values of `marg` greater than 0.3 generally indicate good robustness margins.

`[marg,freq] = ncfmargin(P,C,tol)` calculates the normalized coprime factor/gap metric robust stability of the multivariable feedback loop consisting of `C` in negative feedback with `P`. `tol` specifies a relative accuracy for calculating the normalized coprime factor metric and must be between 10^{-5} and 10^{-2}. `tol=0.001` is the default value.

Examples

Consider the plant model `4/(s-0.001)` an unstable first order, and two constant gain controllers, `k1 = 1` and `k2 = 10`. Both controllers stabilize the closed-loop system

```
x = tf(4,[1 0.001]);
clp1 = feedback(x,1)
```

The transfer function `clp1` is shown as is `clp2`.

```
     4
---------
s + 4.001

clp2 = feedback(x,10)

Transfer function:
  4
------
s + 40
```

The closed-loop system with controller `k1`, `clp1`, has a normalized coprime factor robust stability margin of `0.71` that is achieved at infinite frequency. This indicates that the closed-loop system is very robust to unstructured perturbations. The closed-loop system with controller `k2`, `clp2`, has a normalized coprime factor robust stability margin of `0.10`. This indicates that the closed-loop system is not robust to unstructured perturbations.

```
[marg1,freq1] = ncfmargin(x,1)
marg1 =
    0.7071
freq1 =
   Inf
[marg2,freq2] = ncfmargin(x,10)
marg2 =
    0.0995
freq2 =
   Inf
```

Construct an uncertain system, xu, by adding an 11% unmodeled dynamics to the nominal system x. Calculate the robust stability of the closed-loop system with the feedback gain 1 and 10.

```
xu = x + ultidyn('uncstruc',[1 1],'Bound',0.11);
[stabmarg1, du1, report1] = robuststab(feedback(xu,1));
disp(report1{1})
Uncertain System is robustly stable to modeled uncertainty.
 -- It can tolerate up to 909% of modeled uncertainty.
 -- A destabilizing combination of 909% the modeled uncertainty exists,
causing an instability at 165 rad/s.

[stabmarg10, du10, report10] = robuststab(feedback(xu,10));
disp(report10{1})
Uncertain System is NOT robustly stable to modeled uncertainty.
 -- It can tolerate up to 90.9% of modeled uncertainty.
 -- A destabilizing combination of 90.9% the modeled uncertainty exists,
causing an instability at 1.64e+003 rad/s.
```

The closed-loop system with K=1 is robustly stable in the presence of the unmodeled dynamics based on the robust stability analysis. In fact, the closed-loop system with K=1 can tolerate 909% (or 9.09*11%) of the unmodeled LTI dynamics, whereas the closed-loop system is not robustly stable with a constant gain of 10 controller. The closed-loop system with K=10 implemented can only tolerate 90.9% (or.909*11%) of the unmodeled LTI dynamics.

Algorithm

The computation of the gap amounts to solving 2-block H_∞ problems, Georgiou, Smith, 1988. The particular method used here for solving the H_∞ problems is based on Green *et al.*, 1990. The computation of the nugap uses the method of Vinnicombe, 1993.

References

- McFarlane, D.C. and K. Glover, Robust Controller Design using Normalised Coprime Factor Plant Descriptions, Lecture Notes in Control and Information Sciences, Springer Verlag, Vol. 138, 1989.

- McFarlane, D.C., and K. Glover, “A Loop Shaping Design Procedure using Synthesis,” IEEE Transactions on Automatic Control, Vol. 37, No. 6, 1992, pp. 759-769.
- Vinnicombe, G., “Measuring Robustness of Feedback Systems,” Ph.D. Dissertation, Department of Engineering, University of Cambridge, 1993.

See Also

Performs a comprehensive analysis of feedback loop

`gapmetric`

`norm`

`wcmargin`

Purpose

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

$$U_r N_r + V_r M_r = I$$

$$N_l U_l + M_l V_l = I$$

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 comprimeness 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'`overides `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.

Algorithm

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].

$$G_l = \begin{bmatrix} N_l & M_l \end{bmatrix}$$

$$G_r = \begin{bmatrix} N_r \\ M_r \end{bmatrix}$$

2 Perform k^{th} order square root balanced model truncation on G_l (or G_r) [2].

3 The reduced model `GRED` is :

$$\left[\begin{array}{c|c} \hat{A} & \hat{B} \\ \hline \hat{C} & \hat{D} \end{array}\right] = \left[\begin{array}{c|c} A_c - B_m C_l & B_n - B_m D_l \\ \hline C_l & D_l \end{array}\right]$$

where

$N_l(:= A_c, B_n, C_c, D_n)$

$M_l := (A_c, B_m, C_c, D_m)$

$C_l = (D_m)^{-1} C_c$

$D_l = (D_m)^{-1} D_n$

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:

```
rand('state',1234); randn('state',5678);
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
```

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

reduce

balancmr

schurmr

bstmr

hankelmr

hankelsv

Purpose

Loop shaping design using Glover-McFarlane method

Syntax

```
[K,CL,GAM,INFO]=ncfsyn(G)
[K,CL,GAM,INFO]=ncfsyn(G,W1)
[K,CL,GAM,INFO]=ncfsyn(G,W1,W2)
[K,CL,GAM,INFO]=ncfsyn(G,W1,W2,'ref')
```

Description

`ncfsyn` is a method for designing controllers that uses a combination of loop shaping and robust stabilization as proposed in McFarlane and Glover [1]-[2]. The first step is for you to select a pre- and post-compensator W_1 and W_2, so that the gain of the 'shaped plant' $G_s := W_2 G W_1$ is sufficiently high at frequencies where good disturbance attenuation is required and is sufficiently low at frequencies where good robust stability is required. The second step is to use `ncfsyn` to compute an optimal *positive* feedback controllers `K`.

The optimal `Ks` has the property that the `sigma` plot of the shaped loop

```
Ls=W2*G*W1*Ks
```

matches the target loop shape G_s optimally, roughly to within plus or minus `20*log10(GAM)` decibels. The number margin `GAM=1/ncfmargin(Gs,K)` and is always greater than 1. `GAM` gives a good indication of robustness of stability to a wide class of unstructured plant variations, with values in the range 1<`GAM`<3 corresponding to satisfactory stability margins for most typical control system designs.

`[K,CL,GAM,INFO]=ncfsyn(G,W1,W2,'ref')` computes the Glover-McFarlane H_∞ normalized coprime factor loop-shaping controller `K`, with a reference command, for `lti` plant `G`, weights `W1` and `W2` if the `'ref'`option is included. The closed-loop system `CL` represents the transfer matrix from the reference and disturbance to the feedback error and output of `W1`.

Algorithm

`K=W2*Ks*W1`, where `Ks` $= K_\infty$ is an optimal H_∞ controller that simultaneously minimizes the two H_∞ cost functions

$$\gamma := \min_K \left\| \begin{bmatrix} I \\ K \end{bmatrix} (I - G_s K)^{-1} [G_s, I] \right\|_\infty$$

$$\gamma := \min_K \left\| \begin{bmatrix} I \\ G_s \end{bmatrix} (I - K G_s)^{-1} [K, I] \right\|_\infty$$

Roughly speaking, this means for most plants that

$$\sigma(W_2 G W_1 K_\infty), \text{ db} = \sigma(W_2 G W_1), \text{ db} \pm \gamma, \text{ db}$$

$$\sigma(K_\infty W_2 G W_1), \text{ db} = \sigma(W_2 G W_1), \text{ db} \pm \gamma, \text{ db},$$

so you can use the weights W_1 and W_2 for loopshaping. For a more precise bounds on loopshaping accuracy, see Theorem 16.12 of Zhou and Glover.

Theory ensures that if $G_s = NM^{-1}$ is a normalized coprime factorization (NCF) of the weighted plant model G_s satisfying

$$G_s = N(jw)^* N(jw) + M(jw)^* M(jw) = I,$$

then the control system will remain robustly stable for any perturbation $\tilde{G}_s$ to the weighted plant model G_s that can be written

$$\tilde{G}_s = (N + \Delta_1)(M + \Delta_2)^{-1}$$

for some stable pair Δ_1, Δ_2 satisfying

$$\left\| \begin{bmatrix} \Delta_1 \\ \Delta_2 \end{bmatrix} \right\|_\infty < \text{MARG:=1/GAM} \quad .$$

The closed-loop H_∞-norm objective has the standard signal gain interpretation. Finally it can be shown that the controller, K_∞, 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 $K=W_1K_\infty W_2$.

Input Arguments

`G`	LTI plant to be controlled
`W1,W2`	Stable minimum-phase LTI weights, either SISO or MIMO. Default is $W_1=I$, $W_2=I$
`'ref'`	Reference input to controller. Default is no reference input is included.

Output Arguments

`K`	LTI controller `K= W1*Ks*W2`
`CL`	$\begin{bmatrix} I \\ K_\infty \end{bmatrix}(I - W_2GW_1K_\infty)^{-1}[W_2GW_1, I]$, LTI H_∞ optimal closed loop
`GAM`	H_∞ optimal cost $\gamma = \dfrac{1}{\overline{b}(W_2GW_1, K_\infty)} =$ `hinfnorm(CL)` ≥ 1
`INFO`	Structure array containing additional information

Additional output `INFO` fields

INFO.emax	nugap robustness emax=1/GAM=ncfmargin(Gs,-Ks)=$b(W_2GW_1,K_\infty)$
INFO.Gs	'shaped plant' Gs=W2*G*W1
INFO.Ks	Ks = K[[BULLET]] = NCFSYN(Gs) = NCFSYN(W2*G*W1)

[MARG,FREQ] = ncfmargin(G,K,TOL) calculates the normalized coprime factor/gap metric robust stability margin assuming *negative* feedback.

$$\text{MARG} = b(G,-K) = 1/\left\| \begin{bmatrix} I \\ -K \end{bmatrix} (I+GK)^{-1} [G \ , I] \right\|_\infty$$

where G and K are LTI plant and controller, and TOL (default=.001) is the tolerance used to compute the H_∞ norm. FREQ is the peak frequency. That is, the frequency at which the infinity norm is reached to within TOL.

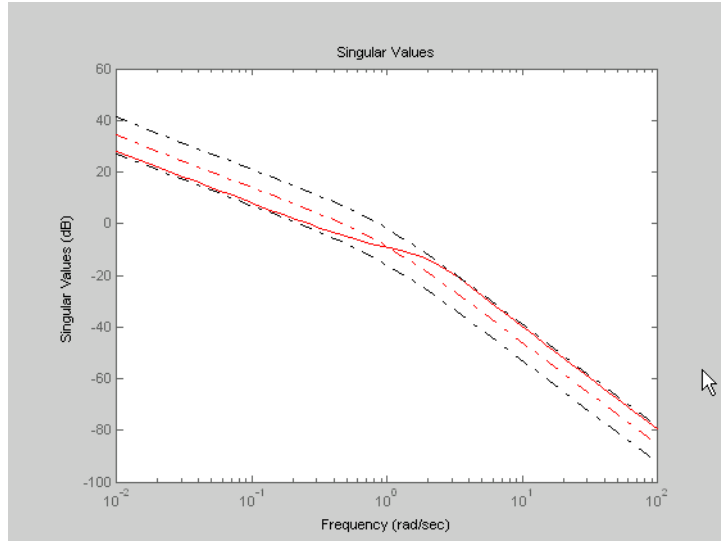
Algorithm

See McFarlane and Glover [1]-[2] for details.

Examples

The following code shows how ncfsyn can be used for loop-shaping. The achieved loop G*K has a sigma plot is equal to that of the target loop G*W1 to within plus or minus 20*log10(GAM) decibels.

```
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-.')
```



Achieved loop G*K and shaped loop Gs, ±20log(GAM) db

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

Purpose

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

`setlmis`

`lmivar`

`lmiterm`

`getlmis`

`lmiedit`

`dellmi`

normalized2actual

Purpose

Convert value for atom in normalized coordinates to corresponding actual value

Syntax

```
avalue = normalized2actual(A,NV)
```

Description

Converts a normalized value NV of a 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`

`robuststab`

`robustperf`

Purpose

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 $p = (p_1, \ldots, p_n) \in \mathbf{R}^n$, `pdlstab` seeks a Lyapunov function of the form

$V(xp,) = x^T Q(p)–1x,\ Q(p) = Q_0 + p_1 Q_1 + \ldots p_n Q_n$

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

$$\begin{pmatrix} A + jE & B \\ C & D \end{pmatrix} = \sum_{i=1}^{n} \alpha_i \begin{pmatrix} A + jE_i & B_i \\ C_i & D_i \end{pmatrix}, \quad \alpha_i \geq 0, \quad \sum_{i=1}^{n} \alpha_i = 1, \tag{2-17}$$

`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 2-17.

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

Remark

For affine parameter-dependent systems with *time-invariant* parameters, there is equivalence between the robust stability of

$$E(p)\dot{x} = A(p)x \tag{2-18}$$

and that of the dual system

$$E(p)^T\dot{z} = A(p)^T z \tag{2-19}$$

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 2-19 when it fails on Equation 2-18.

See Also

`quadstab`

Purpose

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`

polydec

Purpose

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

`pvec`

`pvinfo`

`aff2pol`

`hinfgs`

Purpose

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 $x^T P x$ 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

`quadstab`

`pdlstab`

psinfo

Purpose

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, . . ., S_n, the entries of `p` should be real parameter values p_1, . . ., 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

  $$\mathrm{Co}\{S_1, \ldots, S_n\},$$

the entries of p should be polytopic coordinates $p_1, \ldots, p_n$ satisfying $p_j \geq 0$ and the result is the interpolated LTI system of SYSTEM matrix

$$S = \frac{p_1 S_1 + \ldots + p_n S_n}{p_1 + \ldots + p_n}$$

See Also

psys

Purpose

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.

Two types of uncertain state-space models can be manipulated in the LMI Control Toolbox:

- *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:

  $$\underbrace{\begin{bmatrix} A(t)+jE(t) & B(t) \\ C(t) & D(t) \end{bmatrix}}_{S(t)} \in \mathbf{Co}\left\{ \underbrace{\begin{bmatrix} A_1+jE_1 & B_1 \\ C_1 & D_1 \end{bmatrix}}_{S_1}, \ldots, \underbrace{\begin{bmatrix} A_k+jE_k & B_k \\ C_k & D_k \end{bmatrix}}_{S_k} \right\}$$

  where $S_1, \ldots, S_k$ are given "vertex" systems and

  $$\mathrm{Co}\{S_1, \ldots, S_k\} = \left\{ \sum_{i=1}^{k} \alpha_i S_i \ : \ \alpha_i \geq 0, \ \sum_{i=1}^{k} \alpha_i = 1 \right\}$$

  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 $A(\cdot)$; $B(\cdot)$, . . ., $E(\cdot)$ are fixed affine functions of some vector $p = (p_1, \ldots, p_n)$ of real parameters, i.e.,

$$\underbrace{\begin{bmatrix} A(p) + jE(p) & B(p) \\ C(p) & D(p) \end{bmatrix}}_{S(p)} =$$

$$\underbrace{\begin{bmatrix} A_0 + jE_0 & B_0 \\ C_0 & D_0 \end{bmatrix}}_{S_0} + p_1 \underbrace{\begin{bmatrix} A_1 + jE_1 & B_1 \\ C_1 & D_1 \end{bmatrix}}_{S_1} + \ldots + p_n \underbrace{\begin{bmatrix} A_n + jE_n & B_n \\ C_n & D_n \end{bmatrix}}_{S_n}$$

where S_0, S_1, . . ., S_n are given `SYSTEM` matrices. The parameters p_i can be time-varying or constant but uncertain.

Both types of models are specified with the function `psys`. 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, . . ., 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

`psinfo`

`pvec`

aff2pol

Purpose

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

$$\underline{p}_j \le p_j \le \overline{p}_j$$

The parameter vector p then takes values in a hyperrectangle of $\mathbf{R}^n$ called the parameter box. The second argument range is an n-by-2 matrix that stacks up the extremal values $\underline{p}_j$ and $\overline{p}_j$ 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 $\underline{v}_j$ and $\overline{v}_j$ on $\frac{dp_j}{dt}$:

$$\underline{v}_j \le \frac{dp_j}{dt} \le \overline{v}_j$$

Set $\underline{v}_j$ = `Inf` and $\overline{v}_j$ = `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 $\mathbf{R}^n$. 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

$p_1 \in [-1, 2]$, $p_2 \in [20, 50]$

The corresponding parameter vector $p = (p_1, p_2)$ is specified by

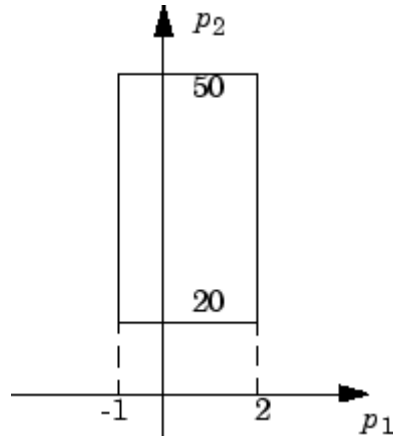
```
pv = pvec('box',[-1 2;20 50])
```

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

$$v_1 = \begin{pmatrix} -1 \\ 20 \end{pmatrix}, \quad v_2 = \begin{pmatrix} -1 \\ 50 \end{pmatrix}, \quad v_3 = \begin{pmatrix} 2 \\ 20 \end{pmatrix}, \quad v_4 = \begin{pmatrix} 2 \\ 50 \end{pmatrix}$$

Hence, you could also specify p by

```
pv = pvec('pol',[v1,v2,v3,v4])
```



Parameter box

See Also

`pvinfo`

`psys`

pvinfo

Purpose

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

`pvec` 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,

$$\mathrm{pmin} \le p_j(t) \le \mathrm{pmax}, \quad \mathrm{dpmin} \le \frac{dp_j}{dt} \le \mathrm{dpmax}$$

For the type `'pol'`:

```
pvinfo(pv,'par',j)
```

returns the j-th vertex of the polytope of $\mathbf{R}^n$ 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 $(V_1, \ldots, V_k)$. 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

pvec

psys

Purpose

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

$$E(t)\dot{x} = A(t)x + B(t)u, \qquad y = C(t)x + D(t)u \tag{2-20}$$

is the smallest $\gamma > 0$ such that

$$\|y\|_{L_2} \le \gamma \|u\|_{L_2} \tag{2-21}$$

for all input $u(t)$ with bounded energy. A sufficient condition for Equation 2-21 is the existence of a quadratic Lyapunov function

$$V(x) = x^T P x,\ P > 0$$

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 2-20 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

`quadstab`

`psys`

Purpose

Quadratic stability of polytopic or affine parameter-dependent systems

Syntax

```
[tau,P] = quadstab(ps,options)
```

Description

For affine parameter-dependent systems

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

or polytopic systems

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

`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

  $$A^TQE + EQA^T < \tau I \text{ for all admissible values of } (A, E)$$

  $$Q > I$$

  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 $P = Q_{opt}^{-1}$. 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

`pdlstab`

`decay`

`quadperf`

`psys`

randatom

Purpose

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')
Uncertain Real Parameter: Name BMSJA, NominalValue -6.75,
Range [-7.70893  -1.89278]
```

The following statement creates the variable `ultidyn` uncertain object `xlti` with three inputs and four outputs. You will get the results shown below if both the random variable seeds are set to 29.

```
rand('seed',29);
randn('seed',29);
xlti = randatom('ultidyn',[4 3])
Uncertain GainBounded LTI Dynamics: Name OOJGS, 4x3,
```

```
   Gain Bound = 0.646
```

See Also

rand

randn

randumat

randuss

ucomplex

ucomplexm

ultidyn

randumat

Purpose

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)
UMAT: 2 Rows, 3 Columns
  ROQAW: complex, nominal = 9.92+4.84i, radius = 0.568,
 1 occurrence
  UEPDY: real, nominal = -5.81, variability = [-1.98681  0.133993],
 3 occurrences
  VVNHL: complex, nominal = 5.64-6.13i, radius = 1.99,
 2 occurrences
```

The following statement creates the `umat` uncertain object `x2` of size 4-by-2 with the seed 91.

```
rand('seed',91); randn('seed',91);
x2 = randumat(4,2)
UMAT: 4 Rows, 2 Columns
  SSAFF: complex, nominal = -0.366+2.81i, radius = 1.76,
3 occurrences
  VDTIH: complex, nominal = -3.03-3i,  +/- 27.5%, 2 occurrences
  XOLLJ: real, nominal = 0.0628, range = [-3.73202  4.28174],
1 occurrence
```

See Also

rand

randn

randatom

randuss

ucomplex

ultidyn

randuss

Purpose

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 nth 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 nth order single-input uncertain continuous-time system with `p` outputs.

`usys = randuss(n,p,m)` generates an nth order uncertain continuous-time system with `p` outputs and `m` inputs.

`usys = randuss(n,p,m,Ts)` generates an nth 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

randn

randatom

randumat

ucomplex

ultidyn

frd/rcond

Purpose

LAPACK reciprocal condition estimator of `frd` object

Syntax

```
r = rcond(x)
```

Description

`rcond(x)` is an estimate for the reciprocal of the condition of the `frd` object `x` in the 1-norm obtained by the LAPACK condition estimator. `rcond` operates on `x.ReponseData` of the `x frd` at each frequency to construct `r`. If `x` is well conditioned, `rcond(x)` is near 1.0. If `x` is badly conditioned, `rcond(x)` is near EPS.

`r=rcond(x)` returns `r` as an `frd` object.

See Also

`cond`

`norm`

`condest`

`normest`

Purpose

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,* $\| \texttt{G-GRED} \|_\infty$, *multiplicative error,* $\|\texttt{G}^{-1}\texttt{(G-GRED)}\|_\infty$, 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.

Argument	Value	Description
'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.
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

Argument	Description
	• 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

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:

```
rand('state',1234); randn('state',5678);G = rss(30,5,4);
[g1, redinfo1] = reduce(G); % display Hankel SV plot
                            % and prompt for order
[g2, redinfo2] = reduce(G,20); % default to balancmr
[g3, redinfo3] = reduce(G,[10:2:18],'algorithm','schur');
                            % select schurmr
[g4, redinfo] = reduce(G,'ErrorType','mult','MaxError',[0.01, 0.05]);
rand('state',12345); randn('state',6789);
wt1 = rss(6,5,5); wt1.d = eye(5)*2;
wt2 = rss(6,4,4); wt2.d = 2*eye(4);
[g5, redinfo5] = reduce(G, [10:2:18], 'weight',{wt1,wt2});
[g6, redinfo6] = reduce(G,'ErrorType','add','algorithm','hankel, ...
       'maxerror',[0.01]);
for i = 1:6
    figure(i); eval(['sigma(G,g' num2str(i) ');']);
end
```

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`

`schurmr`

`bstmr`

`ncfmr`

`hankelmr`

`hankelsv`

Purpose

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])
```

Purpose

Create options object for use with `robuststab` and `robustperf`

Syntax

```
opts = robopt
opts = robopt('name1',value1,'name2',value2,...)
```

Description

`opts = robopt` (with no input arguments) creates an options object with all the properties set to their default values.

`opts = robopt('name1',value1,'name2',value2,...)` creates a `robopt` object in which specified properties have the given values. Any unspecified property is set to its default value. It is sufficient to type only enough leading characters to define the property name uniquely. Case is ignored for property names.

`robopt` with no input or output arguments displays a complete list of option properties and their default values.

Fields

The following are the `robopt` object properties:

Object Property	Description
`Display`	Displays progress of computations `{'on';'off'}`. Default is `'off'`
`Sensitivity`	Computes margin sensitivity to individual uncertainties `{'on';'off'}`. Default is `'on'`.
`VaryUncertainty`	Percentage variation of uncertainty used as a stepsize in finite-difference calculations to estimate sensitivity. Default is 25.
`Mussv`	Option used in internal structured singular value calculations (when calling `mussv`). Default is `'sm9'`.

Object Property	Description
`Default`	Structure, field names are `robopt` properties, and values are the default values.
`Meaning`	Structure, field names are `robopt` properties, and values are the text description of the property.

Examples

You can create a `robopt` options object called `opt` with all default values.

```
opt = robopt
Property Object Values:
            Display: 'off'
        Sensitivity: 'on'
    VaryUncertainty: 25
              Mussv: 'sm9'
            Default: [1x1 struct]
            Meaning: [1x1 struct]
```

An elementary finite-difference scheme is used in estimating local sensitivities. The property `VaryUncertainty` denotes the step size used in estimating the derivatives necessary in computing sensitivities.

In the following statements, you are requesting that the sensitivity of the robust stability margin calculation to a 50% variation in individual uncertainties be calculated. The `robopt` options properties `'Sensitivity'` and `'VaryUncertainty'` are set individually.

```
opt = robopt;
opt.VaryUncertainty = 50;
opt
Property Object Values:
            Display: 'off'
        Sensitivity: 'on'
    VaryUncertainty: 50
              Mussv: 'sm9'
            Default: [1x1 struct]
```

```
                    Meaning: [1x1 struct]
```

See Also

dkitopt

robuststab

robustperf

wcgopt

wcsens

wcmargin

Purpose

Robust performance margin of uncertain multivariable system

Syntax

```
perfmarg = robustperf(sys)
[perfmarg,wcu,report,info] = robustperf(sys)
[perfmarg,wcu,report,info] = robustperf(sys,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. The relationship between `robustperf` and other measures, such as `robuststab` and `wcgain`, is described in "Generalized Robustness Analysis"in the Robust Toolbox User's Guide.

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. If the input system `sys` is a `ufrd`, then the analysis is performed on the frequency grid within the `ufrd`. If the input system `sys` is a `uss`, then 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.

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 `sys`. 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

```
[perfmarg,perfmargunc,Report] = robustperf(sys)
```

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 "Generalized Robustness Analysis" in the online documentation.

`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 "Generalized Robustness Analysis". There are *M* field names, which are the names of uncertain elements of `sys`.

`Report` is a text description of the robust performance analysis results.

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 "Generalized Robustness 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 (it should always be, as described in "Generalized Robustness Analysis").

```
[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 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(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 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 `robopt` object is `'off'`, the values are set to `NaN`.
`Frequency`	*N*-by-1 frequency vector associated with analysis.
`BadUncertainValues`	*N*-by-1 cell array, with one entry for each frequency point. The kth entry `Info.BadUncertainValues{k}` is a `struct` of values of uncertain elements resulting from a robust performance analysis at frequency `Info.Frequency(k)`.
`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`.

Options (e.g., controlling what is displayed during the computation, turning on/off the sensitivity computation, setting the step size in the sensitivity computation, and controlling the option argument used in the underlying call to `mussv`) is specified using the robustness analysis options `robopt` object. For instance, you can turn the display on and turn off the sensitivity by executing

```
opt = robopt('Sensitivity','off','Display','on');
[PerfMarg,Destabunc,Report,Info] = robustperf(sys,opt)
```

Handling Array Dimensions

If `sys` has array dimensions (for example, suppose that the size of `sys` is $r \times c \times d_1 \times d_2 \times ... \times d_F$, refer to the $d_1 \times d_2 \times ... \times d_F$ as the *array dimensions*) then the margin calculation is performed "pointwise" (individually, at each and every array value) and the computed answers all have array dimensions as well. Details are described below. Again, assume that there are N frequency points and M uncertain elements.

The results of

```
[perfmarg,perfmargunc,Report,Info] = robustperf(sys,opt)
```

are `perfmarg` a structure with the following fields

Field	Description
`LowerBound`	$d_1 \times ... \times d$F, lower bound on stability margin across the array dimensions.
`UpperBound`	$d_1 \times ... \times d$F, upper bound on performance margin across the array dimensions. Using single indexing, for each `i`, the upper bound on the performance margin of `sys(:,:,i)` is `perfmarg.UpperBound(i)`.
`CriticalFrequency`	$d_1 \times ... \times d$F, the value of frequency at which the performance degradation curve crosses the $y=1/x$ curve. Using single indexing, for each `i`, the frequency at which the performance degradation curve crosses the $y=1/x$ curve in robust performance analysis of `sys(:,:,i)` is `perfmarg.CriticalFrequency(i)`. See "Generalized Robustness Analysis".

`perfmargunc` is a $d_1 \times ... \times d_F$ structure array of values of uncertain elements, associated with the intersection of the performance degradation curve and the $y=1/x$ curve. See "Generalized Robustness Analysis" in the online documentation. Using single indexing, for each `i`, the `struct` of values of uncertain elements for uncertain system `sys(:,:,i)` is `perfmargunc(i)`.

`Report` is a character array, dimensions 3, 4, ..., `F+2` are $d_1 \times ... \times d_F$, containing text description of the robustness analysis results at each grid in the array dimensions.

In addition to the first 3 output arguments, described previously, `Info` is a structure with the following fields

Field	Description
`Sensitivity`	*A* $d_1 \times ... \times d$F `struct`, field names are names of uncertain elements of `sys`. Using single indexing notation, `Sensitivity(i)` contains the sensitivities of `perfmarg.UpperBound(i)` for the uncertain system `sys(:,:,i)`.
`Frequency`	*N*-by-1 frequency vector associated with analysis.
`BadUncertainValues`	*N*-by-1 cell array, with one entry for each frequency point. The `k`th entry `Info.BadUncertainValues{k}` is a $d_1 \times ... \times d$F `struct` of values of uncertain elements resulting from a $d_1 \times ... \times d$F family of robust performance computations at frequency `Info.Frequency(k)`.
`MussvBnds`	$1 \times 2 \times d_1$x...xdF `frd`, with upper and lower bounds from `mussv`. Using single indexing for the dimensions associated with the array dimensions, it follows that the `(1,1,i)` entry is the μ-upper bound (reciprocal of `perfmarg.UpperBound(i)`) while the `(1,2,i)` entry is the μ-lower bound (reciprocal of `perfmarg.UpperBound(i)`).
`MussvInfo`	Structure of compressed data from `mussv`.

The smallest performance margin over all array dimensions is computed `min(perfmarg.UpperBound(:))`. Computing

```
i = find(UpperBound==min(UpperBound(:)))
```

and then selecting `perfmargunc(i)` yields values for an uncertainty corresponding to the smallest performance margin across all array dimensions.

Algorithm

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, but might require user attention.

If `sys` is a `uss` object, then the first requirement of stability of nominal value is explicitly checked within `robustperf`. 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*.

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`.

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 the Robust Control Toolbox demo 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.

See Also

Comprehensive analysis of feedback loop

`mussv`

```
norm
robopt
robuststab
actual2normalized
wcgain
wcsens
wcmargin
```

Purpose

Calculate robust stability margins of uncertain multivariable system

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` will equal $-_{\infty}$.
`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.
`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 atom 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 robustness analysis results.

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],'SampleStateDim',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 `robopt` object is `'off'`, the values are set to `NaN`.
`Frequency`	*N*-by-1 frequency vector associated with analysis.
`BadUncertainValues`	*N*-by-1 cell array, with one entry for each frequency point. The kth entry`Info.BadUncertainValues{k}` is a `struct` of values of uncertain elements, closest to their nominal values, which cause the system poles to migrate across the stability boundary at frequency `Info.Frequency(k)`. The command `pole(usubs(sys,Info.BadUncertainValues{k}))` shows the migration. The command `usubs(sys,cat(1,Info.BadUncertainValues{:}))` generates an *N*-by-1 `ss` array. Each instance is unstable, with poles on the stability boundary at frequencies given by the vector `Info.Frequency`. This migration to instability has been achieved with the smallest normalized deviations in the uncertain elements from their nominal values.
`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`.

Options (e.g., controlling what is displayed during the computation, turning on/off the sensitivity computation, setting the step-size in the sensitivity computation, and controlling the option argument used in the underlying call to `mussv`) can be specified using the robustness

analysis options `robopt` object. For instance, you can turn the display on, and the sensitivity calculation off by executing

```
opt = robopt('Sensitivity','off','Display','on');
[StabMarg,Destabunc,Report,Info] = robuststab(sys,opt)
```

Handling Array Dimensions

If `sys` has array dimensions (for example, suppose that the size of `sys` is $r \times c \times d_1 \times d_2 \times ... \times d_F$, refer to the $d_1 \times d_2 \times ... \times d_F$ as the *array dimensions*) then the margin calculation is performed pointwise (individually, at each and every array value) and the computed answers all have array dimensions as well. Details are described below. Again, assume that there are N frequency points and M uncertain elements.

The results of

```
[stabmarg,destabunc,Report,Info] = robuststab(sys,opt)
```

are `stabmarg` is a structure with the following fields:

Field	Description
`LowerBound`	A $d_1 \times ... \times d_F$, lower bound on stability margin across the array dimensions.
`UpperBound`	$d_1 \times ... \times d_F$, upper bound on stability margin across the array dimensions. Using single-indexing, for each `i`, the upper bound on the stability margin of `sys(:,:,i)` is `stabmarg.UpperBound(i)`.
`DestabilizingFrequency`	$d_1 \times ... \times d_F$, frequency at which instability occurs, associated with `stabmarg.UpperBound`. Using single-indexing, for each `i`, the frequency at which instability occurs in robust stability analysis of `sys(:,:,i)` is `stabmarg.DestabilizingFrequency(i)`.

`destabunc` is a $d_1 \times ... \times d_F$ structure array of values of uncertain elements, that cause instability. Using single-indexing, for each `i`,

the destabilizing values of uncertain elements for uncertain system `sys(:,:,i)` is `destabunc(i)`.

`Report` is a character array, dimensions 3, 4, ... , `F+2` are $d_1 \times ... \times d_F$, containing text description of the robustness analysis results at each grid in the array dimensions.

In addition to the first 3 output arguments, described previously, `Info` is a structure with the following fields

Field	Description
`Sensitivity`	A $d_1 \times ... \times d_F$ `struct`, field names are names of uncertain elements of `sys`. Using single indexing notation, `Sensitivity(i)` contains the sensitivities of `stabmarg.UpperBound(i)` for the uncertain system `sys(:,:,i)`.
`Frequency`	$N \times 1$ frequency vector associated with analysis.
`BadUncertainValues`	N-by-1 cell array, with one entry for each frequency point. The k'th entry `Info.BadUncertainValues{k}` is a $d_1 \times ... \times d$F `struct` of values of uncertain elements, closest to their nominal values, that cause the system poles to migrate across the stability boundary at frequency `Info.Frequency(k)`. The command `usubs(sys,Info.BadUncertainValues{k})` produces an `ss` array of size d_1x...xdF with the substitutions made. Alternatively, `usubs(sys,cat(F+1,Info.BadUncertainValues{:}))` produces an `ss` array of size $d_1 \times ... \times d_F \times N$ with the substitutions made.
`MussvBnds`	A $1 \times 2 \times d_1 \times ... \times d_f$ `frd`, with upper and lower bounds from `mussv`. Using single-indexing for the dimensions associated with the array dimensions, it follows that the `(1,1,i)` entry is the μ-upper bound (corresponding to `stabmarg.LowerBound(i)`) while the `(1,2,i)` entry is the μ-lower bound (which corresponds to `stabmarg.UpperBound(i)`).
`MussvInfo`	Structure of compressed data from `mussv`.

You can compute the smallest stability margin over all array dimensions via

```
min(stabmarg.UpperBound(:)).
```

Computing `i = find(UpperBound==min(UpperBound(:)))` and then `destabunc(i)` yields values for an uncertainty corresponding to the smallest stability margin across all array dimensions.

Algorithm

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, but might require user attention.

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 the Limitations section below 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.

Limitations

Under most conditions, the robust stability margin that occurs 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 $_{\infty}$.

See the Robust Control Toolbox demo Getting Reliable Estimates of Robustness Margins for more information about circumventing the problem in an engineering-relevant fashion.

See Also

`loopmargin`

`mussv`

`robopt`

`robustperf`

`wcgain`

`wcsens`

`wcmargin`

Purpose

Schur decomposition of `frd` object

Syntax

```
[u,t] = schur(x)
t = schur(x)
[u,t] = schur(x,0)
t = schur(x,0)
[u,t] = schur(x,'econ')
t = schur(x,'econ')
```

Description

`frd/schur` applies the `schur` command to `frd` objects. `[u,t] = schur(x)` operates on the `x.ReponseData` of the `frd` object at each frequency point to construct `u` and `t`. `u` and `t` are `frd` objects. `x` must be square. See the built-in `schur` command for details.

See Also

`qz`

`schur`

Purpose

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, 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* $\| \texttt{G-GRED} \|_\infty$ for well-conditioned model reduced problems [1]:

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

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).
`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'`overides `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.
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.

Algorithm

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,

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

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

3 Find the left/right orthonormal eigen-bases of PQ associated with the k^{th} big Hankel singular values.

$$V_A = [V_{R,SMALL}, \overbrace{V_{L,BIG}}]$$

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

$$V_D = [\overbrace{V_{R,BIG}}, V_{L,SMALL}]$$

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

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

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

6 Finally,

$$\left[\begin{array}{c|c} \hat{A} & \hat{B} \\ \hline \hat{C} & \hat{D} \end{array}\right] = \left[\begin{array}{c|c} S^T_{L,BIG} A S_{R,BIG} & S^T_{L,BIG} B \\ \hline C S_{R,BIG} & D \end{array}\right]$$

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

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:

```
rand('state',1234); randn('state',5678);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]);
rand('state',12345); randn('state',6789);
```

```
wt1 = rss(6,5,5); wt1.d = eye(5)*2;
wt2 = rss(6,4,4); wt2.d = 2*eye(4);
[g5, redinfo5] = schurmr(G, [10:2:18], 'weight',{wt1,wt2});
for i = 1:5
     figure(i); eval(['sigma(G,g' num2str(i) ');']);
end
```

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

`reduce`

`balancmr`

`bstmr`

`ncfmr`

`hankelmr`

`hankelsv`

Purpose

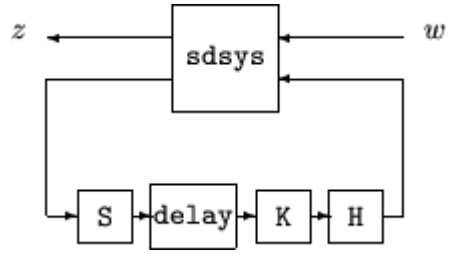
Compute L_2 norm of continuous-time system in feedback with discrete-time system

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
```

Algorithm

`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`

Purpose

Compute H_∞ controller for sampled-data system

Syntax

```
[K,GAM]=sdhinfsyn(P,NMEAS,NCON)
[K,GAM]=sdhinfsyn(P,NMEAS,NCON, KEY1,VALUE1,KEY2,VALUE2,...)
```

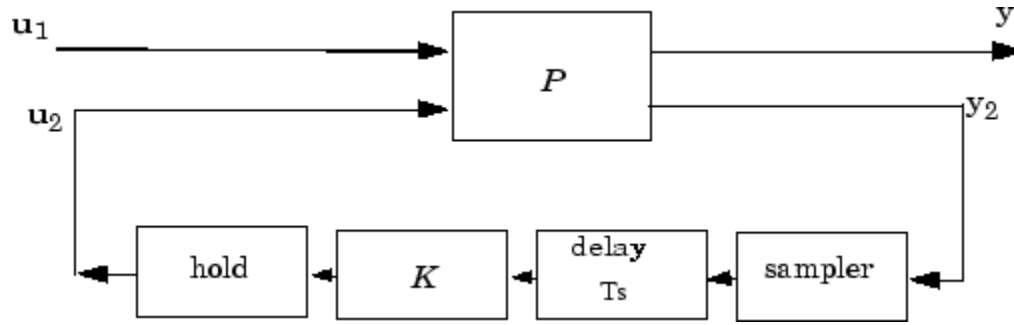
Description

`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:

$$P = \left[\begin{array}{c|cc} A & B_1 & B_2 \\ \hline C_1 & 0 & 0 \\ C_2 & 0 & 0 \end{array}\right]$$

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) sampling period of the controller to be designed
`'DELAY'`	integer	(Default=0) a nonnegative integer giving the number of sample periods delay for the control computation
`'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

Algorithm

`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

`norm`

`hinfsyn`

`sdhinfnorm`

Purpose

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 sampling time (the unspecified sampling 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 sampling time (the unspecified sampling 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

To illustrate the use of `sdlsim`, consider the application of a discrete controller to an integrator with 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 with `lsim` 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]';
```

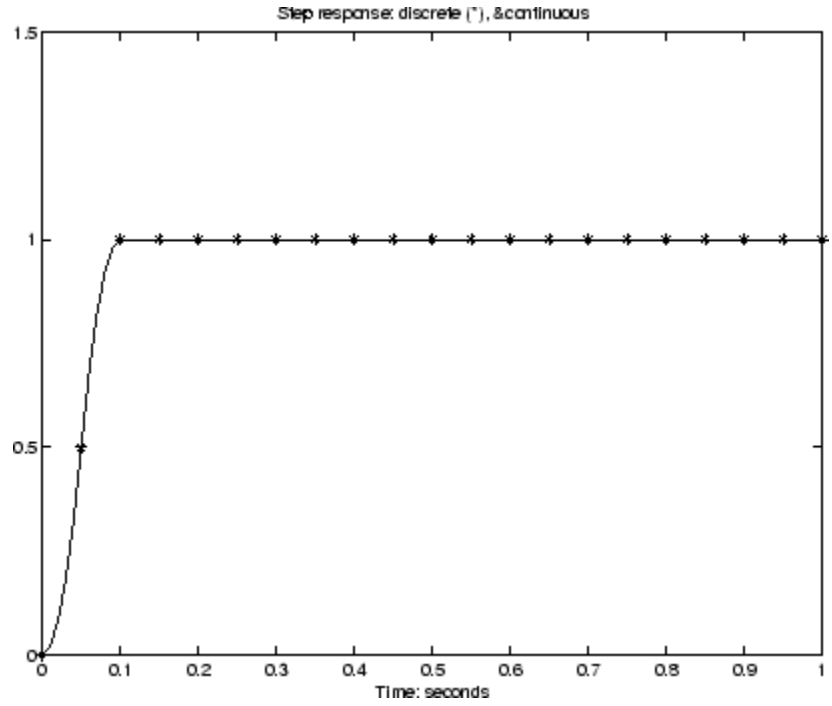
```
sysoutname = 'dclp';
cleanupsysic = 'yes';
sysic;
```

`lsim` is used to simulate the digital step response.

```
[yd,td] = step(dclp,20*T);
```

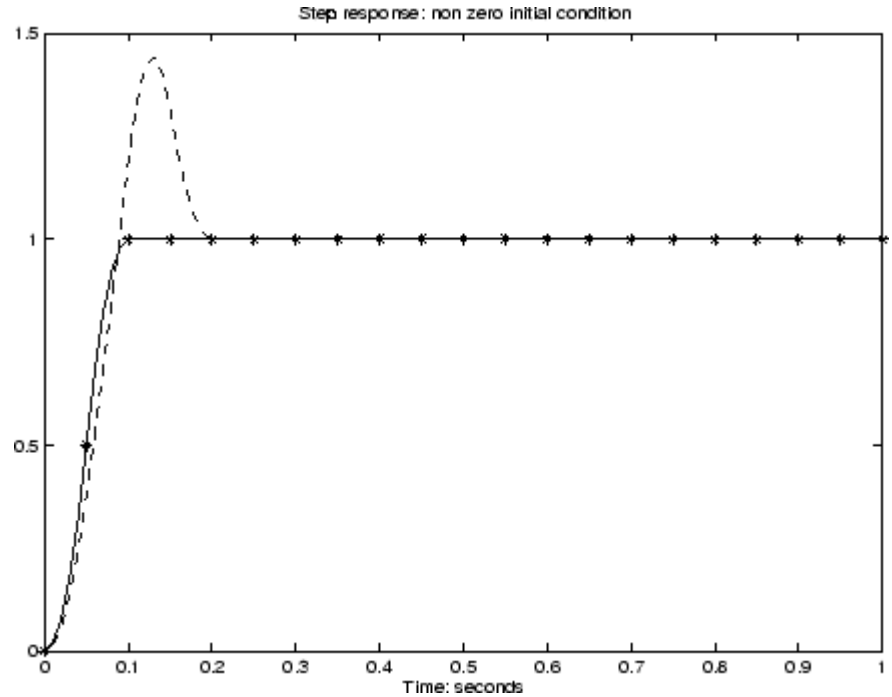
The continuous interconnection is set up and the sampled data response is calculated with `sdlsim`.

```
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 (*), &continuous')
```



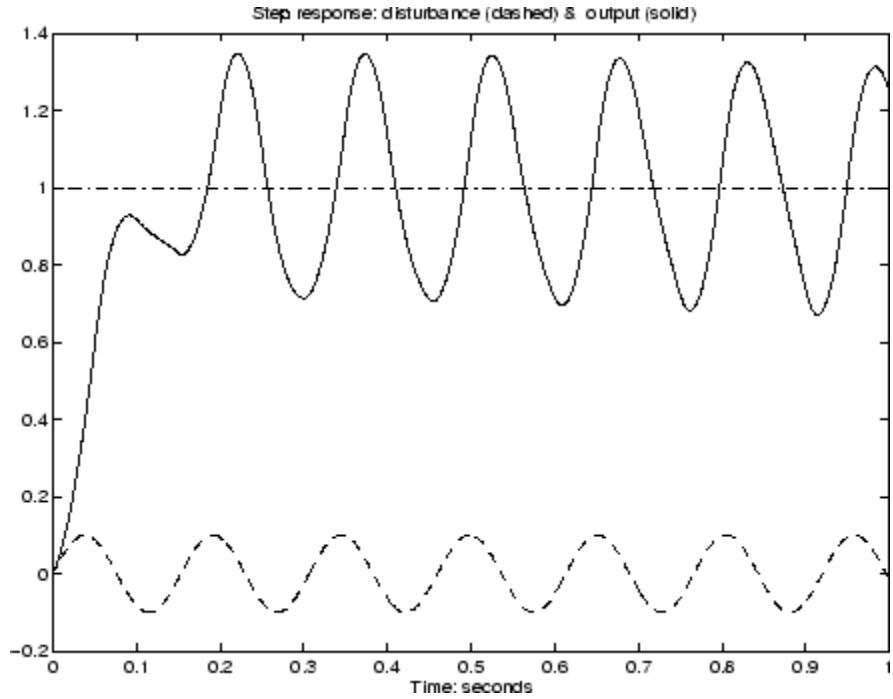
You can see the effect of a nonzero initial condition in the continuous-time system. Note how examining the system at only the sample points will underestimate the amplitude of the overshoot.

```
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: non zero 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) &  output (solid)')
```



Algorithm

`sdlsim` oversamples the continuous-time, `N` times the sample rate of the controller `k`.

See Also

`gapmetric`

`hinfsyn`

`norm`

`sdhinfnorm`

`sdhinfsyn`

`sysic`

Purpose

State-space sector bilinear transformation

Syntax

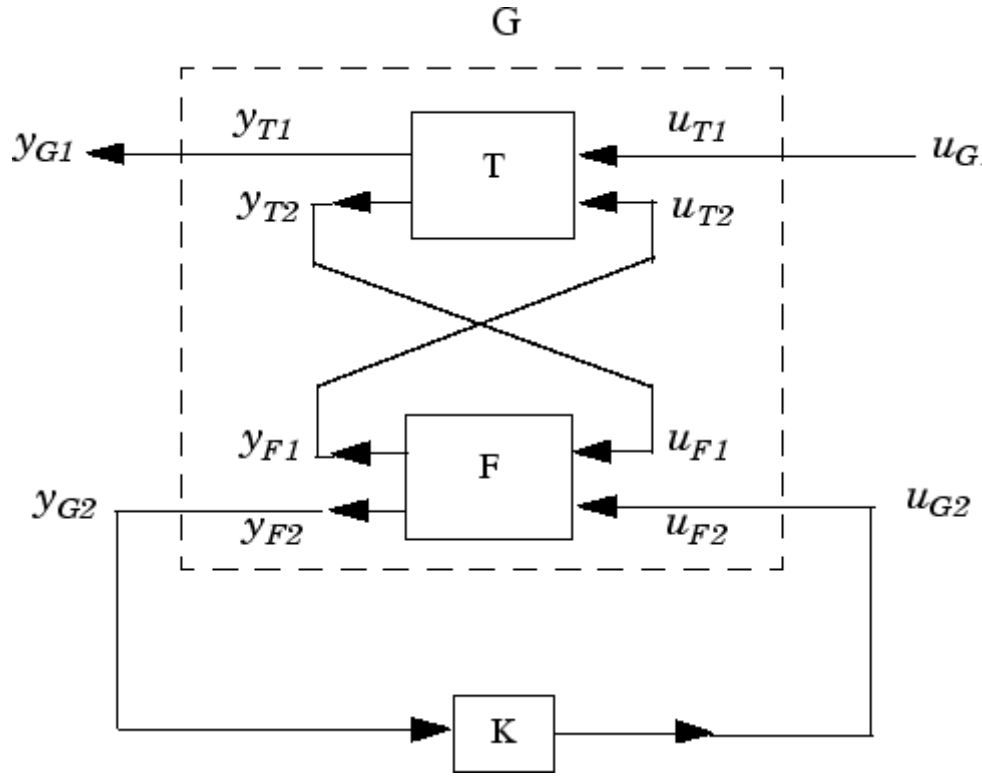
```
[G,T] = sectf(F,SECF,SECG)
```

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:	

Input Arguments		
	`[-1,1]` or `[-1;1]`	$\|y\|^2 \le \|u\|^2$
	`[0,Inf]` or `[0;Inf]`	$0 \le Re[y^*u]$
	`[A,B]` or `[A;B]`	$0 \ge \mathrm{Re}[(y-Au)^*(y-Bu)]$
	`[a,b]` or `[a;b]`	$0 \ge \mathrm{Re}[(y-diag(a)u)^*(y-diag(b)u)]$
	`S`	$0 \ge \mathrm{Re}[(S_{11}u+S_{12}y)^*(S_{21}u+S_{22}y)]$
	`S`	$0 \ge \mathrm{Re}[(S_{11}u+S_{12}y)^*(S_{21}u+S_{22}y)]$

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`

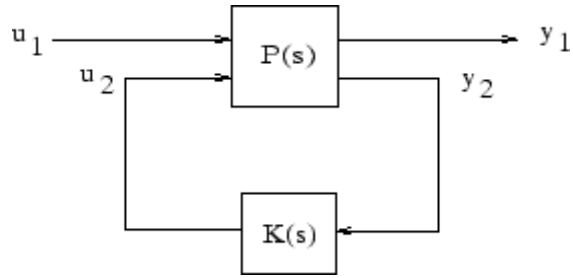
Output Variables	
`G`	The transformed plant $G(s)$=`lftf(T,F)`:
`T`	The linear fractional transformation $T(s)$=`T`

Examples

The statement G(jω) inside *sector*[–1, 1] is equivalent to the H_∞ inequality

$$\sup_{\omega} \bar{\sigma}(G(j\omega)) = \|G\|_\infty \le 1$$

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 2-329.



Sector Transform Block Diagram

Here is a simple example of the sector transform.

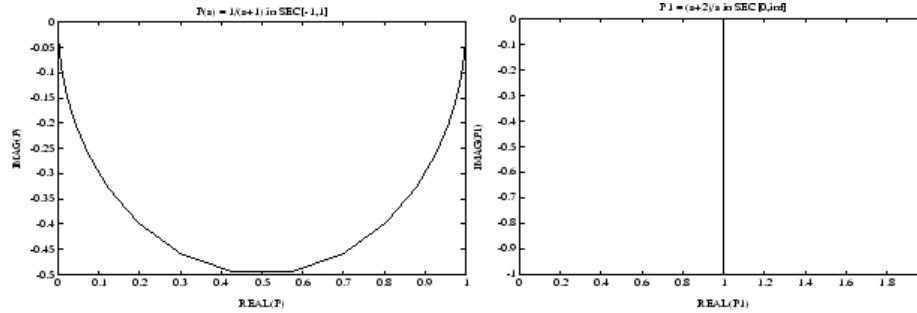
$$P(s) = \frac{1}{s+1} \in \mathbf{sector}[-1,1] \rightarrow P_1(s) = \frac{s+2}{s} \in \mathbf{sector}[0,\infty].$$

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 2-329. The condition $P_1(s)$ inside [0, ∞] implies that $P_1(s)$ is stable and $P_1(j\omega)$ is *positive real*, i.e.,

$$P_1^*(j\omega) + P_1(j\omega) \ge 0 \quad \forall\omega$$



Example of Sector Transform

Algorithm

`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).

Limitations

A well-posed conic sector must have $\det(B - A) \neq 0$ 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.

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

`lft`

`hinfsyn`

Purpose

Semilog scale plot of `frd` object

Syntax

```
semilogx(...) <block>same as </block>plot
semilogy(...) <block>same as </block>plot
```

Description

`semilogx(...)` is the same as `plot(...)`, except a logarithmic (base 10) scale is used for the *x*-axis.

`semilogy(...)` is the same as `plot(...)`, except a logarithmic (base 10) scale is used for the *y*-axis.

See Also

`loglog`

`plot`

`semilogy`

setlmis

Purpose

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`

`lmivar`

`lmiterm`

`newlmi`

Purpose

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

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

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

`evallmi`

`delmvar`

Purpose

Return left- and right-hand 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-hand 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-hand 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`

simplify

Purpose

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 occurences 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
```

Algorithm

`simplify` uses heuristics along with one-dimensional model reduction algorithms to partially reduce the dimensionality of the representation of an uncertain matrix or system.

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 occurences 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.

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

`umat`

`uss`

`ucomplex`

`ureal`

`uss`

Purpose

Form skew-symmetric matrix

Syntax

x = skewdec(m,n)

Description

`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.

See Also

`decinfo`

`lmivar`

slowfast

Purpose

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`.

$[G_1(s)] := (\hat{A}_{11}, \hat{B}_1, \hat{C}_1, \hat{D}_1)$ denotes the slow part of $G(s)$. The slow poles have low frequency and magnitude values.

$[G_2(s)] := (\hat{A}_{22}, \hat{B}_2, \hat{C}_2, \hat{D}_2)$ 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.

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

`schur`

`modreal`

Purpose

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`

Purpose

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` and 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

canon

c2d

d2c

mussv

mussvextract

ss2ss

Purpose

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

Purpose

Singular value decomposition of `frd` object

Syntax

```
S = svd(X)
[U,S,V] = svd(X)
```

Description

`S = svd(X)` operates on `X.ReponseData` at each frequency to construct `S`.

`[U,S,V] = svd(X)` produces a diagonal `frd S` that has the same dimensions as `X` and includes positive diagonal elements in decreasing order. `U` and `V` are unitary matrices and `frd` objects, such that `X = U*S*V'`. For more information, see the built-in `svd` command.

See Also

`schur`

`svd`

symdec

Purpose

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.

See Also

`decinfo`

Purpose

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.

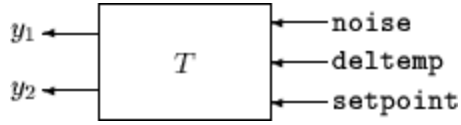
`sysic` also requires that for every subsystem name listed in `systemnames`, a corresponding variable, `input_to_ListedSubSystemName` must exist in the calling workspace. This variable is similar to `outputvar` – it defines the input signals to this particular subsystem as linear combinations of individual 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`.

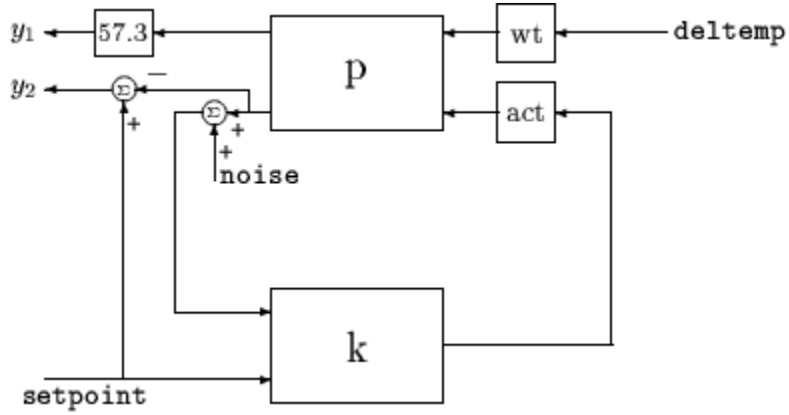
After running `sysic` the variables `systemnames, inputvar, outputvar` and all of the `input_to_ListedSubSystemName` will exist in the workspace. Setting the optional variable `cleanupsysic` to `'yes'` will cause these variables to be removed from the workspace after `sysic` has formed the interconnection.

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`

ucomplex

Purpose

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'`.

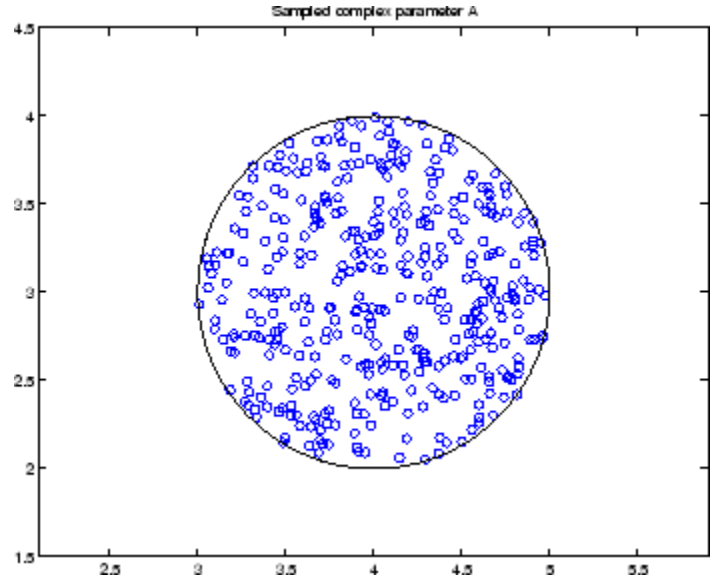
Examples

Create an uncertain complex parameter with internal name `A`. The uncertain parameter's possible values are a complex disc of radius `1`, centered at `4+3j`. The value of `A.percentage` is `20` (radius is 1/5 of the magnitude of the nominal value).

```
A = ucomplex('A',4+3*j)
Uncertain Complex Parameter: Name A, NominalValue 4+3i, Radius 1
```

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

umat

ucomplexm

ultidyn

ureal

ucomplexm

Purpose

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

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]));
```

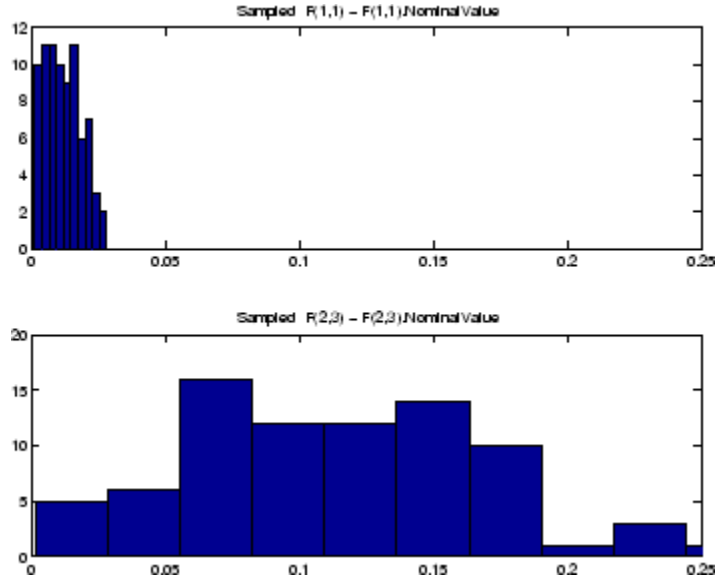
Sample the difference between the uncertain matrix and its nominal value at 80 points, yielding a 2-by-3-by-80 matrix `typicaldev`.

```
typicaldev = usample(F-F.NominalValue,40);
```

Plot histograms of the deviations in the (1,1) entry as well as the deviations in the (2,3) entry.

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);
hist(abs(typicaldev(1,1,:)));xlim([0 .25])
title('Sampled  F(1,1) - F(1,1).NominalValue')
subplot(2,1,2);
hist(abs(typicaldev(2,3,:)));xlim([0 .25])
title('Sampled  F(2,3) - F(2,3).NominalValue')
```



See Also

get

umat

ucomplex

ultidyn

ureal

Purpose

Fit an uncertain model to set of LTI responses

Syntax

```
usys = (Parray,Pnom,ord)
usys = (Parray,Pnom,ord1,ord2,utype)
[usys,info] = (Parray,...)
[usys_new,info_new] = (Pnom,info,ord1_new,ord2_new)
```

Description

`usys = (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 = (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 2-360, 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] = (Parray,...)` returns a structure `info` that contains optimization information. `info.W1opt` and `Info.W2opt` contain the

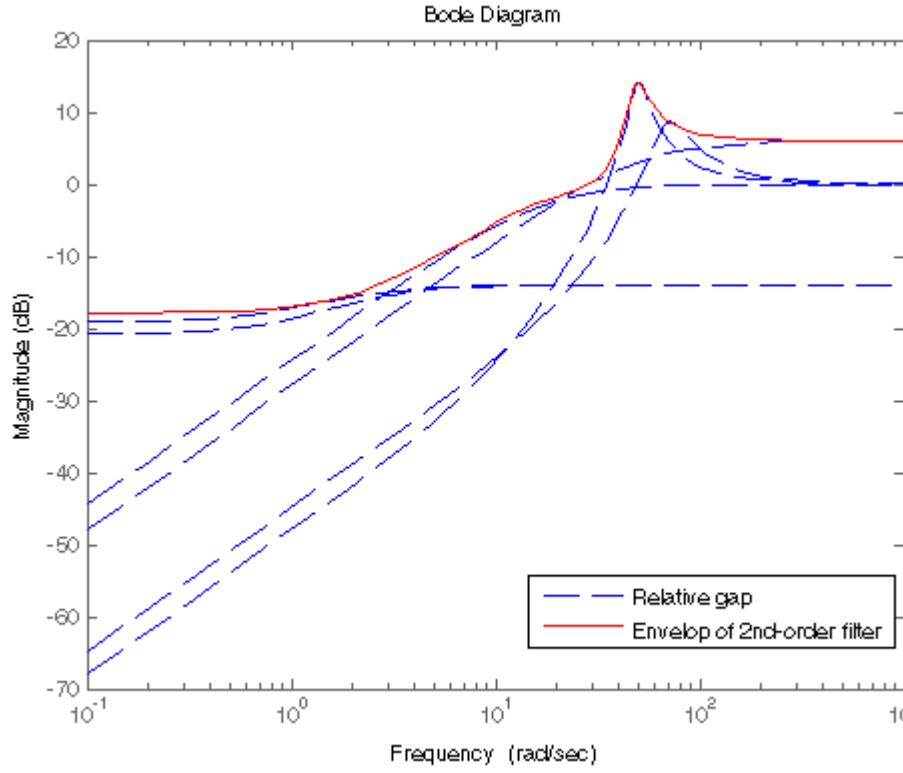
values of W1 and W2 computed on a frequency grid and `info.W1` and `info.W2` contain the fitted shaping filters W1 and W2.

`[usys_new,info_new] = (Pnom,info,ord1_new,ord2_new)` improves the fit using initial filter values in `info and` new orders `ord1_new` and `ord2_new` of W1 and W2. This syntax speeds up command execution by reusing previously computed information. Use this syntax when you are changing filter orders in an iterative workflow.

Definitions

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 $usys = Pnom \times (I + W1 \times \Delta \times W2)$.
- Output Multiplicative of the form $usys = (I + W1 \times \Delta \times W2) \times Pnom$.

- Additive of the form $usys = Pnom + W1 \times \Delta \times W2$.

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 $W1 = W \times I$ and $W1 = 1$.

Examples

1 Fit an uncertain model to multiple LTI responses:

Create the nominal plant.

```
Pnom = tf(2,[1 -2]);
```

2 Create an LTI array whose responses the uncertain model should fit.

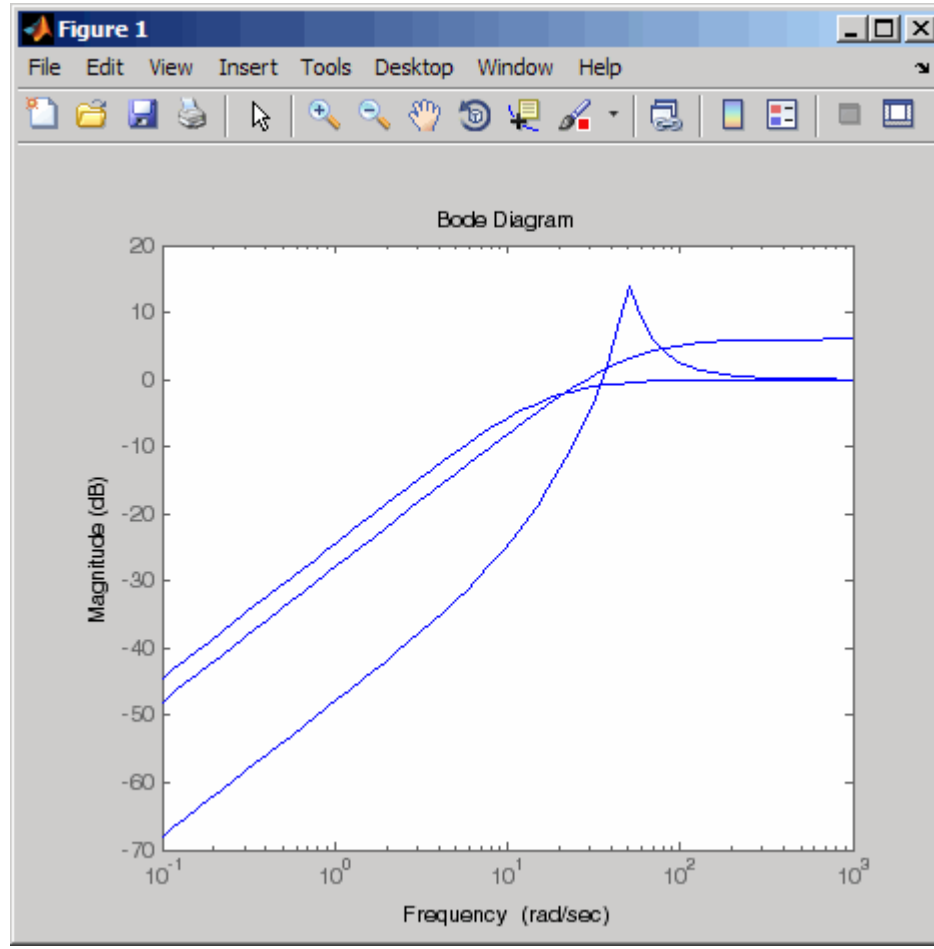
```
p1 = Pnom*tf(1,[.06 1]);
p2 = Pnom*tf([-.02 1],[.02 1]);
p3 = Pnom*tf(50^2,[1 2*.1*50 50^2]);
array = stack(1,p1,p2,p3);
Parray = frd(array,logspace(-1,3,60));
```

3 Plot relative errors between the nominal plant response and the three models in the LTI array.

```
bodemag((Pnom-Parray)/Pnom)
```

The set of relative errors is shown in the following figure. If you use a multiplicative uncertainty model structure, the magnitude of

the shaping filter should fit the maximum relative errors at each frequency.



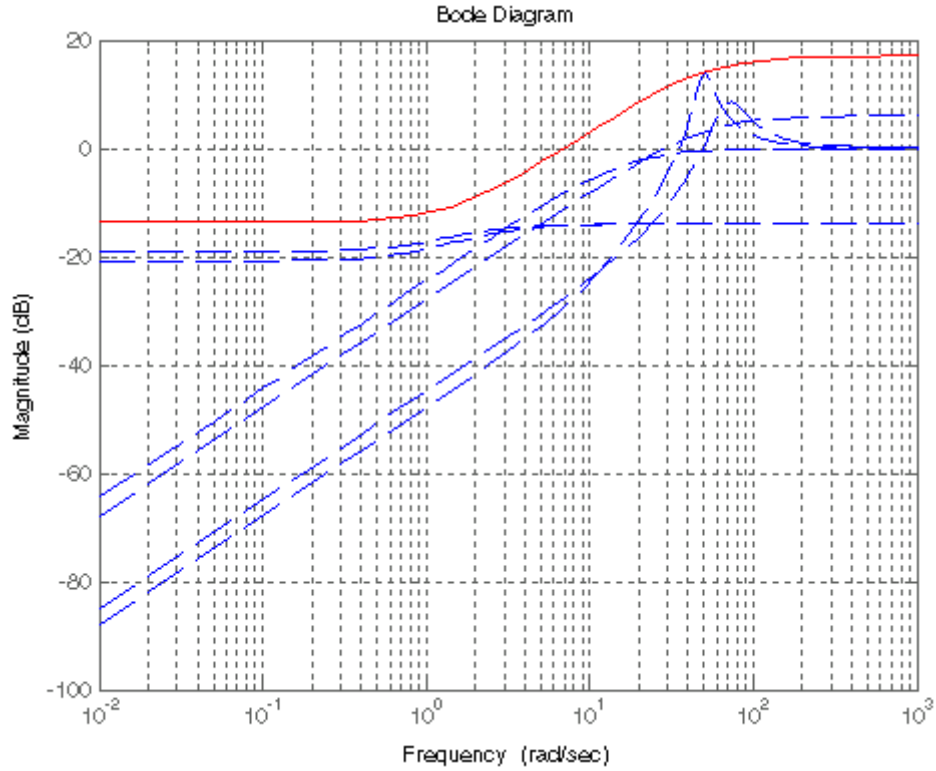
4 Use a 1st-order shaping filter to fit the maximum relative errors.

```
[P,Info] = ucover(Parray,Pnom,1);
```

5 Plot a Bode magnitude plot to see how well the shaping filter fits the relative errors.

```
bodemag((Pnom-Parray)/Pnom,'b--',Info.W1,'r'); grid
```

The plot, as shown in the following figure, shows that the filter W1 is too conservative and exceeds the maximum relative error at most frequencies.



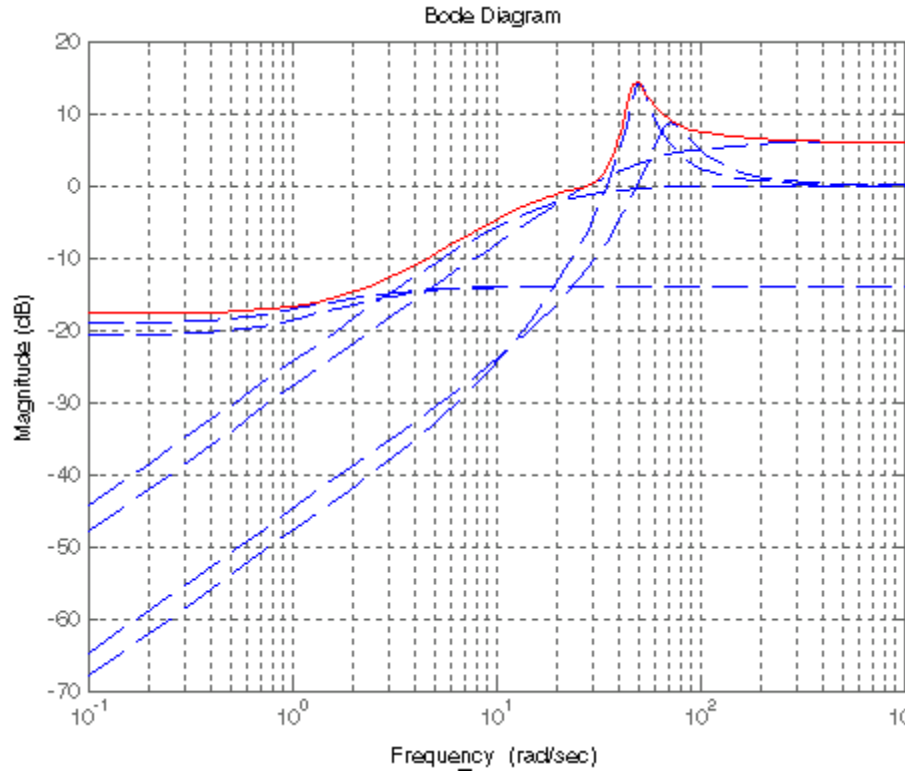
6 To obtain a tighter fit, rerun the function using a 4th-order filter.

```
[P,Info] = ucover(Parray,Pnom,4);
```

7 Evaluate the fit by plotting the Bode magnitude plot.

```
bodemag((Pnom-Parray)/Pnom,'b--',Info.W1,'r'); grid
```

The plot, as shown in the following figure, shows that magnitude of W1 closely matches the minimum uncertainty amount.



Algorithm

The `ucover` command designs the minimum-phase shaping filters W1 and W2 in two steps:

1 Computes the optimal values of W1 and W2 on a frequency grid.

2 Fits W1 and W2 values with the dynamic filters of the specified orders using the `fitmagfrd` command.

See Also

`ss`, `tf`, `zpk`, `frd`, `usample`

Tutorials

- Modeling a Family of Responses as an Uncertain System
- Simultaneous Stabilization Using Robust Control
- First-Cut Robust Design

Purpose

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., `robuststab`) 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

`ureal`

`ultidyn`

`ucomplex`

`ucomplexm`

ufind

Purpose

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.

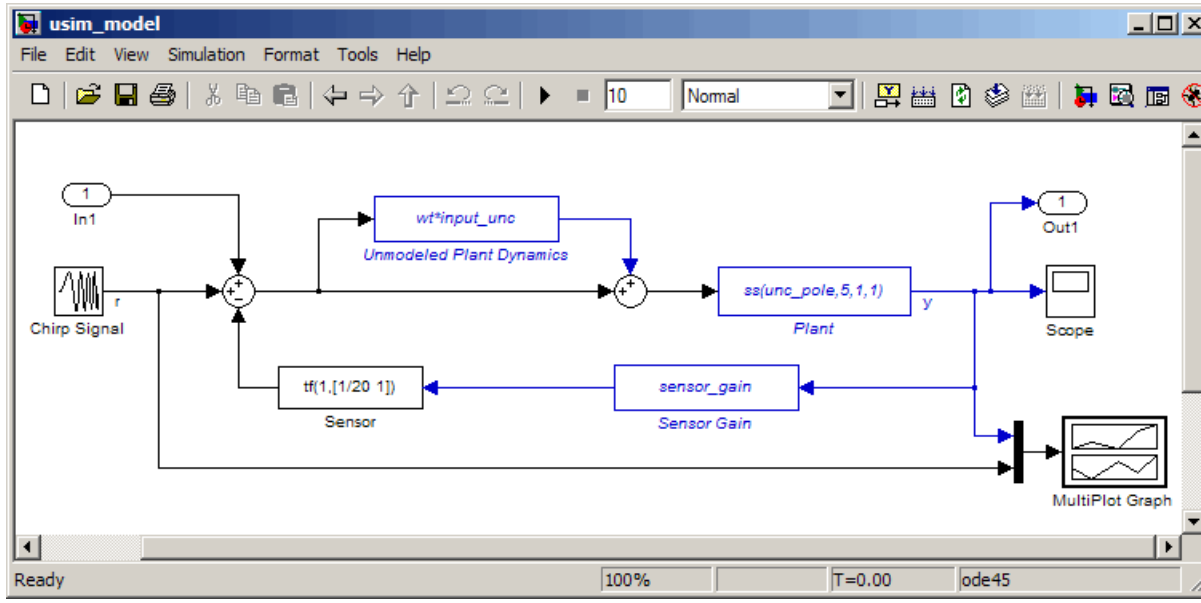
Examples

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'
```

See Also

`usample`

Uncertain State Space block

Tutorials

"Varying Uncertainty Values Using Individual Uncertain State Space Blocks"

"Varying Uncertainty Values Across Multiple Uncertain State Space Blocks"

Robustness Analysis in Simulink

How To "Simulating Uncertainty Effects"

Purpose

Create uncertain frequency response data (ufrd) object, or convert another model type to ufrd model

Syntax

```
usysfrd = ufrd(usys,frequency)
usysfrd = ufrd(usys,frequency,'Units',units)
usysfrd = ufrd(sysfrd)
usys = ufrd(response,frequency)
usys = ufrd(response,frequency,Ts)
usys = ufrd(response,frequency,RefSys)
usys = ufrd(response,frequency,'Units',units,Ts)
usys = ufrd(response,frequency,'Units',units,Ts,RefSys)
```

Description

Uncertain frequency response data (ufrd) models result from the conversion of an uncertain state-space (uss) system to its uncertain frequency response. ufrd models also result when frequency response data models (frd) are combined with uncertain matrices (umat).

`usysfrd = ufrd(usys,frequency,'Units',units)` converts a uss model usys to a ufrd model usysfrd by frequency response.'Units' specifies the units of the frequencies in frequency, which can be 'rad/s' or 'Hz'. If the last two arguments are omitted, the default for frequency units is 'rad/s'.

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

`usysfrd = ufrd(usys,frequency,'Units',units,'P1',V1,'P2',V2,...)` sets the properties P1, P2, ... to the values V1, V2, ...

`usys = ufrd(response,frequency)` creates a ufrd from the response and frequency arguments. response should be a umat array, whose first array dimension (i.e., size(response,3)) aligns with the frequency. Note that you are unlikely to use this option.

`usysfrd = ufrd(sysfrd)` converts an frd model sysfrd to a ufrd model usysfrd with no uncertain elements.

Examples

In the first example, you create a continuous-time uncertain system with both parametric uncertainty and unmodeled dynamics uncertainty. Compute the uncertain frequency response and plot the Bode plot, using 20 random samples, with a color choice of red for random samples, and blue for nominal.

```
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));
bode(usysfrd,'r',usysfrd.NominalValue,'b+')
```

Example 2

In this example, you convert a not-uncertain `frd` model to `ufrd` without uncertainties. You can verify the equality of the nominal value of the `ufrd` and simplified representation to the original system.

```
G = frd(tf([1 2 3],[1 2 3 4]),logspace(-2,2,40));
usys = ufrd(G)
UFRD: 1 Output, 1 Input, Continuous System, 40 Frequency points
isequal(usys.NominalValue,G)
ans =
     1
isequal(simplify(usys,'class'),G)
ans =
     1
```

See Also

`frd`

`ss`

ulinearize

Purpose

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 `linoptions` 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.
```

See Also

`ureal`

`udyn`

`ultidyn`

`uss`

`linearize`, `getlinio`, `linio`, `operpoint`, `findop`, `linoptions` in the Simulink Control Design documentation.

Tutorials

"Example - Specifying a Block to Linearize To an Uncertain Variable at the Command Line"

Linearization of Simulink Models with Uncertainty

How To

"Computing Uncertain State-Space Models from Simulink Models"

Purpose

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).

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.

Trailing Property/Value pairs are allowed in the construction.

`H=ultidyn('name',iosize,'Property1',Value1,'Property2',Value2,...)`

The property `SampleStateDim` 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.

Examples

Example 1

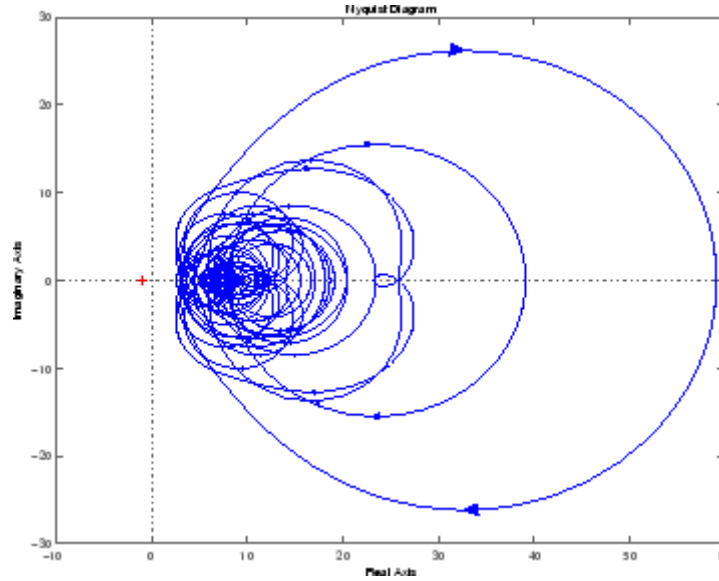
Create an `ultidyn` object with internal name `'H'`, dimensions `2-by-3`, norm bounded by `7`.

```
H = ultidyn('H',[2 3],'Bound',7)
Uncertain GainBounded LTI Dynamics: Name H, 2x3, Gain Bound = 7
```

Example 2

Create a scalar `ultidyn` object with an internal name `'B'`, whose frequency response has a real part greater than `2.5`. Change the `SampleStateDim` to `5`, and plot the Nyquist plot of `30` random samples.

```
B = ultidyn('B',[1 1],'Type','PositiveReal','Bound',2.5)
Uncertain PositiveReal LTI Dynamics: Name B, 1x1, M+M' >= 2*(2.5)
B.SampleStateDim = 5;
nyquist(usample(B,30))
```



See Also

`get`

`ureal`

`uss`

Purpose

Create uncertain matrix

Syntax

```
h = umat(m)
```

Description

Uncertain matrices are usually created by manipulation of uncertain atoms (`ureal, ucomplex, ultidyn,` etc.), `double` matrices, and other uncertain matrices. Most standard matrix manipulations are valid, including addition, multiplication, inverse, horizontal and vertical concatenation. Specific rows/columns of an uncertain matrix can be referenced and assigned also.

The command `umat` is rarely used. There are two situations where it may be useful. If `M` is a `double`, then `H = umat(M)` recasts `M` as an uncertain matrix (`umat` object) without any uncertainties. Similarly, if `M` is an uncertain atom, then `H = umat(M)` recasts `M` as an uncertain matrix (`umat` object) whose value is merely the uncertain atom. In both cases, `simplify(H,'class')` is the same as `M`.

If `M` is a `umat`, then `M.NominalValue` is the result obtained by replacing each uncertain atom in `M` with its own nominal value.

If `M` is a `umat`, then `M.Uncertainty` is an object describing all the uncertain atoms in `M`. All atoms 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 atom `B` in `M`.

Examples

Create 3 uncertain atoms 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]
UMAT: 3 Rows, 2 Columns
  a: real, nominal = 5, range = [2  6], 1 occurrence
  b: complex, nominal = 1+1i, radius = 0.5, 4 occurrences
  c: real, nominal = 3, variability = [-0.4  0.4], 1 occurrence
```

View the properties of `M` with `get`

```
get(M)
    NominalValue: [3x2 double]
     Uncertainty: [1x1 atomlist]
```

The nominal value of `M` is the result when all atoms are replaced by their nominal values. View the properties of `M` with `get`

```
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)
```

```
  UMAT: 2 Rows, 1 Columns
    b: complex, nominal = 1+1i, radius = 0.5, 3 occurrences
```

See Also

ureal

ultidyn

ucomplex

ucomplexm

usample

Purpose

Plot multiple frequency response objects and doubles 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)

Type	Description
'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

Two SISO second-order systems are created, and their frequency responses are calculated over different frequency ranges.

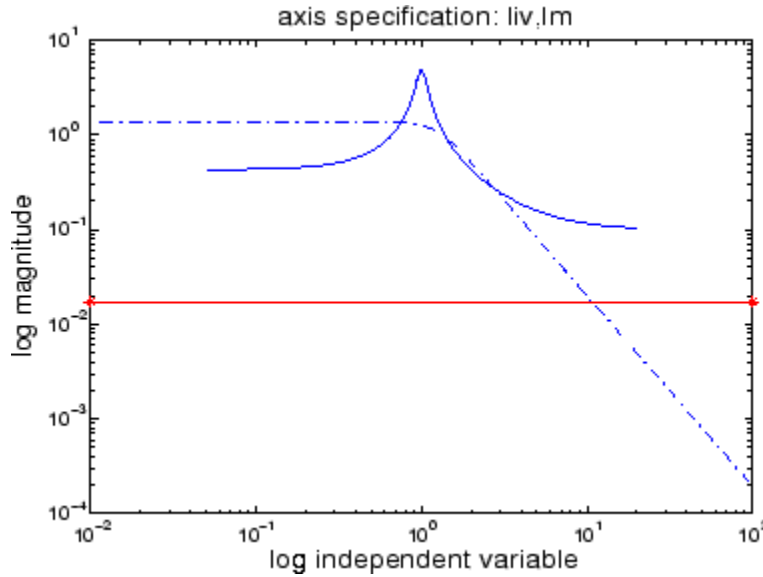
```
a1 = [-1,1;-1,-0.5];
b1 = [0;2]; c1 = [1,0]; d1 = 0;
sys1 = ss(a1,b1,c1,d1);
a2 = [-.1,1;-1,-0.05];
b2 = [1;1]; c2 = [-0.5,0]; d2 = 0.1;
sys2 = ss(a2,b2,c2,d2);
omega = logspace(-2,2,100);
sys1g = frd(sys1,omega);
omega2 = [ [0.05:0.1:1.5] [1.6:.5:20] [0.9:0.01:1.1] ];
omega2 = sort(omega2);
sys2g = frd(sys2,omega2);
```

An `frd` object with a single frequency is also created. Note the distinction between the `frd` object and the constant matrix in the subsequent plots.

```
sys3 = rss(1,1,1);
rspot = frd(sys3,2);
```

The following plot uses the `'liv,lm' plot_type` specification.

```
uplot('liv,lm',sys1g,'b-.',rspot,'r*',sys2g);
xlabel('log independent variable')
ylabel('log magnitude')
title('axis specification: liv,lm')
```



See Also

`bode`

`plot`

`nichols`

```
nyquist
semilogx
semilogy
sigma
```

Purpose

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

`ucomplex`

`umat`

`uss`

Purpose

Generate random samples of uncertain object

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)]`.

`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)
```

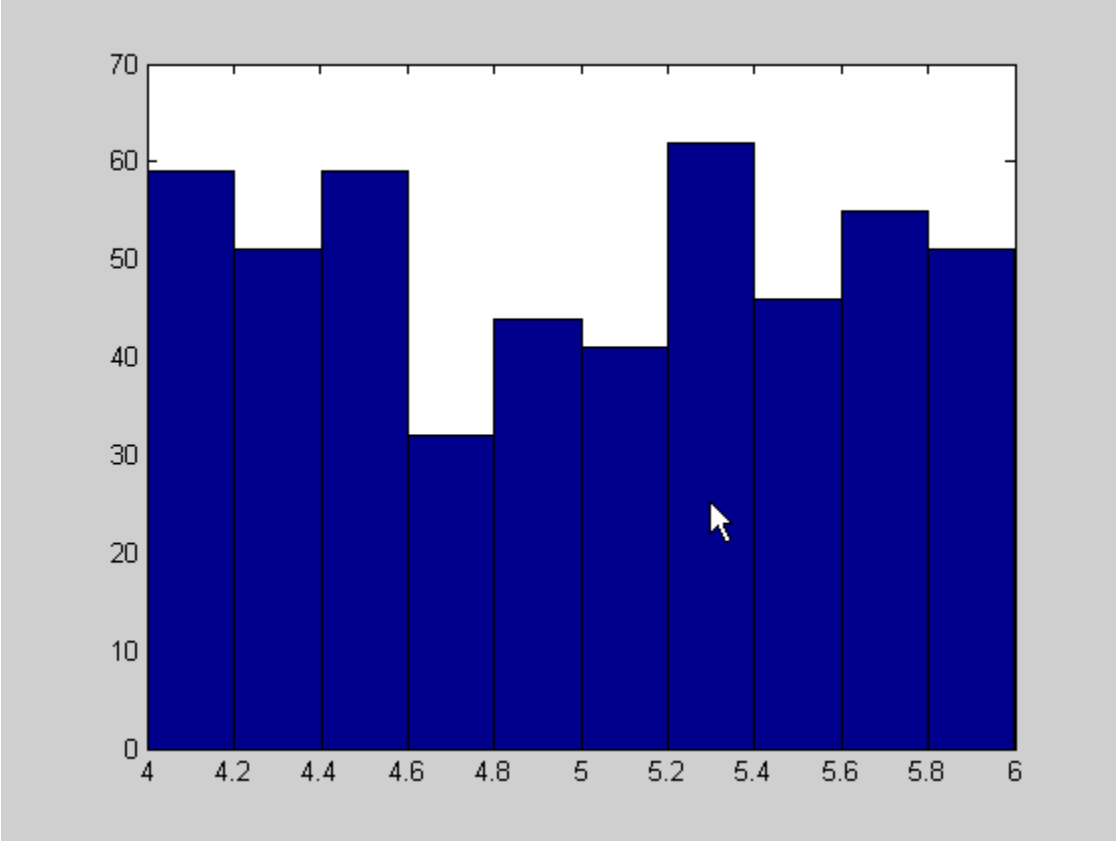
affects how ultidyn elements within A are sampled, restricting the poles of the samples. If A is a continuous-time uss or ufrd, then the poles of sampled GainBounded ultidyn elements in SampleValues will each have magnitude <= BW. If A is a discrete-time, then sampled GainBounded ultidyn elements are 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 Atoms") the GainBounded elements described above.

Examples

Example 1

Sample a real parameter and plot a histogram.

```
A = ureal('A',5);
Asample = usample(A,500);
size(A)
ans =
     1     1
size(Asample)
ans =
     1     1   500
class(Asample)
ans =
double
hist(Asample(:))
```



Example 2

This example illustrates how to sample the open and closed-loop response of an uncertain plant model for Monte Carlo analysis. You can 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 nominal plant parameter.

```
KI = 1/(2*tau.Nominal*gamma.Nominal);
C = tf(KI,[1 0]);
```

Now create an uncertain closed-loop system.

```
CLP = feedback(P*C,1);
```

You can sample the plant at 20 values (distributed uniformly about the `tau` and `gamma` parameter cube).

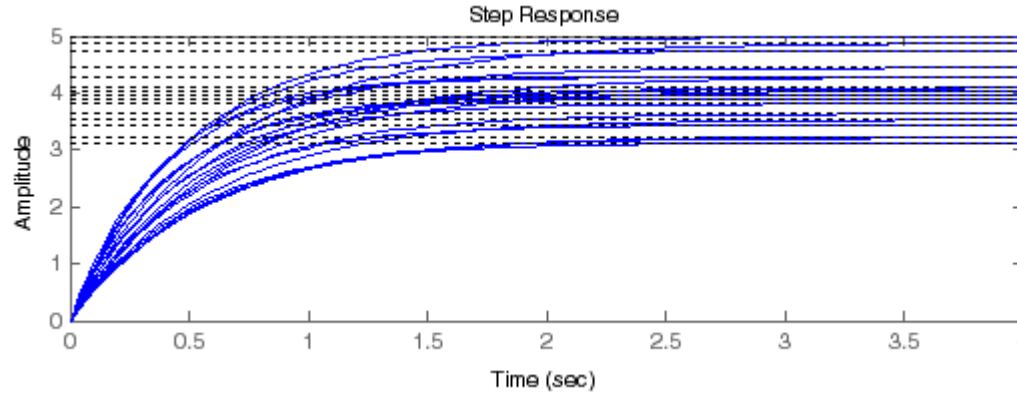
```
[Psample1D,Values1D] = usample(P,20);
size(Psample1D)
20x1 array of state-space models
Each model has 1 output, 1 input, and 1 state.
```

You can sample the plant `P` at 10 values in the `tau` parameter and 15 values in the `gamma` parameter.

```
[Psample2D,Values2D] = usample(P,'tau',10,'gamma',15);
size(Psample2D)
10x15 array of state-space models
Each model has 1 output, 1 input, and 1 state.
```

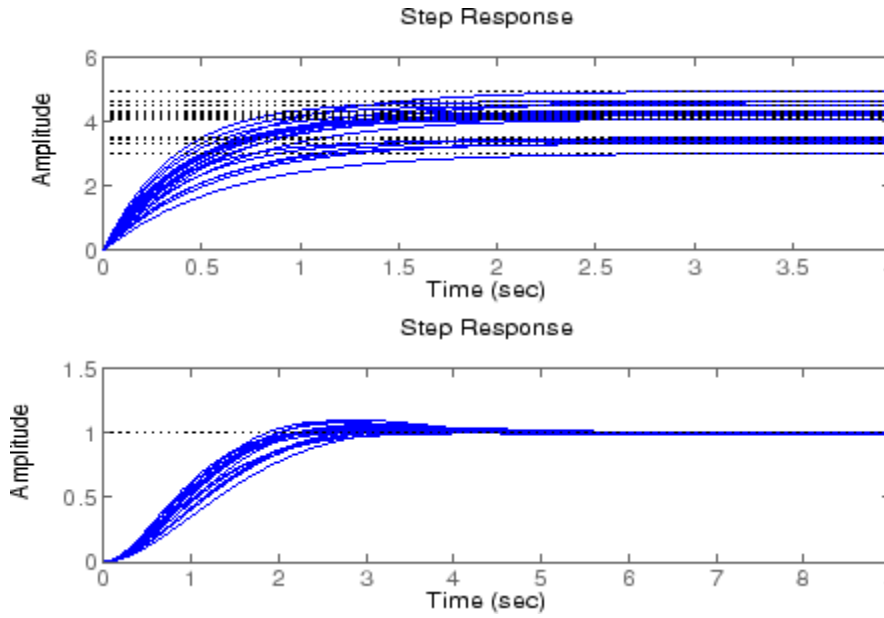
You can plot the 1-D sampled plant step responses

```
subplot(2,1,1); step(Psample1D)
```



You can also evaluate the uncertain closed-loop at the same values, and plot the step response using `usubs`.

```
subplot(2,1,2); step(usubs(CLP,Values1D))
```



Example 3

To see the effect of `Wmax`, create two `ultidyn` objects

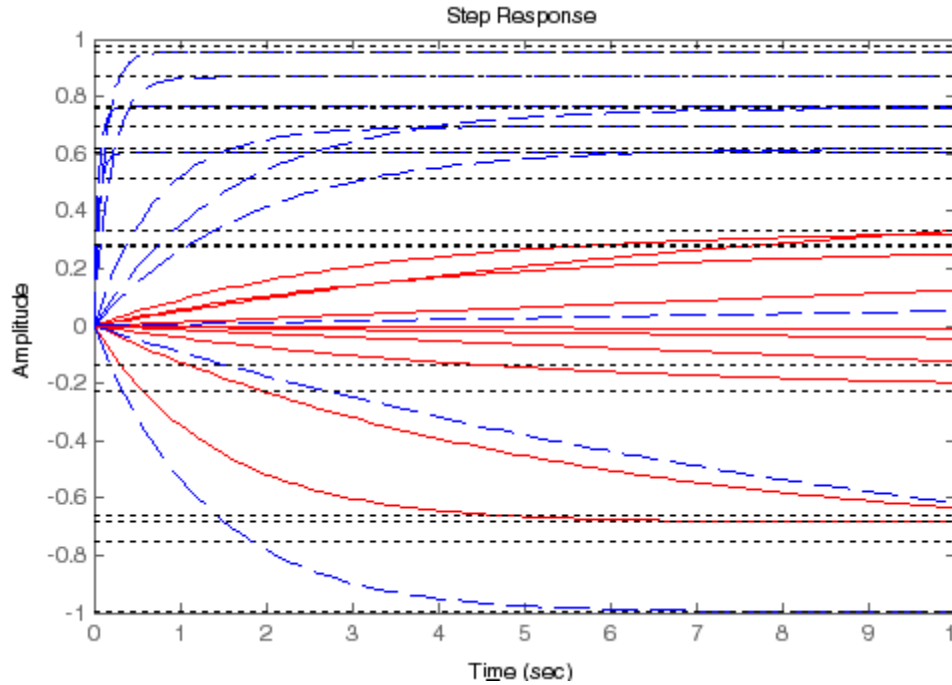
```
A = ultidyn('A',[1 1]);
B = ultidyn('B',[1 1]);
```

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. Plot the slow sample (from `A`) in red, and the faster samples (from `B`.) in blue.

```
step(As,'r',Bs,'b--',10)
```



See Also

`usample`, `usubs`, `ufind`, `ureal`, `ucomplex`, `ultidyn`, `umat`, `ufrd`, `uss`

Purpose

Generate random samples of uncertain variables

Syntax

```
samples = usample(uvars,N)
samples = usample(uvars)
samples = usample(uvars,N,Wmax)
```

Description

`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

Example 1

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)
```

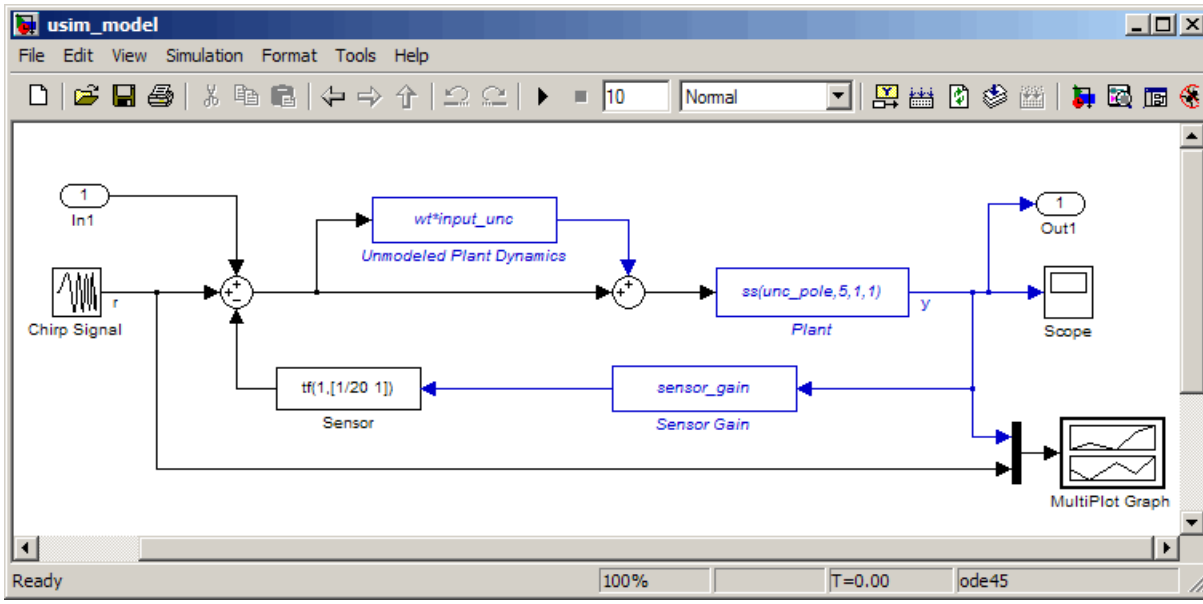
Example 2

Generate random samples of 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.

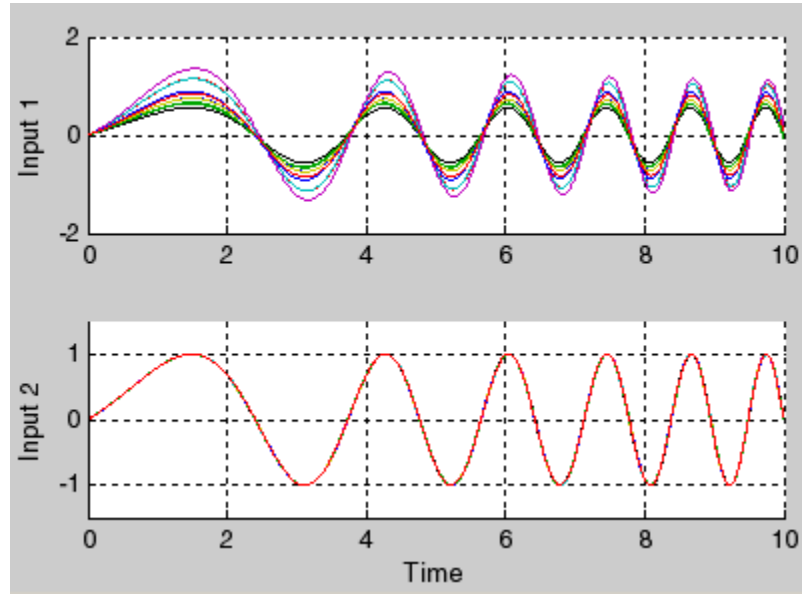
```
uvars = ufind('usim_model');
```

3 Use `usample` to generate random samples of `unc_pole`, `input_unc`, and `sensor_gain` and simulate the closed-loop response.

```
for i=1:10;
   uval = usample(uvars);
    sim('usim_model',10);
```

```
end
```

The MultiPlot Graph block displays the simulated responses, as shown in the following figure.



See Also

`ufind`, `usubs`, `ureal`, `ucomplex`, `ultidyn`, `umat`, `ufrd`, `uss`, `uss/usample`

Tutorials

"Varying Uncertainty Values Using Individual Uncertain State Space Blocks"

"Varying Uncertainty Values Across Multiple Uncertain State Space Blocks"

Robustness Analysis in Simulink

How To

"Simulating Uncertainty Effects"

usimfill

Purpose

Helper function for USS System blocks to set "User-defined Uncertainty" field or state of "Uncertainty value" menu

---

Note `usimfill` will be removed in a future release. Use `ufind` instead.

Syntax

```
usimfill(ModelName,str)
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,str)` pushes the string in `str` 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

See the Robust Control Toolbox demo Robustness Analysis in Simulink for a more detailed example of how to use `usimfill`.

Open the model file associated with the demo,

```
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 string `'newData'`.

```
usimfill('usim_model','newData');
```

View all of the dialog boxes, and see that the string `'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`

Purpose

Find USS System blocks within specified Simulink® model and check for consistency

Note `usiminfo` will be removed in a future release. 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 conpatiblilty conditions are satisfied. It is possible to have uncertain objects of the same name through out 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	
`sname`	Simulink diagram name
`silent`	Display inconsistencies between uncertain atoms, when not empty. Default is empty.

Output Arguments	
`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).

Output Arguments	
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

usimsamp

Purpose

Generate random instance of all uncertain atoms present in all USS System blocks of Simulink model

---

Note `usimsamp` will be removed in a future release. 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

See the Robust Control Toolbox demo Robustness Analysis in Simulink for a more detailed example of how to use `usimsamp`.

Open the model file associated with the demonstration,

```
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

Purpose

Specify uncertain state space models or convert LTI model to uncertain state space model

Syntax

```
usys = uss(a,b,c,d)
usys = uss(a,b,c,d,Ts)
usys = uss(d)
usys = uss(a,b,c,d,Property,Value,...)
usys = uss(a,b,c,d,Ts,Property,Value,...)
usys = uss(sys)
```

Description

`uss` creates uncertain state-space models (`uss` objects) or to convert LTI models to the `uss` class.

`usys = uss(a,b,c,d)` creates a continuous-time uncertain state-space object. The matrices `a, b, c` and `d` can be `umat` and/or `double` and/or uncertain atoms. These are the 4 matrices associated with the linear differential equation model to describe the system.

`usys = uss(a,b,c,d,Ts)` creates a discrete-time uncertain state-space object with sampling time `Ts`.

`usys = uss(d)` specifies a static gain matrix and is equivalent to `usys = uss([],[],[],d)`.

Any of these syntaxes can be followed by property name/property value pairs.

`usys = uss(a,b,c,d,'P1',V1,'P2',V2,...)` set the properties `P1, P2, ...` to the values `V1, V2, ...`

`usys = uss(sys)` converts an arbitrary `ss`, `tf` or `zpk` model `sys` to an uncertain state space object without uncertainties. Both `usys.NominalValue` and `simplify(usys,'class')` are the same as `ss(sys)`.

Examples

You can first create two uncertain atoms and use them to create two uncertain matrices. These four matrices can be packed together to form a 1-output, 1-input, 2-state continuous-time uncertain state space system.

```
p1 = ureal('p1',5,'Range',[2 6]);
p2 = ureal('p2',3,'Plusminus',0.4);
A = [-p1 0;p2 -p1];
B = [0;p2];
C = [1 1];
usys = uss(A,B,C,0);
```

In the second example, you can convert a not-uncertain `tf` model to an uncertain state-space model without uncertainties. You can verify the equality of the nominal value of the `usys` object and simplified representation to the original system.

```
G = tf([1 2 3],[1 2 3 4]);
usys = uss(G)
USS: 3 States, 1 Output, 1 Input, Continuous System
isequal(usys.NominalValue,ss(G))
ans =
     1
isequal(simplify(usys,'class'),ss(G))
ans =
     1
```

See Also

`frd`

`ss`

usubs

Purpose

Substitute given values for uncertain elements of uncertain objects

Syntax

```
B = usubs(M,atomname1,value1,atomname2,value2,...)
B = usubs(M,{atomname1;atomname2;...},{value1;value2;...})
B = usubs(M,StrucArray)
```

Description

`usubs` is used to substitute a specific value for an uncertain element of an uncertain 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,atomname1,value1,atomname2,value2,...)` sets the atoms in `M`, identified by `atomname1`, `atomname2`, etc., to the values in `value1`, `value2`, etc., respectively.

`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.

The `names` and `values` can also be grouped in cell arrays, as

```
B = usubs(M,atomname1,value1,...)
```

In this case, if the `value` cell is 1-by-1, then that value is substituted for all the listed atoms. For this situation, it is not required that the value be in a cell array.

`value` can also be the string `'NominalValue'` or `'Random'` (or only partially specified) in which case the nominal value, or a random instance of the atom is used.

Combinations of the above syntaxes are also allowed, so that

```
B = usubs(M,{atomname1;atomname2},{value1;value2})
```

is allowed.

The names and values can also be grouped in a structure, with its field names constituting the Names, and the field values constituting the Values. In the following function call, StrucArray is a structure with field names and values.

```
B = usubs(M,StrucArray)
```

Robustness analysis commands such as wcnorm, wcgain and robuststab return the offending uncertain element values in this manner. usample, which randomly samples uncertain objects, also returns the sample points in this manner.

Examples

Create an uncertain matrix and perform identical substitution in two different manners.

```
p = ureal('p',5);
m = [1 p;p^2 4];
size(m)
ans =
     2     2
m1 = usubs(m,'p',5)
m1 =
     1     5
    25     4
NamesValues.p = 5;
m2 = usubs(m,NamesValues)
m2 =
     1     5
    25     4
m1 - m2
ans =
     0     0
     0     0
```

You can make an array-valued substitution using the structure-based syntax,

```
NamesValues.p = rand(1,1,6);
m3 = usubs(m,NamesValues);  % 2-by-2-by-6
size(m3)
ans =
     2     2     6
```

You can use `usubs` to substitute for individual uncertainties. Create three uncertain real parameters, and form a simple `2-by-2` uncertain matrix with the parameters

```
a = ureal('a',5); b = ureal('b',3); c = ureal('c',1);
m = [a b;c a*b*c];
```

You can perform a single parameter substitution and check the results

```
m1 = usubs(m,'a',10);
simplify(m1(1,1))
ans =
    10
simplify(10*m1(1,2)*m1(2,1) - m1(2,2))
ans =
     0
```

You can replace one real parameter with a transfer function, and other parameters with `doubles`. You can do this using two different forms of the syntax and check that the results are identical.

```
m2 = usubs(m,'a',tf([5],[1 1]),'b',2.6,'c',1.3);
nv.a = tf([5],[1 1]);
nv.b = 2.6;
nv.c = 1.3;
m3 = usubs(m,nv);
norm(m2-m3,'inf')
ans =
     0
```

In `m`, replace `'a'` with `'b'`, obtaining `'b'` directly from `m` :

```
m4 = usubs(m,'a',m.Uncertainty.b);
```

See Also

gridureal

usample

simplify

Purpose

Calculate bounds on worst-case gain of uncertain system

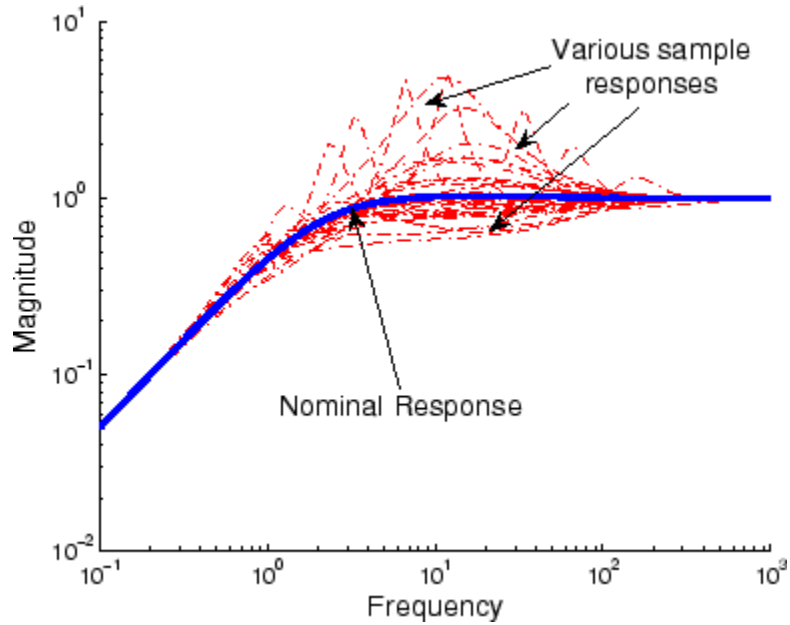
Syntax

```
[maxgain,wcu,info] = wcgain(sys)
[maxgain,wcu,info] = wcgain(sys,opts)
```

Description

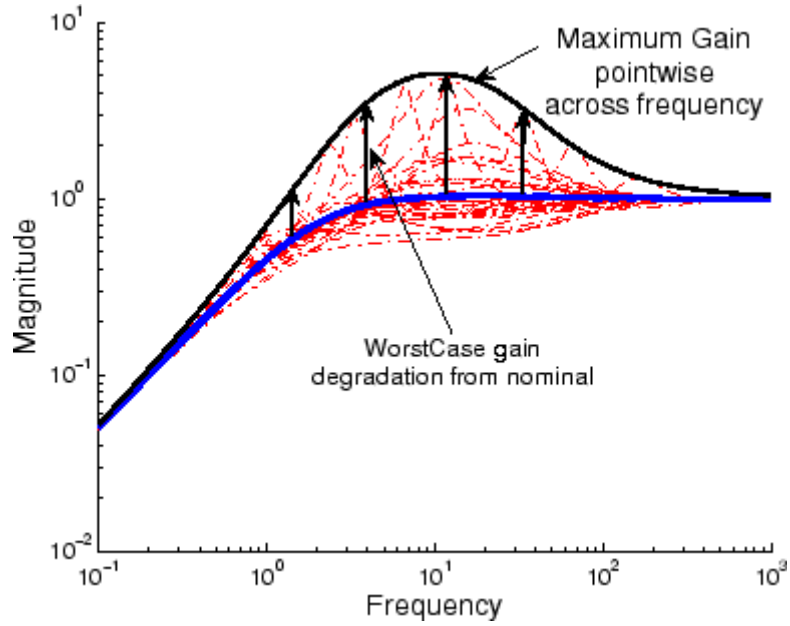
The gain of an uncertain system generally depends on the values of its uncertain elements. Here “gain” refers to the frequency response magnitude. Determining the maximum gain over all allowable values of the uncertain elements is referred to as a *worst-case gain* analysis. This maximum gain is called the *worst-case gain*.

The following figure shows the frequency response magnitude of many samples of an uncertain system model.



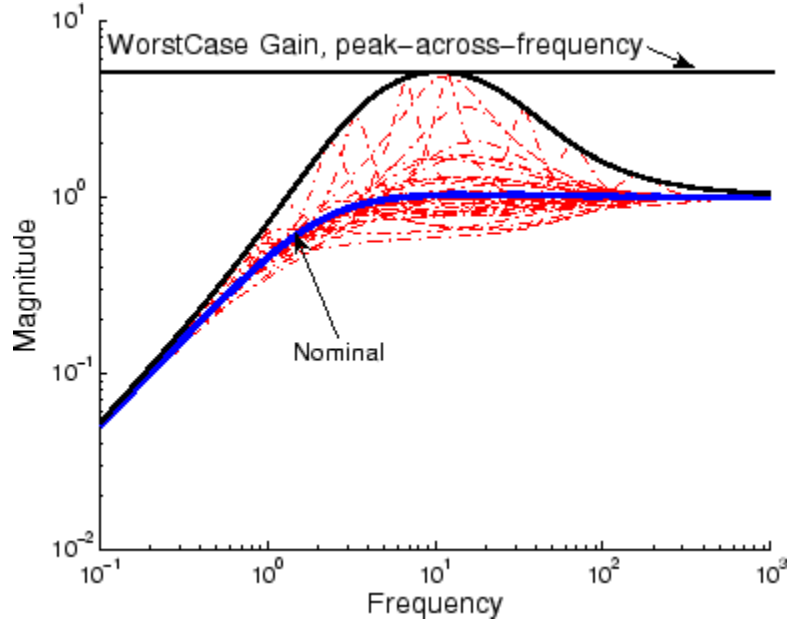
`wcgain` can perform two types of analysis on uncertain systems.

- A *pointwise-in-frequency* worst-case gain analysis yields the frequency-dependent curve of maximum gain, shown in the figure below.



  This plot shows the maximum frequency-response magnitude at each frequency due to the uncertain elements within the model.

- A *peak-over-frequency* worst-case gain analysis only aims to compute the largest value of the frequency-response magnitude across all frequencies. During such an analysis, large frequency ranges can be quickly eliminated from consideration, thus reducing the computation time.



The default analysis performed by `wcgain` is *peak-over-frequency*. You can control which analysis is performed by using the `wcgopt` options object. For multi-input, multi-output systems, the gain is the maximum singular value of the frequency response matrix.

As with other *uncertain-system* analysis tools, only bounds on the worst-case gain are computed. The exact value of the worst-case gain is guaranteed to lie between these upper and lower bounds.

The computation used in `wcgain` is a frequency-domain calculation. If the input system `sys` is an uncertain frequency response object (`ufrd`), then the analysis is performed on the frequency grid within the `ufrd`. If the input system `sys` is an uncertain state-space object (`uss`), then an appropriate frequency grid is generated (automatically), and the analysis performed on that frequency grid. In all descriptions below, *N* denotes the number of points in the frequency grid.

Basic Syntax

Suppose `sys` is an `ufrd` or `uss` with M uncertain elements. Calculate the worst-case gain of

```
[maxgain,maxgainunc] = wcgain(sys)
```

`maxgain` is a structure with the following fields

Field	Description
`LowerBound`	Lower bound on worst-case gain, positive scalar.
`UpperBound`	Upper bound on worst-case gain, positive scalar. If the nominal value of the uncertain system is unstable, then `maxgain.LowerBound` and `maxgain.UpperBound` equal $_\infty$.
`CriticalFrequency`	The critical value of frequency at which maximum gain occurs (this is associated with `maxgain.LowerBound`).

`maxgainunc` is a structure containing values of uncertain elements that maximize the system gain. 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 combined lead to the gain value in `maxgain.LowerBound`. The command

```
norm(usubs(sys,maxgainunc),'inf')
```

shows the gain.

Examples

Create a plant with 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 rad/s).

Design a proportional controller K_1 that puts the nominal closed-loop bandwidth at 0.8 rad/s. Roll off K_1 at a frequency 25 times the nominal closed-loop bandwidth. Repeat the design for a controller K_2 that puts

the nominal closed-loop bandwidth at 2.0 rad/s. In each case, form the closed-loop sensitivity function.

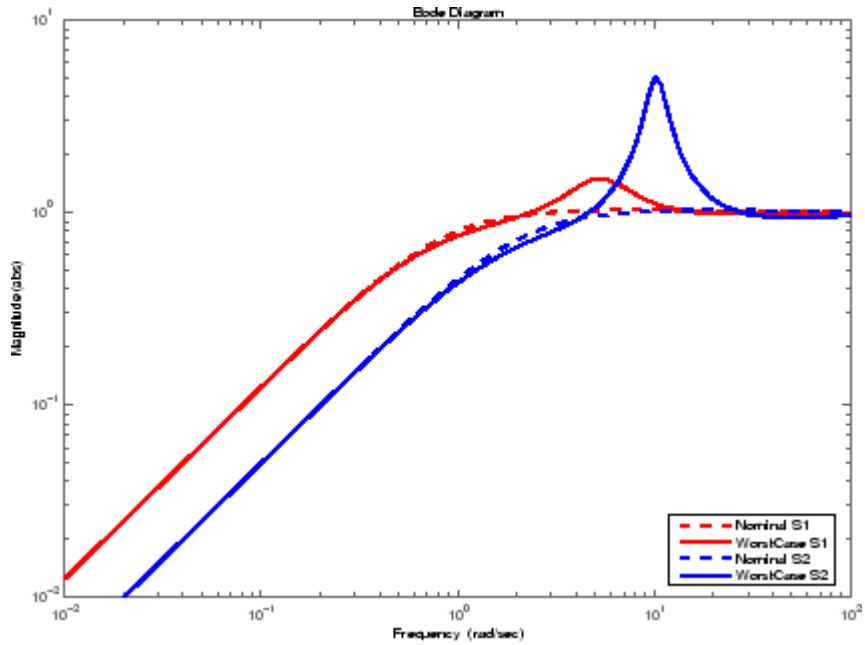
```
P = tf(1,[1 0]) + ultidyn('delta',[1 1],'bound',0.4);
BW1 = 0.8;
K1 = tf(BW1,[1/(25*BW1) 1]);
S1 = feedback(1,P*K1);
BW2 = 2.0;
K2 = tf(BW2,[1/(25*BW2) 1]);
S2 = feedback(1,P*K2);
```

Assess the worst-case gain of the closed-loop sensitivity function.

```
[maxgain1,wcunc1] = wcgain(S1);
[maxgain2,wcunc2] = wcgain(S2);
maxgain1
maxgain1 =
           LowerBound: 1.5070e+000
           UpperBound: 1.5080e+000
    CriticalFrequency: 5.3096e+000
maxgain2
maxgain2 =
           LowerBound: 5.1024e+000
           UpperBound: 5.1034e+000
    CriticalFrequency: 1.0215e+001
```

The `maxgain` variables indicate that controller K_1 achieves better worst-case performance than K_2. Plot Bode magnitude plots of the nominal closed-loop sensitivity functions, as well as the *worst* instances, using `usubs` to replace the uncertain element with the worst value returned by `wcgain`.

```
bodemag(S1.Nom,'r--',usubs(S1,wcunc1),'r',...
S2.Nom,'b--',usubs(S2,wcunc2),'b')
```



Note that although the nominal closed-loop sensitivity resulting from K_2 is superior to that with K_1, the worst-case behavior is much worse.

Basic Syntax with Third Output Argument

A third output argument yields more specialized information, including sensitivities of the worst-case gain to the uncertain element's ranges and frequency-by-frequency information.

```
[maxgain,maxgainunc,info] = wcgain(sys)
```

The third output argument `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 numbers, each entry indicating the local sensitivity of the worst-case gain in `maxgain.LowerBound` to all the individual uncertain element's uncertainty ranges. For instance, a value of 25 indicates that if the uncertainty range is enlarged by 8%, then the worst-case gain should increase by about 2%. If the `Sensitivity` property of the `wcgopt` object is `'off'`, the values are `NaN`.
`Frequency`	*N*-`by-1` frequency vector associated with analysis.
`ArrayIndex`	`1-by-1` scalar matrix whose value is 1. In more complicated situations (described later) the value of this field is dependent on the input data.

Options (e.g., turning on/off the sensitivity computation, setting the step-size in the sensitivity computation, adjusting the stopping criteria, and controlling behavior across frequency and array dimensions) can be specified using the worst-case gain analysis options `wcgopt` object. For instance, you can turn the sensitivity calculation off by executing

```
opt = wcgopt('Sensitivity','off');
[maxgain,maxgainunc,info] = wcgain(sys,opt)
```

Advanced Options: Pointwise-in-Frequency Calculations

It is also possible to perform the computation pointwise-in-frequency, determining the worst-case gain at each and every frequency point. To do this, the `wcgopt` options object must be used.

```
opt = wcgopt('FreqPtWise',1);
[maxgain,maxgainunc,info] = wcgain(sys);
```

Because the calculation is pointwise-in-frequency, many results are *N*-by-1 cell arrays, often containing scalar information relevant to each particular frequency. `maxgain` is a structure with the following fields

Field	Description
`LowerBound`	Lower bound on worst-case gain, positive scalar (`frd` with *N* frequency points).
`UpperBound`	Upper bound on worst-case gain, positive scalar. If the nominal value of the uncertain system is unstable, then `maxgain.LowerBound` and `maxgain.UpperBound` equal ∞ (`frd` with *N* frequency points).
`CriticalFrequency`	Scalar. The critical value of frequency at which maximum gain occurs (this is associated with `norm(maxgain.LowerBound,inf)`).

`maxgainunc` is a *N*-by-1 cell array of values of uncertain elements that maximize the system gain. Each entry of the cell array is a `struct` whose *M* field names are the names of uncertain elements of `sys`. The maximum singular value of `usubs(sys,maxgainunc{k})` at the `k`th frequency (in `info.Frequency(k)`) is equal to `maxgain.LowerBound{k}`.

`info` is a structure with the following fields:

Field	Description
`Sensitivity`	*N*-by-1 cell. Each entry is a `struct` corresponding to the sensitivities in the worst-case gain at each individual frequency.
`Frequency`	*N*-by-1 frequency vector associated with analysis.
`ArrayIndex`	*N*-by-1 cell array. Each value is the 1-by-1 matrix whose numerical value is 1. In more complicated situations (described later) the value of this field is dependent on the input data.

Advanced Options: Handling Array Dimensions

If `sys` has array dimensions, the default behavior is to maximize over all of these dimensions as well. This can be controlled however, and it is also possible to perform the computation pointwise-in-the-array-dimensions, determining the worst-case gain at each and every grid point. Moreover, any combination of "peak-over" and "pointwise-over" is allowed. To specify the desired computation, you must use `wcgopt`. For concreteness, suppose that `sys` is a r×c×7×5×8 uncertain system (i.e., a 7-by-5-by-8 array of uncertain *r* output, *c* input systems). In order to perform the worst-case gain calculation pointwise over the 2nd and 3rd array dimensions (the slot with 5 points and the slot with 8 points), set the `ArrayDimPtWise` property as follows:

```
opt = wcgopt('ArrayDimPtWise',[2 3]);
```

In this case, the worst-case gain calculation is performed "pointwise" on the 5-by-8 grid, but only the "peak value" over the first array dimension (the slot with 7 points) is kept track of. For that reason, many of the results are of dimension 1-by-5-by-8.

In general, suppose that the array dimensions of `sys` are $d_1 \times ... \times d_F$ (7×5×8 in the above example). Furthermore, assume that the `ArrayDimPtWise` property of the `wcgopt` object is set to some of the integers between 1 and F. Let $e_1,e_2,...,e_F$ denote the dimensions of the array on which the results are computed. By definition, if `j` is an integer listed in `ArrayDimPtWise`, then $e_j=d_j$ (all grid points in slot `j` are computed); otherwise, $e_j=1$ (only the maximum in slot `j` is computed). In the above example, with `ArrayDimPtWise` set to `[2 3]`, it follows that $e_1=1,e_2=5,e_3=8$.

Assume `FreqPtWise` is set to `'off'` (you will return to that case later). In this case, the results of

```
[maxgain,maxgainunc,info] = wcgain(sys,opt)
```

are that `maxgain` is a structure with the following fields:

Field	Description
`LowerBound`	1-by-1 `frd`, with array dimensions $e_1 \times ... \times e$F, lower bound on worst-case gain, computed pointwise over all array dimensions listed in `ArrayDimPtWise` property, and peaked over all others.
`UpperBound`	Upper bound, analogous to `LowerBound`
`CriticalFrequency`	$e_1 \times ... \times e$F array with the critical value of the frequency at which maximum gain occurs (this is associated with `maxgain.LowerBound`).

`maxgainunc` is a $e_1 \times ... \times e_F$ `struct`, containing values of uncertain elements that maximize the system gain. 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 combined lead to the gain value in `maxgain.LowerBound`. The command `norm(usubs(sys,maxgainunc),'inf')` shows the gain, and should be identical to `maxgain.LowerBound` (to within the tolerance used in `norm`).

`info` is a structure with the following fields:

Field	Description
`Sensitivity`	$e_1 \times ... \times e_F$ `struct` array. Each entry is the local sensitivity of the worst-case gain in `maxgain.LowerBound` to all the individual uncertain elements uncertainty ranges.
`Frequency`	N-by-1 frequency vector associated with analysis.
`ArrayIndex`	At each value in the $e_1 \times ... \times e_F$ grid, there is a corresponding value in the $d_1 \times ... \times d_F$ grid where the maximum occurs. The variable `info.ArrayIndex` is an $e_1 \times ... \times e_F$ matrix whose value is the single-index representation of the maximizing location in the $d_1 \times ... \times d_F$ grid.

Advanced Options:

Array Dimension Handling with `FreqPtWise` Set to `'on'`

The final case involves array dimensions and pointwise-in-frequency calculations. Again, suppose that the array dimensions of `sys` are $d_1 \times ... \times d_F$. Furthermore, assume that the `ArrayDimPtWise` property of the `wcgopt` object is set to some of the integers between 1 and `F`. Let $e_1, e_2, ..., e_F$ denote the dimensions of the array on which the results are computed.

Because the calculation is pointwise-in-frequency, many results are `N-by-1` cell arrays, often containing $e_1 \times ... \times e_F$ arrays in each cell.

`maxgain` is a structure with the following fields

Field	Description
`LowerBound`	`N-by-1` cell array, `maxgain.LowerBound{k}` is a 1-by-1 `frd` with array dimensions $e_1 \times ... \times e_F$, and is a lower bound on worst-case gain at frequency `info.Frequency(k)`, computed pointwise over all array dimensions listed in the `ArrayDimPtWise` property, and "peaked" over all others.
`UpperBound`	Upper bound on worst-case gain, analogous to `maxgain.LowerBound`.
`CriticalFrequency`	$e_1 \times ... \times e_F$ array with the critical value of the frequency at which maximum gain (pointwise over all array dimensions listed in `ArrayDimPtWise` property, and "peaked" over all others) occurs.

`maxgain.CriticalFrequency`$e_1 \times ... \times e_F$ array with the critical value of the frequency at which maximum gain (pointwise over all array dimensions listed in the `ArrayDimPtWise` property, and "peaked" over all others) occurs.

`maxgainunc` is a `N-by-1` cell array, `k`th entry is a $e_1 \times ... \times e_F$ `struct`, containing values of uncertain elements that maximize the system gain at frequency `info.Frequency(k)`.

`info` is a structure with the following fields

Field	Description
`Sensitivity`	`N-by-1` cell. Each entry is the $e_1 \times ... \times e_F$ `struct` array which holds the local sensitivity of the worst-case gain at one frequency to all of the individual uncertain elements uncertainty ranges.
`Frequency`	`N-by-1` frequency vector associated with analysis.
`ArrayIndex`	`N-by-1` cell array, `k`th $e_1 \times ... \times e_F$, at each value in the $e_1 x ... x e_F$ grid, there is a corresponding value in the $d_1 \times ... \times d_F$ grid where the maximum occurs. The variable `info.ArrayIndex` is a $e_1 \times ... \times e_F$ matrix whose value is the single-index representation of the maximizing location in the $d_1 \times ... \times d_F$ grid.

Behavior on Not-Uncertain Systems

`wcgain` can also be used on not-uncertain systems (e.g., `ss` and `frd`). If `sys` is a single `ss` or `frd`, then the worst-case gain is simply the gain of the system (identical to `norm(sys,'inf')`). However, if `sys` has array dimensions, then the possible combinations of "peak-over" and "pointwise-over" can be used to customize the computation.

Algorithm

The worst-case gain is guaranteed to be at least as large as `LowerBound` (some value of allowable uncertain elements yield this gain – one instance is returned in the structure `maxgainunc`. The frequency at which the gain in `LowerBound` occurs is in `CriticalFrequency`. Lower bounds for `wcgain` are computed using a power iteration on `ultidyn`, `ucomplex` and `ucomplexm` uncertain atoms, (holding uncertain real parameters fixed) and a coordinate aligned search on the uncertain real parameters (while holding the complex blocks fixed).

Similarly, the worst-case gain is guaranteed to be no larger than `UpperBound`. In other words, for all allowable modeled uncertainty, the gain is provably less than or equal to `UpperBound`. These bounds are derived using the upper bound for the structured singular value, which is essentially optimally-scaled, small-gain theorem analysis. Upper bounds are obtained by solving a semidefinite program. `wcgain` uses branch and bound on the uncertain real parameters to tighten the lower and upper bounds.

Limitations

Because the calculation is carried out with a frequency grid, 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, compared with `robuststab`, the problem in `wcgain` is less acute. Thought of as a function of problem data and frequency, the worst-case gain is a continuous function (unlike the robust stability margin, which in special cases is not; see the demo Getting Reliable Estimates of Robustness Margins). Hence, in worst-case gain 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.

See Also

Comprehensive analysis of feedback loops

`mussv`

`norm`

`robuststab`

`wcgopt`

`wcsens`

`wcmargin`

Purpose

Create options object for use with `wcgain`, `wcsens`, and `wcmargin`

Syntax

```
options = wcgopt
options = wcgopt('name1',value1,'name2',value2,...)
wcgopt
```

Description

`options = wcgopt` (with no input arguments) creates an options object with all the properties set to their default values.

`options = wcgopt('name1',value1,'name2',value2,...)` creates a `wcgain`, `wcsens` and `wcmargin` options object called options in which specified properties have specific values. Any unspecified property is set to its default value. It is sufficient to type only enough leading characters to define the property name uniquely. Case is ignored for property names.

`wcgopt` with no input or output arguments displays a complete list of option properties and their default values.

The following are the `wcgopt` object properties:

Object Property	Description
`Sensitivity`	Computes margin sensitivity to individual uncertainties {`'off'; 'on'`}. Default is `'on'`
`LowerBoundOnly`	If `LowerBoundOnly` is `'on'`, then only the lower bound computation is performed. The default value is `'off'`, which implies that both upper and lower bounds for worst-case gain are computed.
`FreqPtWise`	Apply stopping criteria based on upper/lower bounds (described below) at every frequency point (as opposed to just the peak value). `FreqPtWise=1` activates the pointwise criteria. In order to only compute the peak value to within tolerance, use 0. Default = 0.

Object Property	Description
`ArrayDimPtWise`	Relevant for `uss/ufrd/ss/frd` arrays. For indices specified in `ArrayDimPtWise`, the stopping criteria based on upper/lower bounds (described below) is used at every point in array dimensions specified in `ArrayDimPtWise`, being applied to the peak value over all other array dimensions. Default = []. If `FreqPtWise==1`, the computation terminates when at least one of the following four conditions is true at *every* frequency: • `UpperBound - LowerBound <= AbsTol` • `UpperBound - LowerBound <= Reltol*UpperBound` • `UpperBound <= AGoodThreshold + MGoodThreshold*Norm(NominalValue)` `LowerBound >= ABadThreshold + MBadThreshold*Norm(NominalValue)` If `FreqPtWise==0`, the computation terminates when any one of the following four conditions is true: • `PeakUpperBound - PeakLowerBound <= AbsTol` • `PeakUpperBound - PeakLowerBound <= Reltol*PeakUpperBound` at every frequency • `UpperBound <= AGoodThreshold + MGoodThreshold*Norm(NominalValue))` at some frequency `LowerBound >= ABadThreshold + MBadThreshold*Norm(NominalValue)` In both situations above, the stopping condition is applied at every point in array dimensions specified in `ArrayDimPtWise`. `UpperBound` and `LowerBound` are the peak values over all other array dimensions.
`Default`	Structure with default values of all wcgopt properties.

Object Property	Description
`Meaning`	Structure. Field names are `wcgopt` properties, and values are the text description of property.
`VaryUncertainty`	Percentage variation of uncertainty used as a step size in finite-difference calculations to estimate sensitivity. Default is 25.
`AbsTol`	Upper and lower absolute stopping tolerance. Default=0.02.
`RelTol`	Upper and lower absolute stopping tolerance. Default = 0.02.
`AbsTol`	Upper and lower relative stopping tolerance. Default=0.05.
`MGoodThreshold`	Multiplicative (`UpperBound`) stopping threshold. Default = 1.04.
`AGoodThreshold`	Additive (`UpperBound`) stopping threshold. Default = 0.05.
`MBadThreshold`	Multiplicative (`LowerBound`) stopping threshold, Default = 5.
`AGoodThreshold`	Additive (`LowerBound`) stopping threshold. Default = 20.
`NTimes`	Number of restarts in lower bound search (positive integer).
`MaxCnt`	Number of cycles in lower bound search (positive integer). Default = 3.
`MaxTime`	Maximum computation time allowed (in seconds). The computation is prematurely terminated if this much real time elapses before the computation is complete. All quantities that have been computed are returned. Default = 720.

Examples

You can create a `wcgopt` options object called `opt` with all default values.

```
opt = wcgopt
Property Object Values:
        Sensitivity: 'on'
     LowerBoundOnly: 'off'
         FreqPtWise: 0
      ArrayDimPtWise: []
    VaryUncertainty: 25
```

```
              Default: [1x1 struct]
              Meaning: [1x1 struct]
               AbsTol: 0.0200
               RelTol: 0.0500
      MGoodThreshold: 1.0400
      AGoodThreshold: 0.0500
       MBadThreshold: 20
       ABadThreshold: 5
              NTimes: 2
              MaxCnt: 3
             MaxTime: 720
```

The following statements change the absolute tolerance stopping criterion from `0.02` to `0.04` and the point wise over frequency test from the peak worst-case value, `opt.FreqPtWise=0`, to the worst-case value at every frequency.

```
opt = wcgopt;
opt.AbsTol = 0.04;
opt.FreqPtWise = 1;
opt
Property Object Values:
        Sensitivity: 'on'
      LowerBoundOnly: 'off'
          FreqPtWise: 1
      ArrayDimPtWise: []
              Default: [1x1 struct]
              Meaning: [1x1 struct]
     VaryUncertainty: 25
               AbsTol: 0.0400
               RelTol: 0.0500
      MGoodThreshold: 1.0400
      AGoodThreshold: 0.0500
       MBadThreshold: 20
       ABadThreshold: 5
              NTimes: 2
              MaxCnt: 3
```

```
MaxTime: 720
```

This statement makes a single call to `wcgopt` to set the maximum computation time to 10000 seconds and disables the `Sensitivity` calculation.

```
opt = wcgopt('MaxTime',10000,'Sensitivity','off');
```

See Also

`dkitopt`

`robopt`

`wcgain`

`wcnorm`

`wcsens`

`wcmargin`

wcmargin

Purpose

Worst-case disk stability margins of uncertain feedback loops

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 loop-at-a-time allowable, independent 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 calculates the largest region for each channel such that for all gain and phase variations inside the region the nominal closed-loop system is stable. The guaranteed bound is calculated based on the balanced sensitivity function. See the `dmplot` and reference pages to learn more about the algorithm.

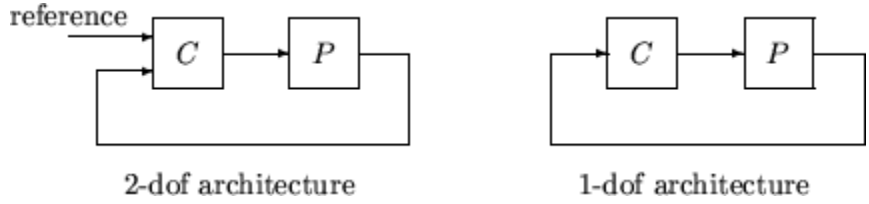
Consider a system with uncertain elements. It is of interest to determine the gain and phase margins of each individual channel in the presence of uncertainty. These margins are called worst-case margins. Worst-case margin, `wcmargin` calculates the largest disk margin such that for values of the uncertainty 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).
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 wcgopt 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.



2-dof architecture 1-dof architecture

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 `NY` outputs and `NU` inputs and hence the controller `C` must have `NU` outputs and `NY` 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.

Field	Description
`Frequency`	Frequency associated with the worst-case margin (rad/s).
`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 `wcgopt` 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 `NY`-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 `wcgopt` 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 `wcgopt` options object, and set the `Sensitivity` property to `'on'`.

```
opt = wcgopt('Sensitivity','on');
[maxgain,maxgainunc,info] = wcgain(sys,opt)
```

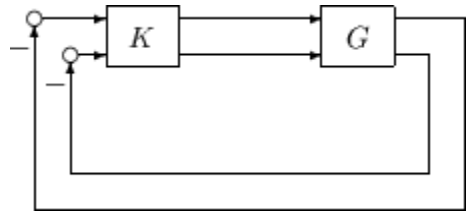
Examples

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

$$G := \frac{1}{s^2 + \alpha^2}\begin{bmatrix} s - \alpha^2 & \alpha(s+1) \\ -\alpha(s+1) & s - \alpha^2 \end{bmatrix}, \quad K = I_2$$

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.

See Also

`dmplot`

`loopsens`

Performs comprehensive analysis of feedback loops

`robuststab`

`usubs`

`wcgain`

`wcgopt`

`wcsens`

Purpose

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.
`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* fieldnames, 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] = wcgain(mat)
```

The third output argument `info` is a structure with the following fields:

Field	Description
`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 `wcgopt` object is `'off'`, the values are `NaN`.
`ArrayIndex`	1-by-1 scalar matrix with the value of 1. In more complicated situations (described later) the value of this field depends on the input data.

Advanced options: Handling array dimensions

If `mat` has array dimensions, the default behavior is to maximize over all dimensions. It is also possible to perform the computation pointwise-in-the-array-dimensions to determine the worst-case norm at each grid point. Any combination of "peak-over" and "pointwise-over" is allowed.

To specify the desired computation, the `wcgopt` must be used. For concreteness, suppose that `mat` is an r×c×7×5×8 uncertain system (i.e., a 7-by-5-by-8 array of uncertain *r* output, *c* input systems). To perform the worst-case gain calculation pointwise over the second and third

array dimensions (the slots with 5 points and 8 points, respectively), set the `ArrayDimPtWise` property:

```
opt = wcgopt('ArrayDimPtWise',[2 3]);
```

In this case, the worst-case norm calculation is performed "pointwise" on the 5-by-8 grid. Only the "peak value" in the first array dimension (the slot with 7 points) is tracked. For that reason, many of the results will be of dimension 1-by-5-by-8.

In general, suppose that the array dimensions of `sys` are $d_1 \times ... \times d_F$ (7×5×8 in the above example). Furthermore, assume that the `ArrayDimPtWise` property of the `wcgopt` object has been set to some of the integers between 1 and `F`. Let $e_1, e_2, ..., e_F$ denote the dimensions of the array on which the results are computed. By definition, if `j` is an integer listed in `ArrayDimPtWise`, then $e_j = d_j$ (all grid points in slot `j` are computed), otherwise $e_j = 1$ (only the maximum in slot `j` is computed). In the above example, with `ArrayDimPtWise` set to `[2 3]`, it follows that $e_1 = 1$, $e_2 = 5$, $e_3 = 8$.

In this case, the following command

```
[maxgain,maxgainunc,info] = wcgain(sys,opt)
```

produces the `maxgain` a structure with the following fields

Field	Description
`LowerBound`	$e_1 \times ... \times e_F$ matrix of lower bounds on worst-case norm, computed pointwise over all array dimensions listed in `ArrayDimPtWise` property and "peaked" over all others.
`UpperBound`	Upper bound, analogous to `LowerBound`

`maxgainunc` is a $e_1 \times ... \times e_F$ `struct`, containing values of uncertain elements which maximize the system norm. There are M fieldnames, which are the names of uncertain elements of `mat`. The value of each

field is the corresponding value of the uncertain element, which lead to the gain value in `maxnorm.LowerBound` when jointly combined.

`info` is a structure with the following fields

Field	Description
`Sensitivity`	$e_1 \times ... \times e_F$ `struct` array, where each entry is the local sensitivity of the worst-case norm in `maxnorm.LowerBound` to the uncertainty range of each uncertain element.
`ArrayIndex`	At each value in the $e_1 \times ... \times e_F$ grid, there is a corresponding value in the $d_1 \times ... \times d_F$ grid where the maximum occurs. The variable `info.ArrayIndex` is an $e_1 \times ... \times e_F$ matrix, where the value is the single-index representation of the maximizing location in the $d_1 \times ... \times d_F$ grid.

Examples

You can construct an uncertain matrix and compute the worst-case norm of the matrix, as well as its inverse. Your objective is to 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',2,'Range',[1 3]);
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 =
    LowerBound: 14.7199
    UpperBound: 14.7327
[maxnormMi] = wcnorm(Mi)
maxnormMi =
    LowerBound: 2.5963
```

```
    UpperBound: 2.5979
```

The condition number of `M` must be less than the product of the two upper bounds for all values of the uncertain elements making up `M`. Conversely, the largest value of `M` condition number must be at least equal to the condition number of the nominal value of `M`. Compute these crude bounds on the worst-case value of the condition number.

```
condUpperBound = maxnormM.UpperBound*maxnormMi.UpperBound;
condLowerBound = cond(M.NominalValue);
[condLowerBound condUpperBound]
ans =
    5.0757   38.2743
```

How can you get a more accurate estimate? Recall that the condition number of an `nxm` matrix `M` can be expressed as an optimization, where a free norm-bounded matrix Δ tries to align the gains of `M` and `M-`[1]

$$\kappa(M) = \max_{\substack{\Delta \in C^{m \times m} \\ \sigma_{max}(\Delta) \le 1}} (\sigma_{max}(M\Delta M^{-1}))$$

If `M` is itself 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 then by using `wcnorm` to carry out the maximization.

Create a 2-by-2 `ucomplexm` object, with nominal value equal to zero.

```
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.

You can create the expression involving `M, Delta` and `inv(M)`.

```
H = M*Delta*Mi;
```

Finally, consider the stopping criteria and call `wcnorm`. One stopping criteria for `wcnorm(H)` is based on the norm of the nominal value of `H`. During the computation, if `wcnorm` determines that the worst-case norm is at least

$$ABadThreshold + MBadThreshold*norm(H.NominalValue)$$

then the calculation is terminated. In our case, since `H.NominalValue` equals 0, the stopping criteria is governed by `ABadThreshold`. The default value of `ABadThreshold` is 5. To keep `wcnorm` from prematurely stopping, set `ABadThreshold` to 38 (based on our crude upper bound above).

```
opt = wcgopt('ABadThreshold',38);
[maxKappa,wcu,info] = wcnorm(H,opt);
maxKappa
maxKappa =
    LowerBound: 26.9629
    UpperBound: 27.9926
```

You can verify that `wcu` makes the condition number as large as `maxKappa.LowerBound`.

```
cond(usubs(M,wcu))
ans =
   26.9629
```

Algorithm

See `wcgain`

See Also

`norm`

`frd/svd`

`wcgain`

`wcgopt`

Purpose

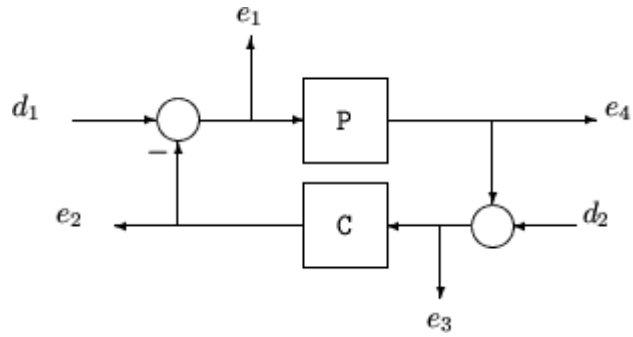
Calculate worst-case sensitivity and complementary sensitivity functions of plant-controller feedback loop

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.



Description	Equation
Input sensitivity ($TF_{e1 \leftarrow d1}$)	*(I+CP)*-1
Input complementary sensitivity ($TF_{e2 \leftarrow d1}$)	*CP(I+CP)*-1
Output sensitivity ($TF_{e3 \leftarrow d2}$)	*(I+PC)*-1
Output complementary sensitivity ($TF_{e4 \leftarrow d}$)	*PC(I+PC)*-1
Input loop transfer function	*CP*
Output loop transfer function	*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

Fields of `wcst` (Continued)

Field	Description
`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`).

Fields of Si, So, Ti, To, PSi, CSo (Continued)

Field	Description
`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 `wcgopt` object is `'off'`, the values are `NaN`.

`wcst = wcsens(L,type)` and `wcst = wcsens(P,C,type)` allows 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 containing a collection of strings, i.e. `'Si','To'`, as well as a comma separated list.

`wcst = wcsens(L,type,scaling)` and `wcst = wcsens(P,C,type,scaling)` adds a `scaling` to the worst-case sensitivity analysis. `scaling` is either the character strings `'Absolute'` (default), `'Relative'` or a `ss/tf/zpk/frd` object. The default scaling `'Absolute'` calculates bounds on the maximum gain of the uncertain sensitivity function. The `'Relative'` scaling 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, Similarly if `scaling` is a `ss/tf/zpk/frd` object, bounds on the maximum scaled gain of the uncertain sensitivity function are found. If `scaling` is `'Relative'`or a `ss/tf/zpk/frd` object, the worst-case analysis peaks over frequency. If `scaling` is an object, its input/output dimensions should be 1-by-1 or dimensions compatible with `P` and `C`. `type` and `scaling` can also be combined 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 `wcgopt` 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 `wcgopt` options object, and set the `Sensitivity` property to `'on'`.

```
opt = wcgopt('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.

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

Comprehensive analysis of feedback loops

loopsens

robuststab

usubs

wcgain

wcgopt

wcmargin

3

Block Reference

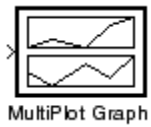
MultiPlot Graph
Uncertain State Space
USS System

MultiPlot Graph

Purpose

Plot multiple signals

Description



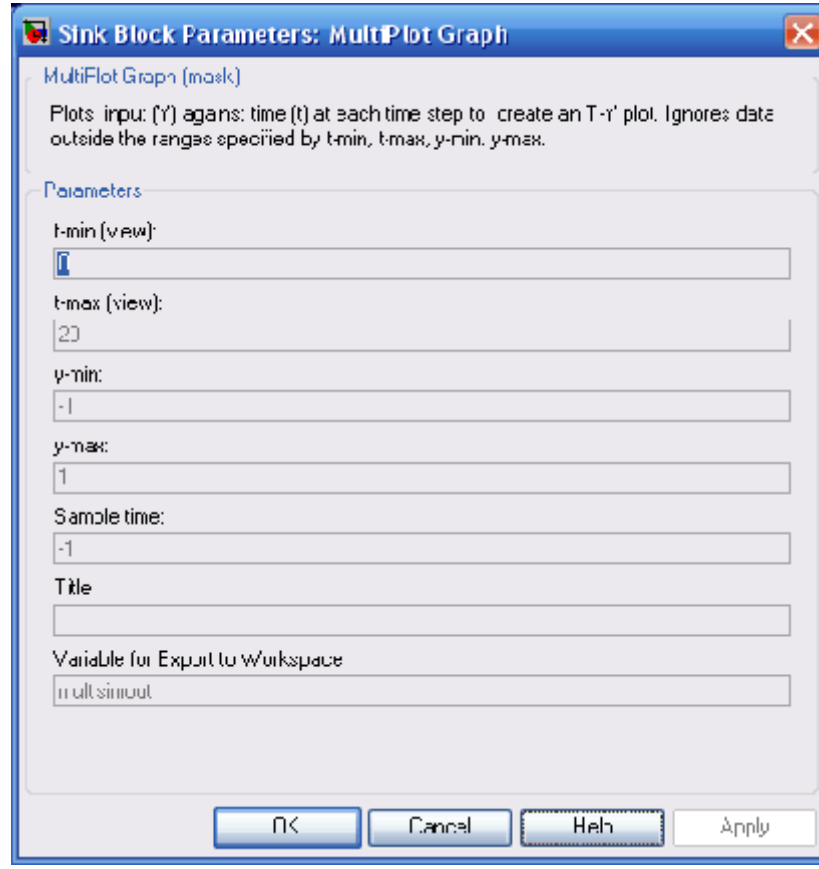
The MultiPlot Graph block displays signals in a MATLAB figure.

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.

Dialog Box



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.

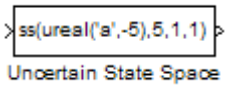
Variable for Export to Workspace

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.

Purpose

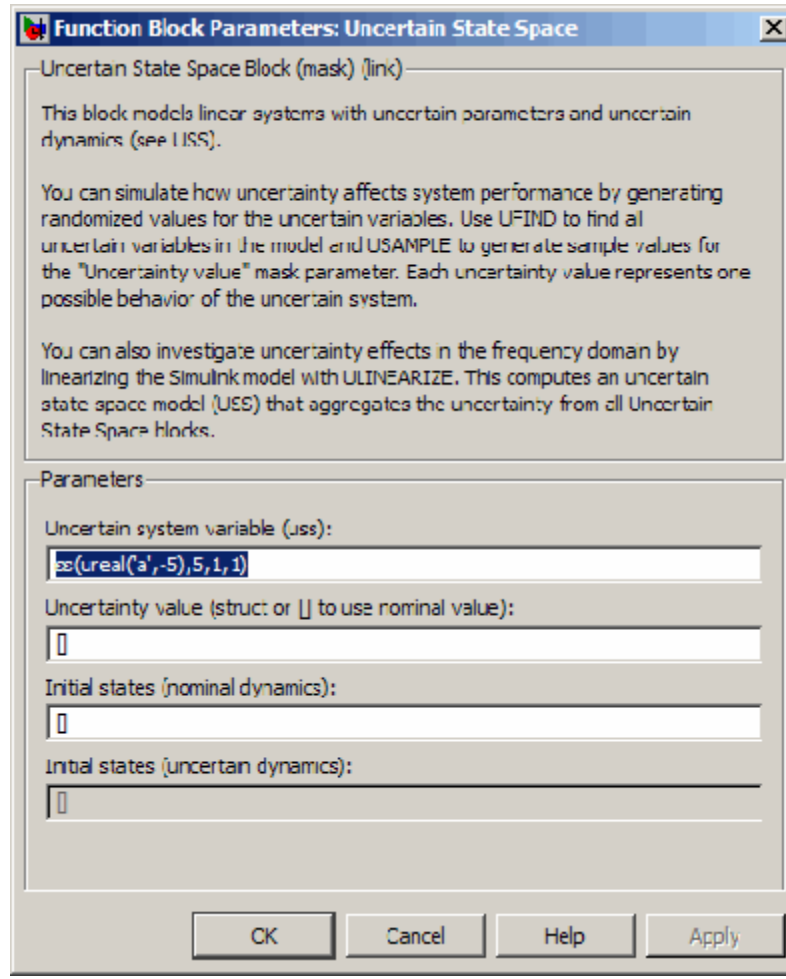
Specify uncertain system in Simulink

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.

Dialog Box



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`).

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 "Varying Uncertainty Values Using Individual Uncertain State Space Blocks" and "Varying 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`, `usample`, `ulinearize`, `uss`, `umat`, `ureal`, `ultidyn`

Tutorials

Robustness Analysis in Simulink

Linearization of Simulink Models with Uncertainty

How To

"Simulating Uncertainty Effects"

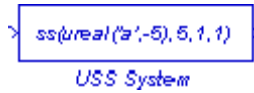
"Computing Uncertain State-Space Models from Simulink Models"

Purpose

Import uncertain systems into Simulink

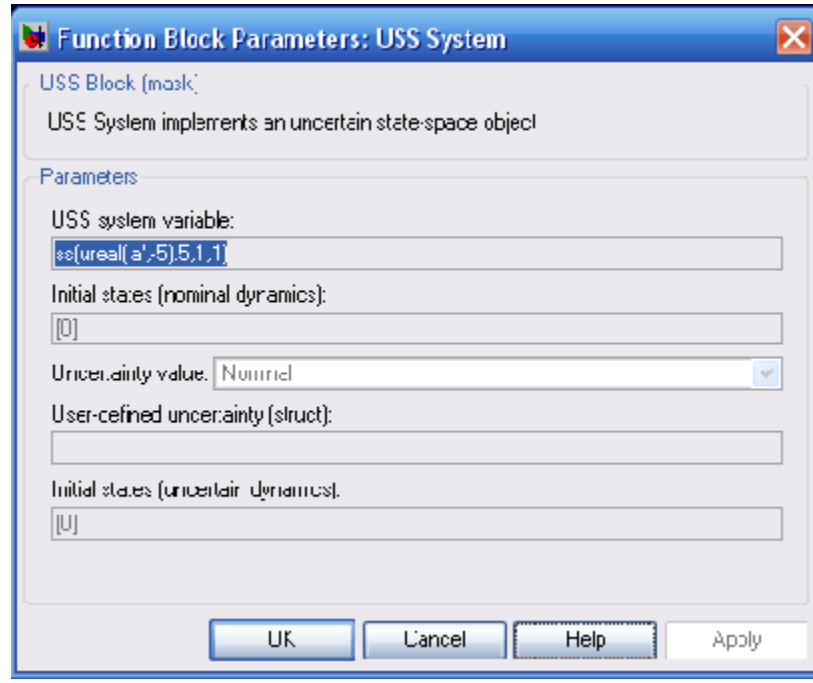
Note USS System block will be removed in a future release. Use Uncertain State Space block instead.

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.

Dialog Box



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)`.

Index

Symbols and Numerics

A

B

C

D

F

G

H

I

L

M

N

P

R

S

T

Z