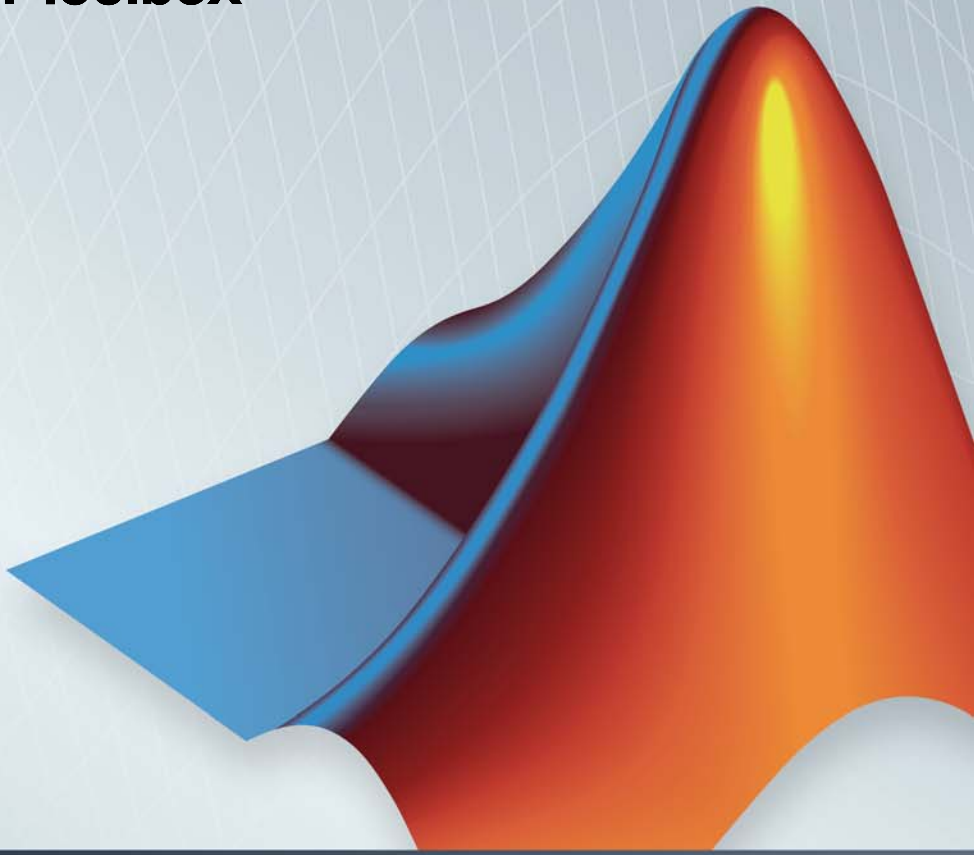
Robust Control Toolbox™

Reference

R2013**b**

Gary Balas
Richard Chiang
Andy Packard
Michael Safonov



MATLAB®

MathWorks®

Revision History

September 2005	First printing	New for Version 3.0.2 (Release 14SP3)
March 2006	Online only	Revised for Version 3.1 (Release 2006a)
September 2006	Online only	Revised for Version 3.1.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)
September 2007	Online only	Revised for Version 3.3 (Release 2007b)
March 2008	Online only	Revised for Version 3.3.1 (Release 2008a)
October 2008	Online only	Revised for Version 3.3.2 (Release 2008b)
March 2009	Online only	Revised for Version 3.3.3 (Release 2009a)
September 2009	Online only	Revised for Version 3.4 (Release 2009b)
March 2010	Online only	Revised for Version 3.4.1 (Release 2010a)
September 2010	Online only	Revised for Version 3.5 (Release 2010b)
April 2011	Online only	Revised for Version 3.6 (Release 2011a)
September 2011	Online only	Revised for Version 4.0 (Release 2011b)
March 2012	Online only	Revised for Version 4.1 (Release 2012a)
September 2012	Online only	Revised for Version 4.2 (Release 2012b)
March 2013	Online only	Revised for Version 4.3 (Release 2013a)
September 2013	Online only	Revised for Version 5.0 (Release 2013b)

Contents

Class Reference

Purpose

Gain constraint for control system tuning

Description

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

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

You can use a gain constraint to:

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

Construction

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

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

`Req = TuningGoal.Gain(inputname,outputname,gainprofile)` specifies the maximum gain as a function of frequency. You can specify the target gain profile (maximum gain across the I/O pair) as a smooth transfer function. Alternatively, you can sketch a piecewise error profile using an `frd` model.

Input Arguments

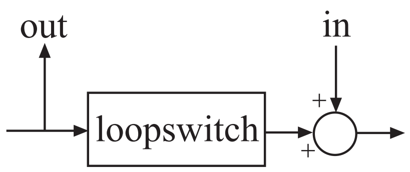
inputname

Input signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as input signals for the tuning requirement are as follows.

- If you are using the requirement to tune a Simulink® model of a control system, then `inputname` can include:
  - Any model input
  - Any linearization input point in the model
  - Any signal identified as a `Controls`, `Measurements`, `Switches`, or `IOs` signal in an `slTunable` interface associated with the Simulink model
- If you are using the requirement to tune a generalized state-space model (`genss`) of a control system using `systune`, then `inputname` can include:
  - Any input of the control system model
  - Any `loopswitch` channel in the control system model

  For example, if you are tuning a control system model, `T`, then `inputname` can be a string contained in `T.InputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `inputname` can include `X`.
- If you are using the requirement to tune a controller model, `C0` for a plant `G0`, using `looptune`, then `inputname` can include:
  - Any input of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `inputname` is a `loopswitch` channel of a generalized model, the input signal for the requirement is the implied input associated with the switch:



outputname

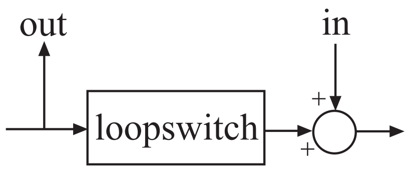
Output signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as output signals for the tuning requirement are as follows.

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

  For example, if you are tuning a control system model,`T`, then `outputname` can be a string contained in `T.OutputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `outputname` can include `X`.

- If you are using the requirement to tune a controller model, `C0`, for a plant, `G0`, using `looptune`, then `outputname` can include:
  - Any output of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `outputname` is a `loopswitch` channel of a generalized model, the output signal for the requirement is the implied output associated with the switch:



gainvalue

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

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

gainprofile

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

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

Properties

Input

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

Output

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

MaxGain

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

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

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB®, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

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

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

$T(s,x)$ is the closed-loop transfer function from `Input` to `Output`. $\|\cdot\|_\infty$ denotes the H_∞ norm (see `norm`).

Examples

Disturbance rejection

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

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

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

Custom roll-off specification

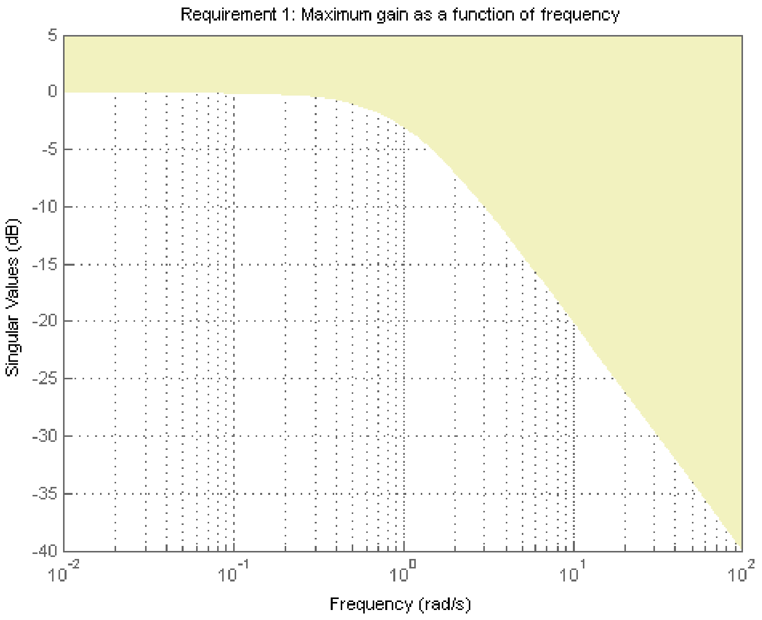
Create a gain constraint that constrains the response from a signal `'du'` to a signal `'u'` to roll off at 20 dB/decade at frequencies greater than 1. The gain constraint also specifies disturbance rejection (maximum gain of 1) in the frequency range [0,1].

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

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

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

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

Constrain Open-Loop Gain

Create a gain constraint that limits to 100 the open-loop gain from a signal `'u'` to a signal `'y'`. Set the `Openings` property to the name of signal at which you want to open the loop.

```
Req = TuningGoal.Gain('u','y',100);
Req.Openings = 'u';
```

See Also

`slTunable.looptune` | `looptune` | `slTunable.systune` | `viewSpec` | `systune` | `TuningGoal.Tracking` | `TuningGoal.LoopShape` | `slTunable`

How To

- “Specifying Design Requirements for systune”
- “Performance and Robustness Specifications for looptune”
- “Using Design Requirement Objects”
- “Control of a Linear Electric Actuator”
- “Multi-Loop PID Control of a Robot Arm”
- “MIMO Control of Diesel Engine”

Purpose

Target loop shape for control system tuning

Description

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

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

Use `TuningGoal.LoopShape` when the loop shape near crossover is simple or well understood (such as integral action). To specify only high gain or low gain constraints in certain frequency bands, use `TuningGoal.MinLoopGain` and `TuningGoal.MaxLoopGain`. When you do so, the software determines the best loop shape near crossover.

Construction

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

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

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

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

Input Arguments

location

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

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, `location` contains the names of one or more feedback loops in the `Location` property of `loopswitch` blocks in the control system model. For example, the following code creates a PI loop with a loop switch marking the plant input `'u'`.

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

You can use the string `'u'` to refer to the sensitivity at the plant input.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, `location` contains the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, `location` contains the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

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

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

loopgain

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

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

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

crosstol

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

Default: 0.1

wc

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

wcrange

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

Properties

Location

Location at which the open-loop response shape to be constrained is measured, specified as a string or cell array of strings that identify one or more loop opening locations in the control system to tune.

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

LoopGain

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

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

CrossTol

Tolerance on gain crossover frequency, in decades.

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

Default: `0.1`

LoopScaling

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

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

Default: `'on'`

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

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

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

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

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

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

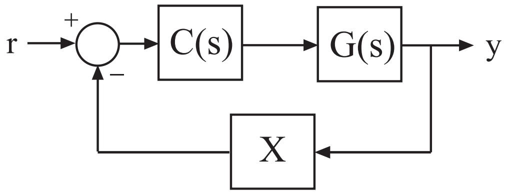
$T = S - I$ is the complementary sensitivity function.

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

Examples

Loop Shape and Crossover Tolerance

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



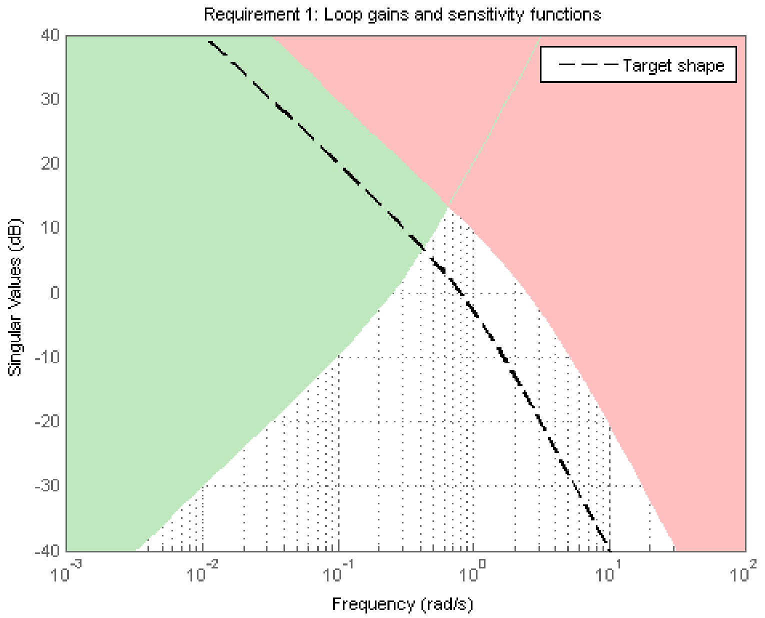
The requirement should apply to the open-loop response measured at the `loopswitch` block `X`. Specify a crossover tolerance of 0.5 decades.

Use an `frd` model to sketch the target loop shape.

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

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

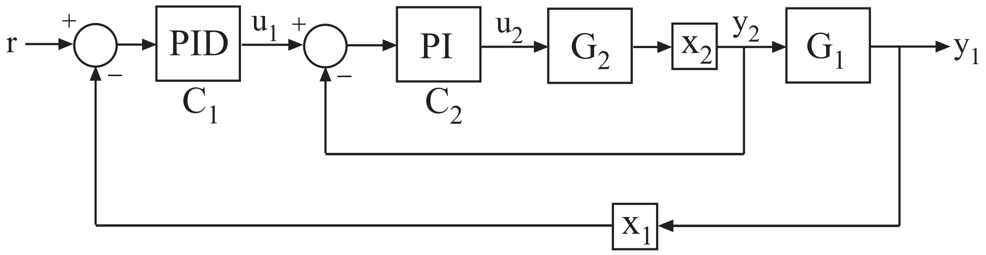
```
viewSpec(Req)
```



The red and green regions indicate the bounds for the sensitivity, `S = 1/(1-G*C)`, and the complementary sensitivity, `T = (G*C)/(1-G*C)`, respectively. When you use `viewSpec(Req,CL)` to validate a tuned closed-loop model of this control system, `CL`, the tuned values of `S` and `T` are also plotted.

Specify Different Loop Shapes for Multiple Loops

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



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

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

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

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

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

Specifying `'X1'` for the `location` indicates that `Req1` applies to the point-to point, open-loop transfer function at the loop opening location

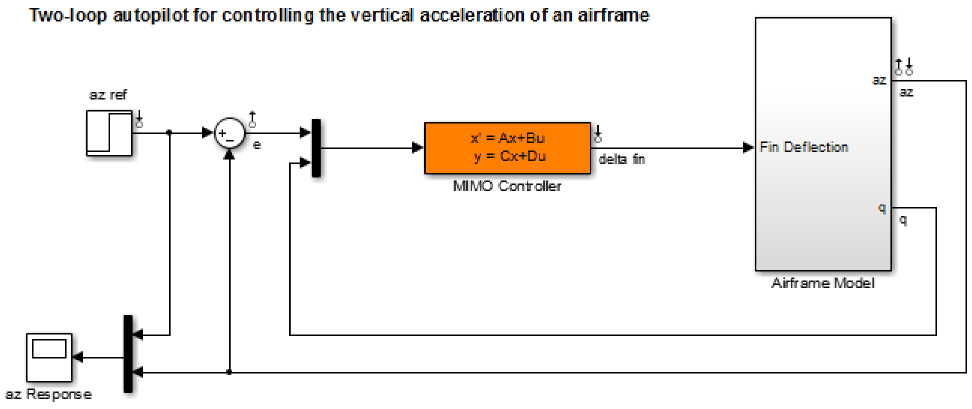
X1. You do not have to set Req1.Openings because this loop shape is enforced with the inner loop closed.

You may need to tune the control system with both loop shaping requirements Req1 and Req2. To do so, use both requirements as inputs to the tuning command. For example, suppose CL0 is a tunable genss model of the closed-loop control system. In that case, the following command tunes the control system to both requirements.

```
[CL,fSoft] = systune(CL0,[Req1,Req2]);
```

Loop Shape for Tuning Simulink Model

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



Open the model.

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

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

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

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

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

To use this requirement to tune the Simulink model, create an slTunable interface to the model.

```
STO = slTunable('rct_airframe2','MIMO Controller');
```

Designate both az and q as potential loop-opening locations in the slTunable interface.

```
STO.Openings = {'az','q'};
```

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

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

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

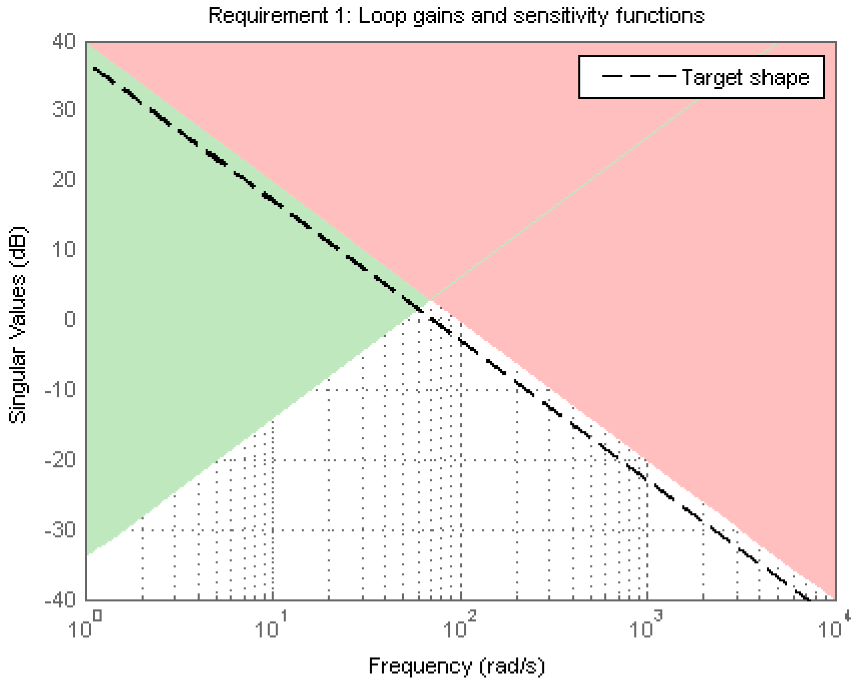
Loop Shape Requirement with Crossover Range

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

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

Examine the resulting requirement to see the target loop shape.

```
viewSpec(Req)
```



The plot shows that the requirement specifies an integral loop shape, with crossover around 70 rad/s, the geometrical mean of the range `[50,100]`.

See Also

`slTunable.looptune` | `slTunable.systune` | `looptune` | `systune` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain` | `viewSpec` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `slTunable` | `frd`

How To

- “Specifying Design Requirements for systune”

- “Performance and Robustness Specifications for looptune”
- “Using Design Requirement Objects”
- “Tuning Multi-Loop Control Systems”
- “Tuning of a Digital Motion Control System”

TuningGoal.Margins

Purpose

Stability margin requirement for control system tuning

Description

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

After you create a requirement object, you can further configure the tuning requirement by setting "Properties" on page 1-28 of the object.

Construction

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

Input Arguments

location

Loop-opening locations at which the minimum gain and phase margins apply, specified as a string or cell array of strings. These strings identify one or more loop-opening locations in the control system to tune. What loop-opening locations are available depends on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, `location` contains the names of one or more feedback loops in the `Location` property of `loopswitch` blocks in the control system model. For example, the following code creates a PI loop with a loop switch marking the plant input `'u'`.

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

You can use the string `'u'` to refer to the sensitivity at the plant input.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, `location` contains the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, `location` contains the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

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

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

gainmargin

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

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

phasemargin

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

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

Properties

Location

Location at which the minimum gain and phase margins apply, specified as a string or cell-array of strings. These strings identify one or more loop-opening locations in the control system to tune.

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

GainMargin

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

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

PhaseMargin

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

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

Focus

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

Set the `Focus` property to limit enforcement of the requirement to a particular frequency band. For best results with stability margin requirements, pick a frequency band extending about one decade on each side of the gain crossover frequencies. For example, suppose `Req` is a `TuningGoal.Margins` requirement that you are using to tune a system with approximately 10 rad/s bandwidth. To limit the enforcement of the requirement, use the following command:

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

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

Models

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

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

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

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

Default: `NaN`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Algorithms

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

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

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

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

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

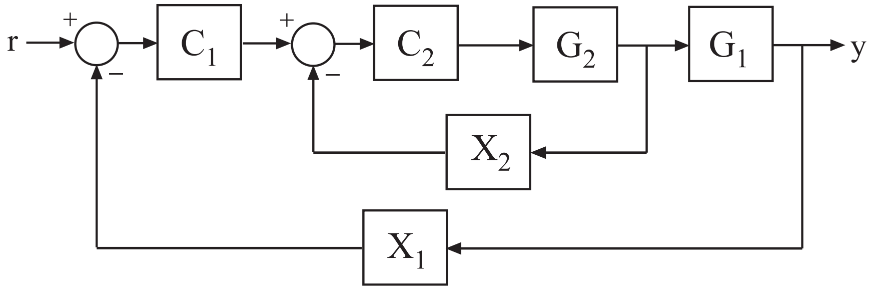
D is an automatically-computed loop scaling factor.

α is a scalar parameter computed from the specified gain and phase margin.

Examples

SISO Margin Requirement Evaluated with Additional Loop Opening

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



Create a model of the system. To do so, specify and connect the numeric plant models `G1` and `G2`, and the tunable controllers `C1` and `C2`. Also specify and connect the `loopswitch` blocks `X1` and `X2` that mark potential loop-opening locations.

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

Create a tuning requirement object.

```
Req = TuningGoal.Margins('X2',5,40);
```

This requirement imposes the specified stability margins on the feedback loop identified by the `loopswitch` channel `'X2'`, which is the inner loop.

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

```
Req.Openings = {'X1'};
```

Adding `'X1'` to the `Openings` property of the tuning requirements object ensures that `systune` evaluates the requirement with the loop open at that location.

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

MIMO Margin Requirement in Frequency Band

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

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

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

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

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

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

See Also

`slTunable.looptune` | `looptune` | `systune` | `slTunable.systune` | `viewSpec` | `evalSpec`

How To

- “Specifying Design Requirements for systune”
- “Performance and Robustness Specifications for looptune”
- “Using Design Requirement Objects”
- “Tuning Control Systems with SYSTUNE”
- “Digital Control of Power Stage Voltage”
- “Tuning of a Two-Loop Autopilot”

- “Fixed-Structure Autopilot for a Passenger Jet”

Purpose

Minimum loop gain constraint for control system tuning

Description

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

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

Construction

`Req = TuningGoal.MinLoopGain(location,loopgain)` creates a tuning requirement for boosting the gain of a SISO or MIMO feedback loop. The requirement specifies that the open-loop frequency response measured at the specified locations exceeds the minimum gain profile specified by `loopgain`. You can specify the minimum gain profile as a smooth transfer function or sketch a piecewise error profile using an `frd` model. Only gain values greater than 1 are enforced.

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

Input Arguments

location

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

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, `location` contains the names of one or more feedback loops in the `Location` property of `loopswitch` blocks in the control system model. For example, the following code creates a PI loop with a loop switch marking the plant input `'u'`.

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

You can use the string `'u'` to refer to the sensitivity at the plant input.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, `location` contains the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, `location` contains the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

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

loopgain

Minimum open-loop gain as a function of frequency.

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

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

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

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

fmin

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

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

gmin

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

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

Properties

Location

Location at which minimum loop gain is constrained, specified as a string or cell array of strings. These strings identify one or more loop-opening locations in the control system to tune.

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

MinGain

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

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

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

LoopScaling

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

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

Default: `'on'`

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

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

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

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

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

Examples

Minimum Loop Gain Requirement

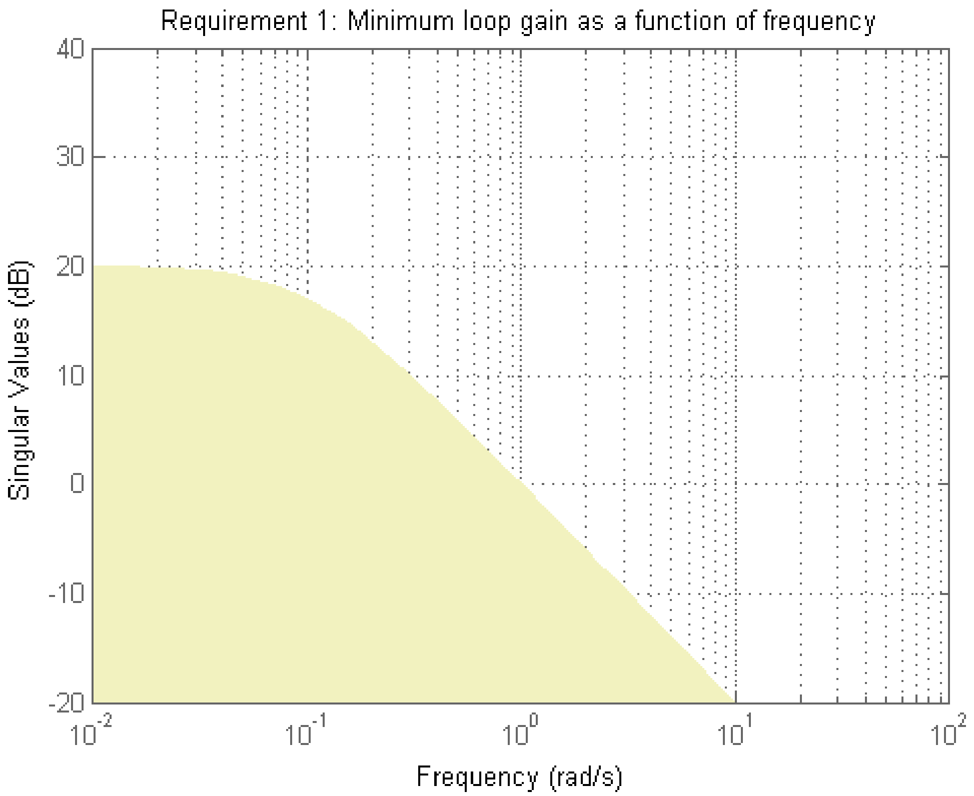
Create a requirement that boosts the open-loop gain of a feedback loop to greater than a specified profile.

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

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

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

```
viewSpec(Req)
```



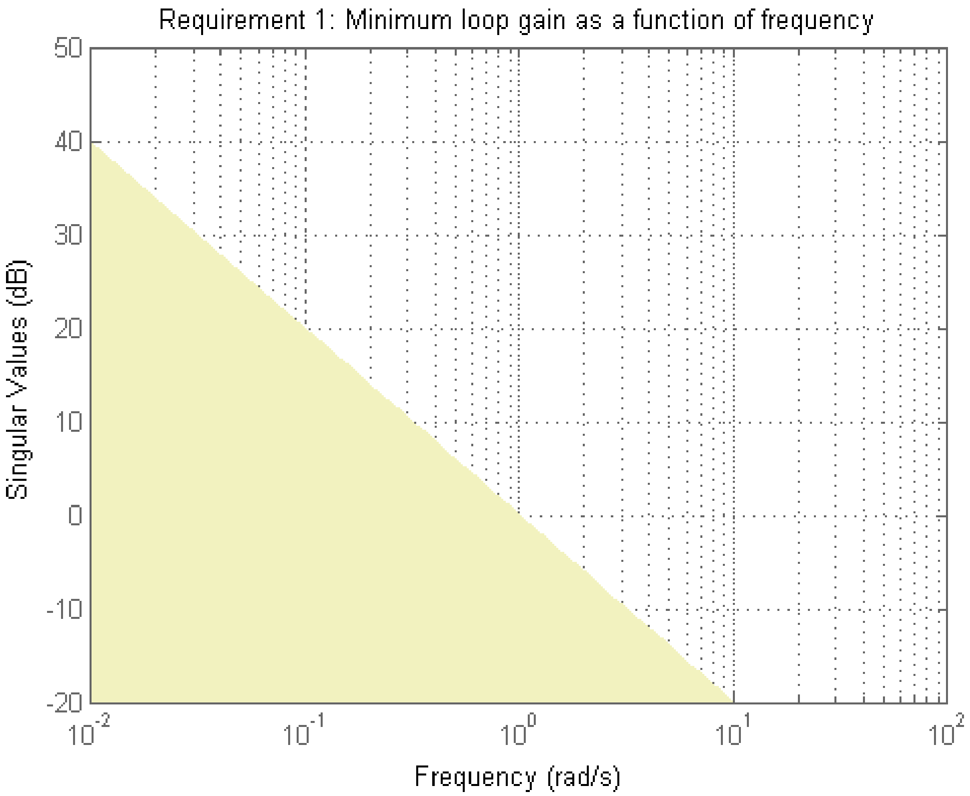
The yellow region indicates where the requirement is violated, except that gain values less than 1 are not enforced. Therefore, this requirement only specifies a minimum gain at frequencies below 1 rad/s.

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

Integral Minimum Gain Specified as Gain Value at Single Frequency

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

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



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

Requirement with Limited Model Application and Additional Loop Openings

Create a requirement that specifies a minimum loop gain of –20 dB (0.01) at 100 rad/s. Set the Models and Openings properties to further configure the requirement's applicability.

```
Req = TuningGoal.MinLoopGain('InnerLoop',100,0.01);
Req.Models = [2 3];
Req.Openings = 'OuterLoop'
```

When tuning a control system that has a loop-opening location called 'InnerLoop', you can use Req as an input to looptune or systune. Setting the Openings property specifies that the loop gain at 'InnerLoop' is measured with the loop opened at a location in the control system identified by 'OuterLoop'. When tuning an array of control system models, setting the Models property restricts how the requirement is applied. In this example, the requirement applies only to the second and third models in an array.

See Also

slTunable.looptune | looptune | slTunable.systune | systune | viewSpec | evalSpec | TuningGoal.Gain | TuningGoal.LoopShape | TuningGoal.MaxLoopGain | TuningGoal.Margins | slTunable | sigma

How To

- "Using Design Requirement Objects"
- "Performance and Robustness Specifications for looptune"
- "Specifying Design Requirements for systune"
- "PID Tuning for Setpoint Tracking vs. Disturbance Rejection"

Purpose

Maximum loop gain constraint for control system tuning

Description

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

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

Construction

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

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

Input Arguments

location

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

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, `location` contains the names of one or more feedback loops in the `Location` property of `loopswitch` blocks in the control system model. For example, the following code creates a PI loop with a loop switch marking the plant input `'u'`.

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

You can use the string `'u'` to refer to the sensitivity at the plant input.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, `location` contains the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, `location` contains the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

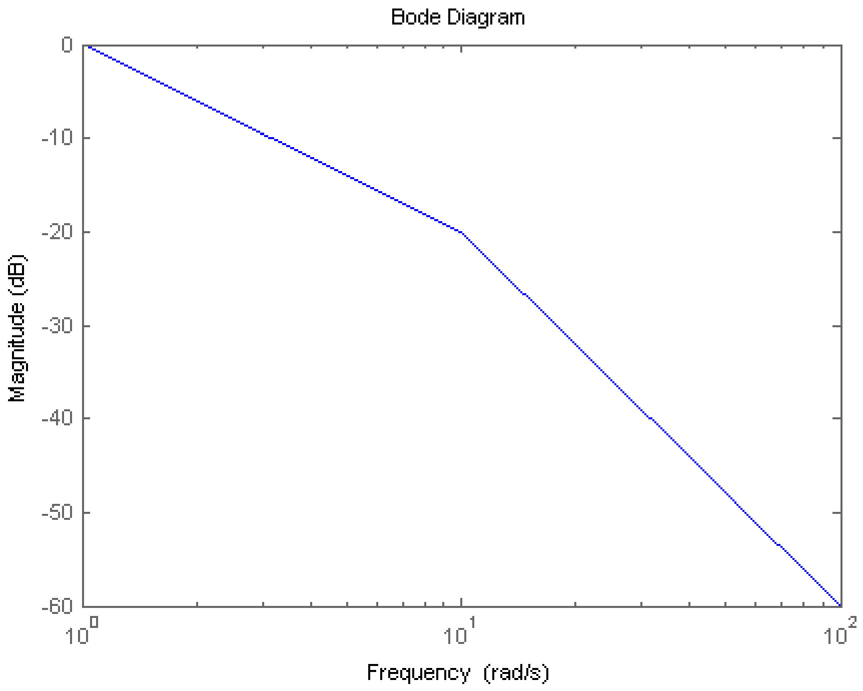
If `location` is a cell array of loop-opening locations, then the maximum gain requirement applies to the resulting MIMO loop.

loopgain

Maximum open-loop gain as a function of frequency.

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

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



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

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

fmax

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

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

gmax

Value of maximum gain occurring at `fmax`, specified as a scalar absolute value.

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

Properties

Location

Location at which maximum loop gain is constrained, specified as a string or cell array of strings. These strings identify one or more loop-opening locations in the control system to tune.

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

MaxGain

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

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

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

LoopScaling

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

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

Default: `'on'`

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.

  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

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

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

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

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

Examples

Maximum Loop Gain Requirement

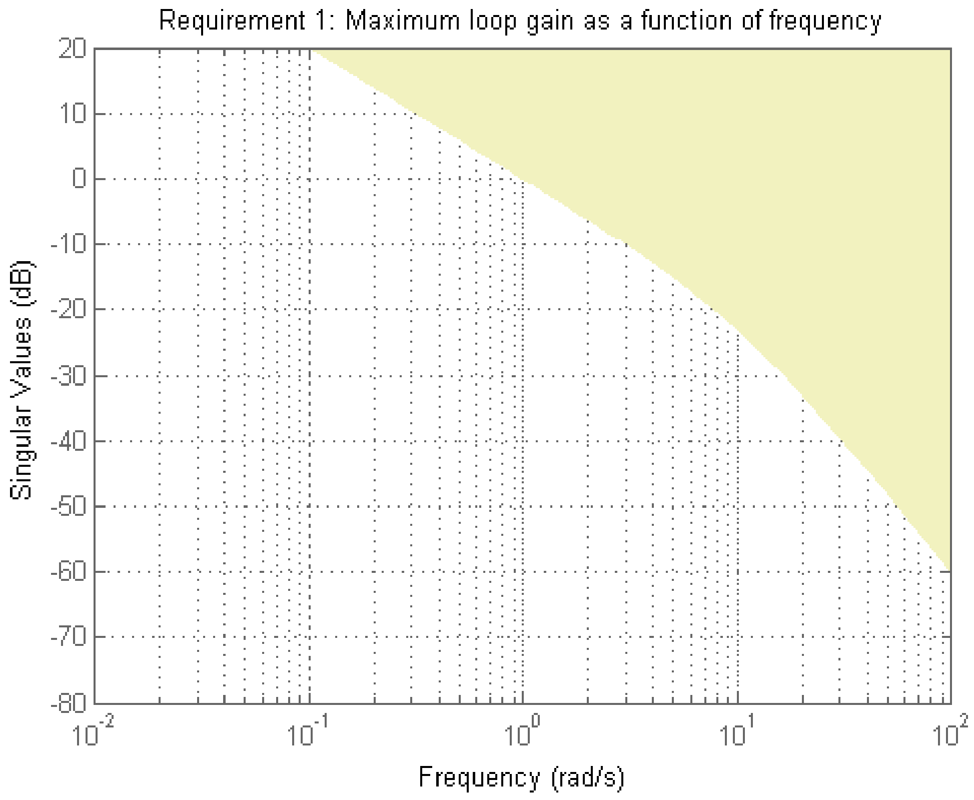
Create a requirement that limits the maximum open-loop gain of a feedback loop to a specified profile.

Suppose that you are tuning a control system that has a loop-opening location identified by `PILoop`. Limit the open-loop gain measured at that location to 1 (0 dB) at 1 rad/s, rolling off at a rate of -20 dB/dec up to 10 rad/s, and a rate of -40 dB/dec at higher frequencies. Use an `frd` model to sketch this gain profile.

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

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

```
viewSpec(Req)
```



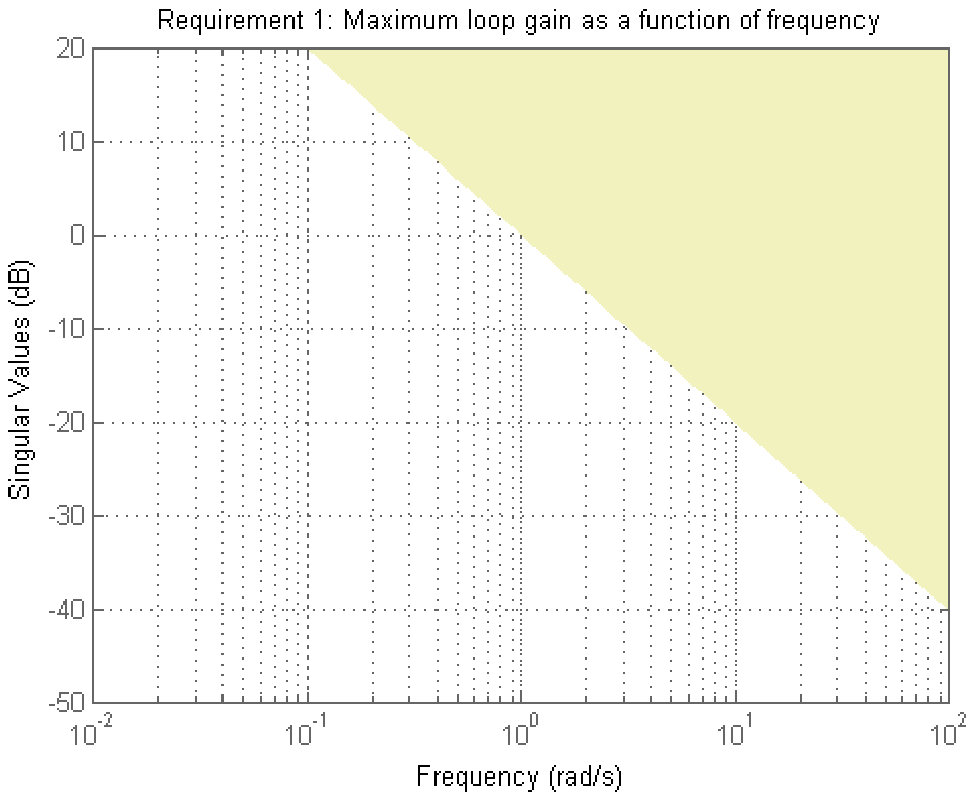
The yellow region indicates where the requirement is violated, except that gain values greater than 1 are not enforced. Therefore, this requirement only specifies minimum roll-off rates at frequencies above 1 rad/s.

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

Integral Loop Gain Specified as Gain Value at Single Frequency

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

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



viewSpec confirms that the requirement is correctly specified. You can use this requirement to tune a control system that has a loop-opening location identified as 'X'. Since loop gain values greater than 1 are ignored, this requirement specifies a rolloff of 20 dB/decade above 1 rad/s, with no restriction on loop gain below that frequency.

Requirement with Limited Model Application and Additional Loop Openings

Create a requirement that specifies a maximum loop gain of –20 dB (0.01) at 100 rad/s. SSet the Models and Openings properties to further configure the requirement's applicability.

```
Req = TuningGoal.MaxLoopGain('InnerLoop',100,0.01);
Req.Models = [2 3];
Req.Openings = 'OuterLoop'
```

When tuning a control system that has a loop-opening location called 'InnerLoop', you can use Req as an input to looptune or systune. Setting the Openings property specifies that the loop gain at 'InnerLoop' is measured with the loop opened at a location in the control system identified by 'OuterLoop'. When tuning an array of control system models, setting the Models property restricts how the requirement is applied. In this example, the requirement applies only to the second and third models in an array.

See Also

slTunable.looptune | looptune | slTunable.systune | systune | viewSpec | evalSpec | TuningGoal.Gain | TuningGoal.LoopShape | TuningGoal.MinLoopGain | TuningGoal.Margins | slTunable | sigma

How To

- "Using Design Requirement Objects"
- "Performance and Robustness Specifications for looptune"
- "Specifying Design Requirements for systune"
- "PID Tuning for Setpoint Tracking vs. Disturbance Rejection"
- "MIMO Control of Diesel Engine"

- “Tuning of a Two-Loop Autopilot”

Purpose

Overshoot constraint for control system tuning

Description

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

Construction

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

Input Arguments

inputname

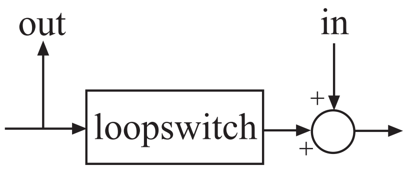
Input signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as input signals for the tuning requirement are as follows.

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

For example, if you are tuning a control system model, `T`, then `inputname` can be a string contained in `T.InputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `inputname` can include `X`.

- If you are using the requirement to tune a controller model, `C0` for a plant `G0`, using `looptune`, then `inputname` can include:
  - Any input of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `inputname` is a `loopswitch` channel of a generalized model, the input signal for the requirement is the implied input associated with the switch:



outputname

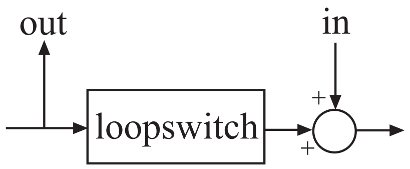
Output signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as output signals for the tuning requirement are as follows.

- If you are using the requirement to tune a Simulink model of a control system, then `outputname` can include:
  - Any model output
  - Any linearization output point in the model
  - Any signal identified as a `Controls`, `Measurements`, `Switches`, or `IOs` signal in an `slTunable` interface associated with the Simulink model

- If you are using the requirement to tune a generalized state-space model (`genss`) of a control system using `systune`, then `outputname` can include:
  - Any output of the control system model
  - Any `loopswitch` channel in the control system model

  For example, if you are tuning a control system model,`T`, then `outputname` can be a string contained in `T.OutputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `outputname` can include `X`.
- If you are using the requirement to tune a controller model, `C0`, for a plant, `G0`, using `looptune`, then `outputname` can include:
  - Any output of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `outputname` is a `loopswitch` channel of a generalized model, the output signal for the requirement is the implied output associated with the switch:



maxpercent

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

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

Properties

Input

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

Output

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

MaxOvershoot

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

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

$$f(x) = \left\| \frac{1}{PG} T(s,x) \right\|_{\infty}.$$

$T(s,x)$ is the closed-loop transfer function from `Input` to `Output`. PG is the peak gain based on second-order characteristics, given by:

$$PG = \begin{cases} 1, & \text{MaxOvershoot} < 4.32\% \\ \text{Inf}, & \text{MaxOvershoot} > 100\% \\ \frac{1}{2}\left(y + \frac{1}{y} \right), & \text{otherwise.} \end{cases}$$

$$y = -\frac{1}{\pi} \log\left(\frac{\text{MaxOvershoot}}{100} \right).$$

Examples

Overshoot Constraint

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

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

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

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

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

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

See Also

`slTunable.looptune` | `looptune` | `slTunable.systune` | `systune` | `viewSpec` | `evalSpec` | `TuningGoal.Gain` | `TuningGoal.Sensitivity` | `slTunable`

How To

- “Using Design Requirement Objects”
- “Performance and Robustness Specifications for looptune”
- “Specifying Design Requirements for systune”
- “Multi-Loop PID Control of a Robot Arm”

Purpose

Constraint on closed-loop dynamics

Description

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

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-64 of the object.

Construction

`Req = TuningGoal.Poles()` creates a default template for constraining the closed-loop pole locations. Modify the `MinDecay`, `MinDamping`, and `MaxFrequency` properties to shape the region where the closed-loop poles should lie. To limit the enforcement of this requirement to poles having natural frequency within a specified frequency range, set the `Focus` property. If you want to constrain the poles of the system with one or more feedback loops opened, set the `Openings` property. (See “Properties” on page 1-64.)

Properties

MinDecay

Minimum decay rate of closed-loop poles of tuned system, specified as a positive scalar value. When you tune the control system using this requirement, closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDecay`. This constraint helps ensure stable dynamics in the tuned system.

Change the value of this property to set a minimum decay rate other than the default 10^{-6}. For example, suppose `Req` is a `TuningGoal.Poles` requirement. Change the minimum decay rate to 0.001:

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

This constraint only applies to poles with natural frequencies in the range specified by the `Focus` property.

Default: `1e-6`

MinDamping

Minimum damping of closed-loop poles of tuned system, specified as a scalar value between 0 and 1. When you tune the control system using this requirement, closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDamping*|s|`.

Change the value of this property to set a minimum damping other than the default 10^{-6}. For example, suppose `Req` is a `TuningGoal.Poles` requirement. Change the minimum damping to 0.5:

```
Req.MinDamping = 0.5;
```

This constraint only applies to poles with natural frequencies in the range specified by the `Focus` property.

Default: `1e-6`

MaxFrequency

Maximum natural frequency of poles of tuned system, specified as a positive scalar value in the units of the control system model you are tuning (rad/`TimeUnit`). When you tune the control system using this requirement, closed-loop system poles that depend on the tunable parameters are constrained to satisfy `|s| < MaxFrequency`. This constraint prevents fast dynamics in the tunable component.

Change the value of this property to set maximum frequency other than the default 10^6. For example, suppose `Req` is a `TuningGoal.Poles` requirement. Change the maximum frequency to 1000:

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

This constraint only applies to poles with natural frequencies in the range specified by the `Focus` property.

Default: `1e6`

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

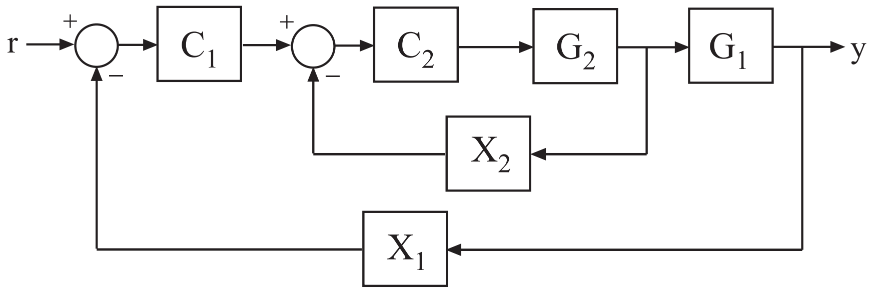
```
Req.Name = 'LoopReq';
```

Default: `[]`

Examples

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

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



Create a model of the system. To do so, specify and connect the numeric plant models, `G1` and `G2`, and the tunable controllers `C1` and `C2`. Also, create and connect the `loopswitch` blocks, `X1` and `X2`, which mark potential loop-opening sites.

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

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

```
Req = TuningGoal.Poles();
```

Specify that the constraint on the tuned system poles is applied with the outer loop open. Further restrict the poles of the inner loop to the region Re(s) < –0.1, |s| < 30.

```
Req.Openings = 'X1';
Req.MinDecay = 0.1;
Req.MaxFrequency = 30;
```

When you tune `T` using this requirement, the constraint applies only to the poles of the inner loop, evaluated with the outer loop open.

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

Algorithms

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

For the `TuningGoal.Poles` requirement, *f*(*x*) is given by:

$$f(x) = \max\left(f_{Abs}(x), f_{Rad}(x)\right),$$

where

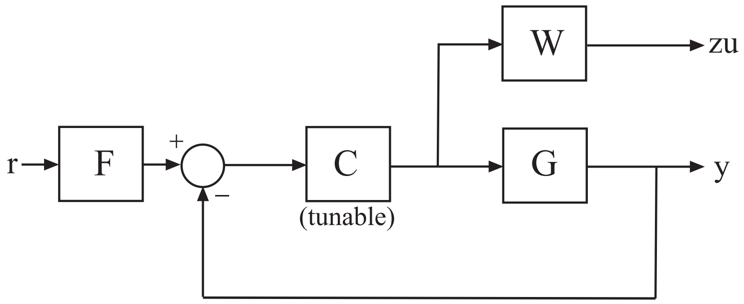
$$f_{Abs}(x) = 2 + \max_j \left(\frac{\mathrm{Re}\left(\lambda_j\right)}{\mathrm{MinDecay} + \mathrm{MinDamping}\left|\lambda_j\right|} \right)$$

$$f_{Rad}(x) = \max_j \left(\frac{\left|\lambda_j\right|}{\mathrm{MaxFrequency}} \right).$$

λ_j are the locations of closed-loop poles that depend on the tunable parameters x, with any specified loop openings taken into account.

Tips

- `TuningGoal.Poles` restricts the closed-loop dynamics of the tuned control system. To constrain the dynamics or ensure the stability of a single tunable component, use `TuningGoal.StableController`.
- The requirement only constrains dynamics that are in a feedback loop with the tuned elements of the control system. `TuningGoal.Poles` does not constrain dynamics that are independent of the tuned elements (such as weighting functions or open-loop dynamics). For example, consider the following control system, in which `C` is a tunable component.



  Tuning this control system with a `TuningGoal.Poles` requirement constrains the dynamics of the feedback loop containing `G` and `C`. However, the requirement does not constrain the dynamics of `F` or the weighting function, `W`.

See Also

`slTunable.looptune | looptune | systune | slTunable.systune | viewSpec | evalSpec | ltiblock.tf | ltiblock.ss | TuningGoal.StableController`

How To

- "Using Design Requirement Objects"
- "Performance and Robustness Specifications for looptune"
- "Specifying Design Requirements for systune"

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

TuningGoal.Rejection

Purpose

Disturbance rejection requirement for control system tuning

Description

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

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

Construction

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

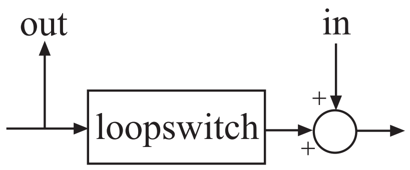
Input Arguments

distloc

Disturbance location, specified as a string or a cell array of strings for vector-valued signals.

- If you are using the requirement to tune a Simulink model of a control system, then `distloc` can include any signal identified as a `Controls`, `Measurements`, or `Switches` signal in an `slTunable` interface associated with the Simulink model.
- If you are using the requirement to tune a generalized state-space model (`genss`) with `systune` or `looptune`, then `inputname` can include any `loopswitch` channel in the model. For example, if you are tuning a control system model, `T`, that contains a `loopswitch` block with a switch channel named `X`,

then `distloc` can include `X`. The disturbance location is the implied input associated with the switch:

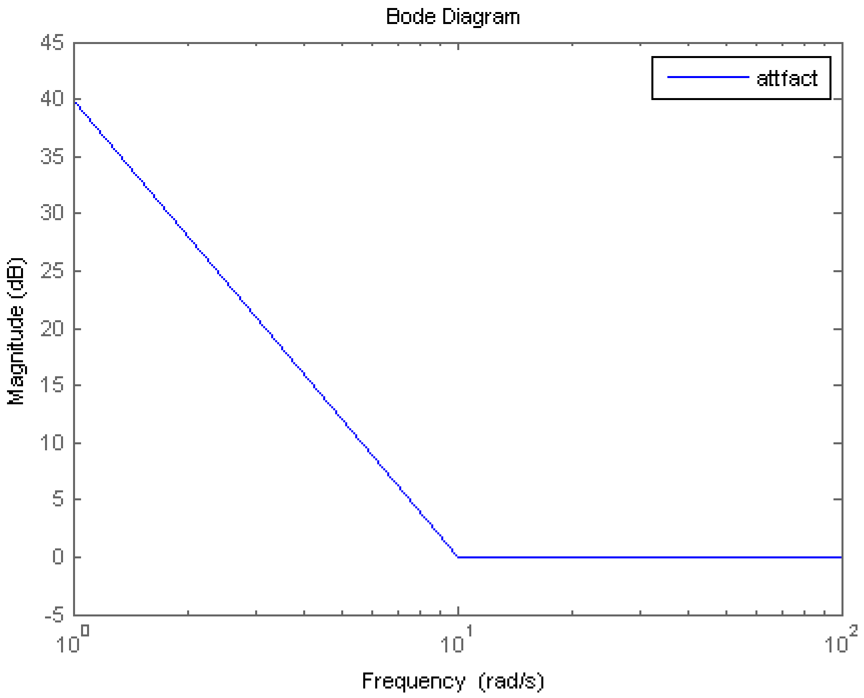


attfact

Attenuation factor as a function of frequency, specified as a numeric LTI model.

The `TuningGoal.Rejection` requirement constrains the minimum disturbance attenuation to the frequency-dependent value `attfact`. You can specify `attfact` as a smooth transfer function (`tf` , `zpk`, or `ss` model). Alternatively, you can specify a piecewise gain profile using a `frd` model. For example, the following code specifies an attenuation factor of 100 (40 dB) below 1 rad/s, gradually dropping to 1 (0 dB) past 10 rad/s, for a disturbance injected at `u`.

```
attfact = frd([100 100 1 1],[0 1 10 100]);
Req = TuningGoal.Rejection('u',attfact);
bodemag(attfact)
```



When you use an `frd` model to specify `attfact`, the gain profile is automatically mapped onto a `zpk` model. The magnitude of this `zpk` model approximates the desired gain profile. Use `viewSpec(Req)` to visualize the resulting attenuation profile.

Properties

DisturbanceInput

Disturbance location signal names, specified as a cell array of strings. The initial value of the `DisturbanceInput` property is set by the `distloc` input argument when you construct the requirement object.

MinAttenuation

Minimum disturbance attenuation as a function of frequency, expressed as a SISO `zpk` model.

The software automatically maps the `attfact` input argument to a `zpk` model. The magnitude of this `zpk` model approximates the desired attenuation factor and is stored in the `MinAttenuation` property. Use `viewSpec(Req)` to plot the magnitude of `MinAttenuation`.

LoopScaling

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

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

Default: `'on'`

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
    - A *control signal* is a signal that is an output of the controller model and an input of the plant model.

  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

When you tune a control system using a `TuningGoal` object to specify a tuning requirement, the requirement is converted into a normalized scalar value *f*(*x*). In this case, *x* is the vector of free (tunable) parameters in the control system. The parameter values are adjusted automatically to minimize *f*(*x*) or drive *f*(*x*) below 1 if the tuning requirement is a hard constraint.

For the `TuningGoal.Rejection` requirement, *f*(*x*) is given by:

$$f(x) = \max_{\omega \in \Omega} \left\| W(j\omega) S(j\omega, x) \right\|.$$

W(*jω*) is the rational transfer function of the `MinAttenuation` property. This transfer function's magnitude approximates minimum disturbance attenuation that you specify for the requirement. $S(j\omega,x)$ is the closed-loop sensitivity function measured at the disturbance location. Ω is the frequency interval over which the requirement is enforced, specified in the `Focus` property.

Examples

Constant Minimum Attenuation in Frequency Band

Create a tuning requirement that enforces a attenuation of at least a factor of 10 between 0 and 5 rad/s. The requirement applies to a disturbance entering a control system at a point identified as `'u'`.

```
Req = TuningGoal.Rejection('u',10);
Req.Name = 'Rejection spec';
Req.Focus = [0 5]
```

Frequency-Dependent Attenuation Profile

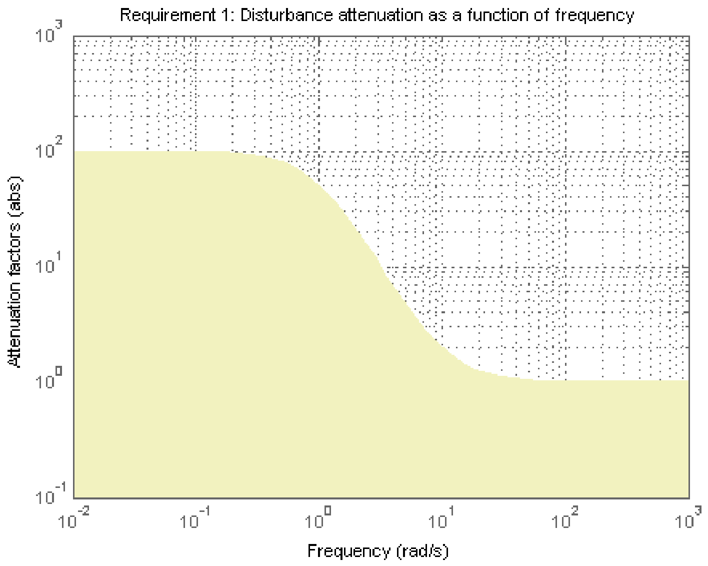
Create a tuning requirement that enforces an attenuation factor of at least 100 (40 dB) below 1 rad/s, gradually dropping to 1 (0 dB) past 10 rad/s. The requirement applies to a disturbance entering a control system at a point identified as `'u'`.

```
attfact = frd([100 100 1 1],[0 1 10 100]);
Req = TuningGoal.Rejection('u',attfact);
```

These commands use a `frd` model to specify the minimum attenuation profile as a function of frequency. The minimum attenuation of 100 below 1 rad/s, together with the minimum attenuation of 1 at the frequencies of 10 and 100 rad/s, specifies the desired rolloff of the requirement.

`attfact` is converted into a smooth function of frequency that approximates the piecewise specified requirement. Display the error requirement using `viewSpec`.

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

See Also

`slTunable.looptune` | `looptune` | `slTunable.systune` | `viewSpec` | `systune` | `TuningGoal.Tracking` | `TuningGoal.LoopShape` | `slTunable`

How To

- "Specifying Design Requirements for systune"
- "Performance and Robustness Specifications for looptune"
- "Using Design Requirement Objects"
- "Decoupling Controller for a Distillation Column"
- "Tuning of a Two-Loop Autopilot"

TuningGoal.Sensitivity

Purpose Sensitivity requirement for control system tuning

Description Use the `TuningGoal.Sensitivity` object to limit the sensitivity of a feedback loop to disturbances. Constrain the sensitivity to be smaller than one at frequencies where you need good disturbance rejection. Use this requirement for control system tuning with tuning commands such as `systune` or `looptune`.

Construction `Req = TuningGoal.Sensitivity(location,maxsens)` creates a tuning requirement for limiting the sensitivity to disturbances entering a feedback loop at the specified location. `maxsens` specifies the maximum sensitivity as a function of frequency. You can specify the maximum sensitivity profile as a smooth transfer function or sketch a piecewise error profile using an `frd` model.

Input Arguments

location

Location at which the sensitivity to disturbances is constrained, specified as a string or cell array of strings that identify one or more loop-opening locations in the control system to tune. What loop-opening locations are available depends on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, `location` contains the names of one or more feedback loops in the `Location` property of `loopswitch` blocks in the control system model. For example, the following code creates a PI loop with a loop switch marking the plant input `'u'`.

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

You can use the string `'u'` to refer to the sensitivity at the plant input.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, `location` contains the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, `location` contains the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

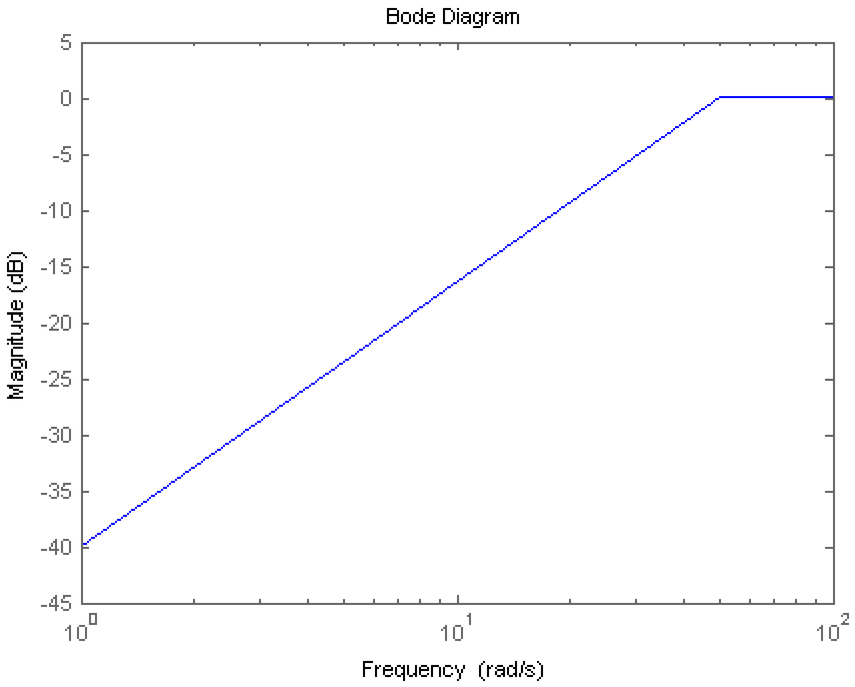
If `location` is a cell array of loop-opening locations, then the sensitivity requirement applies to the MIMO loop.

maxsens

Maximum sensitivity to disturbances as a function of frequency.

You can specify `maxsens` as a smooth SISO transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise gain profile using a `frd` model. For example, the following `frd` model specifies a maximum sensitivity of 0.01 (–40 dB) at 1 rad/s, increasing to 1 (0 dB) past 50 rad/s.

```
maxsens = frd([0.01 1 1],[1 50 100]);
bodemag(maxsens)
ylim([-45,5])
```



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

Properties

Location

Location at which the sensitivity to disturbances is constrained, specified as a string or cell array of strings that identify one or more loop-opening locations in the control system to tune.

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

MaxSensitivity

Maximum sensitivity as a function of frequency, specified as a SISO `zpk` model.

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

LoopScaling

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

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

Default: `'on'`

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.

  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

For the `TuningGoal.Sensitivity` requirement, *f*(*x*) is given by:

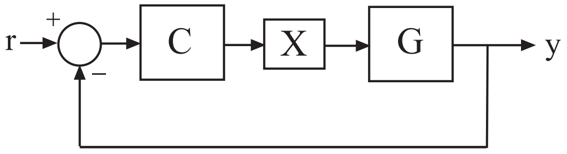
$$f(x) = \left\| \frac{1}{\text{MaxSensitivity}} S(s,x) \right\|_\infty .$$

$S(s,x)$ is the sensitivity function of the control system at `location`.

Examples

Disturbance Sensitivity at Plant Input

Create a requirement that limits the sensitivity to disturbance at the plant input of the following control system. The control system contains a `loopswitch` block at the plant input, having a switch location identified by `'X'`.

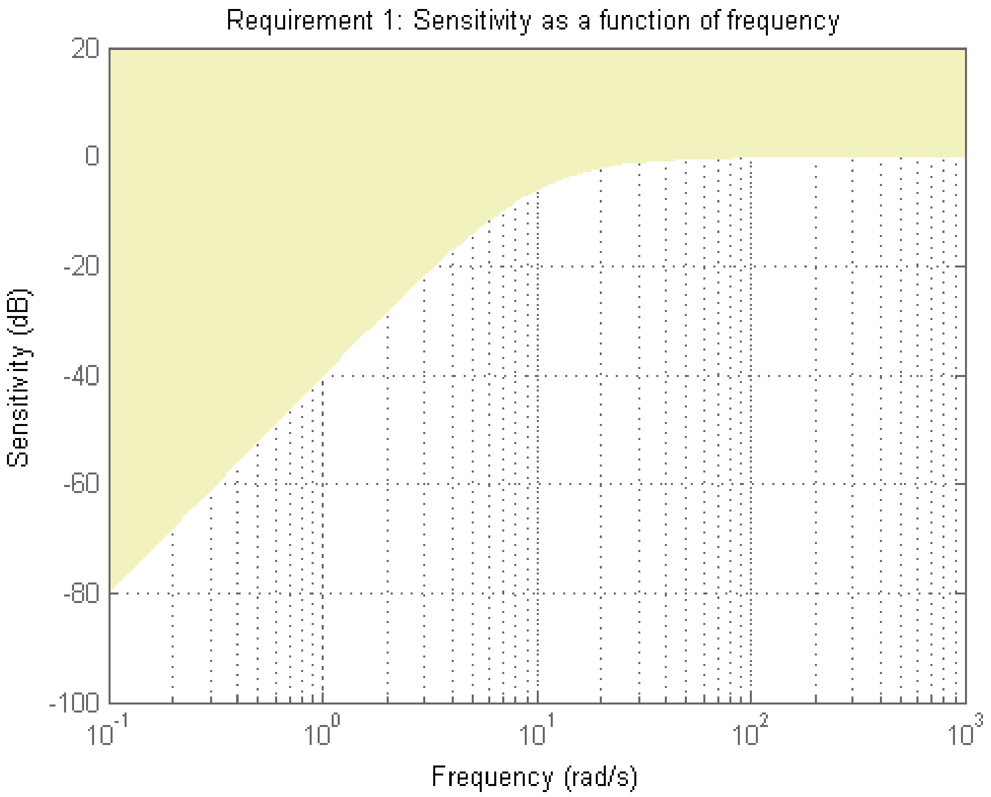


Specify a maximum sensitivity of 0.01 (–40 dB) at 1 rad/s, increasing to 1 (0 dB) past 10 rad/s. Use an `frd` model to sketch this target sensitivity.

```
maxsens = frd([0.01 1 1],[1 10 100]);
Req = TuningGoal.Sensitivity('X',maxsens);
```

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

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

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

Requirement with Limited Frequency Range and Model Application

Create a requirement that specifies a maximum sensitivity of 0.1 (10%) at frequencies below 5 rad/s. Configure the requirement to apply only to the second and third plant models.

```
Req = TuningGoal.Sensitivity('u',0.1);
Req.Focus = [0 5];
Req.Models = [2 3];
```

You can use `Req` as an input to `looptune` or `systune` when tuning a control system that has a loop-opening location called `'u'`. Setting the `Focus` property limits the application of the requirement to frequencies between 0 and 5 rad/s. Setting the `Models` property restricts application of the requirement to the second and third models in an array, when you use the requirement to tune an array of control system models.

See Also

`slTunable.looptune` | `looptune` | `slTunable.systune` | `systune` | `viewSpec` | `evalSpec` | `TuningGoal.Gain` | `TuningGoal.LoopShape` | `TuningGoal.Rejection` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain` | `slTunable`

How To

- “Using Design Requirement Objects”
- “Performance and Robustness Specifications for looptune”
- “Specifying Design Requirements for systune”

Purpose

Constraint on controller stability and fast dynamics for control system tuning

Description

Use the `TuningGoal.StableController` requirement object to specify a tuning requirement that constrains the dynamics of a tunable component in a control system model. Use this requirement for constraining `ltiblock.tf` or `ltiblock.ss` components for control system tuning with tuning commands such as `systune` or `looptune`. The `TuningGoal.StableController` requirement ensures that the tuned value of the tunable component is stable and free of fast dynamics.

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-89 of the object.

Construction

`Req = TuningGoal.StableController(blockID)` creates a tuning requirement. The tuning requirement specifies that the tunable component identified by `blockID` is stable and free of fast dynamics.

By default, the `TuningGoal.StableController` requirement imposes a minimum decay rate of 10^{-6} rad/s for poles of the specified tunable component. The requirement also limits the maximum magnitude of such poles to 10^{6} rad/s. To change these default values, set the `MinDecay` and `MaxDecay` properties, respectively. (See “Properties” on page 1-89.)

Input Arguments

blockID

Tunable component to constrain, specified as a string.

The string `blockID` must correspond to the `Name` property of a Control Design Block in the control system you are tuning.

Properties

Block

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

MinDecay

Minimum decay rate of poles of tunable component, specified as a positive scalar value in radians per second. When you tune the control system using this requirement, all poles of the tunable component are constrained to satisfy `Re(s) < -MinDecay`. This constraint helps ensure stable dynamics in the tunable component.

Change the value of this property to set a minimum decay rate other than the default 10^{-6}. For example, suppose `Req` is a `TuningGoal.StableContoller` requirement. Change the minimum decay rate to 0.001:

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

Default: `1e-6`

MaxFrequency

Maximum natural frequency of poles of tunable component, specified as a positive scalar value in radians per second. When you tune the control system using this requirement, all poles of the tunable component are constrained to satisfy `|s| < MaxFrequency`. This constraint prevents fast dynamics in the tunable component.

Change the value of this property to set maximum frequency other than the default 10^{6}. For example, suppose `Req` is a `TuningGoal.StableContoller` requirement. Change the maximum frequency to 1000:

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

Default: `1e6`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Examples

Constrain Dynamics of Tunable Transfer Function

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

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

Create a tunable component that represents the compensator.

```
C = ltiblock.tf('Compensator',2,2);
```

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

Create a tuning requirement to constrain the dynamics of the compensator block.

```
Req = TuningGoal.StableController('Compensator');
```

Set the minimum decay rate to 0.1 rad/s, and set the maximum frequency to 30 rad/s.

```
Req.MinDecay = 0.1;
Req.MaxFrequency = 30;
```

If you tune `T` using `systune` and the tuning requirement `Req`, the poles of the compensator block are constrained satisfy these values.

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

Algorithms

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

For the `TuningGoal.StableController` requirement, *f*(*x*) is given by:

$$f(x) = \max\left(f_{Abs}(x), f_{Rad}(x)\right),$$

E represents the locations of the poles of the specified tunable element.

β is a parameter that depends on `MinDecay`.

Tips

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

See Also

`slTunable.looptune | looptune | systune | slTunable.systune | viewSpec | evalSpec | ltiblock.tf | ltiblock.ss | TuningGoal.Poles`

How To

- “Using Design Requirement Objects”
- “Performance and Robustness Specifications for looptune”
- “Specifying Design Requirements for systune”
- “Models with Tunable Coefficients”

Purpose

Tracking requirement for control system tuning

Description

Use the `TuningGoal.Tracking` object to specify a tracking requirement that constrains a specified output to track a specified input. Use this requirement for control system tuning with tuning commands such as `systune` or `looptune`.

Construction

`Req = TuningGoal.Tracking(inputname,outputname,responsetime, dcerror)` creates a tuning requirement `Req`. This tuning requirement specifies that the output signal `outputname` tracks a step change in the reference signal `inputname`. `Req` also specifies a target `responsetime` and maximum steady-state error `dcerror`.

`Req = TuningGoal.Tracking(inputname,outputname,maxerror)` specifies the maximum relative error as a function of frequency. You can specify the target error profile (maximum gain from reference signal to tracking error signal) as a smooth transfer function. Alternatively, you can sketch a piecewise error profile using an `frd` model.

Input Arguments

inputname

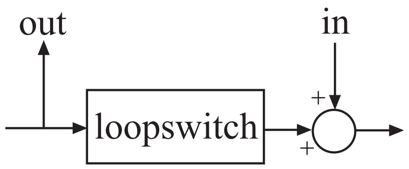
Input signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as input signals for the tuning requirement are as follows.

- If you are using the requirement to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input
  - Any linearization input point in the model
  - Any signal identified as a `Controls`, `Measurements`, `Switches`, or `IOs` signal in an `slTunable` interface associated with the Simulink model

- If you are using the requirement to tune a generalized state-space model (`genss`) of a control system using `systune`, then `inputname` can include:
  - Any input of the control system model
  - Any `loopswitch` channel in the control system model

  For example, if you are tuning a control system model, `T`, then `inputname` can be a string contained in `T.InputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `inputname` can include `X`.
- If you are using the requirement to tune a controller model, `C0` for a plant `G0`, using `looptune`, then `inputname` can include:
  - Any input of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `inputname` is a `loopswitch` channel of a generalized model, the input signal for the requirement is the implied input associated with the switch:



outputname

Output signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as output signals for the tuning requirement are as follows.

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

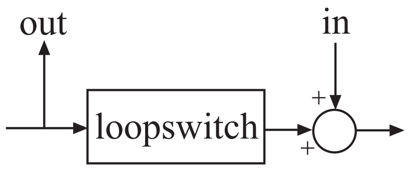
  - Any linearization output point in the model
  - Any signal identified as a `Controls`, `Measurements`, `Switches`, or `IOs` signal in an `slTunable` interface associated with the Simulink model

- If you are using the requirement to tune a generalized state-space model (`genss`) of a control system using `systune`, then `outputname` can include:
  - Any output of the control system model
  - Any `loopswitch` channel in the control system model

  For example, if you are tuning a control system model,`T`, then `outputname` can be a string contained in `T.OutputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `outputname` can include `X`.

- If you are using the requirement to tune a controller model, `C0`, for a plant, `G0`, using `looptune`, then `outputname` can include:
  - Any output of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `outputname` is a `loopswitch` channel of a generalized model, the output signal for the requirement is the implied output associated with the switch:



responsetime

Target response time.

Express the target response time in the time units of the models to be tuned with `looptune`. For example, when tuning a plant

GO and controller CO, if CO.TimeUnit and GO.TimeUnit are 'minutes', then express the target response time in minutes.

dcerror

Maximum steady-state fractional tracking error.

dcerror is a scalar value. If inputname or outputname are vector-valued, dcerror applies to all I/O pairs from inputname to outputname.

maxerror

Target tracking error profile as a function of frequency.

maxerror is the maximum gain from reference signal to tracking error signal. You can specify maxerror as a smooth transfer function (tf, zpk, or ss model). Alternatively, you can sketch a piecewise error profile using a frd model. When you do so, the software automatically maps the error profile to a zpk model. The magnitude of the zpk model. approximates the desired error profile. Use show(Req) to plot the magnitude of the zpk model.

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

Properties

Input

Reference signal names. String or cell array of strings specifying the names of the signals to be tracked, populated by the inputname argument.

Output

Output signal names. String or cell array of strings specifying the names of the signals that must track the reference signals, populated by the outputname argument.

MaxError

Maximum error as a function of frequency, expressed as a SISO zpk model. This property stores the maximum tracking error as a function of frequency (maximum gain from reference signal to tracking error signal).

The software automatically maps the `dcerror` or `maxerror` input arguments onto a `zpk` model. The magnitude of this `zpk` model approximates the desired error profile. The `zpk` model is stored in the `MaxError` property. Use `show(Req)` to plot the magnitude of `MaxError`.

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

For the `TuningGoal.Tracking` requirement, *f*(*x*) is given by:

$$f(x) = \left\| \frac{1}{\text{MaxError}} \left(T(s,x) - I \right) \right\|_\infty .$$

T(*s*,*x*) is the closed-loop transfer function from `Input` to `Output`. $\|\cdot\|_\infty$ denotes the H_∞ norm (see `norm`).

Examples

Tracking requirement with response time and maximum steady-state tracking error

Create a tracking requirement specifying that a signal `'theta'` track a signal `'theta_ref'`. The required response time is 2, in the time units of the control system you are tuning. The maximum steady-state error is 0.1%.

```
Req = TuningGoal.Tracking('theta_ref','theta',2,0.001);
```

Tracking requirement with maximum tracking error as a function of frequency

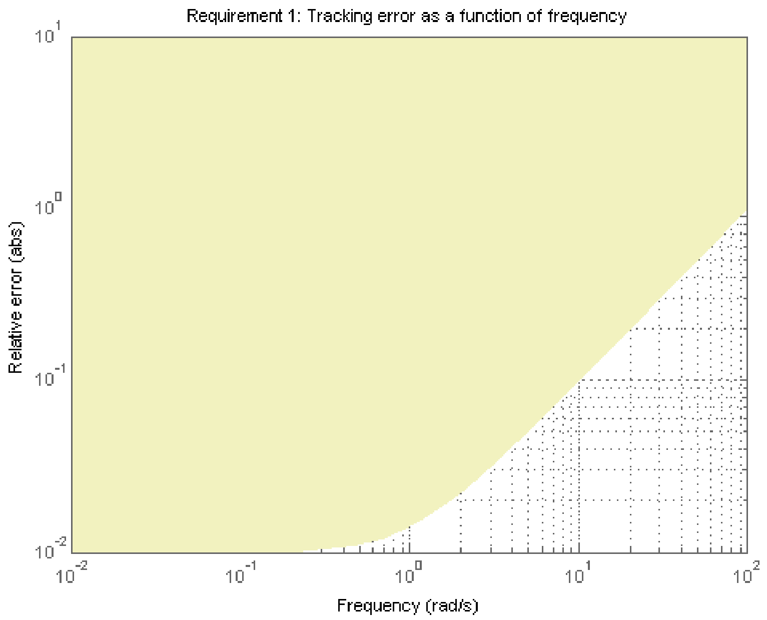
Create a tracking requirement specifying that a signal `'theta'` track a signal `'theta_ref'`. The maximum relative error is 0.01 (1%) in the frequency range `[0,1]`. The relative error increases to 1 (100%) at the frequency 100.

Use a `frd` model to specify the error profile as a function of frequency.

```
err = frd([0.01 0.01 1],[0 1 100]);
Req = TuningGoal.Tracking('theta_ref','theta',err);
```

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

```
viewSpec(Req)
```



The yellow region indicates where the requirement is violated.

See Also

`slTunable.looptune` | `looptune` | `slTunable.systune` | `systune` | `viewSpec` | `evalSpec` | `TuningGoal.Gain` | `TuningGoal.LoopShape` | `slTunable`

How To

- “Using Design Requirement Objects”
- “Performance and Robustness Specifications for looptune”
- “Specifying Design Requirements for systune”
- “Tuning Control Systems with SYSTUNE”
- “Tuning Control Systems in Simulink”
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”

- “Decoupling Controller for a Distillation Column”
- “Digital Control of Power Stage Voltage”
- “Tuning of a Two-Loop Autopilot”

Purpose

Noise amplification constraint for control system tuning

Description

Use the `TuningGoal.Variance` object to specify a tuning requirement that limits the noise amplification from specified inputs to outputs. The noise amplification is defined as either:

- The square root of the output variance, for a unit-variance white-noise input
- The root-mean-square of the output, for a unit-variance white-noise input
- The H_2 norm of the transfer function from the specified inputs to outputs, which equals the total energy of the impulse response

These definitions are different interpretations of the same quantity. `TuningGoal.Variance` imposes the same limit on these quantities.

You can use the `TuningGoal.Variance` requirement for control system tuning with tuning commands, such as `systune` or `looptune`. Specifying this requirement allows you to tune the system response to white-noise inputs. For stochastic inputs with a nonuniform spectrum (colored noise), use `TuningGoal.WeightedVariance` instead.

After you create a requirement object, you can further configure the tuning requirement by setting “Properties” on page 1-106 of the object.

Construction

`Req = TuningGoal.Variance(inputname,outputname,maxamp)` creates a tuning requirement. This tuning requirement limits the noise amplification of the transfer function from `inputname` to `outputname` to the scalar value `maxamp`.

When you tune a control system in discrete time, this requirement assumes that the physical plant and noise process are continuous. To ensure that continuous-time and discrete-time tuning give consistent results, `maxamp` is interpreted as a constraint on the continuous-time H_2 norm. If the plant and noise processes are truly discrete and you want to constrain the discrete-time H_2 norm instead, multiply `maxamp` by $\sqrt{T_s}$. T_s is the sampling time of the model you are tuning.

Input Arguments

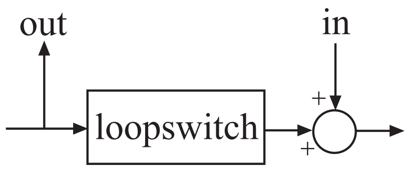
inputname

Input signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as input signals for the tuning requirement are as follows.

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

  For example, if you are tuning a control system model, `T`, then `inputname` can be a string contained in `T.InputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `inputname` can include `X`.
- If you are using the requirement to tune a controller model, `C0` for a plant `G0`, using `looptune`, then `inputname` can include:
  - Any input of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `inputname` is a `loopswitch` channel of a generalized model, the input signal for the requirement is the implied input associated with the switch:



outputname

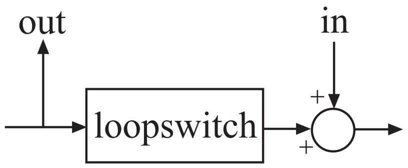
Output signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as output signals for the tuning requirement are as follows.

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

  For example, if you are tuning a control system model,`T`, then `outputname` can be a string contained in `T.OutputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `outputname` can include `X`.

- If you are using the requirement to tune a controller model, `C0`, for a plant, `G0`, using `looptune`, then `outputname` can include:
  - Any output of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `outputname` is a `loopswitch` channel of a generalized model, the output signal for the requirement is the implied output associated with the switch:



maxamp

Maximum noise amplification from `inputname` to `outputname`, specified as a positive scalar value. This value specifies the maximum value of the output variance at the signals specified in `outputname`, for unit-variance white noise signal at `inputname`. This value corresponds to the maximum H_2 norm from `inputname` to `outputname`.

When tuning in discrete time, to constrain the discrete-time H_2 norm, multiply `maxamp` by $\sqrt{T_s}$. T_s is the sampling time of the model you are tuning.

Properties

Input

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

Output

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

MaxAmplification

Maximum noise amplification, specified as a positive scalar value. This property specifies the maximum value of the output variance at the signals specified in `Output`, for unit-variance white noise signal at `Input`. This value corresponds to the maximum H_2 norm from `Input` to `Output`. The initial value of `MaxAmplification` is set by the `maxamp` input argument when you construct the requirement.

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

For the `TuningGoal.Variance` requirement, *f*(*x*) is given by:

$$f(x) = \left\| \frac{1}{\text{MaxAmplification}} T(s,x) \right\|_2 .$$

T(*s*,*x*) is the closed-loop transfer function from `Input` to `Output`. $\left\| \cdot \right\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, *f*(*x*) is given by:

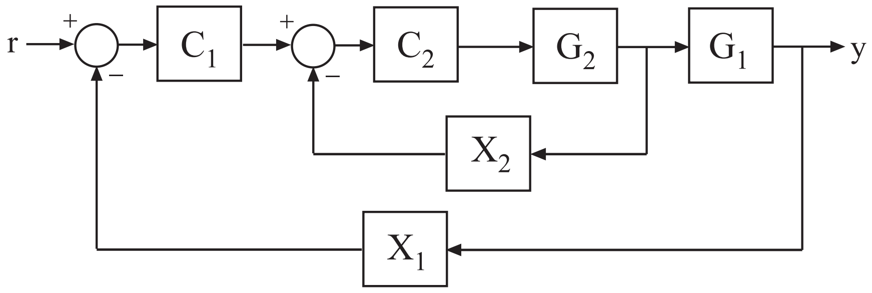
$$f(x) = \left\| \frac{1}{\text{MaxAmplification}\sqrt{T_s}} T(z,x) \right\|_2 .$$

T_s is the sampling time of the discrete-time transfer function *T*(*z*,*x*).

Examples

Constrain Noise Amplification Evaluated with a Loop Opening

Create a requirement that constrains the amplification of the variance from the switch `X2` to the output `y` of the following control system, measured with the outer loop open.



Create a model of the system. To do so, specify and connect the numeric plant models `G1` and `G2`, and the tunable controllers `C1` and `C2`. Also specify and connect the `loopswitch` blocks `X1` and `X2` that mark potential loop-opening sites.

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

Create a tuning requirement that constrains the noise amplification from the implicit input associated with the switch, `X2`, to the output `y`.

```
Req = TuningGoal.Variance('X2','y',0.1);
```

This constraint limits the amplification to a factor of 0.1.

Specify that the transfer function from `X2` to `y` is evaluated with the outer loop open when tuning to this constraint.

```
Req.Openings = {'X1'};
```

Use `systune` to tune the free parameters of `T` to meet the tuning requirement specified by `Req`. You can then validate the tuned control system against the requirement using `viewSpec(Req,T,Info)`.

See Also

`slTunable.looptune | looptune | systune | slTunable | slTunable.systune | viewSpec | evalSpec | norm | TuningGoal.WeightedVariance`

How To

- "Using Design Requirement Objects"
- "Performance and Robustness Specifications for looptune"
- "Specifying Design Requirements for systune"
- "Active Vibration Control in Three-Story Building"
- "Fault-Tolerant Control of a Passenger Jet"

Purpose

Frequency-weighted gain constraint for control system tuning

Description

Use the `TuningGoal.WeightedGain` object to specify a tuning requirement that limits the weighted gain from specified inputs to outputs. The weighted gain is the maximum across frequency of the gain from input to output, multiplied by weighting functions that you specify. You can use the `TuningGoal.WeightedGain` requirement for control system tuning with tuning commands such as `systune` or `looptune`.

After you create a requirement object, you can further configure the tuning requirement by setting "Properties" on page 1-115 of the object.

Construction

`Req = TuningGoal.WeightedGain(inputname,outputname,WL,WR)` creates a tuning requirement. This tuning requirement specifies that the closed-loop transfer function, $H(s)$, from the specified input to output meets the requirement:

$$||W_L(s)H(s)W_R(s)||_\infty < 1.$$

The notation $||\cdot||_\infty$ denotes the maximum gain across frequency (the H_∞ norm).

Input Arguments

inputname

Input signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as input signals for the tuning requirement are as follows.

- If you are using the requirement to tune a Simulink model of a control system, then `inputname` can include:
  - Any model input
  - Any linearization input point in the model

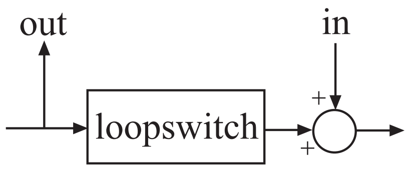
  - Any signal identified as a `Controls`, `Measurements`, `Switches`, or `IOs` signal in an `slTunable` interface associated with the Simulink model

- If you are using the requirement to tune a generalized state-space model (`genss`) of a control system using `systune`, then `inputname` can include:
  - Any input of the control system model
  - Any `loopswitch` channel in the control system model

  For example, if you are tuning a control system model, `T`, then `inputname` can be a string contained in `T.InputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `inputname` can include `X`.

- If you are using the requirement to tune a controller model, `C0` for a plant `G0`, using `looptune`, then `inputname` can include:
  - Any input of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `inputname` is a `loopswitch` channel of a generalized model, the input signal for the requirement is the implied input associated with the switch:



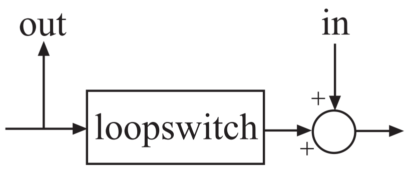
outputname

Output signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as output signals for the tuning requirement are as follows.

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

  For example, if you are tuning a control system model,`T`, then `outputname` can be a string contained in `T.OutputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `outputname` can include `X`.
- If you are using the requirement to tune a controller model, `C0`, for a plant, `G0`, using `looptune`, then `outputname` can include:
  - Any output of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `outputname` is a `loopswitch` channel of a generalized model, the output signal for the requirement is the implied output associated with the switch:



WL,WR

Frequency-weighting functions, specified as scalars or as SISO or MIMO numeric LTI models.

The functions `WL` and `WR` provide the weights for the tuning requirement. The tuning requirement ensures that the gain $H(s)$ from the specified input to output satisfies the inequality:

$$||WL(s)H(s)WR(s)||_\infty < 1.$$

`WL` provides the weighting for the output channels of $H(s)$, and `WR` provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model. For example:

```
WL = tf(1,[1 0.01]);
WR = 10;
```

If you specify MIMO weighting functions, then `inputname` and `outputname` must be vector signals. The dimensions of the vector signals must be such that the dimensions of $H(s)$ are commensurate with the dimensions of `WL` and `WR`. For example, if you specify `WR = diag([1 10])`, then `inputname` must include two signals. Scalar values, however, automatically expand to any input or output dimension.

A value of `WL = []` or `WR = []` is interpreted as the identity.

Properties

Input

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

Output

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

WL

Frequency-weighting function for the output channels of the transfer function *H*(*s*) to constrain, specified as a scalar, or as a SISO or MIMO numeric LTI model. The initial value of the `WL` property is set by the `WL` input argument when you construct the requirement object.

WR

Frequency-weighting function for the input channels of the transfer function to constrain, specified as a scalar or as a SISO or MIMO numeric LTI model. The initial value of the `WR` property is set by the `WR` input argument when you construct the requirement object.

Focus

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

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

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

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

Models

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

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

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

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

Default: `NaN`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.
- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location

can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.

  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.

- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Algorithms

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

For the `TuningGoal.WeightedGain` requirement, *f*(*x*) is given by:

$$f(x) = \left\| W_L T(s,x) W_R \right\|_\infty .$$

$T(s,x)$ is the closed-loop transfer function from `Input` to `Output`. $\|\cdot\|_\infty$ denotes the H_∞ norm (see `norm`).

Examples

Constrain Weighted Gain of Closed-Loop System

Create a tuning goal requirement that constrains the gain of a closed-loop SISO system from its input, *r*, to its output, *y*. Weight the gain at its input by a factor of 10 and at its output by the frequency-dependent weight 1/(*s* + 0.01).

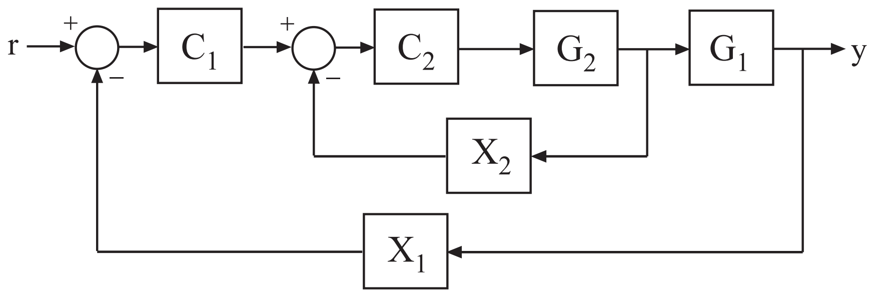
```
WL = tf(1,[1 0.01]);
WR = 10;
Req = TuningGoal.WeightedGain('r','y',WL,WR);
```

You can use the requirement `Req` with `systune` to tune the free parameters of any control system model that has an input signal named `'r'` and an output signal named `'y'`.

You can then use `viewSpec` to validate the tuned control system against the requirement.

Constrain Weighted Gain Evaluated with a Loop Opening

Create a requirement that constrains the gain of the outer loop of the following control system, evaluated with the inner loop open.



Create a model of the system. To do so, specify and connect the numeric plant models, `G1` and `G2`, the tunable controllers `C1` and `C2`. Also, create and connect the `loopswitch` blocks, `X1` and `X2`, that mark potential loop-opening sites.

```
G1 = tf(10,[1 10]);
G2 = tf([1 2],[1 0.2 10]);
```

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

Create a tuning requirement that constrains the gain of this system from r to y. Weight the gain at the output by $s/(s + 0.5)$.

```
WL = tf([1 0],[1 0.5]);
Req = TuningGoal.WeightedGain('r','y',WL,[]);
```

This requirement is equivalent to `Req = TuningGoal.Gain('r','y',1/WL)`. However, for MIMO systems, you can use `TuningGoal.WeightedGain` to create channel-specific weightings that cannot be expressed as `TuningGoal.Gain` requirements.

Specify that the transfer function from r to y be evaluated with the inner loop open for the purpose of tuning to this constraint.

```
Req.Openings = {'X2'};
```

Use `systune` to tune the free parameters of `T` to meet the tuning requirement specified by `Req`. You can then validate the tuned control system against the requirement using the command `viewSpec(Req,T,Info)`.

See Also

`slTunable.looptune` | `looptune` | `systune` | `slTunable.systune` | `slTunable` | `viewSpec` | `evalSpec`

How To

- "Using Design Requirement Objects"
- "Performance and Robustness Specifications for looptune"
- "Specifying Design Requirements for systune"

TuningGoal.WeightedVariance

Purpose

Frequency-weighted H_2 norm constraint for control system tuning

Description

Use the `TuningGoal.WeightedVariance` object to specify a tuning requirement that limits the weighted H_2 norm of the transfer function from specified inputs to outputs. The H_2 norm measures:

- The total energy of the impulse response, for deterministic inputs to the transfer function.
- The square root of the output variance for a unit-variance white-noise input, for stochastic inputs to the transfer function. Equivalently, the H_2 norm measures the root-mean-square of the output for such input.

You can use the `TuningGoal.WeightedVariance` requirement for control system tuning with tuning commands, such as `systune` or `looptune`. By specifying this requirement, you can tune the system response to stochastic inputs with a nonuniform spectrum such as colored noise or wind gusts. You can also use `TuningGoal.WeightedVariance` to specify LQG-like performance objectives.

After you create a requirement object, you can further configure the tuning requirement by setting "Properties" on page 1-125 of the object.

Construction

`Req = TuningGoal.Variance(inputname,outputname,WL,WR)` creates a tuning requirement `Req`. This tuning requirement specifies that the closed-loop transfer function $H(s)$ from the specified input to output meets the requirement:

$$||W_L(s)H(s)W_R(s)||_2 < 1.$$

The notation $||\cdot||_2$ denotes the H_2 norm.

When you are tuning a discrete-time system, `Req` imposes the following constraint:

$$\frac{1}{\sqrt{T_s}}\left\|W_L(z)T(z,x)W_R(z)\right\|_2 < 1.$$

The H_2 norm is scaled by the square root of the sampling time T_s to ensure consistent results with tuning in continuous time. To constrain the true discrete-time H_2 norm, multiply either W_L or W_R by $\sqrt{T_s}$.

Input Arguments

inputname

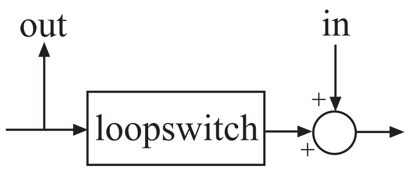
Input signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as input signals for the tuning requirement are as follows.

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

  For example, if you are tuning a control system model, `T`, then `inputname` can be a string contained in `T.InputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `inputname` can include `X`.
- If you are using the requirement to tune a controller model, `C0` for a plant `G0`, using `looptune`, then `inputname` can include:
  - Any input of `C0` or `G0`

- Any `loopswitch` channel in `C0` or `G0`

If `inputname` is a `loopswitch` channel of a generalized model, the input signal for the requirement is the implied input associated with the switch:



outputname

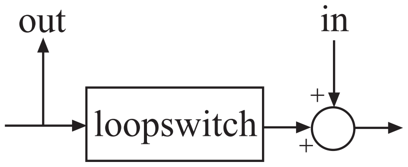
Output signal for requirement, specified as a string or a cell array of strings for vector-valued signals. The signals available to designate as output signals for the tuning requirement are as follows.

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

  For example, if you are tuning a control system model,`T`, then `outputname` can be a string contained in `T.OutputName`. Also, if `T` contains a `loopswitch` block with a switch channel `X`, then `outputname` can include `X`.

- If you are using the requirement to tune a controller model, `C0`, for a plant, `G0`, using `looptune`, then `outputname` can include:
  - Any output of `C0` or `G0`
  - Any `loopswitch` channel in `C0` or `G0`

If `outputname` is a `loopswitch` channel of a generalized model, the output signal for the requirement is the implied output associated with the switch:



WL,WR

Frequency-weighting functions, specified as scalars or as SISO or MIMO numeric LTI models.

The functions `WL` and `WR` provide the weights for the tuning requirement. The tuning requirement ensures that the gain $H(s)$ from the specified input to output satisfies the inequality:

$$||W_L(s)H(s)W_R(s)||_2 < 1.$$

`WL` provides the weighting for the output channels of $H(s)$, and `WR` provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model. For example:

```
WL = tf(1,[1 0.01]);
WR = 10;
```

If you specify MIMO weighting functions, then `inputname` and `outputname` must be vector signals. The dimensions of the vector signals must be such that the dimensions of $H(s)$ are commensurate with the dimensions of `WL` and `WR`. For example, if you specify `WR = diag([1 10])`, then `inputname` must include two signals. Scalar values, however, automatically expand to any input or output dimension.

When you are tuning a discrete-time system, `WL` and `WR` must be either scalars or discrete-time models having the same sampling time (`Ts`) as the model you are tuning.

A value of `WL = []` or `WR = []` is interpreted as the identity.

Properties

Input

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

Output

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

WL

Frequency-weighting function for the output channels of the transfer function $H(s)$ to constrain, specified as a scalar, or as a SISO or MIMO numeric LTI model. The initial value of the `WL` property is set by the `WL` input argument when you construct the requirement object.

WR

Frequency-weighting function for the input channels of the transfer function to constrain, specified as a scalar or as a SISO or MIMO numeric LTI model. The initial value of the `WR` property is set by the `WR` input argument when you construct the requirement object.

Models

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

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

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

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

Default: `NaN`

Openings

Feedback loops to open when evaluating the requirement, specified as a cell array of strings that identify loop-opening locations. The available loop-opening locations depend on what kind of system you are tuning:

- If you are tuning a control system specified as a `genss` model in MATLAB, a loop-opening location can be any feedback channel in a `loopswitch` block in the model. In this case, set `Openings` to a cell array containing the names of one or more loop-opening locations listed in the `Location` property of a `loopswitch` block in the control system model.

- If you are using `looptune` to tune a system that includes a plant model and controller model, a loop-opening location can be any control or measurement signal. In this case, set `Openings` to a cell array containing the names of one or more measurement or control signals.
  - A *control signal* is a signal that is an output of the controller model and an input of the plant model.
  - A *measurement signal* is a signal that is an output of the plant model and an input of the controller model.
- If you are tuning a Simulink model of a control system using the `slTunable` interface, a loop-opening location can be any `Controls`, `Measurements`, or `Switches` signal in the interface. In this case, set `Openings` to a cell array containing the names of one or more signals that you add to the `slTunable` interface. Use `slTunable.addControl`, `slTunable.addMeasurement`, or `slTunable.addSwitch` to add those signals.

All feedback loops are closed by default.

Default: `{}`

Name

Name of the requirement object, specified as a string.

For example, if `Req` is a requirement:

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

Default: `[]`

Algorithms

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

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

$$f(x) = \left\| W_L T(s,x) W_R \right\|_2 .$$

$T(s,x)$ is the closed-loop transfer function from `Input` to `Output`. $\left\| \cdot \right\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, $f(x)$ is given by:

$$f(x) = \frac{1}{\sqrt{T_s}} \left\| W_L(z) T(z,x) W_R(z) \right\|_2 .$$

T_s is the sampling time of the discrete-time transfer function $T(z,x)$.

Examples

Weighted Constraint on H_2 Norm

Create a constraint for a transfer function with one input, `r`, and two outputs, `e` and `y`, that limits the H_2 norm as follows:

$$\left\| \begin{matrix} \dfrac{1}{s+0.001} T_{re} \\ \dfrac{s}{0.001s+1} T_{ry} \end{matrix} \right\|_2 < 1.$$

T_{re} is the closed-loop transfer function from `r` to `e`, and T_{ry} is the closed-loop transfer function from `r` to `y`.

```
s = tf('s');
WL = blkdiag(1/(s+0.001),s/(0.001*s+1));
Req = TuningGoal.WeightedVariance('r',{'e','y'},WL,[]);
```

See Also

`slTunable.looptune` | `looptune` | `TuningGoal.Gain` | `TuningGoal.LoopShape` | `slTunable` | `slTunable.systune` | `norm` | `TuningGoal.Variance`

How To

- "Specifying Design Requirements for systune"

- "Performance and Robustness Specifications for looptune"
- "Using Design Requirement Objects"
- "Fault-Tolerant Control of a Passenger Jet"

2

Alphabetical List

actual2normalized

Purpose

Transform actual values to normalized values

Syntax

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

Description

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

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

The robustness margins computed 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 for all uncertain element values whose normalized distance from the nominal is less than 1.4.

Examples

Uncertain Real Parameter with Symmetric Range

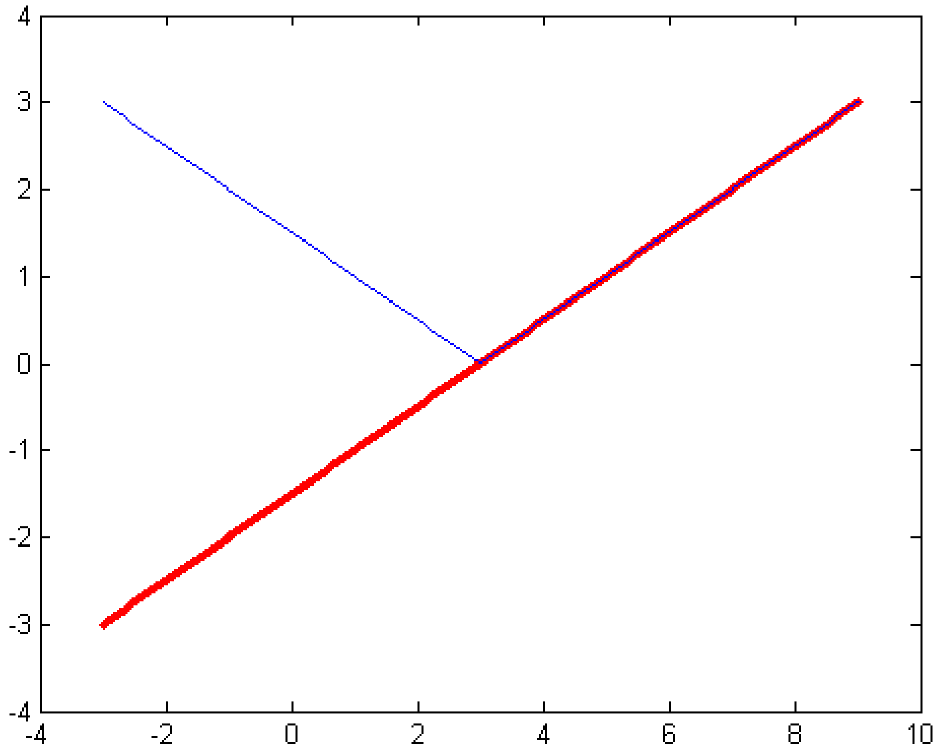
For uncertain real parameters whose range is symmetric about their nominal value, the normalized distance is intuitive, scaling linearly with the numerical difference from the uncertain real parameter's nominal value.

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

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


NV =

   -1.0000         0    1.0000


NV = actual2normalized(a,[2 4])


NV =

   -0.5000    0.5000


NV = actual2normalized(a,[0 6])


NV =

   -1.5000    1.5000
```

Plot the normalized values and normalized distance for several values.

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



Uncertain Real Parameter with Nonsymmetric Range

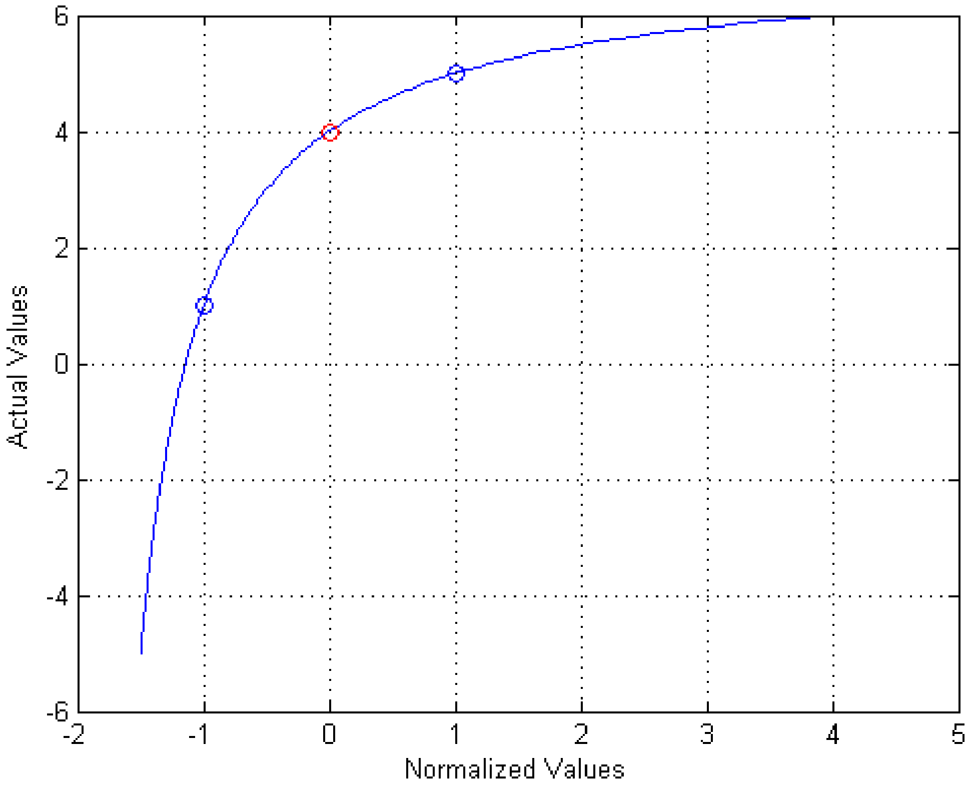
Create a nonsymmetric parameter. The end points are 1 normalized unit from nominal, and the nominal is 0 normalized units from nominal. Moreover, points inside the range are less than 1 unit from nominal, and points outside the range are greater than 1 unit from nominal. However, the relationship between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

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

```
NV = actual2normalized(au,[1 4 5])


NV =

    -1     0     1


NV = actual2normalized(au,[2 4.5])


NV =

   -0.8000    0.4000


NV = actual2normalized(au,[0 6])


NV =

   -1.1429    4.0000
```

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

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

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



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

Algorithms

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

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

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

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)`

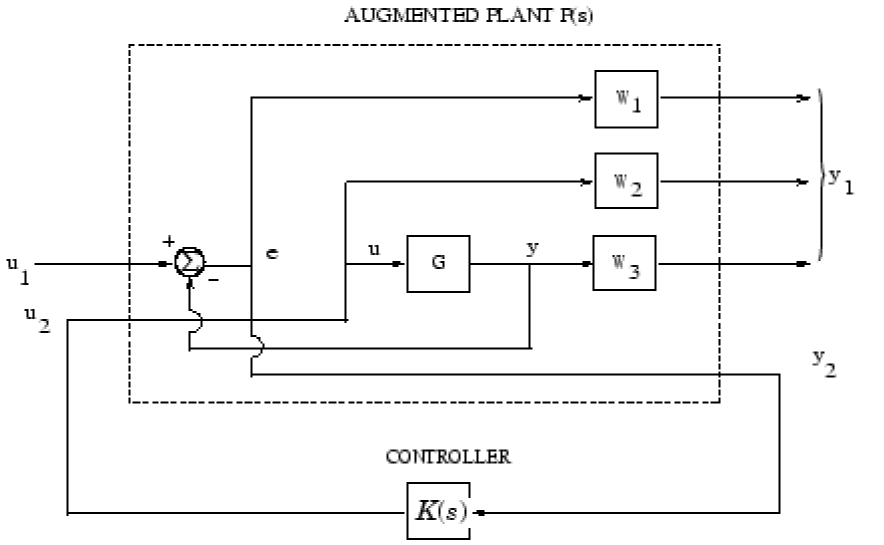
Description `P = AUGW(G,W1,W2,W3)` computes a state-space model of an augmented LTI plant $P(s)$ with weighting functions $W_1(s)$, $W_2(s)$, and $W_3(s)$ penalizing the error signal, control signal and output signal respectively (see block diagram) so that the closed-loop transfer function matrix is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

$$Ty_1u_1 \triangleq \begin{bmatrix} W_1S \\ W_2R \\ W_3T \end{bmatrix}$$

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 "Algorithms" on page 2-9 section below, but without the second row (without the row containing `W2`).

Algorithms

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

$$P(s) = \left[\begin{array}{c|c} W_1 & -W_1G \\ 0 & W_2 \\ 0 & W_3G \\ \hline I & -G \end{array}\right]$$

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

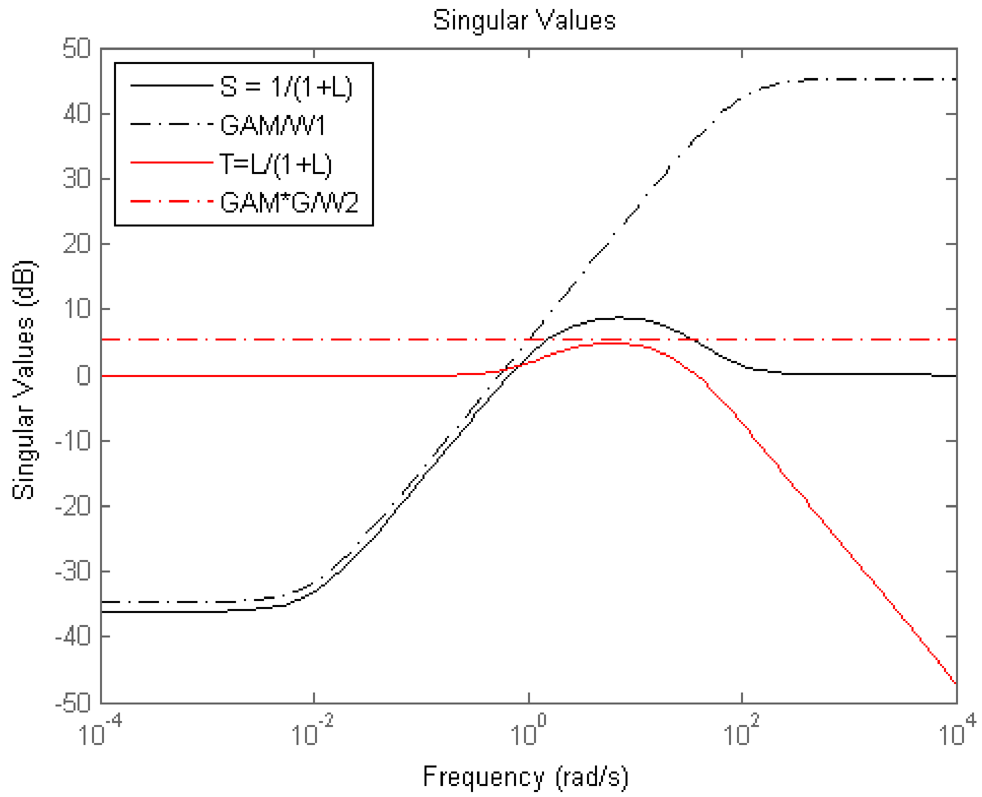
```
[r,c]=size(P);
```

```
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

Create Augmented Plant for H-Infinity Synthesis

```
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`

balancmr

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, $\sigma\iota$.

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

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

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

This table describes input arguments for `balancmr`.

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

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

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

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

Argument	Value	Description
`'MaxError'`	Real number or vector of different errors	Reduce to achieve H_∞ error. When present, *`'MaxError'`* 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 $\| G\text{-}GRED \|_\infty$) • `REDINFO.StabSV` (Hankel SV of stable part of G) • `REDINFO.UnstabSV` (Hankel SV of unstable part of G)

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

Algorithms

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

1 Find the SVD of the controllability and observability grammians

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

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

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

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

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

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,

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

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:

```
rng(1234,'twister');
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]);
for i = 1:4
    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 ($s \rightarrow z$) or $(s \rightarrow \tilde{s})$.

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

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: $s = \frac{z-1}{T}$

Bilinear Transform Types (Continued)

Method	Type of bilinear transform
	`AUG = `T, the sampling period.
`'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 continuous-time: $$s = \frac{\alpha\tilde{s}+\delta}{\gamma\tilde{s}+\beta}$$ `AUG = `$[\alpha\ \beta\ \gamma\ \delta]$.

Examples

Tustin Continuous s-Plane to Discrete z-Plane Transforms

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

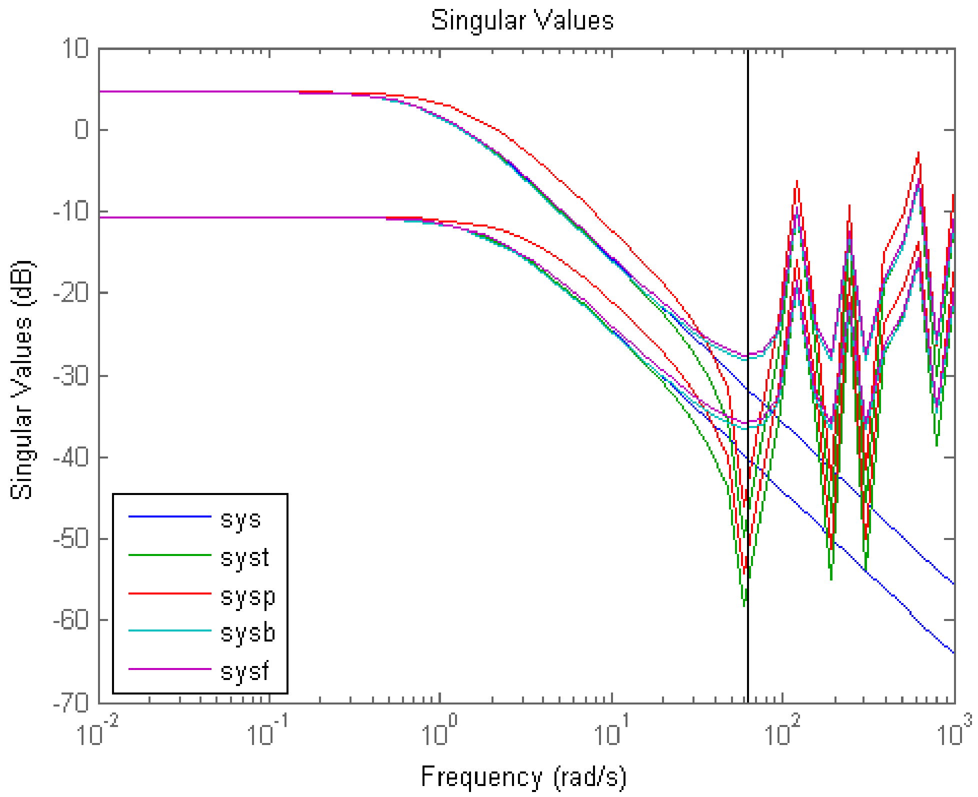
$$A = \begin{bmatrix} -1 & 1 \\ 0 & -2 \end{bmatrix},\ B = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},\ C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix};\ T_s = 0.05$$

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

```
A = [-1 1; 0 -2];
B = [1 0; 1 1];
C = [1 0; 0 1];
D = [0 0; 0 0];
sys = ss(A,B,C,D);                     % ANALOG
Ts = 0.05;  % 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
```

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

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



Bilinear continuous to continuous pole-shifting

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

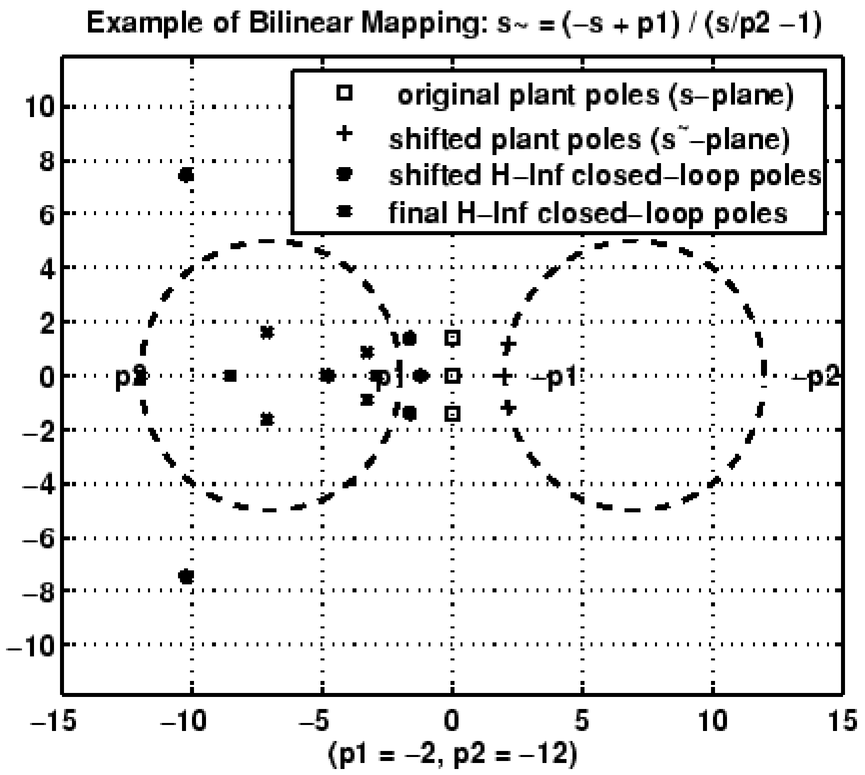
$$G(s) = \frac{1}{s^2(s^2+2)}$$

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

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

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

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



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

Algorithms

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

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

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`

bstmr

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* || `GRED`–1`(G-GRED)` || ∞ or *relative error* || `G`-–1`(G-GRED)` || ∞ for well-conditioned model reduction problems [1]:

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

This table describes input arguments for `bstmr`.

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

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

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

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

Algorithms

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

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

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

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

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

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

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

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

k

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

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

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

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

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

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

6 Finally,

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

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

Examples

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

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

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

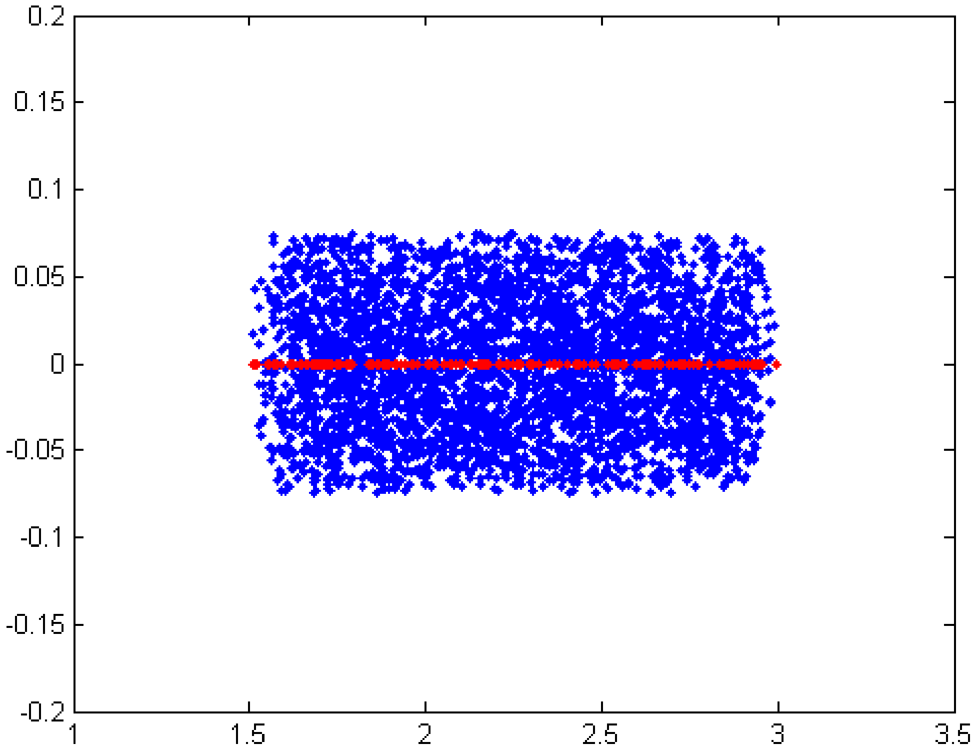
Complexified Uncertain Parameter

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

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

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

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



See Also `icomplexify` | `robuststab`

Related Examples

- Getting Reliable Estimates of Robustness Margins

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.

Algorithms

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

References

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

See Also

`dksyn` | `hinfsyn` | `mussv` | `robuststab` | `robustperf`

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

`gapmetric` | `wcmargin`

dcgainmr

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`

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

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

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

decinfo

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 $x_1, . . ., x_N$. 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 $x_1, . . ., x_N$. This is useful to specify matrix variables with atypical structures (see `lmivar`).

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

Examples

Example 1

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

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

If these variables are defined by

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

the decision variables in X and Y are given by

```
dX = decinfo(lmis,X)

dX =
 1  0  0
 0  1  0
 0  0  1

dY = decinfo(lmis,Y)

dY =
 2
 3
```

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

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

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

Example 2

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

```
setlmis([])
```

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

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

```
decinfo(lmis)

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

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

?> 1

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

X1 :

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

    *********

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

?> 0
```

See Also

`lmivar` | `mat2dec` | `dec2mat`

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`

dec2mat

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, . . ., 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`

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`

dellmi

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

$$\begin{aligned} A_1^T X_1 + X_1 A_1 + Q_1 &< 0 \\ A_2^T X_2 + X_2 A_2 + Q_2 &< 0 \\ A_3^T X_3 + X_3 A_3 + Q_3 &< 0 \end{aligned}$$

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

$$\begin{aligned} A_1^T X_1 + X_1 A_1 + Q_1 &< 0 \\ A_3^T X_3 + X_3 A_3 + Q_3 &< 0 \end{aligned}$$

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

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

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

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

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

See Also

`lmivar` | `setmvar` | `lmiinfo`

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`

dksyn

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 `dksynOptions` to create `opt.`

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

Field	Description
DR	Right D-scale, an ss object
GM	Offset G-scale, an ss object
GR	Right G-scale, an ss object
GFC	Center G-scale, an ss object
MussvBnds	Upper and lower μ bounds, an frd object
MussvInfo	Structure returned from mussv at each iteration.

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

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

Examples

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

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

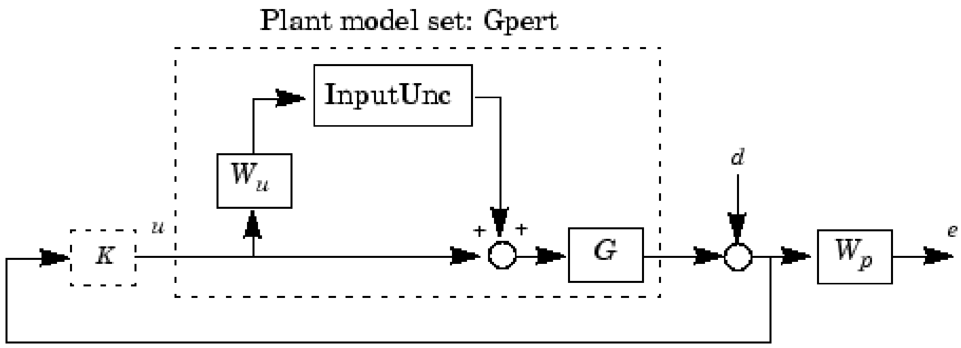
The model itself is uncertain. At low frequency, below 2 rad/s, it can vary up to 25% from its nominal value. Around 2 rad/s the percentage variation starts to increase and reaches 400% at approximately 32 rad/s. The percentage model uncertainty is represented by the weight `Wu` which corresponds to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

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

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

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

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



The sensitivity function, S, is defined as

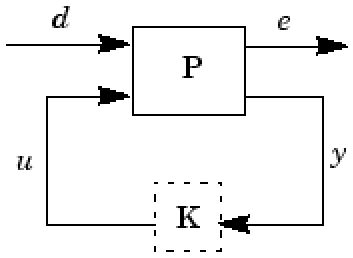
$$S = \frac{1}{1 + PK}$$

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

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

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

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

```
[rpmarg,rpmargunc,report,info] = robustperf(clp);
```

Enter `disp(report)` to display the report.

Algorithms

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

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

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

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

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

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

   and

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

   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 `dksynOptions` to `'on'`, the following information is displayed:

   a. The progress of the γ-iteration is displayed.

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

   c. You are given the option to change the frequency range. If you change it, all relevant frequency responses are automatically recomputed.
   d. You are given the option to rerun the H_∞ synthesis with a set of modified parameters if you set the `AutoIter` field in `dksynOptions` to `'off'`. This is convenient if, for instance, the bisection tolerance was too large, or if `maximum gamma value` was too small.

3. The structured singular value of the closed-loop system is calculated and plotted.
4. An iteration summary is displayed, showing all the controller order, as well as the peak value of μ of the closed-loop frequency responses.
5. The choice of stopping or performing another iteration is given.

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

Interactive Fitting of D-Scalings

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

```
Enter Choice (return for list):
Choices:
nd          Move to Next D-Scaling
nb          Move to Next D-Block

i           Increment Fit Order
d           Decrement Fit Order
apf         Auto-PreFit
```

```
mx 3        Change Max-Order to 3
at 1.01     Change Auto-PreFit tol to 1.01
0           Fit with zeroth order
2           Fit with second order
n           Fit with n'th order
e           Exit with Current Fittings
s           See Status
```

- `nd` and `nb` allow you to move from one *D*-scale data to another. `nd` moves to the next scaling, whereas `nb` moves to the next scaling block. For scalar *D*-scalings, these are identical operations, but for problems with full *D*-scalings, (perturbations of the form 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

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

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

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

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

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

Tutorials

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

See Also

`dksynOptions` | `h2syn` | `hinfsyn` | `mktito` | `mussv` | `robuststab` | `robustperf` | `wcgain` | `wcsens` | `wcmargin`

Purpose

Set options for dksyn

Syntax

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

Description

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

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

Input Arguments

Name-Value Pair Arguments

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

`dksynOptions` takes the following `Name` arguments:

'FrequencyVector'

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

Default: `[]`

'InitialController'

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

Default: `[]`

'AutoIter'

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

Default: `'on'`

'DisplayWhileAutoIter'

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

Default: `'off'`

'StartingIterationNumber'

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

Default: 1

'NumberOfAutoIterations'

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

Default: 10

'MixedMU'

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

Default: `'off'`

'AutoScalingOrder'

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

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

'AutoIterSmartTerminate'

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

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

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

and

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

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

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

Default: `'on'`

'AutoIterSmartTerminateTol'

Tolerance for `AutoIterSmartTerminate` mode.

Default: `0.005`

Output Arguments

options

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

Examples

Create Options Set for `dksyn`

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

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

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

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

See Also

`| dksyn`

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

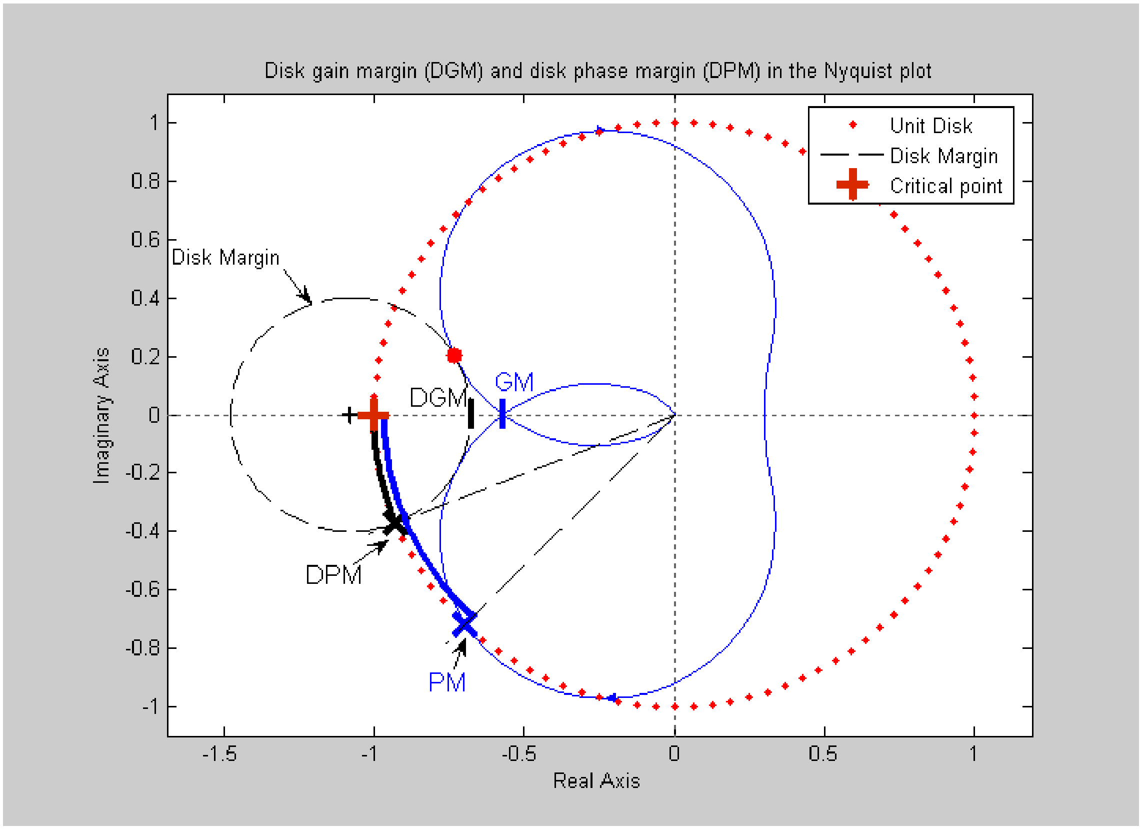
When you call `dmplot` (without an argument), the resulting plot shows a comparison of a disk margin analysis with the classical notations of gain and phase margins. The Nyquist plot is of the loop transfer function L(s)

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

```
dmplot
```

```
This figure shows a comparison of a disk margin analysis
with the classical notations of gain and phase margins.
```

```
The Nyquist plot is of the loop transfer function

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

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

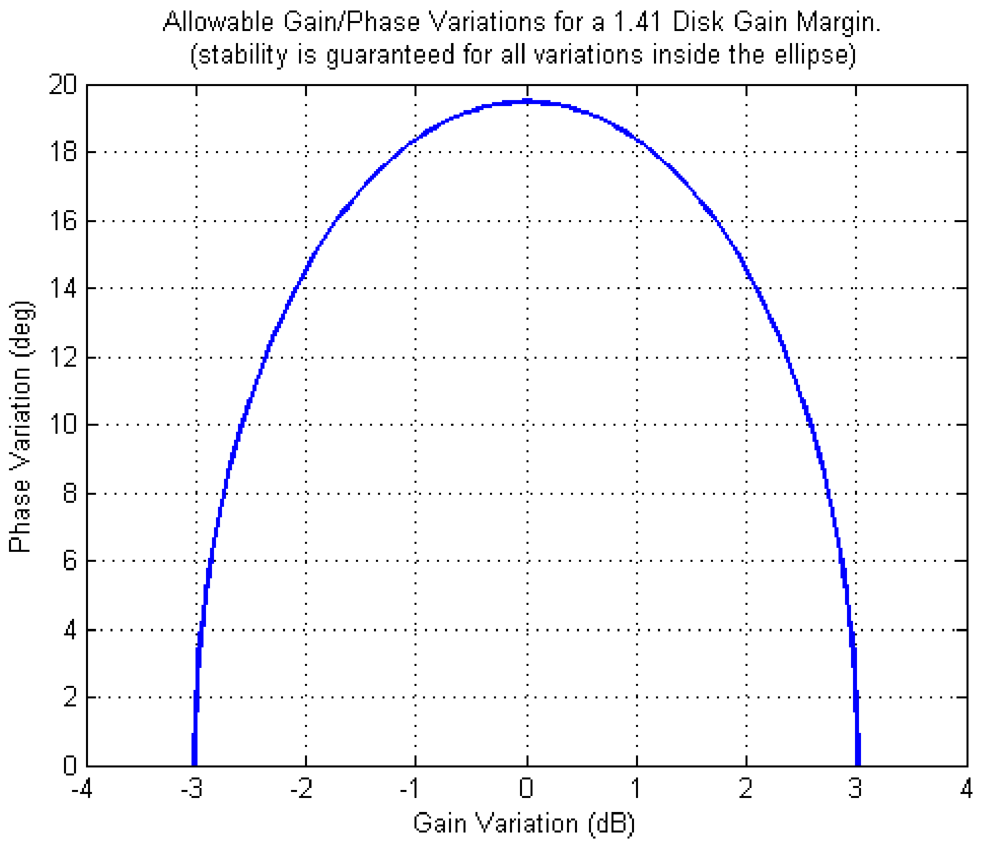


- The Nyquist plot of L corresponds to the blue line.
- The unit disk corresponds to the dotted red line.
- GM and PM indicate the location of the classical gain and phase margins for the system L.
- DGM and DPM correspond to the disk gain and phase margins, respectively. The disk margins provide a lower 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

`wcmargin`

drawmag

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. Be cautious when using this option to avoid unintended changes to variables.

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 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 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

Evaluate tuning requirements for tuned control system

Syntax

```
[Hspec,fval] = evalSpec(Req,T)
[Hspec,fval] = evalSpec(Req,T,Info)
```

Description

`[Hspec,fval] = evalSpec(Req,T)` returns the normalized value, `fval`, of a tuning requirement evaluated for a tuned control system `T`. The `evalSpec` command also returns the transfer function, `Hspec`, used to compute this value.

`[Hspec,fval] = evalSpec(Req,T,Info)` uses the `Info` structure returned by `systune` for correct scaling of MIMO open-loop requirements, such as loop shapes and stability margins.

Input Arguments

Req - Tuning requirement to evaluate

`TuningGoal` requirement object | vector of `TuningGoal` objects

Tuning requirement to evaluate, specified as a `TuningGoal` requirement object or vector of `TuningGoal` objects. `TuningGoal` requirement objects include:

- `TuningGoal.Tracking`
- `TuningGoal.Gain`
- `TuningGoal.WeightedGain`
- `TuningGoal.Variance`
- `TuningGoal.WeightedVariance`
- `TuningGoal.LoopShape`
- `TuningGoal.Margins`
- `TuningGoal.Poles`
- `TuningGoal.StableController`

T - Tuned control system

generalized state-space model | `slTunable` interface object

Tuned control system, specified as a generalized state-space (`genss`) model or an `slTunable` interface to a Simulink model. `T` is typically the result of using the tuning requirement to tune control system parameters with `systune`.

Example: `[T,fSoft,gHard,Info] = systune(T0,SoftReq,HardReq)`, where `T0` is a tunable `genss` model

Example: `[T,fSoft,gHard,Info] = systune(ST0,SoftReq,HardReq)`, where `ST0` is a `slTunable` interface object

Info - System information

data structure returned by `systune`

System information, specified as the data structure returned by `systune` when you use that command to tune a control system. Use `Info` when validating tuned MIMO systems. Doing so ensures that `viewSpec` correctly scales open-loop requirements such as loop shapes and stability margins.

Output Arguments

Hspec - transfer function associated with requirement

State-space model

Transfer function associated with the tuning requirement, returned as a state-space (`ss`) model. `evalSpec` uses `Hspec` to compute the evaluated requirement, `fval`.

For example, suppose `Req` is a `TuningGoal` gain requirement that limits the gain, $H(s)$, between some specified input and output to the gain profile, $w(s)$. In that case, `Hspec` is given by:

$$Hspec(s) = \frac{1}{w(s)} H(s).$$

`fval` is the peak gain of `Hspec`. If $H(s)$ satisfies the tuning requirement, `fval <= 1`.

fval - Normalized value of tuning requirement

positive scalar

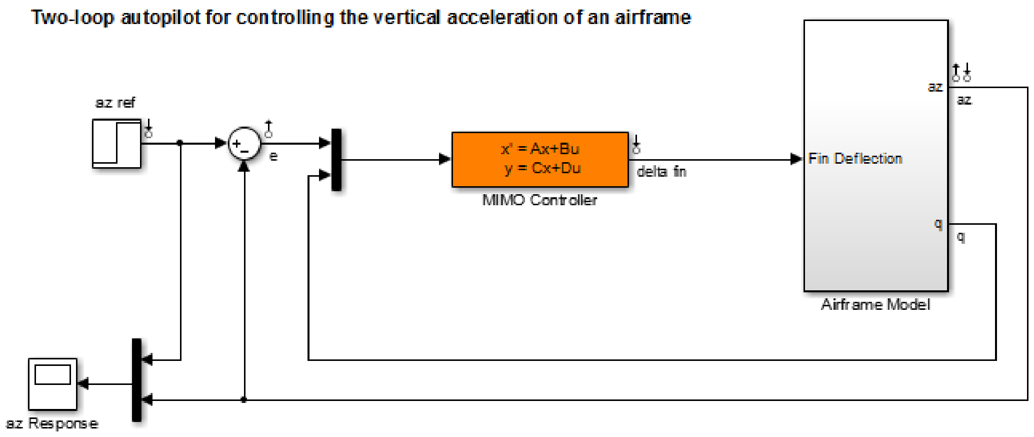
Normalized value of tuning requirement, returned as a positive scalar. The normalized value is a measure of how closely the requirement is met in the tuned system. The tuning requirement is satisfied if `fval < 1`. For information about how each type of `TuningGoal` requirement is converted into a normalized value, see the `TuningGoal` requirement objects.

Examples

Evaluate Requirements for Tuned System

Tune a control system with `systune`, and evaluate the tuning requirements with `evalSpec`.

Create tracking, roll-off, stability margin, and disturbance rejection requirements for tuning the following control system.



```
Req1 = TuningGoal.Tracking('az ref','az',1);
Req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0]));
Req3 = TuningGoal.Margins('delta fin',7,45);
MaxGain = frd([2 200 200],[0.02 2 200]);
Req4 = TuningGoal.Gain('delta fin','az',MaxGain);
```

Tune the model using these tuning requirements.

```
open_system('rct_airframe2')

ST0 = slTunable('rct_airframe2','MIMO Controller');
addControl(ST0,'delta fin');

rng('default');
[ST1,fSoft,~,Info] = systune(ST0,[Req1,Req2,Req3,Req4]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 55
```

`ST1` is a tuned version of the `slTunable` interface to the control system. `ST1` contains the tuned values of the tunable parameters of the MIMO controller in the model.

Evaluate the margin requirement for the tuned system.

```
[hspec,fval] = evalSpec(Req3,ST1,Info);
fval
```

```
fval =

    0.5140
```

The normalized value of the requirement is below 1, indicating that the tuned system satisfies the margin requirement. For more information about how the normalized value of this requirement is calculated, see the `TuningGoal.Margins` reference page.

Evaluate the tracking requirement for the tuned system.

```
[hspec,fval] = evalSpec(Req1,ST1,Info);
fval
```

```
fval =

    1.1327
```

The tracking requirement is nearly met, but the value exceeds 1, indicating a small violation. To further assess the violation, you can use `viewSpec` to examine the requirement against the tuned control system as a function of frequency.

See Also

`systune` | `genss` | `viewSpecslTunable.systune` | `slTunable` | `TuningGoal.Tracking` | `TuningGoal.Sensitivity` | `TuningGoal.Overshoot` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain` | `TuningGoal.Gain` | `TuningGoal.Margins` | `TuningGoal.WeightedGain` | `TuningGoal.Variance` | `TuningGoal.WeightedVariance` | `TuningGoal.LoopShape` | `TuningGoal.Poles` | `TuningGoal.StableController` |

Concepts

- “Generalized Models”
- “Performance and Robustness Specifications for looptune”

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

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

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

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

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 $x = (x_1, . . ., x_N)$ 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 2-272 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

$$A_1^T P + PA_1 < 0 \tag{2-4}$$

$$A_2^T P + PA_2 < 0 \tag{2-5}$$

$$A_3^T P + PA_3 < 0 \tag{2-6}$$

with data

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

This problem arises when studying the quadratic stability of the polytope of matrices 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`).

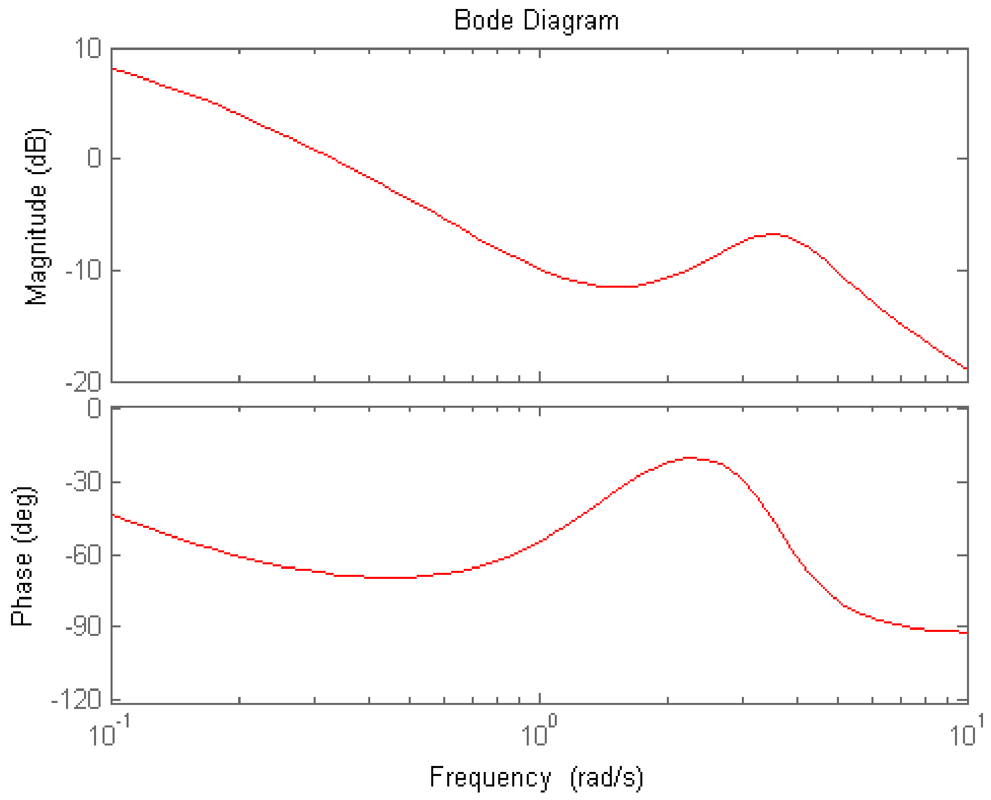
Examples

Fit D-scale Data

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

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

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

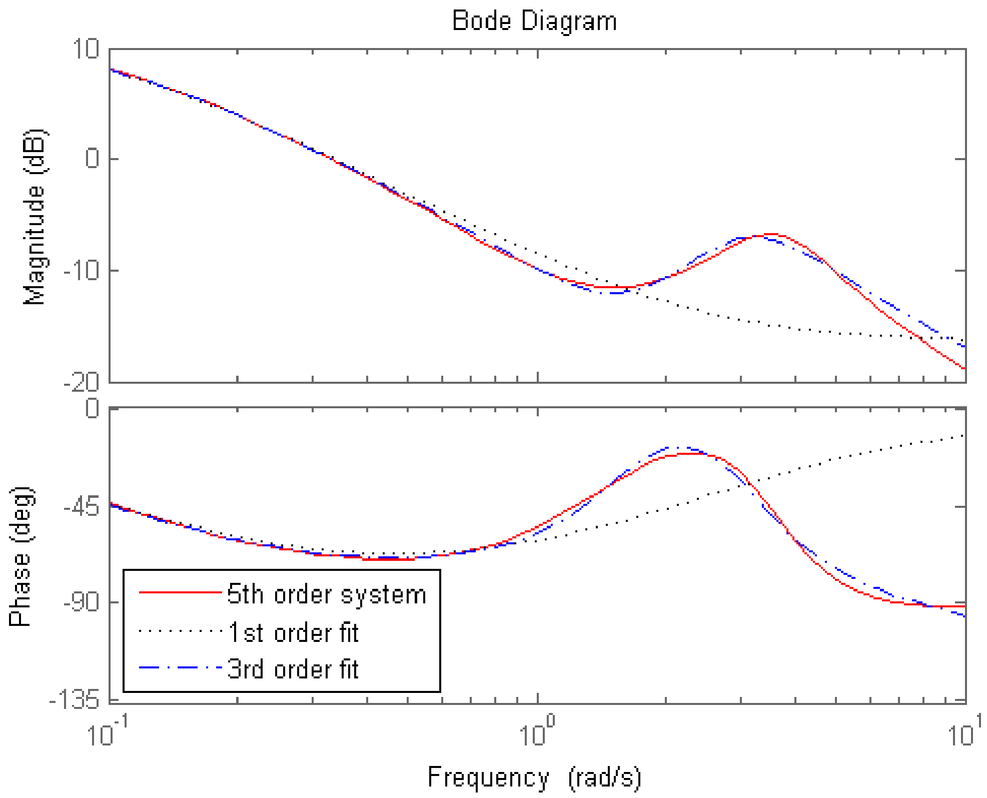


You can try to fit the frequency response *D*-scale data `sysg` with a first-order system, `b1`. Similarly, you can fit the *D*-scale data with a third-order system, `b3`.

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

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

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



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`

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

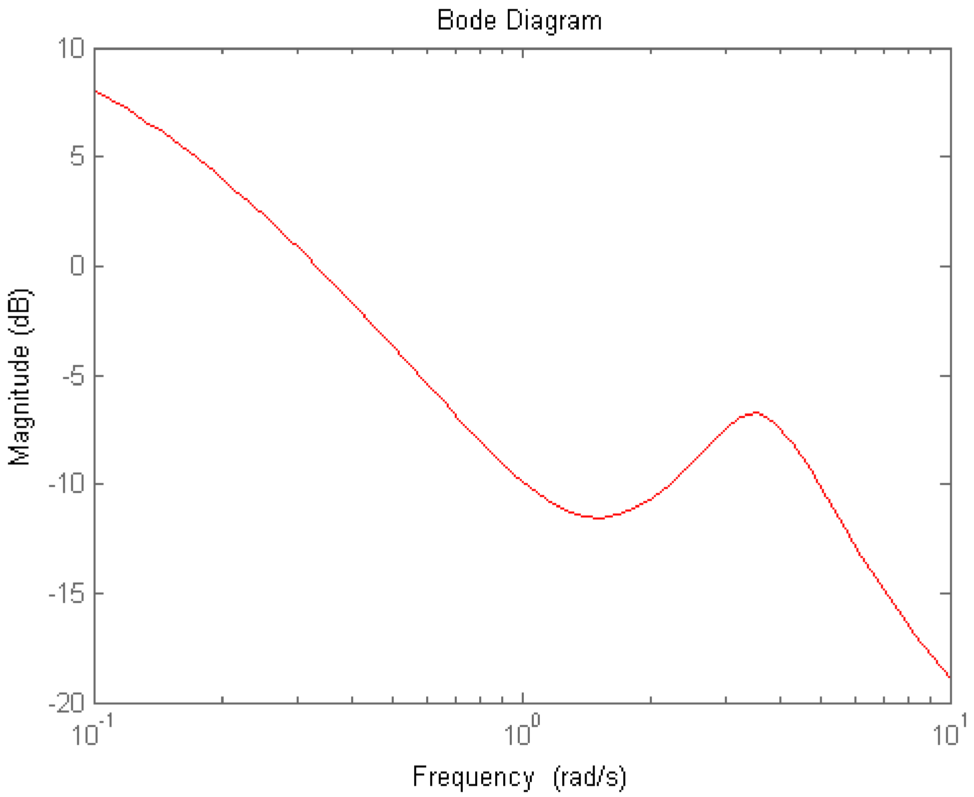
where w denotes the frequency.

Examples

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

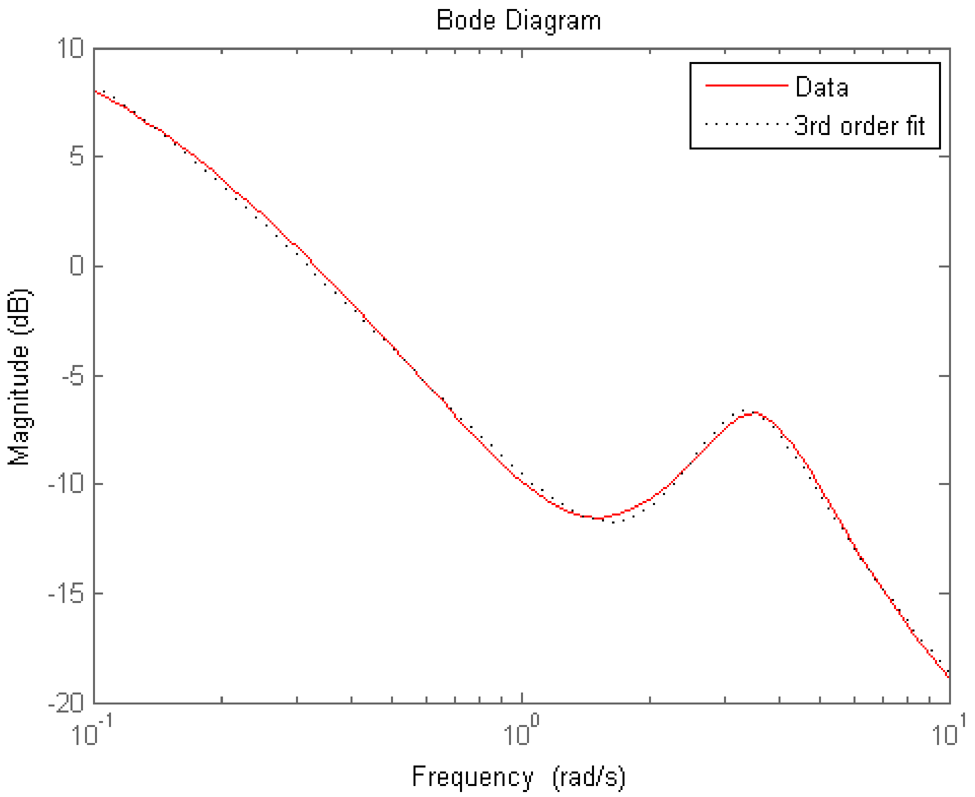
Create frequency response magnitude data from a fifth-order system.

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



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

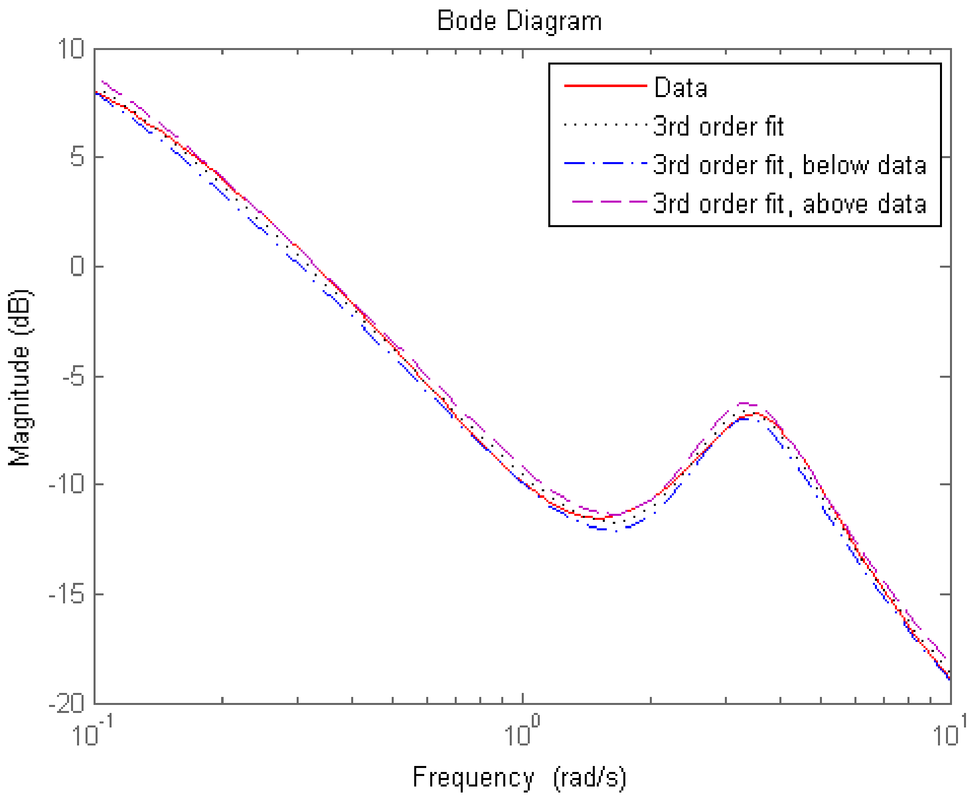
```
ord = 3;
b1 = fitmagfrd(sysg,ord);
b1g = frd(b1,omega);
bodemag(sysg,'r',b1g,'k:');
legend('Data','3rd order fit');
```



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

```
C2.UpperBound = sysg;
C2.LowerBound = [];
b2 = fitmagfrd(sysg,ord,[],[],C2);
b2g = frd(b2,omega);
C3.UpperBound = [];
C3.LowerBound =sysg;
b3 = fitmagfrd(sysg,ord,[],[],C3);
```

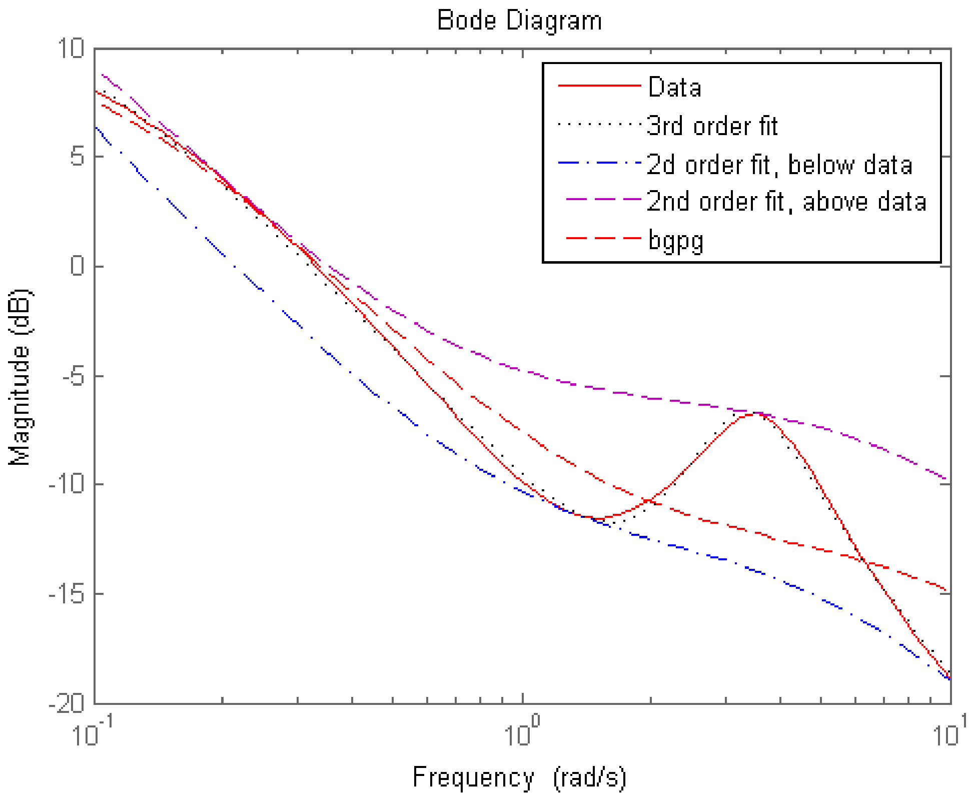
```
b3g = frd(b3,omega);
bodemag(sysg,'r',b1g,'k:',b2g,'b-.',b3g,'m--')
legend('Data','3rd order fit','3rd order fit, below data',...
       '3rd order fit, above data')
```



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

```
ord = 2;
C2.UpperBound = sysg;
```

```
C2.LowerBound = [];
b2 = fitmagfrd(sysg,ord,[],sysg,C2);
b2g = frd(b2,omega);
C3.UpperBound = [];
C3.LowerBound = sysg;
b3 = fitmagfrd(sysg,ord,[],sysg,C3);
b3g = frd(b3,omega);
bgp = fitfrd(genphase(sysg),ord);
bgpg = frd(bgp,omega);
bodemag(sysg,'r',b1g,'k:',b2g,'b-.',b3g,'m--',bgpg,'r--')
legend('Data','3rd order fit','2d order fit, below data',...
       '2nd order fit, above data','bgpg')
```



Algorithms

`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`

gainsurf

Purpose

Create tunable gain surface for gain scheduling

Syntax

```
K = gainsurf(name,K0init,F1,...,FM)
```

Description

In gain-scheduled controllers, each controller gain, $K(\sigma)$, is a function of the scheduling variables, σ. For tuning purposes, it is convenient to parameterize $K(\sigma)$ as a smooth *gain surface* of the form:

$$K(\sigma) = K_0 + K_1 F_1(\sigma) + \ldots + K_M F_M(\sigma).$$

$F_1(\sigma),\ldots,F_M(\sigma)$ are user-selected basis functions. $K_0,\ldots,K_M$ are the coefficients to be tuned. You can use terms in a generic polynomial expansion as basis functions. Or, when you have a priori knowledge of the expected shape of $K(\sigma)$, you can use more specific functions. You can then use `systune`, to tune the coefficients $K_0,\ldots,K_M$, subject to your design requirements.

`K = gainsurf(name,K0init,F1,...,FM)` constructs a tunable model of the gain surface $K(\sigma) = K_0 + K_1 F_1(\sigma) + \ldots + K_M F_M(\sigma)$, sampled at a discrete set of σ values (the *design points*). The arrays `F1,...,FM` contain the values of the basis functions $F_1(\sigma),\ldots,F_M(\sigma)$ at those design points. The gain surface model, `K`, depends on the tunable coefficients $K_0,\ldots,K_M$. You can combine `K` with other static or dynamic elements to construct a closed-loop model of your gain-scheduled control system. Then, use `systune` to tune $K_0,\ldots,K_M$ so that the closed-loop system meets your design requirements at the selected design points.

Input Arguments

name - Identifying label for the tunable gain

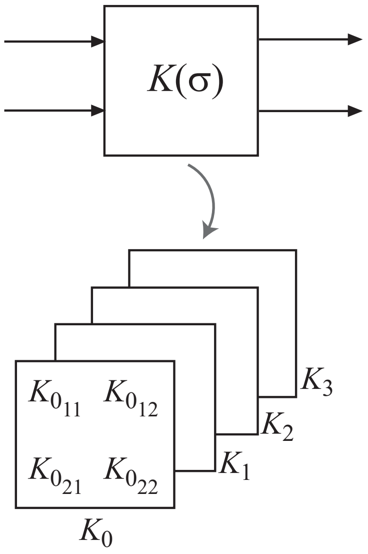
string

Identifying label for the tunable gain surface, specified as a string. The tunable coefficients of the gain surface are assigned names based on this identifying label. For example, suppose you create a gain surface using the name `Kp`. The tunable coefficients are `realp` blocks in the resulting `genmat`. These blocks have names `Kp_0`, `Kp_1`,...,`Kp_M`.

Additionally, you can use this label to refer to the gain surface. For example, you can extract tuned coefficient values from a control system model, `M`, that depends on the gain surface using `[K0,K1,...,KM] = gainsurfdata(M,'Kname')`.

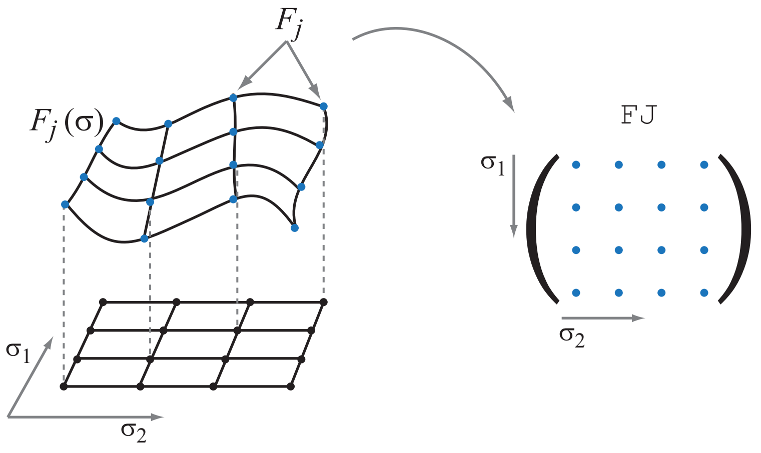
K0init - Initial value of K_0
scalar | array

Initial value of the tunable coefficient K_0, specified as a scalar or an array. The dimensions of `K0init` determine the I/O dimensions of the gain surface. For example, if the gain surface represents a two-input, two-output gain, you can set `K0init = zeros(2)`. Doing so automatically sets the I/O dimensions of the other terms in the gain surface.



F1,...,FM - Values of the basis functions at sample values of scheduling variables
numeric arrays

Function values describing the dependence of the gain surface on the scheduling variables, specified as numeric arrays. Each $F_j(\sigma)$ in the expansion of the gain surface is a scalar-valued function. The corresponding input argument, `Fj`, is a numeric array containing the values of $F_j(\sigma)$ at the corresponding scheduling variable values. For instance, in the following illustration, $F_j(\sigma)$ is a function of two scheduling variables. The corresponding matrix `FJ` contains the values of $F_j(\sigma)$ sampled over a 2-D grid of (σ_1,σ_2) values.



To construct the arguments `F1,...,FM` when your sampling grid is regular, evaluate each $F_j(\sigma)$ over that grid. For example, consider a gain that depends on two scheduling variables, α and β:

$$K(\alpha,\beta) = K_0 + K_1\alpha\beta + K_2\alpha + K_3\beta.$$

For this gain, $F_1 = \alpha\beta$, $F_2 = \alpha$, and $F_3 = \beta$. To create the input argument `F1`, you first create a grid of α and β values that spans the operating range of these variables. Then, you compute the values of F_1 over that grid.

```
[alpha,beta] = ndgrid(0:1:10,50:5:100);
F1 = alpha.*beta;
```

You can improve performance of the tuning algorithm by specifying the function values in terms of normalized scheduling variables that fall in the range [–1,1]. To do this, subtract the mean value from each variable grid and divide by the variable's half-range. For example:

```
alphaN = (alpha-5)/5;
betaN = (beta-75)/25;
F1 = alphaN.*betaN;
```

In this example the values of α and β are regularly spaced. However, regular grid spacing is not required. Suppose your sampling values are arbitrary (α,β) pairs. In this case, α and β are specified as vectors, where ($\alpha(i)$,$\beta(i)$ represents one sample point. Each F_j is also a vector:

$$F_j = \left[F_j(\alpha_1,\beta_1), F_j(\alpha_2,\beta_2),\ldots,F_j(\alpha_N,\beta_N) \right].$$

For an example, see "Gain Surface Over Nonregular Grid" on page 2-104.

Output Arguments

K - Tunable gain surface

generalized matrix

Tunable gain surface, returned as an array of generalized matrices (`genmat`).

The dimensions of each generalized matrix in the array (the I/O dimensions of the gain surface) are determined by the dimensions of `K0init`. The dimensions of the array itself are determined by the sampling grid used to specify the basis functions. Thus, each entry in the array represents the gain at the corresponding scheduling variable value.

The gain surface depends on the tunable coefficients `K0,...,KM`. Each coefficient is modeled as a `realp` block of the same size as `K`. For instance, if the gain surface models a scalar gain, then each coefficient is a scalar `realp` block. If the gain surface models a 2-by-3 gain matrix, then each coefficient is a 2-by-3 `realp` block.

Examples

Tunable Gain With One Independent Variable

Create a scalar gain K that varies as a quadratic function of a single scheduling variable, t:

$$K(t) = K_0 + K_1 t + K_2 t^2.$$

This gain surface can represent a gain that varies with time. The coefficients K_0,K_1, and K_2 are the tunable parameters of this time-varying gain.

To represent the tunable gain surface $K(t)$ in MATLAB, first choose a vector of t values in the range of interest for your problem. Then, obtain the values of each basis function in the expansion of $K(t)$, at those t values.

```
t = 0:5:40;
F1 = t;
F2 = t.^2;
```

Create a tunable model of the gain surface $K(t)$, sampled at the t values.

```
K = gainsurf('K',1,F1,F2)
```

```
K =

  1x9 array of generalized matrices with 1 rows, 1 columns, and the follo
    K_0: Scalar parameter, 1 occurrences.
    K_1: Scalar parameter, 1 occurrences.
    K_2: Scalar parameter, 1 occurrences.

Type "double(K)" to see the current value, "get(K)" to see all properties
```

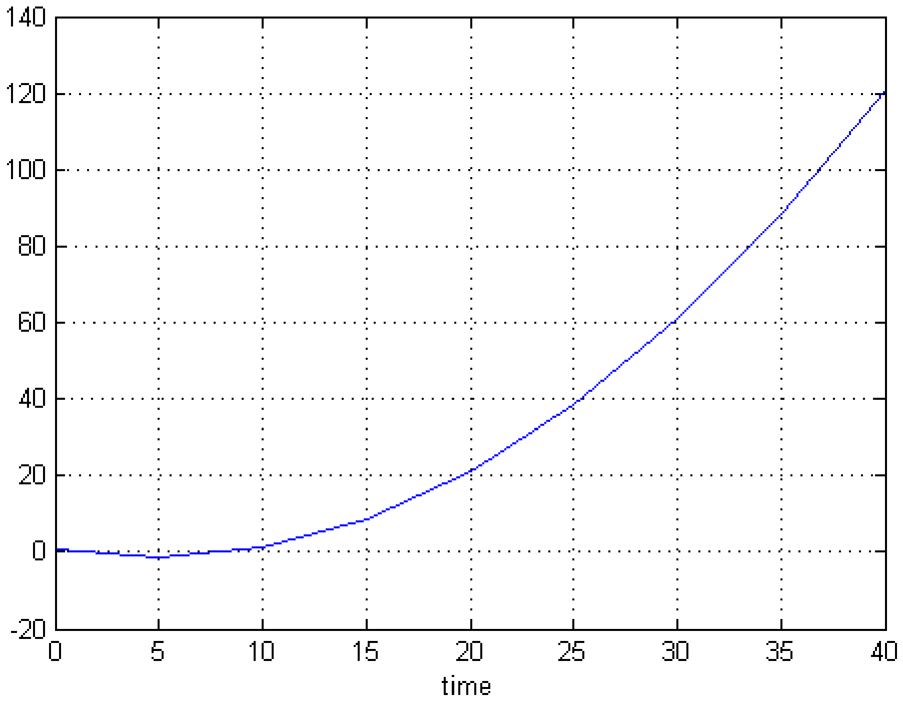
`K` is an array of generalized matrices. Each element in `K` describes $K(t)$ for a particular value of t, and depends on the tunable coefficients `K_0`, `K_1`, and `K_2`. For example, the first element, `K(:,:,1)`, is $K(0) = K_0 + K_1{*}0 + K_2{*}0^2 = K_0$. The second element, `K(:,:,2)`, is $K(5) = K_0 + K_1{*}5 + K_2{*}5^2 = K_0$, and so on.

Associate the independent variable values with the corresponding values of `K`.

```
K.SamplingGrid = struct('time',t);
```

The `SamplingGrid` property keeps track of the scheduling variable values associated with each entry in `K`. This association is convenient for tracing results back to independent variable values. For instance, you can use `view(K)` to inspect the tuned values of the gain surface after tuning. When you do so, `view` takes the axis range and labels from the entries in `SamplingGrid`. For this example, instead of tuning, manually set the values of the tunable blocks to non-zero values. View the resulting gain as a function of time.

```
values = struct('K_0',1,'K_1',-1,'K_2',0.1);
view(setBlockValue(K,values))
```



You can use `K` as a tunable gain to design a gain-scheduled controller. Use `systune` to tune the coefficients K_0, K_1, and K_2 at the sample times t = 0,5,...,40.

Tunable Gain With Two Independent Variables

This example shows how to model a scalar gain K with a bilinear dependence on two scheduling variables, α and V, as follows:

$$K\left(\alpha_N, V_N\right) = K_0 + K_1\alpha_N + K_2V_N + K_3\alpha_N V_N.$$

For this example, α is an angle of incidence that ranges from 0 to 15 degrees, and V is a speed that ranges from 300 to 600 m/s. The coefficients K_0,...,K_3 are the tunable parameters of this variable gain.

Create a grid of design points, (α,V), that are linearly spaced in α and V. These design points are where you will tune the gain surface coefficients.

```
[alpha,V] = ndgrid(0:5:15,300:100:600);
```

These arrays, `alpha` and `V`, represent the independent variation of the two scheduling variables, each across its full range.

When you tune the gain surface coefficients with `systune`, you might obtain better solver performance by normalizing the scheduling variables to fall within the interval [–1,1]. Scale the α and V grid to fall within this range.

```
alphaN = alpha/15;
VN = (V-450)/150;
```

Create the tunable gain surface sampled at the grid of (α_N,V_N) values:

$$K\left(\alpha_N, V_N\right) = K_0 + K_1\alpha_N + K_2V_N + K_3\alpha_N V_N.$$

In this expansion, the basis functions are:

$$F_1\left(\alpha_N, V_N\right) = \alpha_N$$
$$F_2\left(\alpha_N, V_N\right) = V_N$$
$$F_3\left(\alpha_N, V_N\right) = \alpha_N V_N.$$

Specify the values of the basis functions over the (α_N,V_N).

```
F1 = alphaN;
F2 = VN;
F3 = alphaN.*VN;
```

```
K = gainsurf('K',1,F1,F2,F3)
```

```
K =

  4x4 array of generalized matrices with 1 rows, 1 columns, and the follo
    K_0: Scalar parameter, 1 occurrences.
    K_1: Scalar parameter, 1 occurrences.
    K_2: Scalar parameter, 1 occurrences.
    K_3: Scalar parameter, 1 occurrences.

Type "double(K)" to see the current value, "get(K)" to see all properties
```

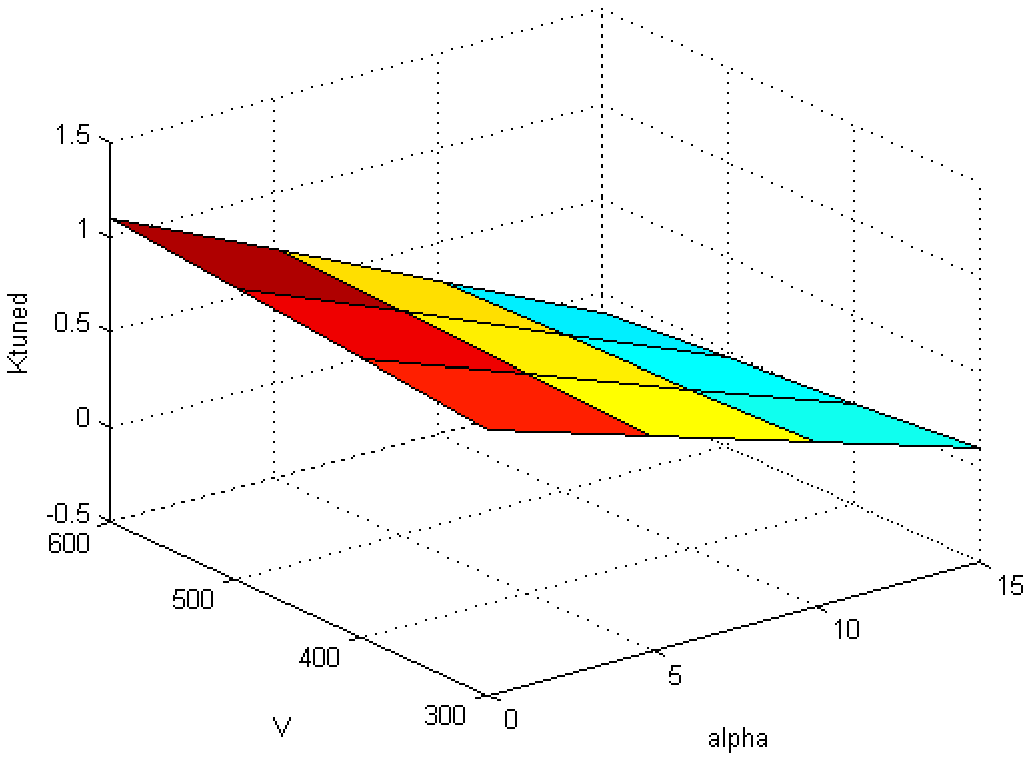
`K` is an array of generalized matrices. Each element in `K` corresponds to $K(a_N,V_N)$ for a particular (a_N,V_N) pair, and depends on the tunable coefficients `K_0`,...,`K_3`.

Associate the independent variable values with the corresponding values of `K`.

```
K.SamplingGrid = struct('alpha',alpha,'V',V);
```

The `SamplingGrid` property keeps track of the scheduling variable values associated with each entry in `K`. This association is convenient for tracing results back to independent variable values. For instance, you can use `view(K)` to inspect the tuned values of the gain surface after tuning. When you do so, `view` takes the axis range and labels from the entries in `SamplingGrid`. For this example, instead of tuning, manually set the values of the tunable blocks to non-zero values. View the resulting gain surface as a function of the scheduling variables.

```
values = struct('K_0',1,'K_1',-1,'K_2',0.1,'K_3',-0.2);
Ktuned = setBlockValue(K,values);
view(Ktuned)
```



The variable names and values that you specified in the `SamplingGrid` property are used to scale and label the axes.

You can use `K` as a tunable gain to build a control system with gain-scheduled tunable components. For example, use `K` to create a gain-scheduled low-pass filter.

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

You can use gain surfaces as arguments to model creation commands like `tf` the same way you would use numeric arguments. The resulting filter is a generalized state-space (`genss`) model array that depends on the four coefficients of the gain surface.

Use model interconnection commands (such as `connect` and `feedback`) to combine `F` with an array of plant models sampled at the same values of α and V. You can then use `systune` to tune the gain-scheduled controller to meet your design requirements. Because you normalized the scheduling variables to model the tunable gain, you must adjust the coefficient values in the implementation of your tuned controller.

Gain Surface Over Nonregular Grid

Create a gain surface sampled at scheduling variable values that do not form a regular grid in the operating domain. The gain surface varies as a bilinear function of variables α and β:

$$K(\alpha,\beta) = K_0 + K_1\alpha + K_2\beta + K_3\alpha\beta.$$

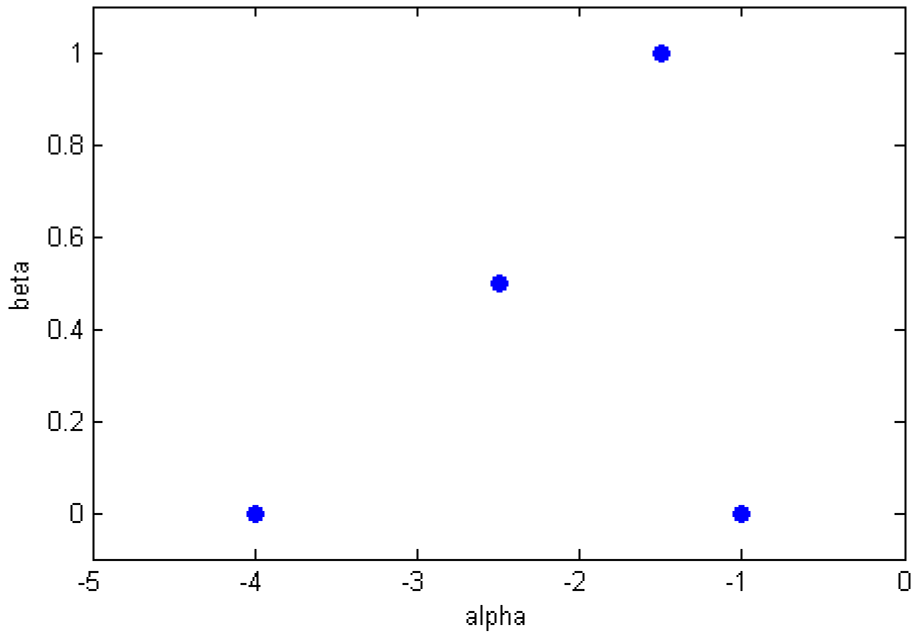
The values of interest of the scheduling variables are the following (α,β) pairs.

$$(\alpha,\beta) = \begin{cases} (-1.0, 0.0) \\ (-1.5, 1.0) \\ (-2.5, 0.5) \\ (-4.0, 0.0) \end{cases}.$$

The basis functions of the expansion of K are:

- $F_1 = \alpha$
- $F_2 = \beta$
- $F_3 = \alpha\beta$

Instead of a regular grid of (α,β) values, here the gain surface is sampled at irregularly spaced points on (α,β)-space.



Specify the (α,β) sample values as vectors. Evaluate the basis functions at these sample points.

```
alpha = [-1.0;-1.5;-2.5;-4.0];
beta = [0.0;1.0;0.5;0.0];
F1 = alpha;
F2 = beta;
F3 = alpha.*beta;
```

Create the tunable model of the gain surface using these sampled function values.

```
K = gainsurf('K',1,F1,F2,F3)
```

```
K =
  4x1 array of generalized matrices with 1 rows, 1 columns, and the follo
    K_0: Scalar parameter, 1 occurrences.
    K_1: Scalar parameter, 1 occurrences.
    K_2: Scalar parameter, 1 occurrences.
    K_3: Scalar parameter, 1 occurrences.

Type "double(K)" to see the current value, "get(K)" to see all properties
```

The gain surface is represented by a 4-by-1 array of generalized matrices. Because `K` is a scalar gain, each element in the array is 1-by-1. Each element in the array represents $K(\alpha,\beta)$ for the corresponding (α,β) sample. Each of these elements depends on the tunable parameters `K_0,...,K_3`.

Use the `SamplingGrid` property to associate the α and β values with the corresponding entries in `K`.

```
SG = struct('alpha',alpha,'beta',beta);
K.SamplingGrid = SG;
```

See Also

`genmatndgridgainsurfdataview (genmat)systune`

Related Examples

- “Gain-Scheduled PID Controller”
- “Tuning of Gain-Scheduled Three-Loop Autopilot”
- Gain Scheduled Control Of a Chemical Reactor

Concepts

- “Gain-Scheduled Control Systems”
- “Parametric Gain Surfaces”

Purpose

Get values of gain surface coefficients

Syntax

```
[K0,K1,...,KM] = gainsurfdata(K)
[K0,K1,...,KM] = gainsurfdata(M,Kname)
```

Description

`[K0,K1,...,KM] = gainsurfdata(K)` returns the current values of the coefficients of a gain surface, `K`. `K` is of the form:

$$K(\sigma) = K_0 + K_1 F_1(\sigma) + \ldots + K_M F_M(\sigma).$$

Typically, you create `K` using `gainsurf`.

`[K0,K1,...,KM] = gainsurfdata(M,Kname)` returns the coefficients of a gain surface having name `Kname` that is incorporated into a control system model, `M`. You can use this syntax to extract tuned coefficient values after using `systune` to tune `M`.

Input Arguments

K - Gain surface

gain surface (`genmat` array created with `gainsurf`)

Gain surface from which to extract coefficients, specified as a generalized matrix (`genmat`) array created using `gainsurf`.

M - Control system model

generalized state-space model (`genss`) | `genss` model array

Control system model from which to extract tunable coefficients, specified as a `genss` model or array of `genss` models. `M` must depend on a gain surface that has the name `Kname`. Typically, you create that gain surface using `gainsurf`, incorporate it into `M`, and tune the coefficients with `systune`. Then you can use `gainsurfdata(M,Kname)` to extract the tuned values of the coefficients associated with that gain surface.

Kname - Name of gain surface

string

Name of a gain surface that the control system model `M` depends on, specified as a string. The value of this string is the `Name` property of a `genmat` that represents a gain surface. For example, suppose you create a tunable gain surface, `Kc`, and combine it with a numeric LTI model array, `sysarr`:

```
K = gainsurf('Kc',1,F1,F2,F3);
M = feedback(sysarr*K,1);
```

`M` is an array of tunable control system models that depend on the gain surface. The following code extracts the values of the coefficients associated with this gain surface.

```
[Kc0,Kc1,Kc2,Kc3] = gainsurfdata(M,'Kc');
```

Output Arguments

K0,K1,...,KM - Current values of tunable coefficients

numeric scalar | numeric array

Current values of the tunable coefficients of the gain surface, returned as numeric values. The gain surface `K` or `Kname` has the form:

$$K(\sigma) = K_0 + K_1 F_1(\sigma) + \ldots + K_M F_M(\sigma).$$

- If the gain surface represents a scalar gain, then the current values are scalars.
- If the gain surface represents a MIMO gain, then the current values are arrays of the same dimensions as the I/O dimensions of the gain surface.

Examples

Get Current Values of Gain Surface Coefficients

Extract the current coefficient values from a tunable gain surface $K = K_0 + K_1 a + K_2 a^2$.

For this example, create a tunable gain surface and extract the initial values of its coefficients. Generally, `gainsurfdata` is useful for extracting tuned coefficient values after control system tuning with `systune`.

Create the tunable gain surface.

```
alpha = 0:10:100;
F1 = alpha;
F2 = alpha.^2;
K = gainsurf('K',1,F1,F2)
```

```
K =

  1x11 array of generalized matrices with 1 rows, 1 columns, and the f
    K_0: Scalar parameter, 1 occurrences.
    K_1: Scalar parameter, 1 occurrences.
    K_2: Scalar parameter, 1 occurrences.

Type "double(K)" to see the current value, "get(K)" to see all propert
```

K is a generalized matrix (genmat) with tunable coefficients K_0, K_1, K_2.

Extract the current values of the tunable coefficients.

```
[K0,K1,K2] = gainsurfdata(K)
```

```
K0 =

     1


K1 =

     0


K2 =

     0
```

You specify the initial value of K_0 with the K0init input argument to gainsurf. The output of gainsurfdata shows that gainsurf automatically assigns initial values of 0 to the remaining coefficients.

Get Coefficient Values from Control System Model

Extract current coefficient values from a generalized model of a control system that depends on a tunable gain surface.

For this example, create a control system model and extract the initial values of its coefficients. Generally, gainsurfdata is useful for extracting tuned coefficient values after control system tuning with systune.

Create an array of plant models in which each plant is sampled at a different value of a scheduling parameter, α.

```
alpha = (0:10:100)';
G = zpk(zeros(1,1,11));
for ii = 1:11
 G(:,:,ii) = zpk([],-1+0.01*alpha(ii),1);
end
```

Create a tunable PI controller with gains that depend the same scheduling parameter.

```
F1 = alpha;
F2 = alpha.^2;
Kp = gainsurf('Kp',1,F1,F2);
Ki = gainsurf('Ki',0.1,F1,F2);
C = pid(Kp,Ki);
```

Combine the plant model with the tunable controller to build a closed-loop control system model array.

```
M = feedback(G*C,1)
```

```
M =

  11x1 array of generalized continuous-time state-space models.
  Each model has 1 outputs, 1 inputs, 2 states, and the following bloc
    Ki_0: Scalar parameter, 1 occurrences.
    Ki_1: Scalar parameter, 1 occurrences.
    Ki_2: Scalar parameter, 1 occurrences.
    Kp_0: Scalar parameter, 1 occurrences.
    Kp_1: Scalar parameter, 1 occurrences.
    Kp_2: Scalar parameter, 1 occurrences.

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

This array of tunable closed-loop models, `M`, depends on the coefficients that parametrize the PI gains in terms of the scheduling variable α.

Extract the initial values of the tunable coefficients.

```
[Kp0,Kp1,Kp2] = gainsurfdata(M,'Kp');
[Ki0,Ki1,Ki2] = gainsurfdata(M,'Ki');
```

`M` depends on the gain surfaces that are assigned the names `Kp` and `Ki`. Therefore, `gainsurfdata` finds and returns the initial values of the coefficients associated with those gain surfaces.

Similarly, use `gainsurfdata` to extract the tuned values of the coefficients after using `systune` to tune `M` against a set of design requirements.

See Also

`gainsurfgenmatgensssystune`

Related Examples

- “Tuning of Gain-Scheduled Three-Loop Autopilot”

Concepts

- “Gain-Scheduled Control Systems”
- “Parametric Gain Surfaces”

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

Compute gap and nugap Metrics for Stable and Unstable Plant Models

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

```
p1 = tf(1,[1 -0.001]);
p2 = tf(1,[1 0.001]);
```

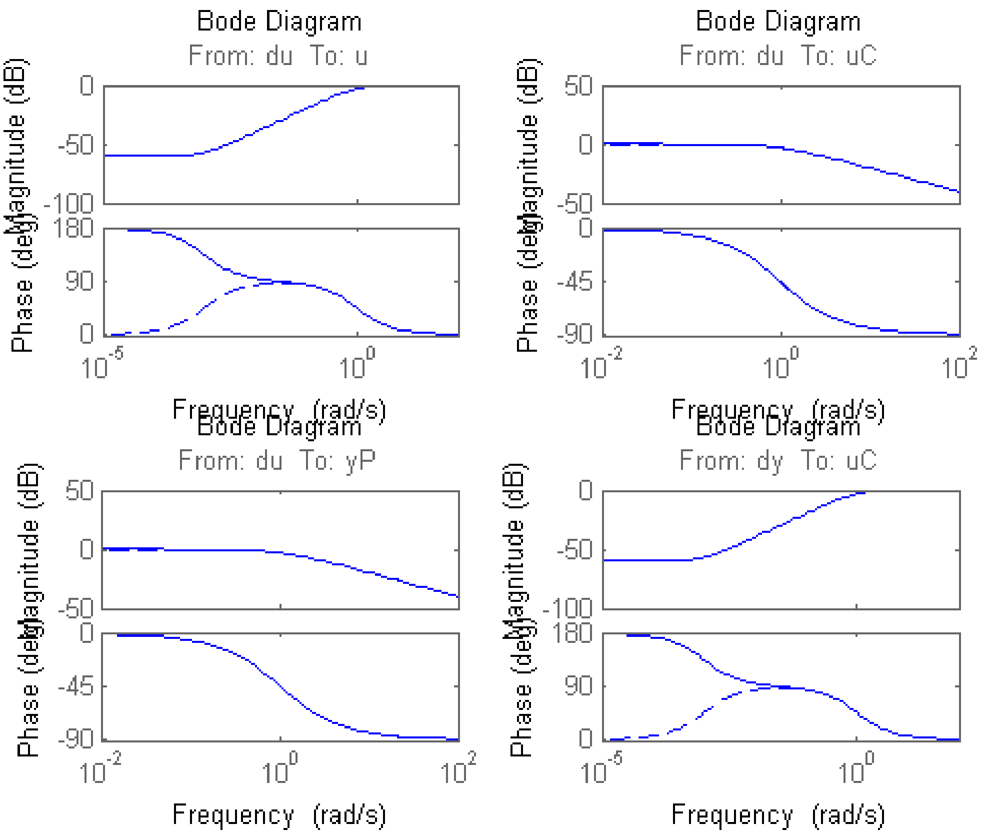
Despite the fact that one plant is unstable and the other is stable, these plants are close in the `gap` and `nugap` metrics.

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

```
g =
    0.0029


ng =

    0.0020
```

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

```
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
```

Algorithms

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 $W_2 G W_1$ and K by $W_1^{-1} K W_2^{-1}$ (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

`loopsyn` | `ncfsyn` | `robuststab` | `wcsens` | `wcmargin`

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`

getlmis

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:

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

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

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

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 λ = 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 λ ≤ 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

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

consider the problem of finding a single Lyapunov function $V(x) = x^TPx$ 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

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

$$A_1^T P + PA_1 < \alpha P \tag{2-11}$$

$$A_2^T P + PA_2 < \alpha P \tag{2-12}$$

$$A_3^T P + PA_3 < \alpha P \tag{2-13}$$

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:

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

Tips

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 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

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

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

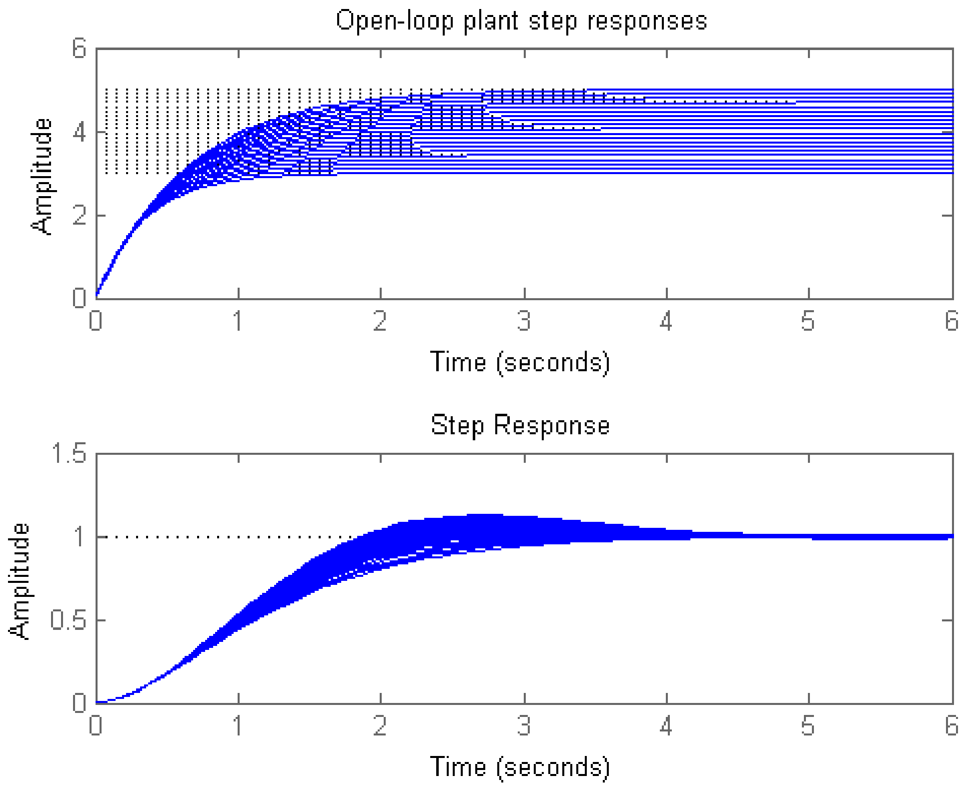
```
gamma = ureal('gamma',4);
tau = ureal('tau',.5,'Percentage',30);
```

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

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

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

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



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

Grid Over Multi-Dimensional Parameter Spaces

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

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

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

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

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

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

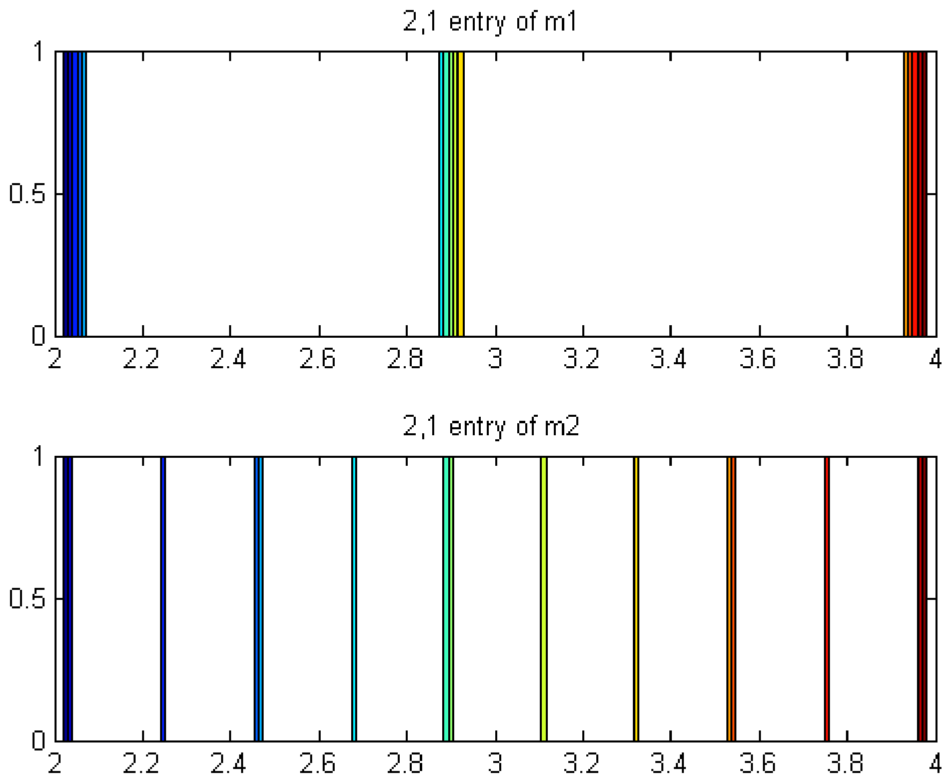
Next, grid the `(a,b,c,d)` space at 15 places.

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

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

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

```
subplot(2,1,1)
hist(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

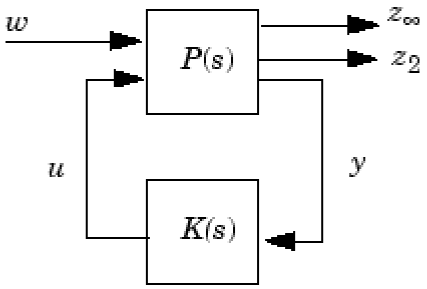
Mixed H_2/H_∞ synthesis with pole placement constraints

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

- $\|T_\infty\|_{\text{[[BULLET]]}} < \gamma_0$
- $\|T_2\|_2 < \text{v}_0$
- 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, v_0, α, β] 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," *IEEE Trans. Aut. Contr.*, 41 (1995), pp. 358–367.

Scherer, C., "Mixed H2/H-infinity Control," *Trends in Control: A European Perspective*, Springer-Verlag (1995), pp.173–216.

See Also

`lmireg` | `msfsyn`

h2syn

Purpose

H_2 control synthesis for LTI plant

Syntax

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

Description

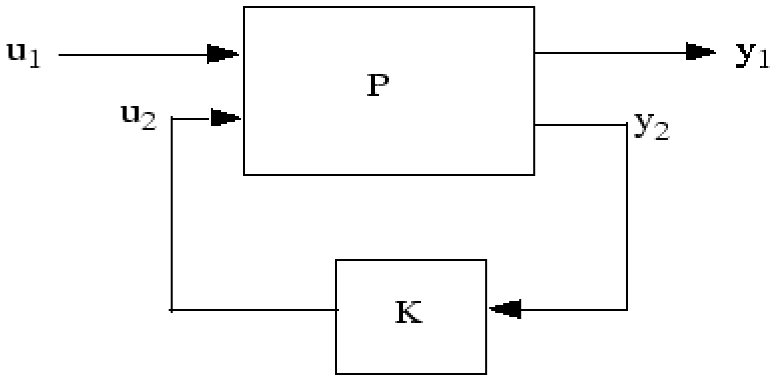
`h2syn` computes a stabilizing H_2 optimal `lti/ss` controller `K` for a partitioned LTI plant `P`. The controller, `K`, stabilizes the plant `P` and has the same number

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

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 Ty_1u_1
`GAM = norm(CL)`	H_2 optimal cost $\gamma = \lVert T_{y_1u_1} \rVert 2$
`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.

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

open_loop_poles =

    0.2593
   15.9497
   20.7994


closed_loop_poles =

  -26.8951
  -22.4817
  -20.6965
  -17.6041
```

```
   -0.8694
   -2.6697
```

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

$$T_{y_1u_1} \square \begin{bmatrix} W_1S \\ (W_2\ /\ G)T \\ W_3T \end{bmatrix}$$

where $G(s) = \dfrac{s-1}{s-2}$, $W_1 = \dfrac{0.1(s+1000)}{100s+1}$, $W_2 = 0.1$, 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')
```

Algorithms

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

$$\begin{aligned}J_{LQG} &= \lim_{T\to\infty} E\left\{\frac{1}{T}\int_0^T y_1^T y_1 dt\right\}\\ &= \lim_{T\to\infty} E\left\{\frac{1}{T}\int_0^T \begin{bmatrix}x^T u_2^T\end{bmatrix}\begin{bmatrix}Q & N_c\\ N_c^T & R\end{bmatrix}\begin{bmatrix}x\\ u_2\end{bmatrix}dt\right\}\\ &= \lim_{T\to\infty} E\left\{\frac{1}{T}\int_0^T \begin{bmatrix}x^T u_2^T\end{bmatrix}\begin{bmatrix}C_1^T\\ D_{12}^T\end{bmatrix}[C_1 \quad D_{12}]\begin{bmatrix}x\\ u_2\end{bmatrix}dt\right\}\end{aligned}$$

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

$$\begin{aligned}E\left\{\begin{bmatrix}\xi(t)\\ \theta(t)\end{bmatrix}[\xi(\tau) \quad \theta(\tau)]^T\right\} &= \begin{bmatrix}\Xi & N_f\\ N_f^T & \Theta\end{bmatrix}\delta(t-\tau)\\ &= \begin{bmatrix}B_1\\ D_{21}\end{bmatrix}\begin{bmatrix}B_1^T & D_{21}^T\end{bmatrix}\delta(t-\tau)\end{aligned}$$

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

$$\begin{aligned}&\dot{\hat{x}} = A\hat{x} + B_2u_2 + K_{FC}(y_2 - C_{2\hat{x}} - D_{22}u_2)\\ &K_{FC} = (YC_2^T + N_f)\Theta^{-1} = (YC_2^T + B_1D_{21}^T)(D_{21}D_{21}^T)^{-1}\end{aligned}$$

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

$$YA^T + AY - (YC_2^T + N_f)\Theta^{-1}(C_2Y + N_f^T) + \Xi = 0$$

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

$$K(s) := \left[\begin{array}{c|c} A - K_{FC}C_2 - B_2K_{FI} + K_{FC}D_{22}K_{FI} & K_f \\ \hline -K_{FI} & 0 \end{array}\right]$$

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 *D11* to zero.

3 Optimal Cost GAM

The full information (FI) cost is given by the equation $\left(\text{trace }(B_1' X_2 B_1)\right)^{\frac{1}{2}}$. The output estimation cost (OEF) is given by $\left(\text{trace }(F_2 Y_2 F_2')\right)^{\frac{1}{2}}$, where $F2 =: -(B_2' X_2 + D_{12}' C_1)$. The disturbance feedforward cost (DFL) is $\left(\text{trace }(L_2' X_2 L_2)\right)^{\frac{1}{2}}$, where L_2 is defined by $-(Y_2 C_2' + B_1 D_{21}')$ and the full control cost (FC) is given by

$(\text{trace}\ (C_1 Y_2 C_1'))^{\frac{1}{2}}$. 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 $\|\| G\text{-}GRED \|\|_\infty$) • 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.

Algorithms

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

1 Find the controllability and observability grammians *P* and *Q*.

2 Form the descriptor

$$E = QP - \rho^2 I$$

where $\sigma_k > \rho \geq \sigma_{k+1}$, and descriptor state-space

Take SVD of descriptor *E* and partition the result into k^{th} order truncation form

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

$$E = [U_{E1}, U_{E2}] \left[\begin{array}{c|c} \Sigma_E 0 & 0 \\ \hline 0 & 0 \end{array}\right] \begin{bmatrix} V_{E1}^T \\ V_{E2}^T \end{bmatrix}$$

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

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

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

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

$$D_1 = D$$

4 Form the equivalent state-space model.

$$\begin{bmatrix} \tilde{A} & \tilde{B} \\ \tilde{C} & \tilde{D} \end{bmatrix} = \begin{bmatrix} \Sigma_E^{-1}(A_{11} - A_{12}A_{22}\ \ A_{21}) & \Sigma_E^{-1}(B_1 - A_{12}A_{22}\ \ B_2) \\ C_1 - C_2 A_{22}{}^{\dagger} A_{21} & D_1 - C_2 A_{22}{}^{\dagger} B_2 \end{bmatrix}$$

The final k^{th} order Hankel MDA is the stable part of the above state-space realization. Its anticausal part is stored in `redinfo.Ganticausal`.

The proof of the Hankel MDA algorithm can be found in [2]. The error system between the original system G and the *Zeroth Order Hankel MDA* G_0 is an all-pass function [1].

Examples

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

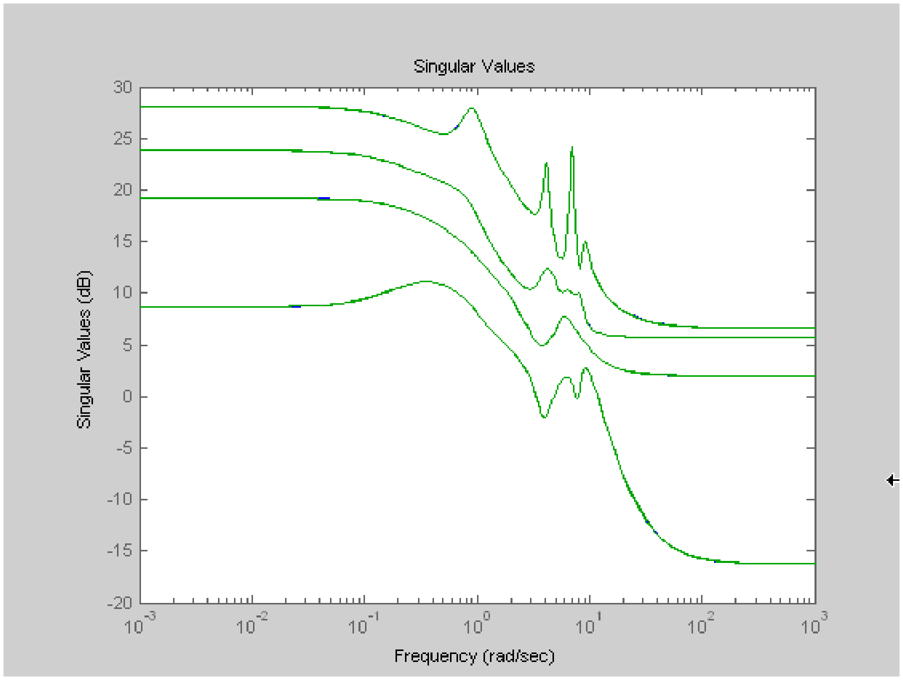
```
rng(1234,'twister');
G = rss(30,5,4);
[g1, redinfo1] = hankelmr(G); % display Hankel SV plot
       % and prompt for order (try 15:20)
[g2, redinfo2] = hankelmr(G,20);
[g3, redinfo3] = hankelmr(G,[10:2:18]);
[g4, redinfo4] = hankelmr(G,'MaxError',[0.01, 0.05]);
for i = 1:4
 figure(i); eval(['sigma(G,g'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

Singular Value Bode Plot of G (30-state, 5 outputs, 4 inputs) on page 2-142 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 2-143.

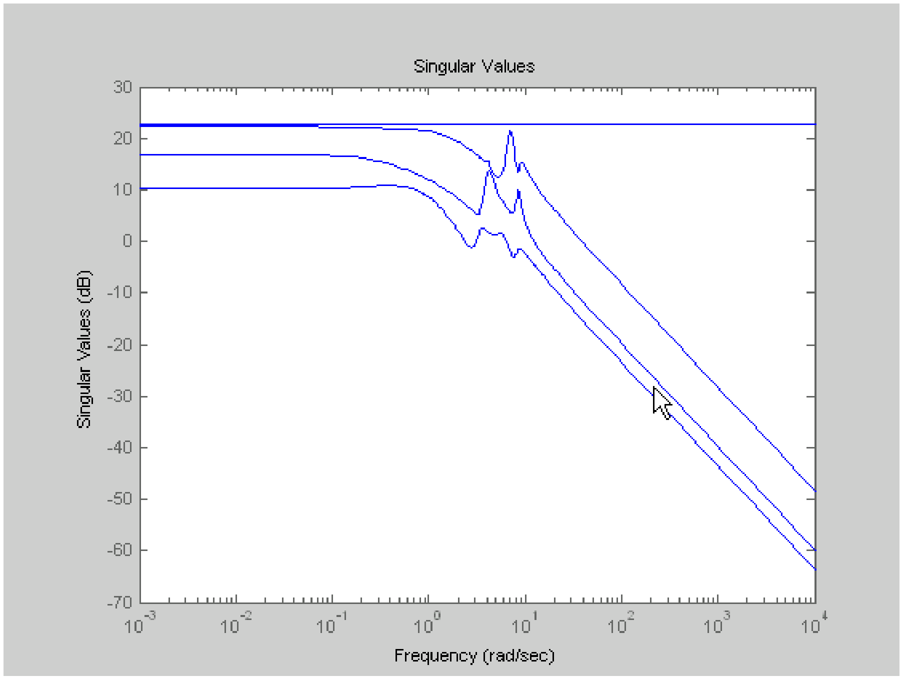
The *Zeroth order Hankel MDA* and its error system sigma plot are obtained via commands

```
[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`

hankelsv

Purpose

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

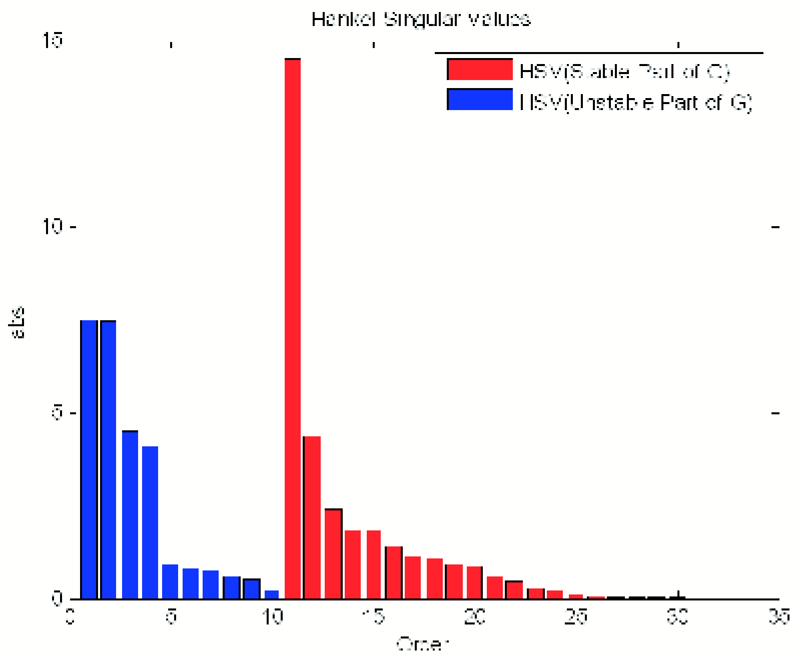
Syntax

```
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

Algorithms

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

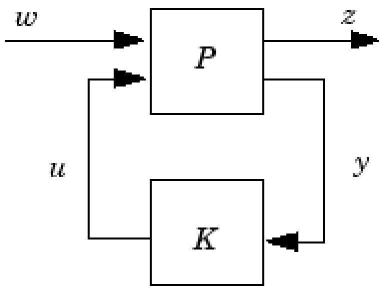
$$P\begin{cases}\dot{x} = A(p)x + B_1(p)w + B_2u\\ z = C_1(p)x + D_{11}(p)w + D_{12}u\\ y = C_2x + D_{21}w + D_{22}u\end{cases}$$

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=[p2,m2]` if $y \in R^{p2}$ and $u \in R^{m2}$). Note that `hinfgs` also accepts the polytopic model of P returned, e.g., by `aff2pol`.

`hinfgs` returns the optimal closed-loop quadratic performance `gopt` and a polytopic description of the gain-scheduled controller `pdK`. To test if a closed-loop quadratic performance γ is achievable, set the third input `gmin` to γ. The arguments `tol` and `tolred` control the required relative accuracy on `gopt` and the threshold for order reduction. Finally, `hinfgs` also returns solutions R, S of the characteristic LMI system.

Controller Implementation

The gain-scheduled controller `pdK` is parametrized by $p(t)$ and characterized by the values $K_{\Pi j}$ of $\begin{pmatrix} A_K(p) & B_K(p) \\ C_K(p) & D_K(p) \end{pmatrix}$ at the corners $^{3}{}_{j}$ of the parameter box. The command

```
Kj = psinfo(pdK,'sys',j)
```

returns the j-th vertex controller $K_{\Pi j}$ while

```
pv = psinfo(pdP,'par')
vertx = polydec(pv)
Pj = vertx(:,j)
```

gives the corresponding corner $^{3}{}_{j}$ of the parameter box (`pv` is the parameter vector description).

The controller scheduling should be performed as follows. Given the measurements $p(t)$ of the parameters at time t,

1. Express $p(t)$ as a convex combination of the $^{3}{}_{j}$:

$$p(t) = \alpha_1{}^{3}{}_{1} + \ldots + \alpha_N{}^{3}{}_{N}, \; \alpha_j \ge 0, \sum_{i=1}^{N} \alpha_j = 1$$

   This convex decomposition is computed by `polydec`.

2 Compute the controller state-space matrices at time t as the convex combination of the vertex controllers $K_{\Pi j}$:

$$\begin{pmatrix} A_K(t) & B_K(t) \\ C_K(t) & D_K(t) \end{pmatrix} = \sum_{i=1}^{N} \alpha_j K_{\Pi_i}.$$

3 Use $A_K(t)$, $B_K(t)$, $C_K(t)$, $D_K(t)$ to update the controller state-space equations.

References

Apkarian, P., P. Gahinet, and G. Becker, "Self-Scheduled H_∞ Control of Linear Parameter-Varying Systems," *Automatica*, 31 (1995), pp. 1251–1261.

Becker, G., Packard, P., "Robust Performance of Linear-Parametrically Varying Systems Using Parametrically-Dependent Linear Feedback," *Systems and Control Letters*, 23 (1994), pp. 205–215.

Packard, A., "Gain Scheduling via Linear Fractional Transformations," *Syst. Contr. Letters*, 22 (1994), pp. 79–92.

See Also

`psys` | `pvec` | `pdsimul` | `polydec`

hinfnorm

Purpose

H_∞ norm of dynamic system

Syntax

```
ninf = hinfnorm(sys)
ninf = hinfnorm(sys,tol)
[ninf,fpeak] = hinfnorm( ___ )
```

Description

`ninf = hinfnorm(sys)` returns the H_∞ in absolute units of the dynamic system model, `sys`.

- If `sys` is a stable SISO system, then the H_∞ norm is the peak gain, the largest value of the frequency response magnitude.
- If `sys` is a stable MIMO system, then the H_∞ norm is the largest singular value across frequencies.
- If `sys` is an unstable system, then the H_∞ norm is defined as `Inf`.
- If `sys` is a model that has tunable or uncertain parameters, then `hinfnorm` evaluates the H_∞ norm at the current or nominal value of `sys`.
- If is a model array, then `hinfnorm` returns an array of the same size as `sys`, where `ninf(k) = hinfnorm(sys(:,:,k))` .

For stable systems, `hinfnorm(sys)` is the same as `getPeakGain(sys)`.

`ninf = hinfnorm(sys,tol)` returns the H_∞ norm of `sys` with relative accuracy `tol`.

`[ninf,fpeak] = hinfnorm( ___ )` also returns the frequency, `fpeak`, at which the peak gain or largest singular value occurs. You can use this syntax with any of the input arguments in previous syntaxes. If `sys` is unstable, then `fpeak` = `Inf`.

Input Arguments

sys - Input dynamic system

dynamic system model | model array

Input dynamic system, specified as any dynamic system model or model array. `sys` can be SISO or MIMO.

tol - Relative accuracy

0.01 (default) | positive real scalar

Relative accuracy of the peak gain, specified as a positive real scalar value. `hinfnorm` calculates `ninf` such that the fractional difference between `ninf` and the true H_∞ norm of `sys` is no greater than `tol`.

Output Arguments

ninf - H_∞ norm of dynamic system

`Inf` | scalar | array

H_∞ norm of `sys`, returned as `Inf`, a scalar value, or an array.

- If `sys` is a single stable model, then `ninf` is a scalar value.
- If `sys` is a single unstable model, then `ninf` is `Inf`.
- If `sys` is a model array, then `ninf` is an array of the same size as `sys`, where `ninf(k) = hinfnorm(sys(:,:,k))`.

fpeak - Frequency of peak gain or largest singular value

`Inf` | nonnegative real scalar | array

Frequency at which the peak gain or largest singular value occurs, returned as `Inf`, a nonnegative real scalar value, or an array. The frequency is expressed in units of rad/`TimeUnit`, relative to the `TimeUnit` property of `sys`.

- If `sys` is a single stable model, then `fpeak` is a scalar.
- If `sys` is a single unstable model, then `fpeak` is `Inf`.
- If `sys` is a model array, then `fpeak` is an array of the same size as `sys`.In this case, `fpeak(k)` is the peak gain or largest singular value frequency of the *k*th model in the array.

Examples

Norm of MIMO System

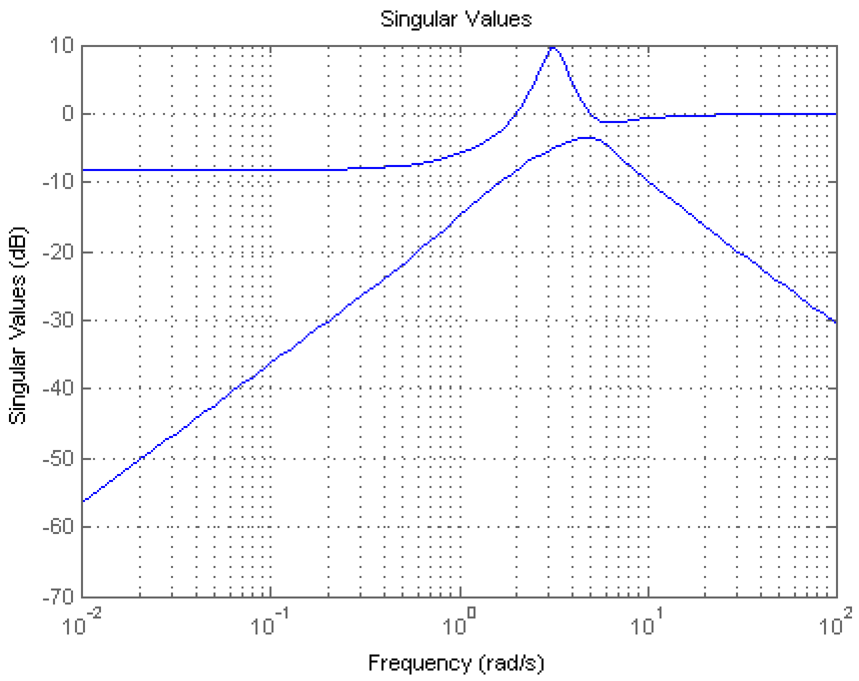
Compute the H_∞ norm of the following 2-input, 2-output dynamic system and the frequency at which the peak singular value occurs.

$$G(s) = \begin{bmatrix} 0 & \dfrac{3s}{s^2+s+10} \\ \dfrac{s+1}{s+5} & \dfrac{2}{s+6} \end{bmatrix}.$$

```
G = [0 tf([3 0],[1 1 10]);tf([1 1],[1 5]),tf(2,[1 6])];
[ninf,fpeak] = hinfnorm(G)

ninf =

    3.0150


fpeak =

    3.1623
```

The H_∞ norm of a MIMO system is its maximum singular value. Plot the singular values of `G` and compare the result from `hinfnorm`.

```
sigma(G),grid
```



The values `ninf` and `fpeak` are consistent with the singular value plot, which displays the values in dB.

See Also

`getPeakGain` | `freqresp` | `sigma`

hinfstruct

Purpose

H_∞ tuning of fixed-structure controllers

Syntax

```
CL = hinfstruct(CL0)
[CL,gamma,info] = hinfstruct(CL0)
[CL,gamma,info] = hinfstruct(CL0,options)
[C,gamma,info] = hinfstruct(P,C0,options)
```

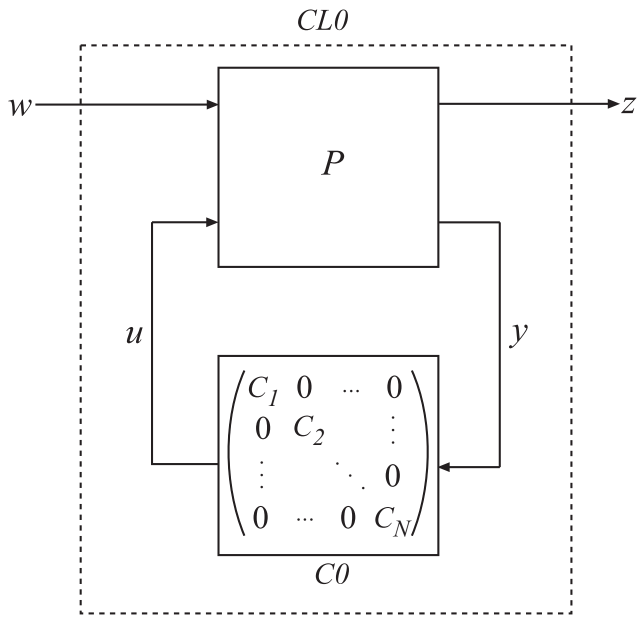
Description

`CL = hinfstruct(CL0)` tunes the free parameters of the tunable `genss` model `CL0`. This tuning minimizes the H_∞ norm of the closed-loop transfer function modeled by `CL0`. The model `CL0` represents a closed-loop control system that includes tunable components such as controllers or filters. `CL0` can also include weighting functions that capture design requirements.

`[CL,gamma,info] = hinfstruct(CL0)` returns `gamma` (the minimum H_∞ norm) and a data structure `info` with additional information about each optimization run.

`[CL,gamma,info] = hinfstruct(CL0,options)` allows you to specify additional options for the optimizer using `hinfstructOptions`.

`[C,gamma,info] = hinfstruct(P,C0,options)` tunes the parametric controller blocks `C0`. This tuning minimizes the H_∞ norm of the closed-loop system `CL0 = lft(P,C0)`. To use this syntax, express your control system and design requirements as a Standard Form model, as in the following illustration:



CL0
w
z
P
u
y
$\begin{pmatrix} C_1 & 0 & \cdots & 0 \\ 0 & C_2 & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & C_N \end{pmatrix}$
C0

`P` is a numeric LTI model that includes the fixed elements of the control architecture. `P` can also include weighting functions that capture design requirements. `C0` can be a single tunable component (for example, a Control Design Block or a `genss` model) or a cell array of multiple tunable components. `C` is a parametric model or array of parametric models of the same types as `C0`.

Tips

- `hinfstruct` is related to `hinfsyn`, which also uses H_∞ techniques to design a controller for a MIMO plant. However, unlike `hinfstruct`, `hinfsyn` imposes no restriction on the structure and order of the controller. For that reason, `hinfsyn` always returns a smaller `gamma` than `hinfstruct`. You can therefore use `hinfsyn` to obtain a lower bound on the best achievable performance.

Input Arguments

CL0

Generalized state-space (`genss`) model describing the weighted closed-loop transfer function of a control system. `hinfstruct` minimizes the H_∞ norm of `CL0`.

`CL0` includes both the fixed and tunable components of the control system in a single `genss` model. The tunable components of the control system are represented as Control Design Blocks, and are stored in the `CL0.Blocks` property of the `genss` model.

P

Numeric LTI model representing the fixed elements of the control architecture to be tuned. `P` can also include weighting functions that capture design requirements. You can obtain `P` in two ways:

- In MATLAB, model the fixed elements of your control system as numeric LTI models. Then, use block-diagram building functions (such as `connect` and `feedback`) to build `P` from the modeled components. Also include any weighting functions that represent your design requirements.
- If you have a Simulink model of your control system and have Simulink Control Design™, use `linlft` to obtain a linear model of the fixed elements of your control system. The `linlft` command linearizes your Simulink model, excluding specified Simulink blocks (the blocks that represent the controller elements you want to tune). If you are using weighting functions to represent your design requirements, connect them in series with the linear model of your plant to obtain `P`.

C0

Single tunable component or cell array of tunable components of the control structure.

Each entry in `C0` represents one tunable element of your control architecture, such as a PID controller, a gain block, or a fixed-order transfer function. The entries of `C0` can be Control Design Blocks or `genss` models.

For more information and examples of creating tunable models, see "Models with Tunable Coefficients" in the *Control System Toolbox™ User's Guide*.

options

Set of options for `hinfstruct`. Use `hinfstructOptions` to define `options`. For information about the available options, see the `hinfstructOptions` reference page.

Output Arguments

CL

Tuned version of the generalized state-space (`genss`) model `CL0`.

The `hinfstruct` command tunes the free parameters of `CL0` to achieve a minimum H_∞ norm. `CL.Blocks` contains the same types of Control Design Blocks as `CL0.Blocks`, except that in `CL`, the parameters have tuned values.

To access the tuned parameter values, use `getValue`. You can also access them directly in `CL.Blocks`.

C

Tuned versions of the parametric models `C0`.

When `C0` is a single parametric model, `C` is a parametric model of the same type, with tuned parameter values.

When `C0` is a cell array of parametric models, `C` is also a cell array. The entries in `C` are parametric models of the same type as the corresponding entries in `C0`.

gamma

Best achieved value for the closed-loop H_∞ norm.

In some cases, `hinfstruct` performs more than one minimization run (when the `hinfstructOptions` option `RandomStarts` > 0). In such cases, `gamma` is the smallest H_∞ norm of all runs.

info

Data structure array containing results from each optimization run. The fields of `info` are:

- `Objective` — Minimum H_∞ norm value for each run.

  When `RandomStarts` = 0, `Objective` = `gamma`.
- `Iterations` — Number of iterations before convergence for each run.
- `TunedBlocks` — Tuned control design blocks for each run.

  `TunedBlocks` differs from `C` in that `C` contains only the result from the best run. When `RandomStarts` = 0, `TunedBlocks` = `C`.

Algorithms

`hinfstruct` uses specialized nonsmooth optimization techniques to enforce closed-loop stability and minimize the H_∞ norm as a function of the tunable parameters. These techniques are based on the work in [1].

`hinfstruct` computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis," *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71-86.

[2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

See Also

`hinfstructOptions | hinfsyn | ltiblock.gain | ltiblock.pid | ltiblock.ss | ltiblock.tf | getValue | genss`

Related Examples

- “Build Tunable Closed-Loop Model for Tuning with hinfstruct”
- Loop Shaping Design with HINFSTRUCT
- Decoupling Controller for a Distillation Column
- Fixed-Structure Autopilot for a Passenger Jet

Concepts

- “What Is hinfstruct?”
- “Formulating Design Requirements as H-Infinity Constraints”
- “Structured H-Infinity Synthesis Workflow”
- “Models with Tunable Coefficients”

hinfstructOptions

Purpose

Set options for hinfstruct

Syntax

```
options = hinfstructOptions
options = hinfstructOptions(Name,Value)
```

Description

`options = hinfstructOptions` returns the default option set for the `hinfstruct` command.

`options = hinfstructOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

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

`hinfstructOptions` takes the following `Name` arguments:

'Display'

String determining the amount of information to display during `hinfstruct` optimization runs.

`Display` takes the following values:

- `'off'` — `hinfstruct` runs in silent mode, displaying no information during or after the run.
- `'iter'` — display optimization progress after each iteration. The display includes the value of the closed-loop H_∞ norm after each iteration. The display also includes a `Progress` value indicating the percent change in the H_∞ norm from the previous iteration.
- `'final'` — display a one-line summary at the end of each optimization run. The display includes the minimized value of the closed-loop H_∞ norm and the number of iterations for each run.

Default: `'final'`

'MaxIter'

Maximum number of iterations in each optimization run.

Default: 300

'RandomStart'

Number of additional optimizations starting from random values of the free parameters in the controller.

If `RandomStart = 0`, `hinfstruct` performs a single optimization run starting from the initial values of the tunable parameters. Setting `RandomStart = N > 0` runs *N* additional optimizations starting from *N* randomly generated parameter values.

`hinfstruct` finds a local minimum of the gain minimization problem. To increase the likelihood of finding parameter values that meet your design requirements, set `RandomStart > 0`. You can then use the best design that results from the multiple optimization runs.

Use with `UseParallel = true` to distribute independent optimization runs among MATLAB workers (requires Parallel Computing Toolbox™ software).

Default: 0

'UseParallel'

Parallel processing flag.

Set to `true` to enable parallel processing by distributing randomized starts among workers in a parallel pool. If there is an available parallel pool, then the software performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

- If **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, then the software starts a parallel pool using the settings in those preferences.
- If **Automatically create a parallel pool** is not selected in your preferences, then the software performs the optimization runs successively, without parallel processing.

If **Automatically create a parallel pool** is not selected in your preferences, you can manually start a parallel pool using `parpool` before running the tuning command.

Using parallel processing requires Parallel Computing Toolbox software.

Default: `false`

'TargetGain'

Target H_∞ norm.

The `hinfstruct` optimization stops when the H_∞ norm (peak closed-loop gain) falls below the specified `TargetGain` value.

Set `TargetGain = 0` to optimize controller performance by minimizing the peak closed-loop gain. Set `TargetGain = Inf` to just stabilize the closed-loop system.

Default: 0

'TolGain'

Relative tolerance for termination. The optimization terminates when the H_∞ norm decreases by less than `TolGain` over 10 consecutive iterations. Increasing `TolGain` speeds up termination, and decreasing `TolGain` yields tighter final values.

Default: 0.001

'MaxFrequency'

Maximum closed-loop natural frequency.

Setting `MaxFrequency` constrains the closed-loop poles to satisfy `|p| <  MaxFrequency`.

To let `hinfstruct` choose the closed-loop poles automatically based upon the system's open-loop dynamics, set `MaxFrequency = Inf`. To prevent unwanted fast dynamics or high-gain control, set `MaxFrequency` to a finite value.

Specify `MaxFrequency` in units of 1/`TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

Default: `Inf`

'MinDecay'

Minimum decay rate for closed-loop poles

Constrains the closed-loop poles to satisfy `Re(p) < -MinDecay`. Increase this value to improve the stability of closed-loop poles that do not affect the closed-loop gain due to pole/zero cancellations.

Specify `MinDecay` in units of 1/`TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

Default: `1e-7`

Output Arguments

options

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

Examples

Create Options Set for `hinfstruct`

Create an options set for a `hinfstruct` run using three random restarts and a stability offset of 0.001. Also, configure the `hinfstruct` run to stop as soon as the closed-loop gain is smaller than 1.

```
options = hinfstructOptions('TargetGain',1,...
                             'RandomStart',3,'StableOffset',1e-3);
```

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

```
options = hinfstructOptions;
options.TargetGain = 1;
options.RandomStart = 3;
options.StableOffset = 1e-3;
```

Configure Option Set for Parallel Optimization Runs

Configure an option set for a `hinfstruct` run using 20 random restarts. Execute these independent optimization runs concurrently on multiple workers in a parallel pool.

If you have the Parallel Computing Toolbox software installed, you can use parallel computing to speed up `hinfstruct` tuning of fixed-structure control systems. When you run multiple randomized `hinfstruct` optimization starts, parallel computing speeds up tuning by distributing the optimization runs among workers.

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences, manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create an `hinfstructOptions` set that specifies 20 random restarts to run in parallel.

```
options = hinfstructOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the `hinfstructOptions` set when you call `hinfstruct`. For example, suppose you have already created a tunable closed loop model

CLO. In this case, the following command uses parallel computing to tune CLO.

```
[CL,gamma,info] = hinfstruct(CLO,options);
```

See Also

hinfstruct

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,...)
```

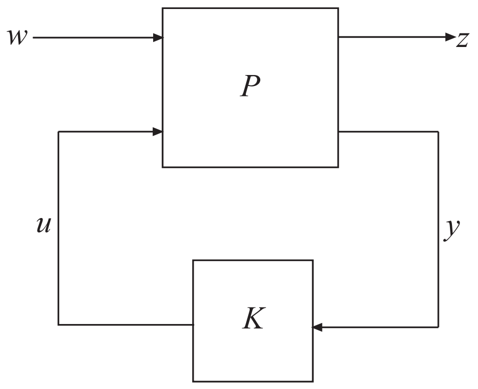
Description

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

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

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`. This closed-loop system is given by `CL = lft(P,K)` as in the following diagram.



The achieved H_∞ cost γ is returned as `GAM`. The `struct` array `INFO` contains additional information about the design.

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)
`'S0'`	real	Frequency `S0` 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'` `'on'`	No command window display, or command window displays synthesis progress information (default)

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 $T_{y_1 u_1}$
`GAM = norm(CL,Inf)`	H_∞ cost $\gamma = \left\| T_{y_1 u_1} \right\|_\infty$
`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}

Algorithms

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.,

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

where $T_{y_1u_1}$ is the closed-loop transfer function `CL`. 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 *bisection 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 γ^2.

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 $\omega \in R$.

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

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. The controller returned by `hinfsyn` is only one particular solution, `K`. When the `'ric'` method is selected, the `INFO.AS` field of `INFO` contains the all- solution controller parameterization K_{AS}. All solutions to the infinity-norm control problem are parameterized by

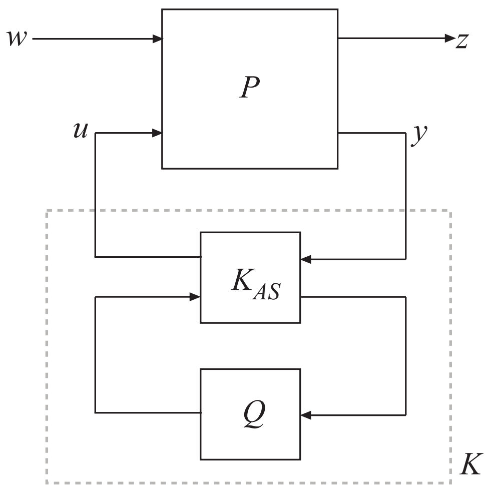
a free stable contraction map Q, which is constrained by $\|Q\|_\infty < 1$. In other words, the solutions include every stabilizing controller $K(s)$ that makes

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

These controllers are given by:

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

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



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 is typically 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`

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

The optimal H_∞ cost in this case is `GAM` = 1.3940. You verify

that $\|T_{y_1u_1}\|_\infty \triangleq \sup_{\omega} \sigma_{max}\left(T_{y_1u_1}(j\omega)\right) < \gamma$ with a sigma plot

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

Example 2: Mixed-Sensitivity

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

```
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 and detectable from the measurement output y:

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

Otherwise, 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 & Control Letters,* vol. 11, no. 8, 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 & Control Letters*, vol. 19, no. 4, pp. 271–280, 1992.

[5] Gahinet, P., and P. Apkarian, “A linear matrix inequality approach to H_∞-control,” *Int J. Robust and Nonlinear Control*, vol. 4, no. 4, pp. 421–448, 1994.

[6] Iwasaki, T., and R.E. Skelton, “All controllers for the general H_∞-control problem: LMI existence conditions and state space formulas,” *Automatica*, vol. 30, no. 8, pp. 1307–1317, 1994.

See Also

`augw` | `h2syn` | `hinfstruct` | `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
```

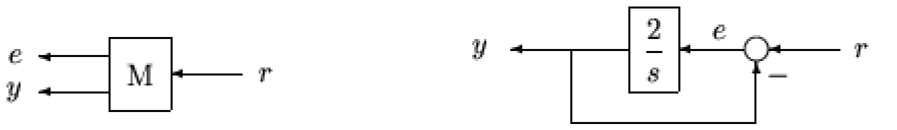
Description

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.

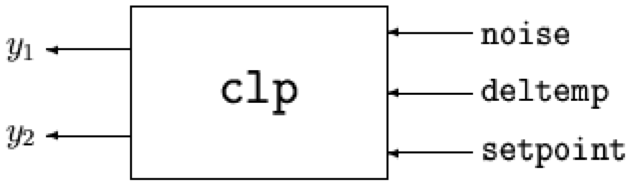
```
       s
 #1:  -----
      s + 2

       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]`.

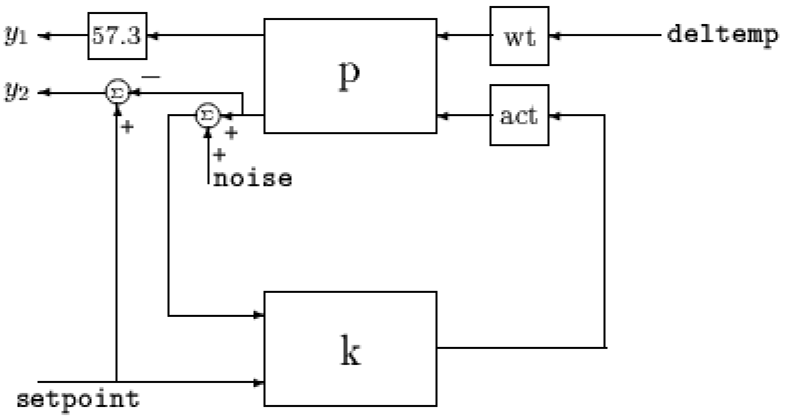
```
R = ureal('R',1,'Percentage',[-10 40]);
```

```
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.
```

Algorithms

Each equation represents an equality constraint among the variables. You choose the input and output variables, and the `imp2exp` function makes the implicit relationship between them explicit.

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`

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

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 H_∞ norm of the error between the approximate realization (*a, b, c, d*) and an exact realization of *y*; the order, say *n*, of the realization (*a, b, c, d*) is determined by the infinity norm error bound specified by the input variable `tol`. The inputs `ts, nu, ny, tol` are optional; if not present they default to the values `ts = 0, nu = 1, ny =` (number of rows of *y*)/`nu`, `tol` = $0.01\bar{\sigma}_1$. The output $hsv = [\bar{\sigma}_1, \bar{\sigma}_2, \ldots]'$ returns the singular values (arranged in descending order of magnitude) of the Hankel matrix:

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

Denoting by G_N a high-order exact realization of y, the low-order approximate model G enjoys the H_∞ norm bound

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

where

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

Algorithms

The realization (*a, b, c, d*) is computed using the Hankel SVD procedure proposed by Kung [2] as a method for approximately implementing the classical Hankel factorization realization algorithm. Kung's SVD realization procedure was subsequently shown to be equivalent to doing balanced truncation (`balmr`) on an exact state-space realization of the finite impulse response $\{y(1),....y(N)\}$ [3]. The infinity norm error bound for discrete balanced truncation was later derived by Al-Saggaf and Franklin [1]. The algorithm is as follows:

1 Form the Hankel matrix Γ from the data y.

2 Perform SVD on the Hankel matrix

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

where Σ_1 has dimension $n \times n$ and the entries of Σ_2 are nearly zero. U_1 and V_1 have *ny* and *nu* columns, respectively.

3 Partition the matrices U_1 and V_1 into three matrix blocks:

$$U1 = \begin{bmatrix} U_{11} \\ U_{12} \\ U_{13} \end{bmatrix} \begin{bmatrix} V_{11} \\ V_{12} \\ V_{13} \end{bmatrix}$$

where $U_{11}, U_{13} \in C^{ny \times n}$ and $V_{11}, V_{13} \in C^{nu \times n}$.

4 A discrete state-space realization is computed as

$$\begin{aligned} A &= \Sigma_1^{-\frac{1}{2}} \bar{U} \Sigma_1^{-\frac{1}{2}} \\ B &= \Sigma_1^{-\frac{1}{2}} V^*{}_{11} \\ C &= U_{11} \Sigma_1^{-\frac{1}{2}} \\ D &= H_0 \end{aligned}$$

where

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

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

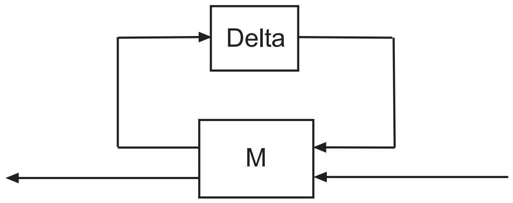
Syntax

```
[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`

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

Tips

Editable text areas have built-in scrolling capabilities. To activate the scroll mode, click in the text area, maintain the mouse button down, and move the mouse up or down. The scroll mode is only active when all visible lines have been used.

See Also

`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 side is zero, it is displayed as

```
N' * L(x) * N < 0
```

Information on the block structure and term content of `L(x)` and `R(x)` is also available. The term content of a block is symbolically displayed as

```
C1 + A1*X2*B1 + B1'*X2*A1' + a2*X1 + x3*Q1
```

with the following conventions:

- `X1, X2, x3` denote the problem variables. Upper-case `X` indicates matrix variables while lower-case `x` indicates scalar variables. The labels 1,2,3 refer to the first, second, and third matrix variable in the order of declaration.
- `Cj` refers to constant terms. Special cases are `I` and –`I` (`I` = identity matrix).

- `Aj, Bj` denote the left and right coefficients of variable terms. Lower-case letters such as `a2` indicate a scalar coefficient.
- `Qj` is used exclusively with scalar variables as in `x3*Q1`.

The index `j` in `Aj, Bj, Cj, Qj` is a dummy label. Hence `C1` may appear in several blocks or several LMIs without implying any connection between the corresponding constant terms. Exceptions to this rule are the notations `A1*X2*A1'` and `A1*X2*B1 + B1'*X2'*A1'` which indicate symmetric terms and symmetric pairs in diagonal blocks.

Examples

Consider the LMI

$$0 \begin{pmatrix} -2X + A^T YB + B^T Y^T A + I & XC \\ C^T X & -zI \end{pmatrix}$$

where the matrix variables are X of Type 1, Y of Type 2, and z scalar. If this LMI is described in `lmis`, information about X and the LMI block structure can be obtained as follows:

```
lmiinfo(lmis)

                  LMI ORACLE
                 ------

This is a system of 1 LMI with 3 variable matrices

Do you want information on
    (v) matrix variables     (l) LMIs     (q) quit

?> v

Which variable matrix (enter its index k between 1 and 3) ? 1
    X1 is a 2x2 symmetric block diagonal matrix
      its (1,1)-block is a full block of size 2

                      -------
```

```
This is a system of 1 LMI with 3 variable matrices
Do you want information on
    (v) matrix variables     (l) LMIs     (q) quit

?> l

Which LMI (enter its number k between 1 and 1) ? 1

    This LMI is of the form
            0 < R(x)
where the inner factor(s) has 2 diagonal block(s)


Do you want info on the right inner factor ?

    (w) whole factor     (b) only one block
    (o) other LMI        (t) back to top level

?> w

Info about the right inner factor

    block (1,1) : I + a1*X1 + A2*X2*B2 + B2'*X2'*A2'

    block (2,1) : A3*X1

    block (2,2) : x3*A4

    (w) whole factor     (b) only one block
    (o) other LMI        (t) back to top level

                     ------

This is a system of 1 LMI with 3 variable matrices

Do you want information on
```

```
    (v) matrix variables     (l) LMIs     (q) quit

?> q

It has been a pleasure serving you!
```

Note that the prompt symbol is ?> and that answers are either indices or letters. All blocks can be displayed at once with option `(w)`, or you can prompt for specific blocks with option `(b)`.

Tips

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

See Also

`decinfo` | `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`

lmireg

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.,

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

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`

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.

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

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

Recall that, by convention, the left side always refers to the smaller side of the LMI. The index `p` is relative to the order of declaration and corresponds to the identifier returned by `newlmi`.

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

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

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 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 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`

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 $\pm x$ where x_n is the n-th decision variable.

Accordingly, `struct` is a matrix of the same dimensions as X such that

- `struct(i,j)=0` if $X(i, j)$ is a hard zero
- `struct(i,j)=n` if $X(i, j) = x_n$
- `struct(i,j)= n` if $X(i, j) = -x_n$

Sophisticated matrix variable structures can be defined with Type 3. To specify a variable X of Type 3, first identify how many *free independent entries* are involved in X. These constitute the set of decision variables associated with X. If the problem already involves n decision variables, label the new free variables as $x_{n+1}, \ldots, x_{n+p}$. The structure of X is then defined in terms of $x_{n+1}, \ldots, x_{n+p}$ as indicated above. To help specify matrix variables of Type 3, `lmivar` optionally returns two extra outputs: (1) the total number n of scalar decision variables used so far and (2) a matrix `sX` showing the entry-wise dependence of X on the decision variables $x_1, \ldots, x_n$.

Examples

Example 1

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

- X_1 is a 3-by-3 symmetric matrix (unstructured),
- X_2 is a 2-by-4 rectangular matrix (unstructured),
- X_3 =

$$\begin{pmatrix} \Delta & 0 & 0 \\ 0 & \delta_1 & 0 \\ 0 & 0 & \delta_2 I_2 \end{pmatrix}$$

  where Δ is an arbitrary 5-by-5 symmetric matrix, δ_1 and δ_2 are scalars, and I_2 denotes the identity matrix of size 2.

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`

Purpose

Stability margin analysis of LTI and Simulink feedback loops

Syntax

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

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

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

Description

`[cm,dm,mm] = loopmargin(L)` analyzes the multivariable feedback loop consisting of the loop transfer matrix `L` (size *N*-by-*N*) in negative feedback with an *N*-by-*N* identity matrix.

`cm`, or classical gain and phase margins, is an *N*-by-1 structure corresponding to loop-at-a-time gain and phase margins for each channel (See `allmargin` for details on the fields of `cm`.)

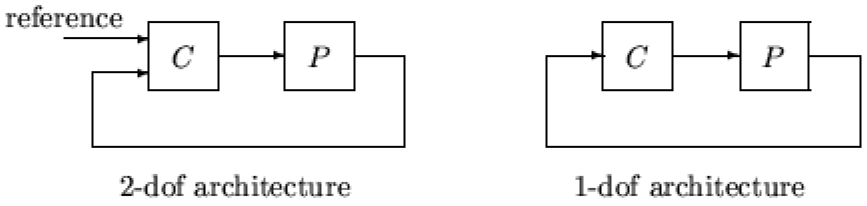
`dm` is an *N*-by-1 structure corresponding to loop-at-a-time disk gain and phase margins for each channel. The disk margin for the i-th feedback channel defines a circular region centered on the negative real axis at the average GainMargin (`GM`), e.g. , (GM_{low}+GM_{high})/2, such that `L(i,i)` does 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] = loopmargin(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.



`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] = loopmargin(Model,Blocks,Ports)` does a multi-loop stability margin analysis using Simulink Control Design software. `Model` specifies the name of the Simulink diagram for analysis. The margin analysis points are defined at the output ports (`Ports`) of blocks (`Blocks`) within the model. `Blocks` is a cell array of full block path names and `Ports` is a vector of the same dimension as `Blocks`. If all `Blocks` have a single output port, then `Ports` would be a vector of ones with the same length as `Blocks`.

Three types of stability margins are computed: loop-at-a-time classical gain and phase margins (`cm`), loop-at-a-time disk margins (`dm`) and a multi-loop disk margin (`mm`).

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

`[cm,dm,mm,info] = loopmargin(Model,Blocks,Ports,OP)` returns `info` in addition to the margins. `info` is a structure with fields `OperatingPoint`, `LinearizationIO` and `SignalNames` corresponding to the analysis.

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] = loopmargin(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] = loopmargin(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 radians-per-second)
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 radians-per-second)
DelayMargin	Delay margins (in seconds for continuous-time systems, and multiples of the sample time for discrete-time systems)
Stable	1 if nominal closed loop is stable, 0 otherwise. If L is a frd or ufrd object, the Stable flag is set to NaN.

dm, or Disk Margin, is 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 (in radians-per-second)

mm is a structure with the following fields.

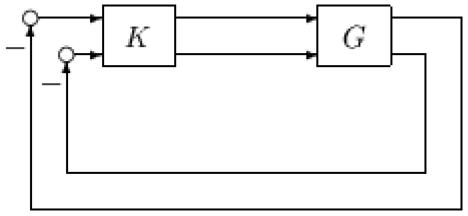
Field	Description
`GainMargin`	Guaranteed bound on simultaneous, independent, gain variations allowed in all plant channels
`PhaseMargin`	Guaranteed bound on simultaneous, independent, phase variations allowed in all plant channels (degrees)
`Frequency`	Associated with `GainMargin`/`PhaseMargin` fields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

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

$$G = \frac{1}{s^2+\alpha^2}\begin{bmatrix} s-\alpha^2 & \alpha(s+1) \\ -\alpha(s+1) & s-\alpha^2 \end{bmatrix},\ K = I_2$$

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] = loopmargin(G,K);
```

First consider the margins at the input to the plant. The first input channel has infinite gain margin and 90 degrees of phase margin based on the results from the `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
```

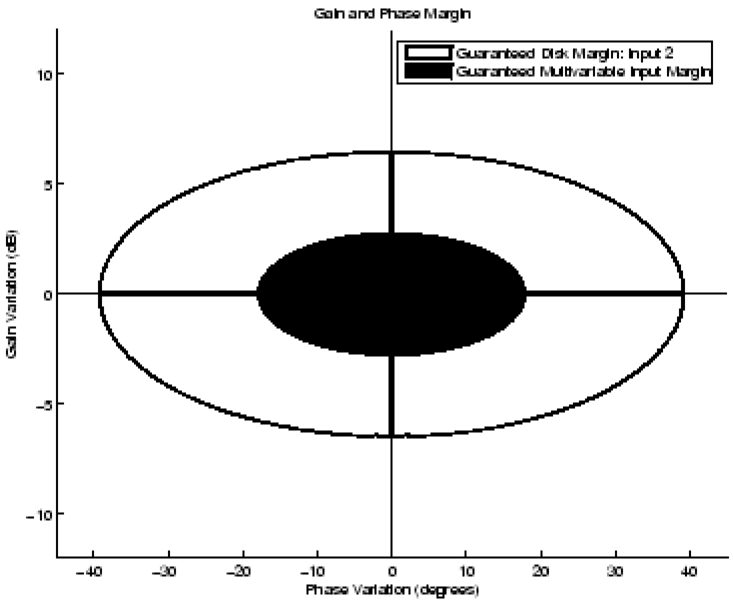
The second input channel has a gain margin of 2.105 and infinite phase margin based on the single-loop analysis, `cmi(2)`. The disk margin analysis, `dmi(2)`, which allows for simultaneous gain and phase variations a loop-at-a-time results in maximum gain margin variations of 0.475 and 2.105 and phase 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.

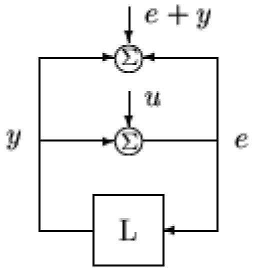
```
mmo
mmo =
     GainMargin: [0.6065 1.6489]
    PhaseMargin: [-27.5293 27.5293]
      Frequency: 0.2287
```

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
```

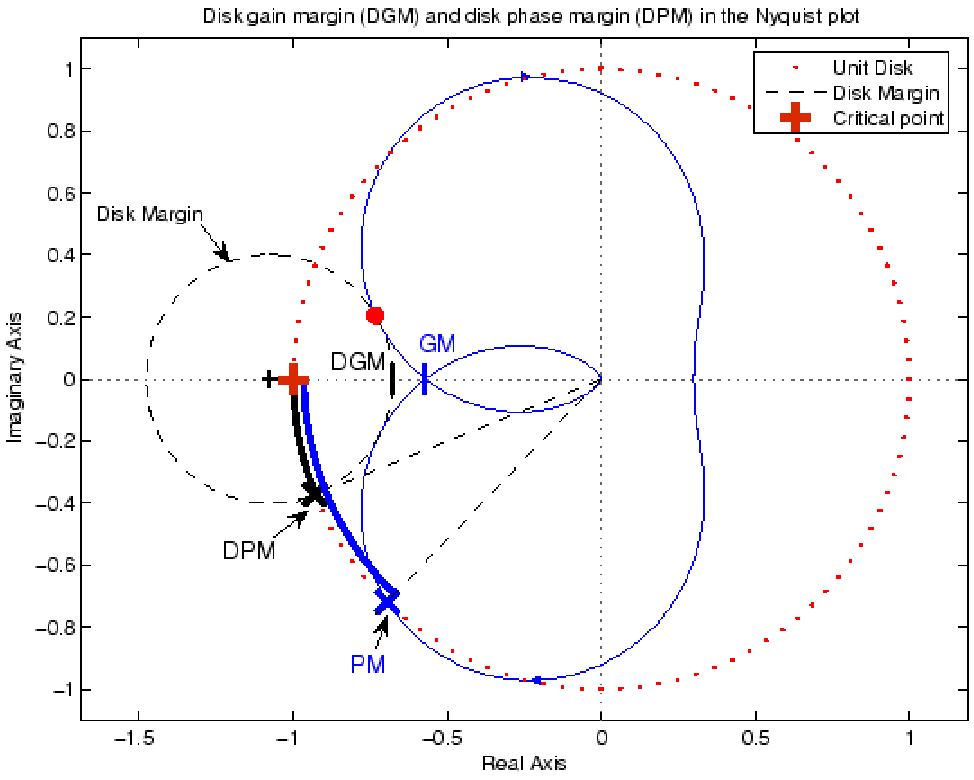
Algorithms

Two well-known loop robustness measures are based on the sensitivity function $S=(I-L)^{-1}$ and the complementary sensitivity function $T=L(I-L)^{-1}$ where L is the loop gain matrix associated with the input or output loops broken simultaneously. In the following figure, S is the transfer matrix from summing junction input u to summing junction output e. T is the transfer matrix from u to y. If signals e and y are summed, the transfer matrix from u to $e+y$ is given by $(I+L)\cdot(I-L)^{-1}$, the balanced sensitivity function. It can be shown (Dailey, 1991, Blight, Daily and Gangass, 1994) that each broken-loop gain can be perturbed by the complex gain $(1+\Delta)(1-\Delta)$ where $|\Delta|<1/\mu(S+T)$ or $|\Delta|<1/\sigma_{max}(S+T)$ at each frequency without causing instability at that frequency. The peak value of $\mu(S+T)$ or $\sigma_{max}(S+T)$ gives a robustness guarantee for all frequencies, and for $\mu(S+T)$ the guarantee is nonconservative (Blight, Daily and Gangass, 1994).



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

This figure shows a comparison of a disk margin analysis with the classical notations of gain and phase margins.



The Nyquist plot is of the loop transfer function $L(s)$

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

- The Nyquist plot of L corresponds to the blue line.
- The unit disk corresponds to the dotted red line.
- GM and PM indicate the location of the classical gain and phase margins for the system L.

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

The disk margin and multiple channel margins calculation involve the balanced sensitivity function $S+T$. For a given peak value of $\mu(S+T)$, 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

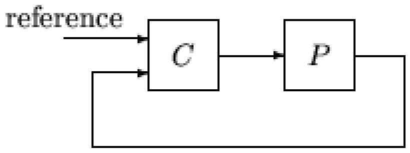
Sensitivity functions of plant-controller feedback loop

Syntax

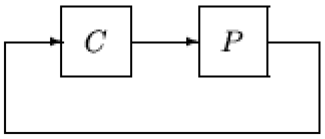
```
loops = loopsens(P,C)
```

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



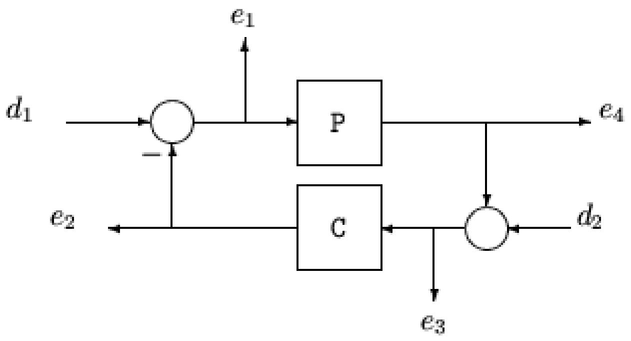
1-dof architecture

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

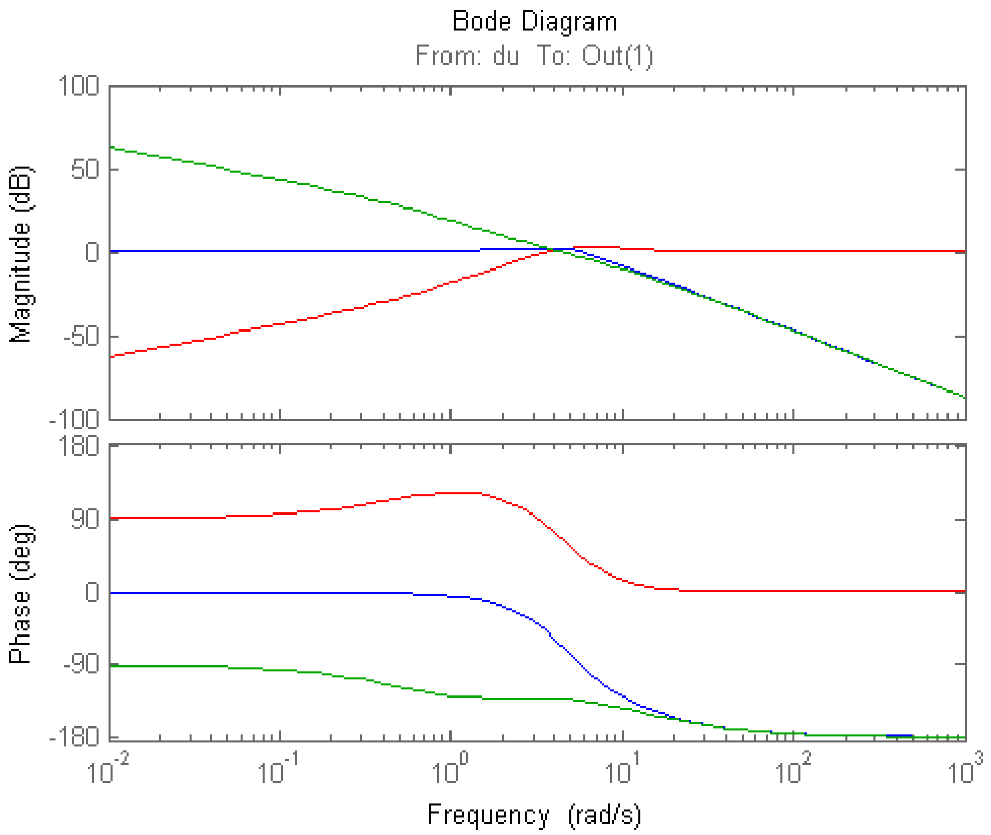
Single Input, Single Output (SISO) Loop Sensitivities

Consider PI controller for a dominantly 1st-order plant, with the closed-loop bandwidth of 2.5 rads/sec. Since the problem is SISO, all gains are the same at input and output.

```
gamma = 2; tau = 1.5; taufast = 0.1;
P = tf(gamma,[tau 1])*tf(1,[taufast 1]);
tauclp = 0.4;
xiclp = 0.8;
wnclp = 1/(tauclp*xiclp);
KP = (2*xiclp*wnclp*tau - 1)/gamma;
KI = wnclp^2*tau/gamma;
C = tf([KP KI],[1 0]);
```

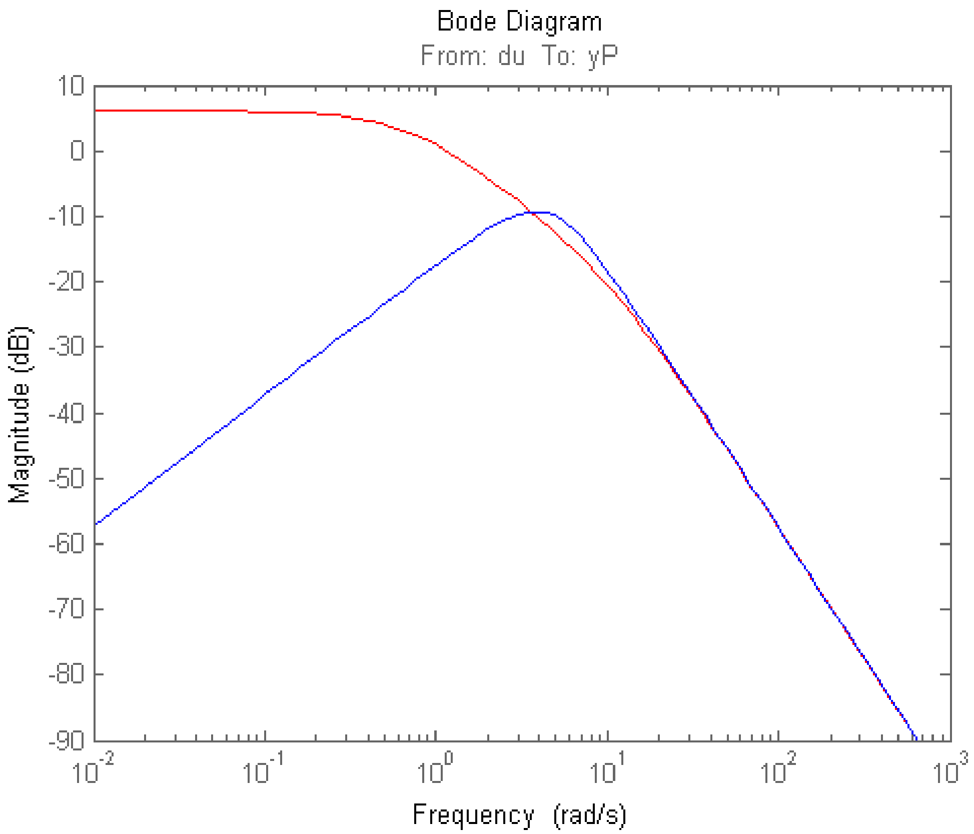
Form the closed-loop (and open-loop) systems with `loopsens`, and plot Bode plots using the gains at the plant input.

```
loops = loopsens(P,C);
bode(loops.Si,'r',loops.Ti,'b',loops.Li,'g')
```



Finally, compare the open-loop plant gain to the closed-loop value of `PSi`.

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

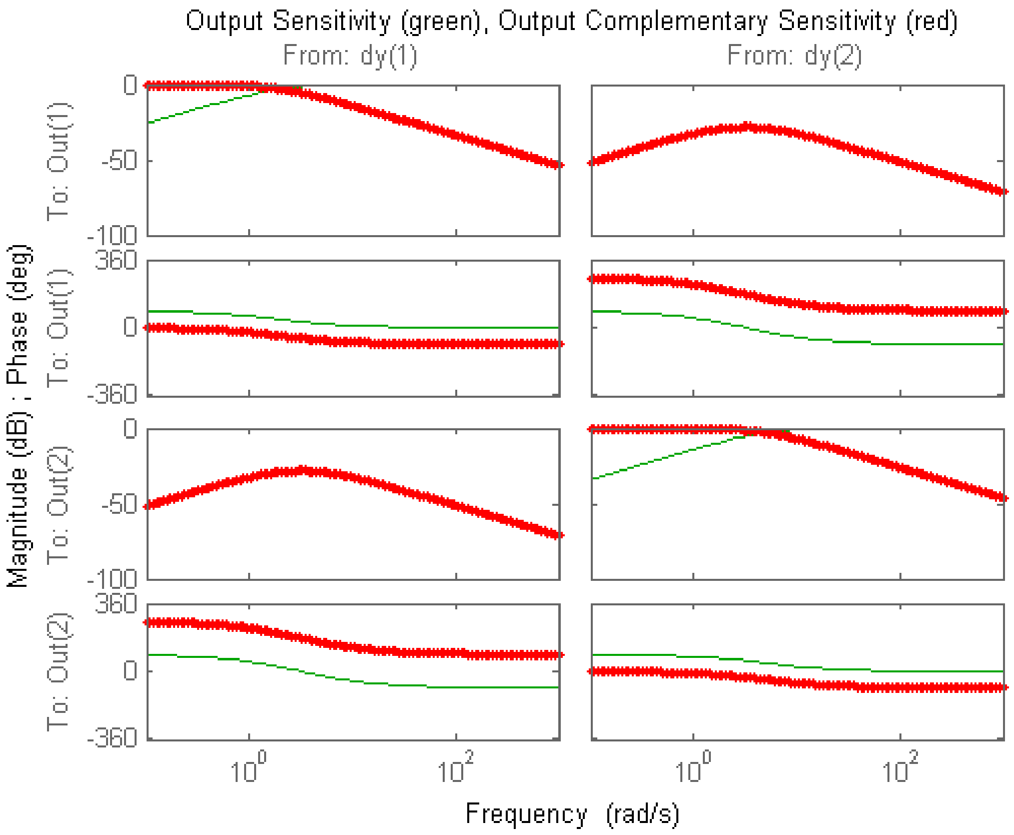


Multi Input, Multi Output (MIMO) Loop Sensitivities

Consider an integral controller for a constant-gain, 2-input, 2-output plant. For purposes of illustration, the controller is designed via inversion, with different bandwidths in each rotated channel.

```
P = ss([2 3;-1 1]);
BW = diag([2 5]);
[U,S,V] = svd(P.d);                % get SVD of Plant Gain
Csvd = V*inv(S)*BW*tf(1,[1 0])*U'; % inversion based on SVD
```

```
loops = loopsens(P,Csvd);
bode(loops.So,'g',loops.To,'r.',logspace(-1,3,120))
title('Output Sensitivity (green), Output Complementary Sensitivity (r
```



See Also

loopmargin | robuststab | wcsens | wcmargin

loopsyn

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

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

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

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

Output Argument	Description
K	LTI controller
CL= G*K/(I+GK)	LTI closed-loop system
GAM	Loop-shaping accuracy (GAM ≥ 1, with GAM=1 being perfect fit
INFO	Additional output information
INFO.W	LTI pre-filter W satisfying $\sigma(G_d) = \sigma\ (GW)$ for all ω; W is always minimum-phase.
INFO.Gs	LTI shaped plant: $G_s = GW$.
INFO.Ks	LTI controller for the shaped plant: $K_s = WK$.
INFO.range	$\{\omega_{\min},\omega_{\max}\}$ 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.

Algorithms

Using the GCD formula of Le and Safonov [1], loopsyn first computes a stable-minimum-phase loop-shaping, squaring-down prefilter W such that the shaped plant $G_s = GW$ is square, and the desired shape G_d is achieved with good accuracy in the frequency range $\{\omega_{\min},\omega_{\max}\}$ by the shaped plant; i.e.,

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

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 $j\omega$-axis, and
3. $\{\omega_{min},\omega_{max}\}=[0,\infty]$,

then GW theoretically achieves a perfect accuracy fit $\sigma(G_d) = \sigma(GW)$ for all frequency ω. Otherwise, `loopsyn` uses a bilinear pole-shifting bilinear transform [3] of the form

```
Gshifted=bilin(G,-1,'S_Tust',[ømin,ømax]),
```

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

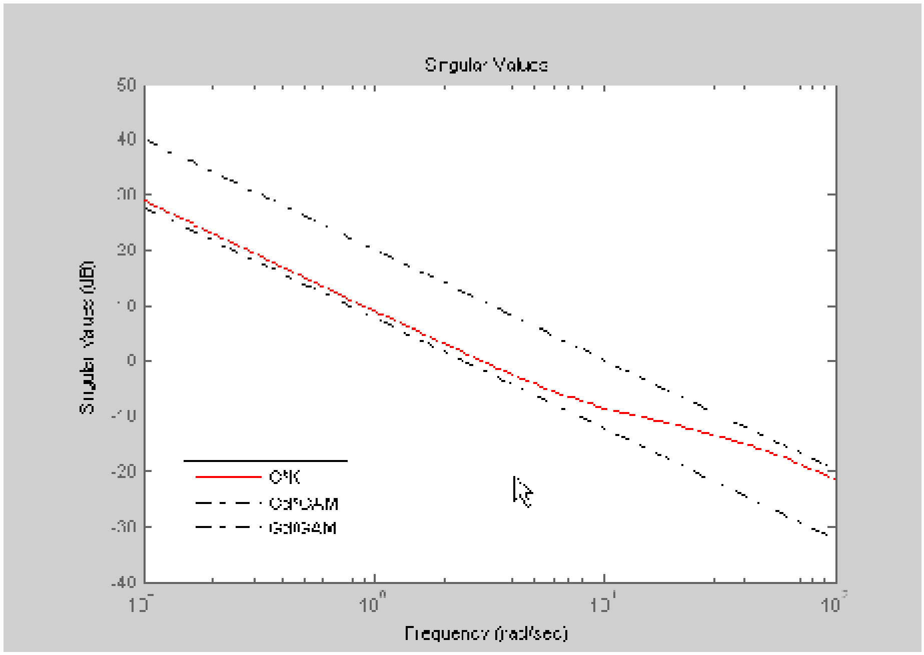
The following code generates the optimal `loopsyn` loopshaping control for the case of a 5-state, 4-output, 5-input plant with a full-rank non-minimum phase zero at $s = +10$. The result is shown in LOOPSYN controller on page 2-229.

```
rng(0,'twister');
s=tf('s'); w0=5; Gd=5/s;          % desired bandwidth w0=5
G=((s-10)/(s+100))*rss(3,4,5);    % 4-by-5 non-min-phase plant
[K,CL,GAM,INFO]=loopsyn(G,Gd);
sigma(G*K,'r',Gd*GAM,'k-.',Gd/GAM,'k-.',{.1,100})  % plot result
```

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`.

References

[1] Le, V.X., and M.G. Safonov. Rational matrix GCD's and the design of squaring-down compensators—a state space theory. *IEEE Trans. Autom.Control*, AC-36(3):384–392, March 1992.

[2] Glover, K., and D. McFarlane. Robust stabilization of normalized coprime factor plant descriptions with H_∞-bounded uncertainty. *IEEE Trans. Autom. Control*, AC-34(8):821–830, August 1992.

[3] Chiang, R.Y., and M.G. Safonov. H_∞ synthesis using a bilinear pole-shifting transform. *AIAA J. Guidance, Control and Dynamics*, 15(5):1111–1115, September–October 1992.

See Also

`mixsyn` | `ncfsyn`

How To

- Loop Shaping of HIMAT Pitch Axis Controller

Purpose

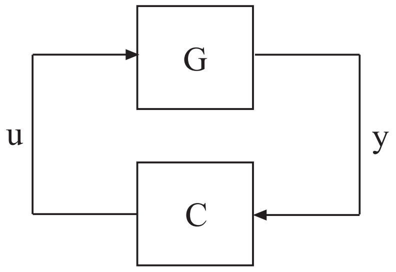
Tune fixed-structure feedback loops

Syntax

```
[G,C,gam] = looptune(G0,C0,wc)
[G,C,gam] = looptune(G0,C0,wc,Req1,...,ReqN)
[G,C,gam] = looptune(...,options)
[G,C,gam,info] = looptune(...)
```

Description

`[G,C,gam] = looptune(G0,C0,wc)` tunes the feedback loop



to meet the following default requirements:

- Bandwidth — Gain crossover for each loop falls in the frequency interval `wc`
- Performance — Integral action at frequencies below `wc`
- Robustness — Adequate stability margins and gain roll-off at frequencies above `wc`

The tunable `genss` model `C0` specifies the controller structure, parameters, and initial values. The model `G0` specifies the plant. `G0` can be a Numeric LTI model, or, for co-tuning the plant and controller, a tunable `genss` model. The sensor signals `y` (measurements) and actuator signals `u` (controls) define the boundary between plant and controller.

`[G,C,gam] = looptune(G0,C0,wc,Req1,...,ReqN)` tunes the feedback loop to meet additional design requirements specified in one or more tuning goal objects `Req1,...,ReqN`. Omit `wc` to use the requirements specified in Req1,...,ReqN instead of an explicit target crossover frequency and the default performance and robustness requirements.

[G,C,gam] = looptune(...,options) specifies further options, including target gain margin, target phase margin, and computational options for the tuning algorithm.

[G,C,gam,info] = looptune(...) returns a structure info with additional information about the tuned result. Use info with the loopview command to visualize tuning constraints and validate the tuned design.

Input Arguments

G0

Numeric LTI model or tunable genss model representing plant in control system to tune.

The plant is the portion of your control system whose outputs are sensor signals (measurements) and whose inputs are actuator signals (controls). Use connect to build G0 from individual numeric or tunable components.

C0

Generalized LTI model representing controller. C0 specifies the controller structure, parameters, and initial values.

The controller is the portion of your control system that receives sensor signals (measurements) as inputs and produces actuator signals (controls) as outputs. Use Control Design Blocks and Generalized LTI models to represent tunable components of the controller. Use connect to build C0 from individual numeric or tunable components.

wc

Vector specifying target crossover region [wcmin,wcmax]. The looptune command attempts to tune all loops in the control system so that the open-loop gain crosses 0 dB within the target crossover region.

A scalar wc specifies the target crossover region [wc/2,2*wc].

Req1,...,ReqN

One or more `TuningGoal` objects specifying design requirements. Available requirement types are:

- `TuningGoal.Tracking` — Setpoint tracking requirement
- `TuningGoal.Gain` — Limit on transfer function gain
- `TuningGoal.LoopShape` — Target shape for open-loop response

For a complete list of the design requirements you can specify, see "Performance and Robustness Specifications for looptune".

options

Set of options for `looptune` algorithm, specified using `looptuneOptions`. See `looptuneOptions` for information about the available options, including target gain margin and phase margin.

Output Arguments

G

Tuned plant.

If `G0` is a Numeric LTI model, `G` is the same as `G0`.

If `G0` is a tunable `genss` model, `G` is a `genss` model with Control Design Blocks of the same number and types as `G0`. The current value of `G` is the tuned plant.

C

Tuned controller. `C` is a `genss` model with Control Design Blocks of the same number and types as `C0`. The current value of `C` is the tuned controller.

gam

Parameter indicating degree of success at meeting all tuning constraints. A value of `gam <= 1` indicates that all requirements are satisfied. `gam >> 1` indicates failure to meet at least one requirement.

Use `loopview` to visualize the tuned result and identify the unsatisfied requirement.

For best results, use the `RandomStart` option in `looptuneOptions` to obtain several minimization runs. Setting `RandomStart` to an integer `N > 0` causes `looptune` to run the optimization `N` additional times, beginning from parameter values it chooses randomly. You can examine `gam` for each run to help identify an optimization result that meets your design requirements.

info

Data for validating tuning results, returned as a structure. To use the data in `info`, use the command `loopview(G,C,info)` to visualize tuning constraints and validate the tuned design.

`info` contains the following tuning data:

Di,Do

Optimal input and output scalings, returned as state-space models. The scaled plant is given by `Do\G*Di`.

Specs

Design requirements that `looptune` constructs for its call to `systune` for tuning (see "Algorithms" on page 2-236), returned as a vector of `TuningGoal` requirement objects.

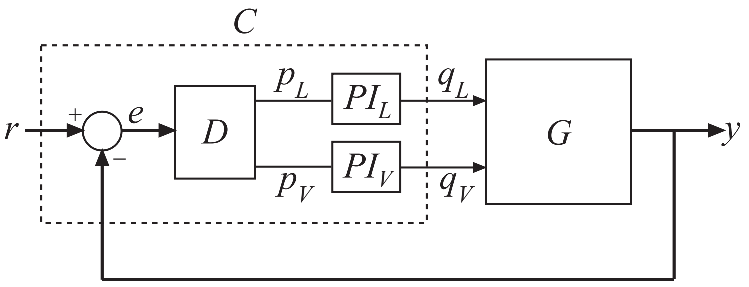
Runs

Detailed information about each optimization run performed by `systune` when called by `looptune` for tuning (see "Algorithms" on page 2-236), returned as a data structure.

The contents of `Runs` are the `info` output of the call to `systune`. For information about the fields of `Runs`, see the `info` output argument description on the `systune` reference page.

Examples

Tune the control system of the following illustration, to achieve crossover between 0.1 and 1 rad/min.



The 2-by-2 plant `G` is represented by:

$$G(s) = \frac{1}{75s+1}\begin{bmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{bmatrix}.$$

The fixed-structure controller, `C`, includes three components: the 2-by-2 decoupling matrix `D` and two PI controllers `PI_L` and `PI_V`. The signals `r`, `y`, and `e` are vector-valued signals of dimension 2.

Build a numeric model that represents the plant and a tunable model that represents the controller. Name all inputs and outputs as in the diagram, so that `looptune` knows how to interconnect the plant and controller via the control and measurement signals.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL','qV'};
G.OutputName = 'y';

D = ltiblock.gain('Decoupler',eye(2));
D.InputName = 'e';
D.OutputName = {'pL','pV'};
PI_L = ltiblock.pid('PI_L','pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';
```

```
PI_V = ltiblock.pid('PI_V','pi');
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';
sum1 = sumblk('e = r - y',2);
C0 = connect(PI_L,PI_V,D,sum1,{'r','y'},{'qL','qV'});

wc = [0.1,1];
[G,C,gam,info] = looptune(G,C0,wc);
```

`C` is the tuned controller, in this case a `genss` model with the same block types as `C0`.

You can examine the tuned result using `loopview`.

Algorithms

`looptune` automatically converts target bandwidth, performance requirements, and additional design requirements into weighting functions that express the requirements as an H_∞ optimization problem. `looptune` then uses `systune` to optimize tunable parameters to minimize the H_∞ norm. For more information about the optimization algorithms, see [1].

`looptune` computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis." *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71–86.

[2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, 14 (1990), pp. 287-293.

Alternatives

For tuning Simulink models with `looptune`, see `slTunable` and `slTunable.looptune` (requires Simulink Control Design).

See Also

`TuningGoal.Tracking` | `slTunable` | `systune` | `slTunable.looptune` | `TuningGoal.Gain` | `TuningGoal.LoopShape` | `hinfstruct` | `looptuneOptions` | `loopview` | `loopmargin` | `genss` | `connect`

Tutorials

- "Tune MIMO Control System for Specified Bandwidth"
- "Tuning Feedback Loops with LOOPTUNE"
- "Decoupling Controller for a Distillation Column"

How To

- "Performance and Robustness Specifications for looptune"

looptuneOptions

Purpose

Set options for looptune

Syntax

```
options = looptuneOptions
options = looptuneOptions(Name,Value)
```

Description

`options = looptuneOptions` returns the default option set for the `looptune` command.

`options = looptuneOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

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

`looptuneOptions` takes the following `Name` arguments:

'GainMargin'

Target gain margin in decibels. `GainMargin` specifies the required gain margin for the tuned control system. For MIMO control systems, the gain margin is the multiloop disk margin. See `loopmargin` for the definition of the multiloop disk margin.

Default: 7.6 dB

'PhaseMargin'

Target phase margin in degrees. `PhaseMargin` specifies the required phase margin for the tuned control system. For MIMO control systems, the phase margin is the multiloop disk margin. See `loopmargin` for the definition of the multiloop disk margin.

Default: 45 degrees

'Display'

String determining the amount of information to display during `looptune` runs.

`Display` takes the following values:

- `'off'` — Run in silent mode, displaying no information during or after the run.
- `'iter'` — Display optimization progress after each iteration. The display includes the value of the objective parameter `gam` after each iteration. The display also includes a `Progress` value, indicating the percent change in `gam` from the previous iteration.
- `'final'` — Display a one-line summary at the end of each optimization run. The display includes the minimized value of `gam` and the number of iterations for each run.

Default: `'final'`

'MaxIter'

Maximum number of iterations in each optimization run.

Default: 300

'RandomStart'

Number of additional optimizations starting from random values of the free parameters in the controller.

If `RandomStart = 0`, `looptune` performs a single optimization run starting from the initial values of the tunable parameters. Setting `RandomStart = N > 0` runs *N* additional optimizations starting from *N* randomly generated parameter values.

`looptune` tunes by finding a local minimum of a gain minimization problem. To increase the likelihood of finding parameter values that meet your design requirements, set `RandomStart > 0`. You can then use the best design that results from the multiple optimization runs.

Use with `UseParallel = true` to distribute independent optimization runs among MATLAB workers (requires Parallel Computing Toolbox software).

Default: 0

'UseParallel'

Parallel processing flag.

Set to `true` to enable parallel processing by distributing randomized starts among workers in a parallel pool. If there is an available parallel pool, then the software performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

- If **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, then the software starts a parallel pool using the settings in those preferences.
- If **Automatically create a parallel pool** is not selected in your preferences, then the software performs the optimization runs successively, without parallel processing.

If **Automatically create a parallel pool** is not selected in your preferences, you can manually start a parallel pool using `parpool` before running the tuning command.

Using parallel processing requires Parallel Computing Toolbox software.

Default: `false`

'TargetGain'

Target value for the objective parameter `gam`.

The `looptune` command converts your design requirements into normalized gain constraints. The command then tunes the free parameters of the control system to drive the objective parameter `gam` below 1 to enforce all requirements.

The default `TargetGain = 1` ensures that the optimization stops as soon as `gam` falls below 1. Set `TargetGain` to a smaller or larger value to continue the optimization or start sooner, respectively.

Default: 1

'TolGain'

Relative tolerance for termination.

The optimization terminates when the objective parameter `gam` decreases by less than `TolGain` over 10 consecutive iterations. Increasing `TolGain` speeds up termination, and decreasing `TolGain` yields tighter final values.

Default: 0.001

'MaxFrequency'

Maximum closed-loop natural frequency.

Setting `MaxFrequency` constrains the closed-loop poles to satisfy `|p| <  MaxFrequency`.

To allow `looptune` to choose the closed-loop poles automatically, based upon the system's open-loop dynamics, set `MaxFrequency = Inf`. To prevent unwanted fast dynamics or high-gain control, set `MaxFrequency` to a finite value.

Specify `MaxFrequency` in units of 1/`TimeUnit`, relative to the `TimeUnit` property of the system you are tuning.

Default: `Inf`

'MinDecay'

Minimum decay rate for closed-loop poles

Constrains the closed-loop poles to satisfy `Re(p) < -MinDecay`. Increase this value to improve the stability of closed-loop poles that do not affect the closed-loop gain due to pole/zero cancellations.

Specify MinDecay in units of 1/TimeUnit, relative to the TimeUnit property of the system you are tuning.

Default: `1e-7`

Output Arguments

options

Option set containing the specified options for the looptune command.

Examples

Create Options Set for looptune

Create an options set for a looptune run using three random restarts. Also, set the target gain and phase margins to 6 dB and 50 degrees, respectively, and limit the closed-loop pole magnitude to 100.

```
options = looptuneOptions('RandomStart',3','GainMargin',6,...
                'PhaseMargin',50,'SpecRadius',100);
```

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

```
options = looptuneOptions;
options.RandomStart = 3;
options.GainMargin = 6;
options.PhaseMargin = 50;
options.SpecRadius = 100;
```

Configure Option Set for Parallel Optimization Runs

Configure an option set for a looptune run using 20 random restarts. Execute these independent optimization runs concurrently on multiple workers in a parallel pool.

If you have the Parallel Computing Toolbox software installed, you can use parallel computing to speed up looptune tuning of fixed-structure control systems. When you run multiple randomized looptune optimization starts, parallel computing speeds up tuning by distributing the optimization runs among workers.

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences, manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a `looptuneOptions` set that specifies 20 random restarts to run in parallel.

```
options = looptuneOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the `looptuneOptions` set when you call `looptune`. For example, suppose you have already created a plant model `G0` and tunable controller `C0`. In this case, the following command uses parallel computing to tune the control system of `G0` and `C0` to the target crossover`wc`.

```
[G,C,gamma] = looptune(G0,C0,wc,options);
```

See Also

| `looptune` | `loopmargin`

looptuneSetup

Purpose

Convert tuning setup for `looptune` to tuning setup for `systune`

Syntax

```
[T0,SoftReqs,HardReqs,sysopt] =
looptuneSetup(looptuneInputs)
```

Description

`[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(looptuneInputs)` converts a tuning setup for `looptune` into an equivalent tuning setup for `systune`. The argument `looptuneInputs` is a sequence of input arguments for `looptune` that specifies the tuning setup. For example,

```
[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(G0,C0,wc,Req1,Req2,loopopt)
```

generates a set of arguments such that `looptune(G0,C0,wc,Req1,Req2,loopopt)` and `systune(T0,SoftReqs,HardReqs,sysopt)` produce the same results.

Use this command to take advantage of additional flexibility that `systune` offers relative to `looptune`. For example, `looptune` requires that you tune all channels of a MIMO feedback loop to the same target bandwidth. Converting to `systune` allows you to specify different crossover frequencies and loop shapes for each loop in your control system. Also, `looptune` treats all tuning requirements as soft requirements, optimizing them but not requiring that any constraint be exactly met. Converting to `systune` allows you to enforce some tuning requirements as hard constraints, while treating others as soft requirements.

You can also use this command to probe into the tuning requirements used by `looptune`.

Input Arguments

looptuneInputs - Plant, controller, and requirement inputs to `looptune`
valid `looptune` input sequence

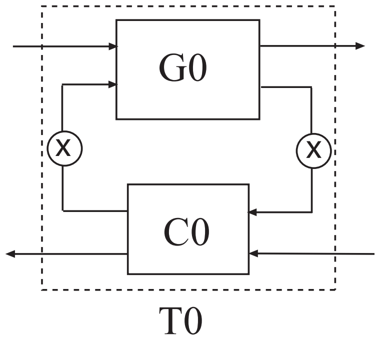
Plant, controller, and requirement inputs to `looptune`, specified as a valid `looptune` input sequence. For more information about the arguments in a valid `looptune` input sequence, see the `looptune` reference page.

Output Arguments

T0 - Closed-loop control system model

generalized state-space model

Closed-loop control system model for tuning with `systune`, returned as a generalized state-space `genss` model. To compute `T0`, the plant, `G0`, and the controller, `C0`, are combined in the feedback configuration of the following illustration.



The connections between `C0` and `G0` are determined by matching signals using the `InputName` and `OutputName` properties of the two models. In general, the signal lines in the diagram can represent vector-valued signals. `loopswitch` switches, indicated by X in the diagram, are inserted between the controller and the plant. This allows definition of open-loop and closed-loop requirements on signals injected or measured at the plant inputs or outputs. For example, the bandwidth `wc` is converted into a `TuningGoal.LoopShape` requirement that imposes the desired crossover on the open-loop signal measured at the plant input.

For more information on the structure of closed-loop control system models for tuning with `systune`, see the `systune` reference page.

SoftReqs - Soft tuning requirements

vector of `TuningGoal` requirement objects

Soft tuning requirements for tuning with `systune`, specified as a vector of `TuningGoal` requirement objects.

`looptune` expresses most of its implicit tuning requirements as soft tuning requirements. For example, a specified target loop bandwidth is expressed as a `TuningGoal.LoopShape` requirement with integral gain profile and crossover at the target frequency. Additionally, `looptune` treats all of the explicit requirements you specify (`Req1,...ReqN`) as soft requirements. `SoftReqs` contains all of these tuning requirements.

HardReqs - Hard tuning requirements

vector of `TuningGoal` requirement objects

Hard tuning requirements (constraints) for tuning with `systune`, specified as a vector of `TuningGoal` requirement objects.

Because `looptune` treats most tuning requirements as soft requirements, `HardReqs` is usually empty. However, if you change the default `MaxFrequency` option of the `looptuneOptions` set, `loopopt`, then this requirement appears as a hard `TuningGoal.Poles` constraint.

sysopt - Algorithm options for `systune` tuning

`systuneOptions` options set

Algorithm options for `systune` tuning, specified as a `systuneOptions` options set.

Some of the options in the `looptuneOptions` set, `loopopt`, are expressed as hard or soft requirements that are returned in `HardReqs` and `SoftReqs`. Other options correspond to options in the `systtuneOptions` set.

Examples

Convert looptune Problem into systune Problem

Convert a set of `looptune` inputs into an equivalent set of inputs for `systune`.

Suppose you have a numeric plant model, `G0`, and a tunable controller model, `C0`. Suppose also that you used `looptune` to tune the feedback loop between `G0` and `C0` to within a bandwidth of `wc = [wmin,wmax]`. Convert these variables into a form that allows you to use `systune` for further tuning.

```
[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(C0,G0,wc);
```

The command returns the closed-loop system and tuning requirements for the equivalent `systune` command, `systune(CL0,SoftReqs,HardReqs,sysopt)`. The arrays `SoftReqs` and `HardReqs` contain the tuning requirements implicitly imposed by `looptune`. These requirements enforce the target bandwidth and default stability margins of `looptune`.

If you used additional tuning requirements when tuning the system with `looptune`, add them to the input list of `looptuneSetup`. For example, suppose you used a `TuningGoal.Tracking` requirement, `Req1`, and a `TuningGoal.Rejection` requirement, `Req2`. Suppose also that you set algorithm options for `looptune` using `looptuneOptions`. Incorporate these requirements and options into the equivalent `systune` command.

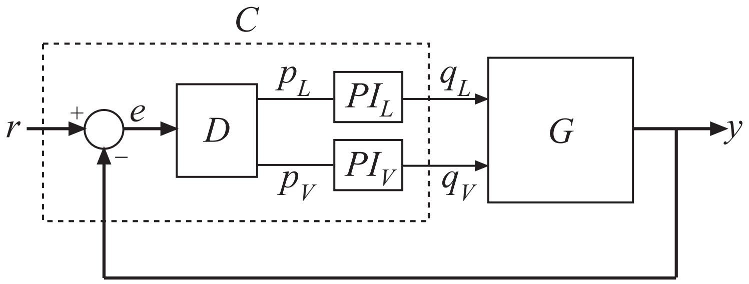
```
[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(C0,G0,wc,Req1,Req2,loopo
```

The resulting arguments allow you to construct an equivalent tuning problem for `systune`. In particular, `[~,C] = looptune(C0,G0,wc,Req1,Req2,loopopt)` yields the same result as the following commands.

```
T = systune(T0,SoftReqs,HardReqs,sysopt);
C = setBlockValue(C0,T);
```

Convert Distillation Column Problem for Tuning With systune

Set up the following control system for tuning with `looptune`. Then convert the setup to a `systune` problem and examine the results. These results reflect the structure of the control system model that `looptune` tunes. The results also reflect the tuning requirements implicitly enforced when tuning with `looptune`.



For this example, the 2-by-2 plant `G` is represented by:

$$G(s) = \frac{1}{75s+1}\begin{bmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{bmatrix}.$$

The fixed-structure controller, `C`, includes three components: the 2-by-2 decoupling matrix `D` and two PI controllers `PI_L` and `PI_V`. The signals `r`, `y`, and `e` are vector-valued signals of dimension 2.

Build a numeric model that represents the plant and a tunable model that represents the controller. Name all inputs and outputs as in the diagram, so that `looptune` and `looptuneSetup` knows how to interconnect the plant and controller via the control and measurement signals.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL','qV'};
G.OutputName = {'y'};

D = ltiblock.gain('Decoupler',eye(2));
D.InputName = 'e';
D.OutputName = {'pL','pV'};
PI_L = ltiblock.pid('PI_L','pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';
PI_V = ltiblock.pid('PI_V','pi');
```

```
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';
sum1 = sumblk('e = r - y',2);
CO = connect(PI_L,PI_V,D,sum1,{'r','y'},{'qL','qV'});
```

This system is now ready for tuning with `looptune`, using tuning goals that you specify. For example, specify a target bandwidth range. Create a tuning requirement that imposes reference tracking in both channels of the system, and a disturbance rejection requirement.

```
wc = [0.1,0.5];
TR = TuningGoal.Tracking('r','y',15);
DR = TuningGoal.Rejection({'qL','qV'},1/s);
DR.Focus = [0 0.1];

[G,C,gam,info] = looptune(G,CO,wc,TR,DR);
```

```
Final: Peak gain = 1, Iterations = 51
```

`looptune` successfully tunes the system to these requirements. However, you might want to switch to `systune` to take advantage of additional flexibility in configuring your problem. For example, instead of tuning both channels to a loop bandwidth inside `wc`, you might want to specify different crossover frequencies for each loop. Or, you might want to enforce the tuning requirements `TR` and `DR` as hard constraints, and add other requirements as soft requirements.

Convert the `looptune` input arguments to a set of input arguments for `systune`.

```
[T0,SoftReqs,HardReqs,sysopt] = looptuneSetup(G,CO,wc,TR,DR);
```

This command returns a set of arguments you can feed to `systune` for equivalent results to tuning with `looptune`. In other words, the following command is equivalent to the `looptune` command.

```
[T,fsoft,ghard,info] = systune(T0,SoftReqs,HardReqs,sysopt);
```

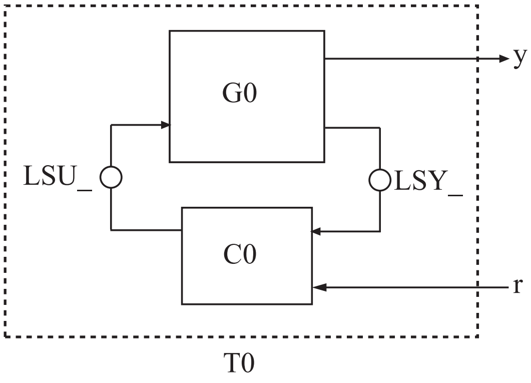
```
Final: Peak gain = 1, Iterations = 51
```

Examine the arguments returned by `looptuneSetup`.

```
T0
```

```
T0 =

  Generalized continuous-time state-space model with 0 outputs, 2 inputs,
    Decoupler: Parametric 2x2 gain, 1 occurrences.
    LSU_: Open/closed switch, 2 channels, 1 occurrences.
    LSY_: Open/closed switch, 2 channels, 1 occurrences.
    PI_L: Parametric PID controller, 1 occurrences.
    PI_V: Parametric PID controller, 1 occurrences.

Type "ss(T0)" to see the current value, "get(T0)" to see all properties,
```

The software constructs the closed-loop control system for `systune` by connecting the plant and controller at their control and measurement signals, inserting and inserting a two-channel `loopswitch` at the connection locations.



When tuning the control system of this example with `looptune`, all requirements are treated as soft requirements. Therefore, `HardReqs` is empty. `SoftReqs` is an array of `TuningGoal` requirements. These requirements together enforce the bandwidth and margins of the `looptune` command, plus the additional requirements that you specified.

```
SoftReqs
```

```
SoftReqs =

  5x1 heterogeneous LoopGeneric (LoopShape, Tracking, Rejection, ...)

    Models
    Openings
    Name
```

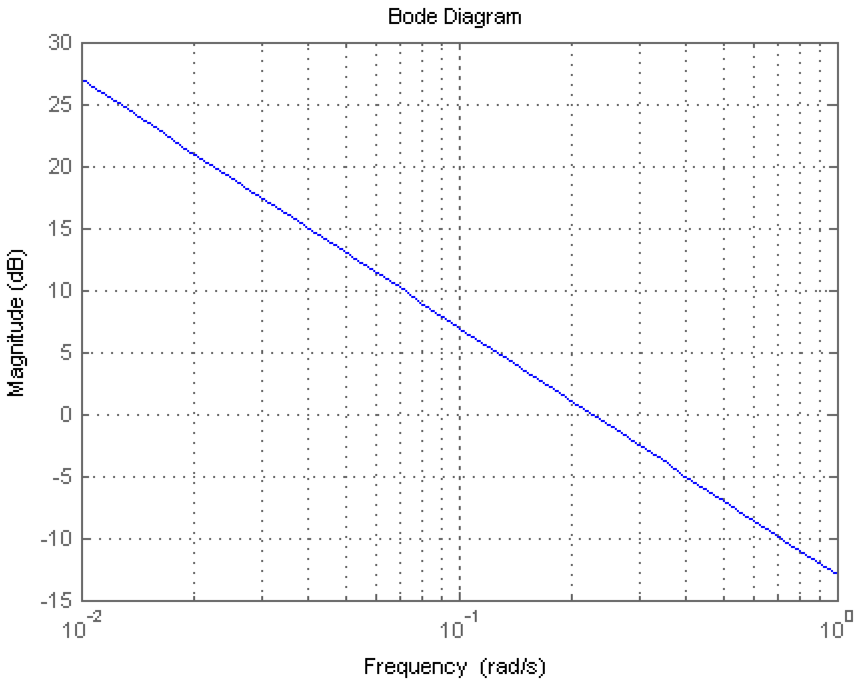
Examine the first entry in `SoftReqs`.

```
SoftReqs(1)
```

```
ans =

  LoopShape with properties:

       Location: {2x1 cell}
       LoopGain: [1x1 zpk]
       CrossTol: 0.3495
    LoopScaling: 'on'
          Focus: [0 Inf]
         Models: NaN
       Openings: {0x1 cell}
           Name: 'Open loop CG'
```

`looptuneSetup` expresses the target crossover frequency range `wc` as a `TuningGoal.LoopShape` requirement. This requirement constrains the open-loop gain profile to the loop shape stored in the `LoopGain` property, with a crossover frequency and crossover tolerance (`CrossTol`) determined by `wc`. Examine this loop shape.

```
bodemag(SoftReqs(1).LoopGain,logspace(-2,0)),grid
```



The target crossover is expressed as an integrator gain profile with a crossover between 0.1 and 0.5 rad/s, as specified by `wc`. If you want to specify a different loop shape, you can alter this `TuningGoal.LoopShape` requirement before providing it to `systune`.

`looptune` also tunes to default stability margins that you can change using `looptuneOptions`. For `systune`, stability margins are specified using `TuningGoal.Margins` requirements. Here, `looptuneSetup` has expressed the default stability margins of `looptune` as soft `TuningGoal.Margins` requirements. For example, examine the fourth entry in `SoftReqs`.

```
SoftReqs(4)
```

```
ans =

  Margins with properties:

       Location: {2x1 cell}
     GainMargin: 7.6000
    PhaseMargin: 45
          Focus: [0 Inf]
         Models: NaN
       Openings: {0x1 cell}
           Name: 'Margins at plant inputs'
```

The last entry in `SoftReqs` is a similar `TuningGoal.Margins` requirement constraining the margins at the plant outputs. `looptune` enforces these margins as soft requirements. If you want to convert them to hard constraints, pass them to `systune` in the input vector `HardReqs` instead of the input vector `SoftReqs`.

Alternatives

When tuning Simulink using an `slTunable`, interface, convert an `slTunable.looptune` problem to `slTunable.systune` using `slTunable.looptuneSetup`.

See Also

`looptune` | `systune` | `looptuneOptions` | `systuneOptions` | `gensss``slTunable.looptuneSetup` |

loopview

Purpose

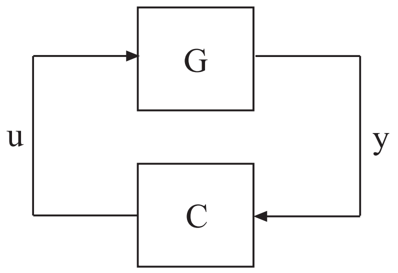
Graphically analyze MIMO feedback loops

Syntax

```
loopview(G,C)
loopview(G,C,info)
```

Description

`loopview(G,C)` plots characteristics of the following positive-feedback, multi-input, multi-output (MIMO) feedback loop with plant `G` and controller `C`.



Use `loopview` to analyze the performance of a tuned control system you obtain using `looptune`.

`loopview` plots the singular values of:

- Open-loop frequency responses `G*C` and `C*G`
- Sensitivity function `S = inv(1-G*C)` and complementary sensitivity `T = 1-S`
- Maximum (target), actual (tuned), and normalized MIMO stability margins. `loopview` plots the multi-loop disk margin (see `loopmargin`). Use this plot to verify that the stability margins of the tuned system do not significantly exceed the target value.

For more information about singular values, see `sigma`.

`loopview(G,C,info)` uses the `info` structure returned by `looptune`. This syntax also plots the target and tuned values of tuning constraints imposed on the system. Additional plots include:

- Singular values of the maximum allowed `S` and `T`. The curve marked `S/T Max` shows the maximum allowed `S` on the low-frequency side of the plot, and the maximum allowed `T` on the high-frequency side. These curves are the constraints that `looptune` imposes on `S` and `T` to enforce the target crossover range `wc`.
- Target and tuned values of constraints imposed by any tuning goal requirements you used with `looptune`.

Use `loopview` with the `info` structure to assist in troubleshooting when tuning fails to meet all requirements.

Input Arguments

G

Numeric LTI model or tunable `genss` model representing the plant in a control system. The plant is the portion of a control system whose outputs are sensor signals (measurements), and whose inputs are actuator signals (controls).

You can obtain `G` as an output argument from `looptune` when you tune your control system.

C

`genss` model representing the controller in a control system. The controller is the portion of your control system that receives sensor signals (measurements) as inputs and produces actuator signals (controls) as outputs.

You can obtain `C` as an output argument from `looptune` when you tune your control system.

info

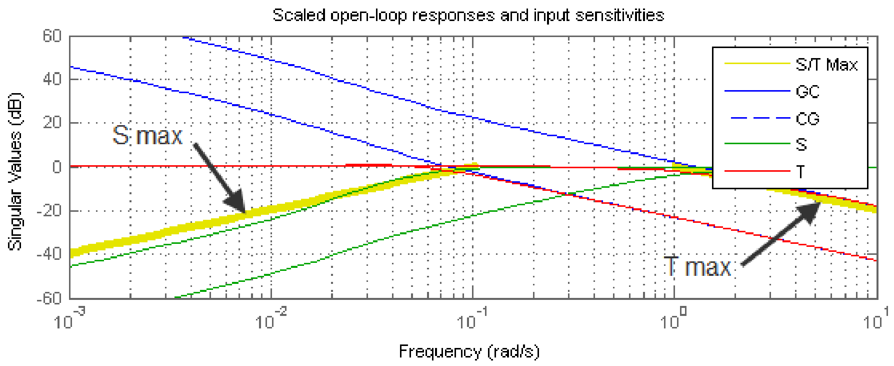
`info` structure returned by `looptune` during control system tuning.

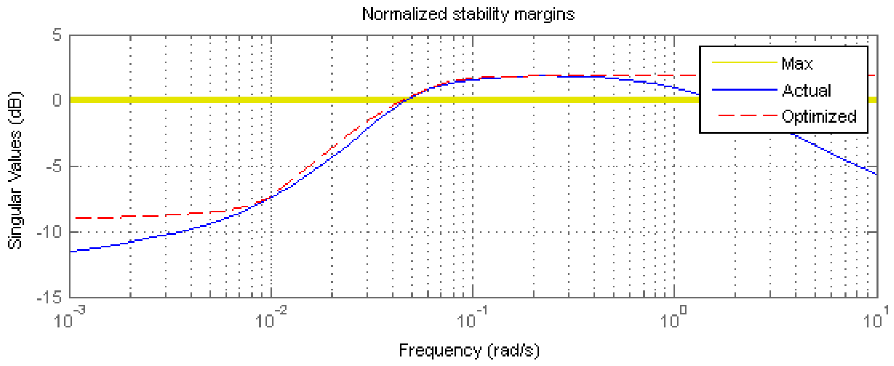
Examples

Tune a control system, and use `loopview` to examine the performance of the tuned controller.

```
s = tf('s');
```

```
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL','qV'};
G.OutputName = 'y';

D = ltiblock.gain('Decoupler',eye(2));
PI_L = ltiblock.pid('PI_L','pi');
PI_L.OutputName = 'qL';
PI_V = ltiblock.pid('PI_V','pi');
PI_V.OutputName = 'qV';

sum = sumblk('e = r - y',2);
C0 = (blkdiag(PI_L,PI_V)*D)*sum;

wc = [0.1,1];
options = looptuneOptions('RandomStart',5);
[G,C,gam,info] = looptune(-G,C0,wc,options);

loopview(G,C,info)
```





The first plot shows that the open-loop gain crossovers fall close to the specified interval `[0.1,1]`. This plot also includes the maximum and tuned values of the sensitivity function `S = inv(1-G*C)` and complementary sensitivity `T = 1-S`. The curve marked `S/T Max` shows the maximum allowed `S` on the low-frequency side of the plot, and the maximum allowed `T` on the high-frequency side. These curves are the constraints that `looptune` imposes on `S` and `T` to enforce the target crossover range `wc`.

The second plot shows that the MIMO stability margins of the tuned system (blue curve) do not significantly exceed the upper limit (yellow curve).

Alternatives

For analyzing Simulink models tuned with `slTunable.looptune`, use `slTunable.loopview` (requires Simulink Control Design).

See Also

`looptune` | `slTunable.looptune` | `slTunable.loopview`

Tutorials

- “Tune MIMO Control System for Specified Bandwidth”
- “Decoupling Controller for a Distillation Column”

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)])).`

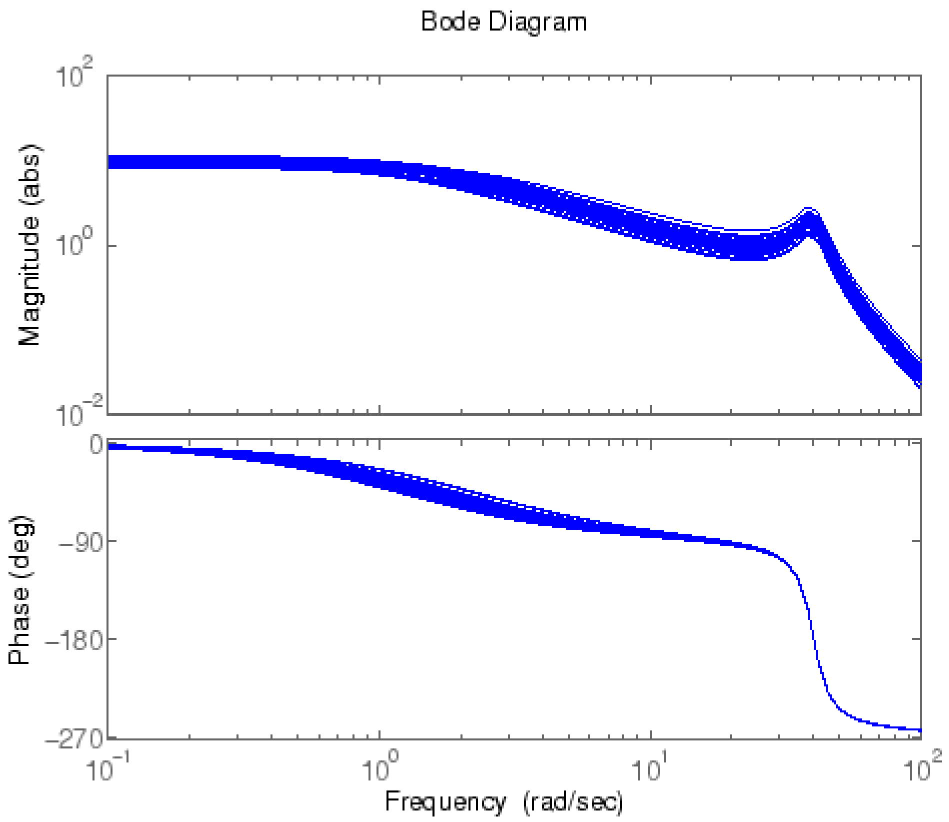
`[usys,wt,diffdata] = ltiarray2uss(P,Parray,ord,type)` returns the norm of the difference (absolute difference for additive, and relative difference for multiplicative uncertainty) between the nominal model `P` and `Parray`. `wt` satisfies `diffdata(w_i) < |wt(w_i)|` at all frequency points.

Examples

See 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.

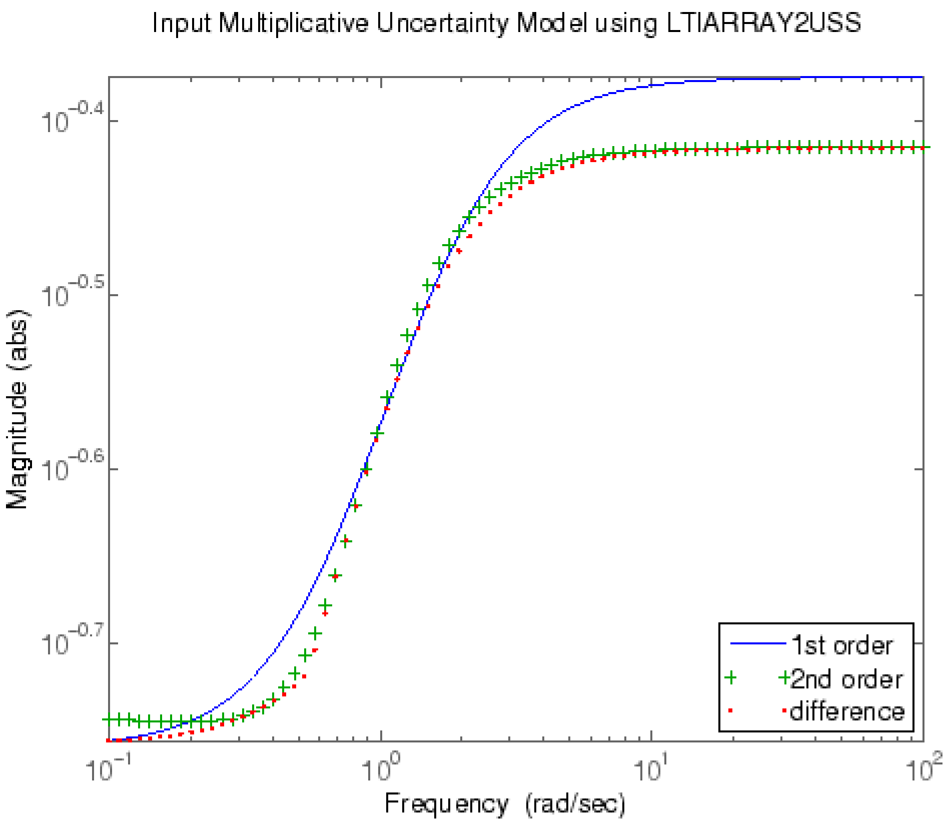
```
gain = ureal('gain',10,'Perc',20);
tau = ureal('tau',.6,'Range',[.42 .9]);
wn = 40;
zeta = 0.1;
usys = tf(gain,[tau 1])*tf(wn^2,[1 2*zeta*wn wn^2]);
sysnom = usys.NominalValue;
parray = usample(usys,30);
om = logspace(-1,2,80);
parrayg = frd(parray,om);
bode(parrayg)
```



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.

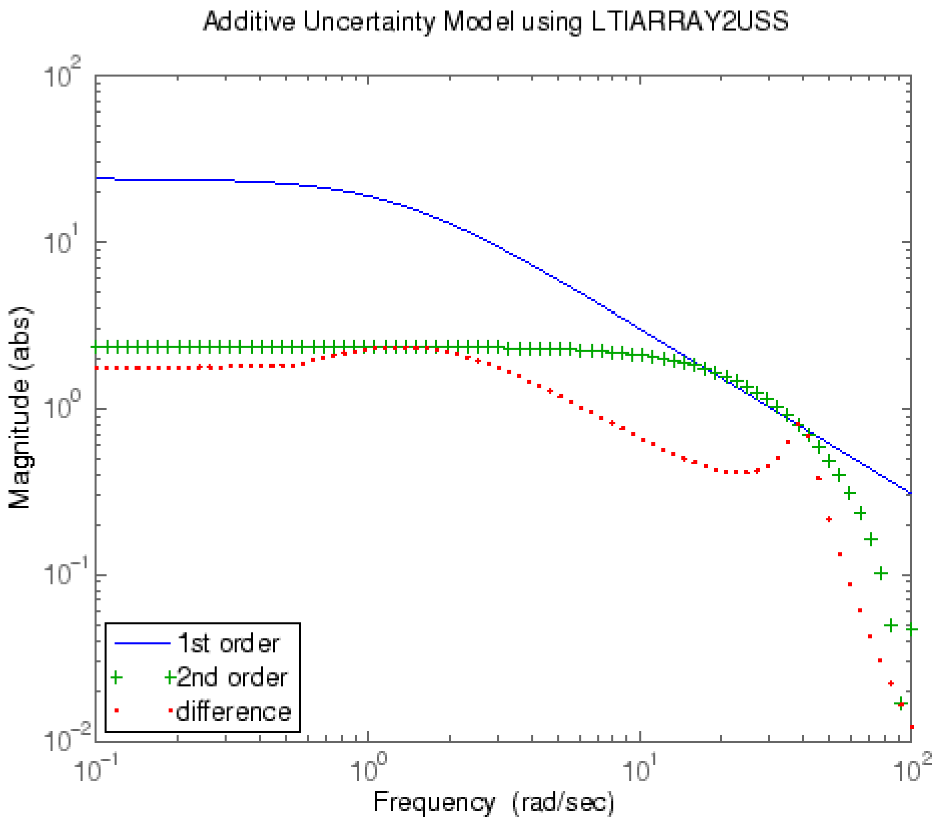
```
[umodIn1,wtIn1,diffdataIn] = ltiarray2uss(sysnom,parrayg,1);
[umodIn2,wtIn2,diffdataIn] = ltiarray2uss(sysnom,parrayg,2);
```

```
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

Algorithms

For each value in the vector `RHO`, `[K,SVL,W1] = ltrsyn(G,F,XI,THETA,RHO)` computes the full-state-feedback (default `OPT='`*`INPUT`*`'`) LTR controller

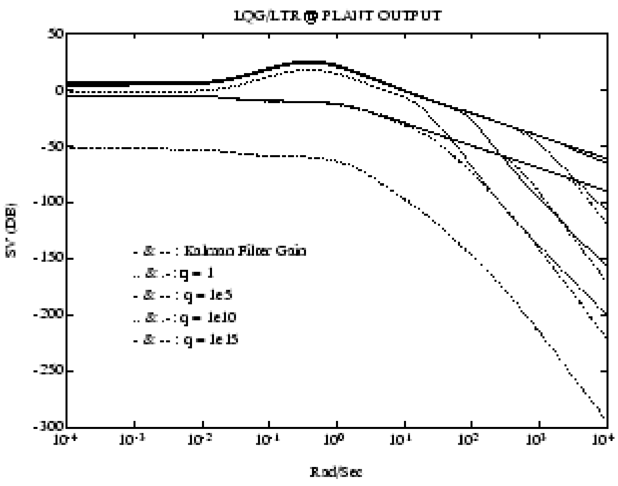
$$K(s) = \left[K_c(Is - A + BK_c + K_fC - K_fDK_c)^{-1}K_f\right]$$

where K_c = `F` and K_f = `lqr(A',C',XI+RHO(i)*B*B',THETA)`. The “fictitious noise” term `RHO(i)*B*B'` results in loop-transfer recovery as `RHO(i)` → ∞. The Kalman filter gain is

$K_f = \Sigma C^T \Theta^{-1}$ 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` | `ncfsyn`

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 $X_1, . . ., X_K$ and given values `X1,...,Xk` of $X_1, . . ., X_K$, `mat2dec` returns the corresponding value `decvec` of the vector of decision variables. Recall that the decision variables are the independent entries of the matrices $X_1, . . ., X_K$ 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 $X_1, . . ., X_K$, `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 $X_1, . . ., X_K$ 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

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

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 $c^T x \le$ `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`

mixsyn

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,...)
```

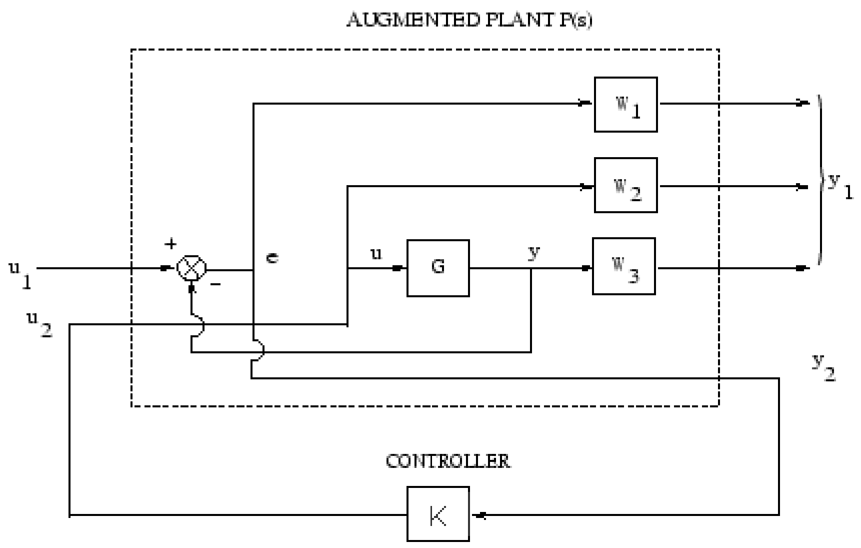
Description

`[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

$$T_{y1u1} \triangleq \begin{bmatrix} W_1 S \\ W_2 R \\ W_3 T \end{bmatrix}$$

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_{y1u1} for mixed sensitivity `mixsyn`.

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

$$\bar{\sigma}(S(j\omega)) \le \gamma \;\; \underline{\sigma}\left(W_1^{-1}(j\omega)\right)$$
$$\bar{\sigma}(R(j\omega)) \le \gamma \;\; \underline{\sigma}\left(W_2^{-1}(j\omega)\right)$$
$$\bar{\sigma}(T(j\omega)) \le \gamma \;\; \underline{\sigma}\left(W_3^{-1}(j\omega)\right)$$

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`).

Algorithms

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

is equivalent to

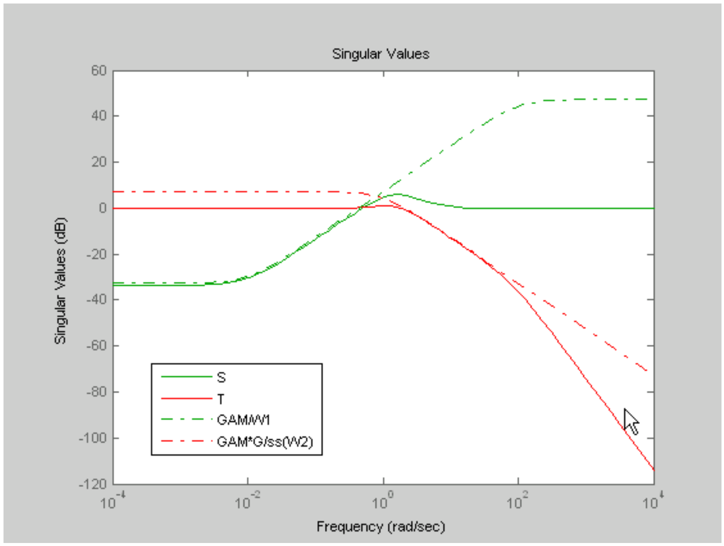
```
[K,CL,GAM,INFO]=...
    hinfsyn(augw(G,W1,W2,W3),KEY1,VALUE1,KEY2,VALUE2,...).
```

`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

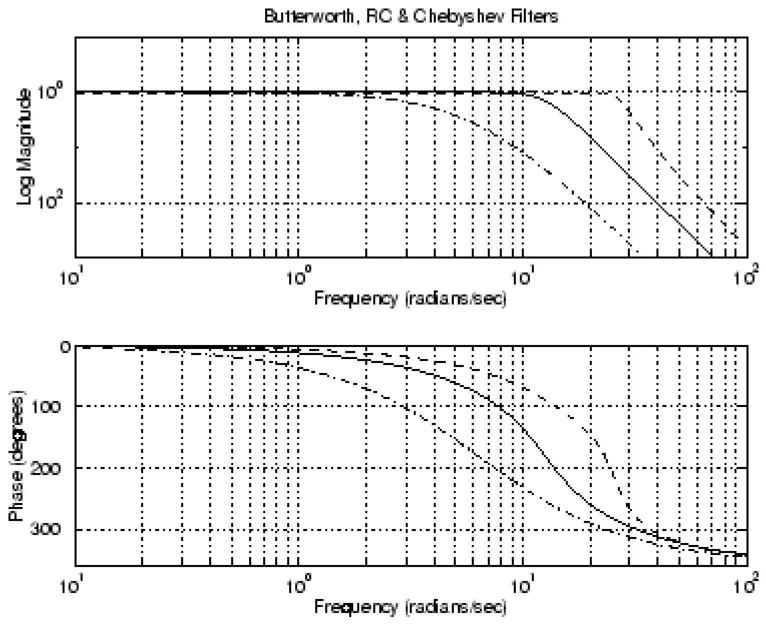
String variable	Description
`'butterw'`	Butterworth filter
`'cheby'`	Chebyshev filter
`'bessel'`	Bessel filter
`'rc'`	Series of resistor/capacitor filters

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

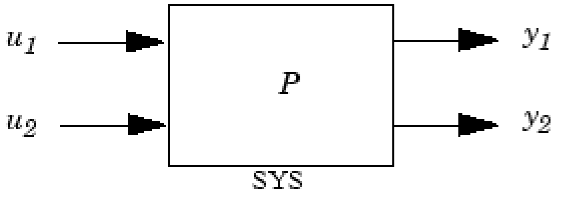
```
SYS=mktito(SYS,NMEAS,NCONT)
```

Description

`SYS=mktito(SYS,NMEAS,NCONT)` adds TITO (two-input/two-output) partitioning to LTI system `SYS`, assigning OutputGroup and InputGroup properties such that

$$\mathrm{NMEAS} = \dim(y_2)$$

$$\mathrm{NCONT} = \dim(u_2)$$



Any preexisting OutputGroup or InputGroup properties of `SYS` are overwritten. TITO partitioning simplifies syntax for control synthesis functions like `hinfsyn` and `h2syn`.

Algorithms

```
[r,c]=size(SYS);
set(SYS,'InputGroup', struct('U1',1:c-NCONT,'U2',c-NCONT+1:c));
set(SYS,'OutputGroup',struct('Y1',1:r-NMEAS,'Y2',r-NMEAS+1:r));
```

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 | sdhinfsyn

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.

Algorithms

Using a real eigen structure decomposition `reig` and ordering the eigenvectors in ascending order according to their eigenvalue magnitudes, we can form a similarity transformation out of these ordered real eigenvectors such that he resulting systems `G1` and/or `G2` are in block diagonal modal form.

Note This routine is extremely useful when model has jω-axis singularities, e.g., rigid body dynamics. It has been incorporated inside Hankel based model reduction routines - `hankelmr`, `balancmr`, `bstmr`, and `schurmr` to isolate those jω-axis poles from the actual model reduction process.

Examples

Given a continuous stable or unstable system, `G`, the following commands can get a set of modal form realizations depending on the split index -- `cut`:

```
rng(1234,'twister');
G = rss(50,2,2);
[G1,G2] = modreal(G,2); % cut = 2 for two rigid body modes
G1.d = zeros(2,2); % remove the DC gain of the system from G1
sigma(G,G1,G2)
```

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` = [γ_0, v_0, α, ß] to specify the problem dimensions and the design parameters γ_0, v_0, α, and ß. 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

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

In this context, `msfsyn` seeks a state-feedback gain that robustly enforces the specifications over the entire polytope of plants. Note that polytopic plants should be defined with `psys` and that the closed-loop system `Pcl` is itself polytopic in such problems. Affine parameter-dependent plants are also accepted and automatically converted to polytopic models.

See Also

`lmireg` | `psys`

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` ≥ 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)

Option	Meaning
`'g`*N*`'`	Use gain-based lower bound method multiple times. The value of *N* sets the number of times, according to `10+`*N*`*10`. For example, `'g6'` uses gain-based lower bound 70 times. Larger numbers typically give better lower bounds. If all uncertainty blocks described by `blk` are real, then the default is `'g1'`. If at least one uncertainty block is complex, then `mussv` uses power iteration lower bound by default.
`'p'`	Use power iteration method to compute lower bound. When at least one of the uncertainty blocks described by `blk` is complex, then `'p'` is the default lower bound method.
`'i'`	Reinitialize lower bound computation at each new matrix (only relevant if `M` is `ND` array or `frd`)
`'m`*N*`'`	Randomly reinitialize lower bound iteration multiple times. *N* is an integer between 1 and 9. For example, `'m7'` randomly reinitializes the lower bound iteration 7 times. Larger numbers typically give better lower bounds.
`'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.

```
rng(929,'twister')
M = randn(5,5) + sqrt(-1)*randn(5,5);
F = randn(2,5) + sqrt(-1)*randn(2,5);
BlockStructure = [-1 0;-1 0;1 1;2 0];
```

```
[ubound,Q] = mussv(M,F,BlockStructure);
bounds = mussv(M,BlockStructure);
optbounds = mussv(M+Q*F,BlockStructure);
```

The quantities `optbounds(1)` and `ubound` should be extremely close, and significantly lower than `bounds(1)` and `bounds(2)`.

```
[optbounds(1) ubound]

ans =

    2.2070    2.1749

[bounds(1)  bounds(2)]

ans =

    4.4049    4.1960
```

Algorithms

The lower bound is computed using a power method, Young and Doyle, 1990, and Packard *et al.* 1988, and the upper bound is computed using the balanced/AMI technique, Young *et al.*, 1992, for computing the upper bound from Fan *et al.*, 1991.

Peter Young and Matt Newlin wrote the original function.

The lower-bound power algorithm is from Young and Doyle, 1990, and Packard *et al.* 1988.

The upper-bound is an implementation of the bound from Fan *et al.*, 1991, and is described in detail in Young *et al.*, 1992. In the upper bound computation, the matrix is first balanced using either a variation of Osborne's method (Osborne, 1960) generalized to handle *repeated scalar* and *full* blocks, or a Perron approach. This generates the standard upper bound for the associated complex μ problem. The Perron eigenvector method is based on an idea of Safonov, (Safonov, 1982). It gives the exact computation of μ for positive matrices with scalar blocks, but is comparable to Osborne on general matrices. Both the Perron and Osborne methods have been modified to handle *repeated scalar* and *full* blocks. Perron is faster for small matrices but has a growth rate of n^3, compared with less than n^2 for Osborne. This is partly due to the `MATLAB` implementation, which greatly favors Perron. The default is to use Perron for simple block structures and Osborne for more complicated block structures. A sequence of improvements to the upper bound is then made based on various equivalent forms of the upper bound. A number of descent techniques are used that exploit the structure of the problem, concluding with general purpose LMI optimization (Boyd *et al.*), 1993, to obtain the final answer.

The optimal choice of `Q` (to minimize the upper bound) in the generalized μ problem is solved by reformulating the optimization into a semidefinite program (Packard *et al.*, 1991).

References

[1] Boyd, S. and L. El Ghaoui, “Methods of centers for minimizing generalized eigenvalues,” *Linear Algebra and Its Applications*, Vol. 188–189, 1993, pp. 63–111.

[2] Fan, M., A. Tits, and J. Doyle, “Robustness in the presence of mixed parametric uncertainty and unmodeled dynamics,” *IEEE Transactions on Automatic Control*, Vol. AC–36, 1991, pp. 25–38.

[3] Osborne, E., “On preconditioning of matrices,” *Journal of Associated Computer Machines*, Vol. 7, 1960, pp. 338–345.

[4] Packard, A.K., M. Fan and J. Doyle, “A power method for the structured singular value,” *Proc. of 1988 IEEE Conference on Control and Decision*, December 1988, pp. 2132–2137.

[5] Safonov, M., “Stability margins for diagonally perturbed multivariable feedback systems,” *IEEE Proc.*, Vol. 129, Part D, 1992, pp. 251–256.

[6] Young, P. and J. Doyle, “Computation of with real and complex uncertainties,” *Proceedings of the 29th IEEE Conference on Decision and Control*, 1990, pp. 1230–1235.

[7] Young, P., M. Newlin, and J. Doyle, “Practical computation of the mixed problem,” *Proceedings of the American Control Conference*, 1992, pp. 2190–2194.

See Also

`mussvextract` | `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 β and matrices D and G, consistent with `BlockStructure`, such that

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

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 β is an upper bound of `mussv(M,BlockStructure)`.

It is true that if `BlockStructure` consists only of complex blocks, then all G matrices will be zero, and the expression above simplifies to

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

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 $M'D_r M$-$ß^2 D_c$ is sufficient to derive an upper bound.

Lower Bound Information

The lower bound of `mussv(M,BlockStructure)` is based on finding a "small" (hopefully the smallest) block-structured matrix `VDelta` that causes `det(I - M*VDelta)` to equal 0. Equivalently, the matrix `M*VDelta` has an eigenvalue equal to 1. It will always be true that the lower bound `(bounds(2))` will be the reciprocal of `norm(VDelta)`.

Examples

Suppose `M` is a 4-by-4 complex matrix. Take the block structure to be two 1-by-1 complex blocks and one 2-by-2 complex block.

```
rng(0,'twister')
M = randn(4,4) + sqrt(-1)*randn(4,4);
BlockStructure = [1 1;1 1;2 2];
```

You can calculate bounds on the structured singular value using the `mussv` command and extract the scaling matrices using `mussvextract`.

```
[bounds,muinfo] = mussv(M,BlockStructure);
[VDelta,VSigma,VLmi] = mussvextract(muinfo);
```

You can first verify the Newlin/Young upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dl` and `Dr`.

```
Dl = VSigma.DLeft


Dl =

    1.0000         0         0         0
         0    0.7437         0         0
         0         0    1.0393         0
         0         0         0    1.0393


Dr = VSigma.DRight


Dr =

    1.0000         0         0         0
         0    0.7437         0         0
         0         0    1.0393         0
         0         0         0    1.0393


[norm(Dl*M/Dr) bounds(1)]

ans =

    6.2950    6.2950
```

You can first verify the LMI upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dr` and `Dc`.

```
Dr = VLmi.Dr;
Dc = VLmi.Dc;
eig(M'*Dr*M - bounds(1)^2*Dc)
```

```
ans =

  -0.0000 - 0.0000i
 -17.7242 - 0.0000i
 -33.8550 + 0.0000i
 -41.2013 - 0.0000i
```

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

```
VDelta

VDelta =

   0.1301 - 0.0922i        0                  0                  0
        0             -0.0121 - 0.1590i       0                  0
        0                  0             -0.0496 - 0.0708i   0.1272 - 0.00
        0                  0              0.0166 - 0.0163i   0.0076 + 0.03

[norm(VDelta) 1/bounds(2)]

ans =

    0.1595    0.1595
```

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

```
eig(M*VDelta)

ans =

   1.0000 - 0.0000i
  -0.2501 - 0.1109i
   0.0000 + 0.0000i
  -0.3022 + 0.2535i
```

Keep the matrix the same, but change `BlockStructure` to be a 2-by-2 repeated, real scalar block and two complex 1-by-1 blocks. Run `mussv` with the `'C'` option to tighten the upper bound.

```
BlockStructure2 = [-2 0; 1 0; 1 0];
[bounds2,muinfo2] = mussv(M,BlockStructure2,'C');
```

You can compare the computed bounds. Note that `bounds2` should be smaller than `bounds`, because the uncertainty set defined by `BlockStructure2` is a proper subset of that defined by `BlockStructure`.

```
[bounds; bounds2]

ans =

    6.2950    6.2704
    5.1840    5.1750
```

You can extract the *D*, *G* and `Delta` from `muinfo2` using `mussvextract`.

```
[VDelta2,VSigma2,VLmi2] = mussvextract(muinfo2);
```

As before, you can first verify the Newlin/Young upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dl, Dr, Gl, Gm and Gr`.

```
Dl = VSigma2.DLeft;
Dr = VSigma2.DRight;
Gl = VSigma2.GLeft;
Gm = VSigma2.GMiddle;
Gr = VSigma2.GRight;
dmd = Dl*M/Dr/bounds2(1) - sqrt(-1)*Gm;
SL = (eye(4)+Gl*Gl)^-0.25;
SR = (eye(4)+Gr*Gr)^-0.25;
norm(SL*dmd*SR)

ans =
    1.0000
```

You can first verify the LMI upper bound with the information extracted from `muinfo`. The corresponding scalings are `Dr`, `Dc`, `Grc` and `Gcr`.

```
Dr = VLmi2.Dr;
Dc = VLmi2.Dc;
Grc = VLmi2.Grc;
Gcr = VLmi2.Gcr;
eig(M'*Dr*M - bounds(1)^2 *Dc + j*(Gcr*M-M'*Grc))

ans =

 -69.9757 + 0.0000i
 -11.2139 - 0.0000i
 -19.2766 - 0.0000i
 -40.2869 - 0.0000i
```

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

```
VDelta2

VDelta2 =

   0.1932                  0                  0                  0
        0             0.1932                  0                  0
        0                  0       -0.1781 - 0.0750i             0
        0                  0                  0           0.0941 + 0.16

[norm(VDelta2) 1/bounds2(2)]

ans =

    0.1932    0.1932
```

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

```
eig(M*VDelta2)

  ans =

 1.0000 + 0.0000i
  -0.4328 + 0.1586i
   0.1220 - 0.2648i
  -0.3688 - 0.3219i
```

See Also

mussv

ncfmargin

Purpose

Calculate normalized coprime stability margin of plant-controller feedback loop

Syntax

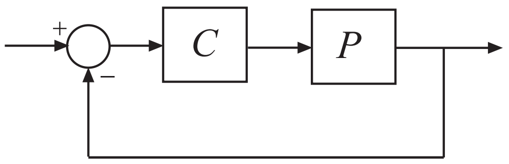
```
[marg,freq] = ncfmargin(P,C)
[marg,freq] = ncfmargin(P,C,sign)
[marg,freq] = ncfmargin(P,C,sign,tol)
```

Description

`[marg,freq] = ncfmargin(P,C)` returns the normalized coprime stability margin (also called the gap metric stability margin) of the multivariable feedback loop consisting of a controller, `C`, in negative feedback with a plant, `P`. This margin, `marg`, is achieved at the frequency `freq`. The normalized coprime stability margin is defined as:

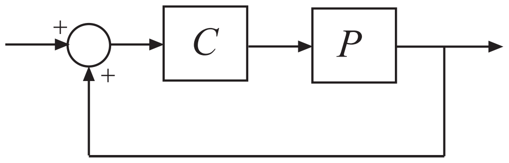
$$\left\| \begin{bmatrix} I \\ C \end{bmatrix} (I - PC)^{-1} \begin{bmatrix} P & I \end{bmatrix} \right\|_{\infty}^{-1}.$$

The calculation assumes the feedback structure of the following diagram:



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

`[marg,freq] = ncfmargin(P,C,sign)` specifies the sign of the feedback connection assumed for the margin calculation. The default value, `sign = -1`, specifies negative feedback. Setting `sign = +1` assumes a positive feedback connection for the margin calculation, as in the following diagram:



`[marg,freq] = ncfmargin(P,C,sign,tol)` calculates the normalized coprime factor metric with the specified relative accuracy. `tol` is a scalar value between 10^{-5} and 10^{-2}. The default value is `tol = 0.001` (0.1% accuracy).

Examples

Consider an unstable first-order plant, `p`, stabilized by high-gain and low-gain controllers, `cL` and `cH`.

```
p = tf(4,[1 -0.001]);
cL = 1;
cH = 10;
```

Compute the stability margin of the closed-loop system with the low-gain controller.

```
[margL,~] = ncfmargin(p,cL)
```

```
margL =

    0.7069
```

Similarly, compute the stability margin of the closed-loop system with the high-gain controller.

```
[margH,~] = ncfmargin(p,cH)
```

```
margH =

    0.0995
```

The closed-loop systems with low-gain and high-gain controllers have normalized coprime stability margins of about 0.71 and 0.1, respectively. This result indicates that the closed-loop system with low-gain controller is more robust to unstructured perturbations than the system with the high-gain controller.

To observe this difference in robustness, construct an uncertain plant, `punc`, that has an additional 11% unmodeled dynamics compared to the nominal plant.

```
punc = p + ultidyn('uncstruc',[1 1],'Bound',0.11);
```

Calculate the robust stability of the closed-loop systems formed by the uncertain plant and each controller.

```
[stabmargL,duL,reportL] = robuststab(feedback(punc,cL));
reportL
[stabmargH,duH,reportH] = robuststab(feedback(punc,cH));
reportH
```

```
reportL =

Uncertain system is robustly stable to modeled uncertainty.
 -- It can tolerate up to 909% of the modeled uncertainty.
 -- A destabilizing combination of 909% of the modeled uncertainty was found.
 -- This combination causes an instability at 2e+03 rad/seconds.
 -- Sensitivity with respect to the uncertain element is:
     'uncstruc' is 100%.  Increasing 'uncstruc' by 25% leads to a 25% decrease in the margin.


reportH =

Uncertain system is not robustly stable to modeled uncertainty.
 -- It can tolerate up to 90.9% of the modeled uncertainty.
 -- A destabilizing combination of 90.9% of the modeled uncertainty was found.
 -- This combination causes an instability at 2e+04 rad/seconds.
 -- Sensitivity with respect to the uncertain element is:
     'uncstruc' is 100%.  Increasing 'uncstruc' by 25% leads to a 25% decrease in the margin.
```

As expected, the robust stability analysis shows that the closed-loop system with low-gain controller is more robustly stable in the presence of the unmodeled LTI dynamics. In fact, this closed-loop system can tolerate 909% (or 9.09*11%) of the unmodeled dynamics. In contrast, closed-loop system with the high-gain controller is not robustly stable. That closed-loop system can only tolerate 90.9% (or 0.909*11%) of the unmodeled dynamics.

Algorithms

The computation of the normalized coprime stability margin amounts to solving 2-block H_∞ problems. [1] The function, `ncfmargin`, is based on [2].

References

[1] Georgiou, T.T., "On the computation of the gap metric," *Systems & Control Letters*, Vol. 11, No. 4, 1988, pp. 253-257.

[2] Green, M., Glover, K., D. Limebeer, and J.C. Doyle, "A J-spectral factorization approach to H_∞ control," *SIAM Journal on Control and Optimization*, Vol. 28, No. 6, 1990, pp. 1350-1371.

See Also

`ncfsyn` | `loopmargin` | `gapmetric` | `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

$$\begin{aligned} U_rN_r + V_rM_r &= I \\ N_lU_l + M_lV_l &= I \end{aligned}$$

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.

Algorithms

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

1. Find the normalized coprime factors of *G* by solving Hamiltonian described in [1].

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

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:

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

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`

ncfsyn

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' $G_s{:} = W_2GW_1$ 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)` db. 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`.

Algorithms

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

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

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

Roughly speaking, this means for most plants that

$\sigma(W_2 G W_1\ K_\infty)$, db = $\sigma(W_2 G W_1)$, db ± γ, db

$\sigma(K_\infty W_2 G W_1)$, db = $\sigma(W_2 G W_1)$, db ± γ, 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 $\Delta_{1,}$ Δ_2 satisfying

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

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 $W_2 G W_1$ 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 $\dfrac{1}{b(W_2GW_1, K_\infty)}$ = `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.

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

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.

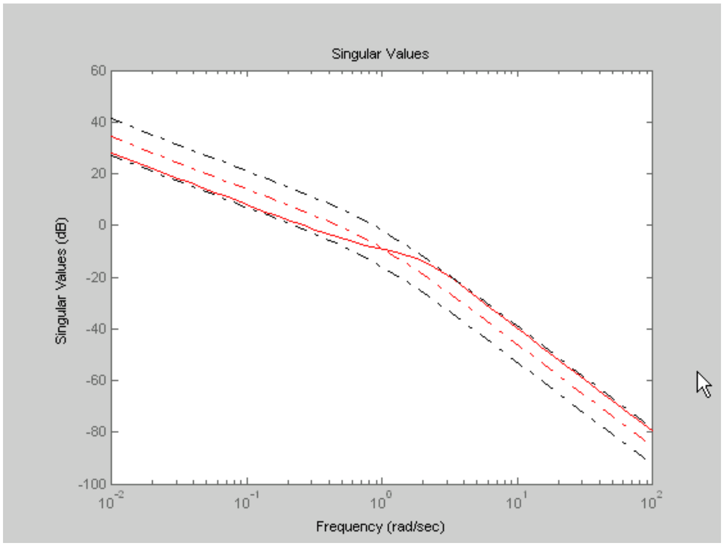
Algorithms

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) db.

```
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

$E(p)x^{\cdot} = A(p)x + B(p)u$

$y = C(p)x + D(p)u$

with $p = (p_1, . . ., p_n) \epsilon R^n$, `pdlstab` seeks a Lyapunov function of the form

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

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

$Ex^{\cdot} = Ax + Bu$

$y = Cx + Du$

with

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

`pdlstab` seeks a Lyapunov function of the form

$V(x, α) = x^T Q(α)–1x,\ Q(α) = α_1 Q_1 + . . .+ α_n Q_n$

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

Tips

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

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

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`

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^TPx of the Lyapunov function and `D` and `S` are the Popov multipliers.

If the uncertainty `delta` contains real parameter blocks, the conservatism of the Popov criterion can be reduced by first performing a simple loop transformation. To use this refined test, call `popov` with the syntax

```
[t,P,S,N] = popov(sys,delta,1)
```

See Also

`quadstab` | `pdlstab`

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

  $$S(p) = S_0 + p_1S_1 + . . . + p_nS_n$$

  For *polytopic* systems with `SYSTEM` matrix ranging in

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

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

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

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.

`psys` supports two types of uncertain state-space models:

- *Polytopic* systems

  $E(t)\,\dot{x} = A(t)x + B(t)u$

  $y = C(t)x + D(t)u$

  whose `SYSTEM` matrix takes values in a fixed polytope:

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

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

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

  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 $p = (p_1, . . ., p_n)$ of real parameters, i.e.,

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

$$\underbrace{\begin{bmatrix} A_0+jE_0 & B_0 \\ C_0 & D_0 \end{bmatrix}}_{S_0} + p1 \underbrace{\begin{bmatrix} A_1+jE_1 & B_1 \\ C_1 & D_1 \end{bmatrix}}_{S_1} + \ldots + p_n \underbrace{\begin{bmatrix} A_n+jE_n & B_n \\ C_n & D_n \end{bmatrix}}_{S_n}$$

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

The argument `syslist` lists the `SYSTEM` matrices S_i characterizing the polytopic value set or parameter dependence. In addition, the description `pv` of the parameter vector (range of values and rate of variation) is required for affine parameter- dependent models (see `pvec` for details). Thus, a polytopic model with vertex systems S_1, . . ., 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 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 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

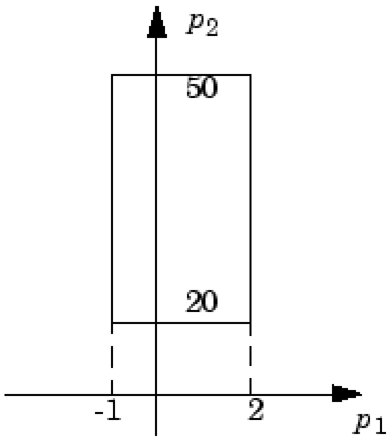
```
pv = pvec('box',[-1 2;20 50])
```

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

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

Hence, you could also specify p by

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



Parameter box

See Also

`pvinfo` | `psys`

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 $p = (p_1, . . ., p_n)$ of real parameters declared with `pvec` and stored in `pv`. The command `pvinfo(pv)` displays the type of parameter vector (`'box'` or `'pol'`), the number n of scalar parameters, and for the type `'pol'`, the number of vertices used to specify the parameter range.

For the type `'box'`:

```
[pmin,pmax,dpmin,dpmax] = pvinfo(pv,'par',j)
```

returns the bounds on the value and rate of variations of the j-th real parameter p_j. Specifically,

$$p\min \le p_j(t) \le p\max, dp\min \le \frac{dp_j}{dt} \le dp\max$$

For the type `'pol'`:

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

returns the j-th vertex of the polytope of R^n in which p ranges, while

```
pvinfo(pv,'eval',c)
```

returns the value of the parameter vector p given its barycentric coordinates `c` with respect to the polytope vertices $(V_1, . . ., V_k)$. The vector `c` must be of length k and have nonnegative entries. The corresponding value of p is then given by

$$p = \frac{\sum_{i=1}^{k} c_i V_i}{\sum_{i=1}^{k} c_i}$$

See Also

`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

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

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

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

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

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

`quadstab` seeks a fixed Lyapunov function $V(x) = x^T P x$ 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

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

  $$Q > I$$

  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 $P = Q_{\text{opt}}^{-1}$. This matrix is returned in `P`.

Other control parameters can be accessed through `options(2)` and `options(3)`:

- if `options(2)=0` (default), `quadstab` runs in fast mode, using the least expensive sufficient conditions. Set `options(2)=1` to use the least conservative conditions
- `options(3)` is a bound on the condition number of the Lyapunov matrix P. The default is 10^9.

See Also

`pdlstab` | `decay` | `quadperf` | `psys`

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')

xr =

  Uncertain real parameter "NMGXC" with nominal value 5.34 and variability [-2.99,1.92].
```

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

```
rng(29,'twister');
xlti = randatom('ultidyn',[4 3])
xlti =

  Uncertain LTI dynamics "LOSWT" with 4 outputs, 3 inputs, and gain less than 0.293.
```

See Also

`rand` | `randn` | `randumat` | `randuss` | `ucomplex` | `ucomplexm` | `ultidyn`

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)

x1 =

  Uncertain matrix with 2 rows and 3 columns.
  The uncertainty consists of the following blocks:
    AWYRT: Uncertain real, nominal = 7.09, variability = [-7.84,16.4]%, 2 occurrences
    HRRED: Uncertain complex, nominal = 3.14+5.47i, radius = 1.92, 1 occurrences
    VSIYA: Uncertain real, nominal = -4.05, variability = [-1.53,3.83], 3 occurrences
    YZEZY: Uncertain complex, nominal = -6.54-2.17i, variability = 24%, 1 occurrences

Type "x1.NominalValue" to see the nominal value, "get(x1)" to see all properties, and
"x1.Uncertainty" to interact with the uncertain elements.
```

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

```
rng(91,'twister');
x2 = randumat(4,2)

x2 =
  Uncertain matrix with 4 rows and 2 columns.
  The uncertainty consists of the following blocks:
    YQZBI: Uncertain complex, nominal = 3.61+1.88i, radius = 1.42, 1 occurrences

Type "x2.NominalValue" to see the nominal value, "get(x2)" to see all properties,
and "x2.Uncertainty" to interact with the uncertain elements.
```

See Also

rand | randn | randatom | randuss | ucomplex | ultidyn

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 `n`th order single-input/single-output uncertain continuous-time system. `randuss` randomly selects from uncertain atoms of type `'ureal'`, `'ultidyn'`, and `'ucomplex'`.

`usys = randuss(n,p)` generates an `n`th order single-input uncertain continuous-time system with `p` outputs.

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

`usys = randuss(n,p,m,Ts)` generates an `n`th order uncertain discrete-time system with `p` outputs and `m` inputs. The sample time is `Ts`.

`usys = randuss` (without arguments) results in a 1-by-1 uncertain continuous-time `uss` object with up to four uncertain objects.

In general, both `rand` and `randn` are used internally. You can control the result of `randuss` by setting seeds for both random number generators before calling the function.

Examples

The statement creates a fifth order, continuous-time uncertain system `s1` of size 2-by-3. Note your display can differ because a random seed is used.

```
s1 = randuss(5,2,3)
USS: 5 States, 2 Outputs, 3 Inputs, Continuous System
  CTPQV: 1x1 LTI, max. gain = 2.2, 1 occurrence
  IGDHN: real, nominal = -4.03, variability =
[-3.74667  22.7816]%, 1 occurrence
  MLGCD: complex, nominal = 8.36+3.09i,  +/- 7.07%, 1 occurrence
  OEDJK: complex, nominal = -0.346-0.296i, radius = 0.895,
1 occurrence
```

See Also

rand | randn | randatom | randumat | ucomplex | ultidyn

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

```
rng(1234,'twister');
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 schurm
[g4, redinfo] = reduce(G,'ErrorType','mult','MaxError',[0.01, 0.05]);


[g5, redinfo5] = reduce(G,'ErrorType','add','algorithm','hankel', ...
       'maxerror',[0.01]);
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. 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`

repmat

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

Options object for use with `robuststab` and `robustperf`

Note `robopt` will be removed in a future version. Use `robuststabOptions` or `robustperfOptions` instead.

Purpose

Robust performance margin of uncertain multivariable system

Syntax

```
perfmarg = robustperf(usys)
[perfmarg,wcu,report,info] = robustperf(usys)
[perfmarg,wcu,report,info] = robustperf(usys,opt)
```

Description

The performance of a nominally stable uncertain system model will generally degrade for specific values of its uncertain elements. `robustperf`, largely included for historical purposes, computes the robust performance margin, which is one measure of the level of degradation brought on by the modeled uncertainty.

As with other *uncertain-system* analysis tools, only bounds on the performance margin are computed. The exact robust performance margin is guaranteed to lie between these upper and lower bounds.

The computation used in `robustperf` is a frequency-domain calculation. Coupled with stability of the nominal system, this frequency domain calculation determines robust performance of `usys`. If the input system `usys` is a `ufrd`, then the analysis is performed on the frequency grid within the `ufrd`. Note that the stability of the nominal system is not verified by the computation. If the input system sys is a `uss`, then the stability of the nominal system is first checked, an appropriate frequency grid is generated (automatically), and the analysis performed on that frequency grid. In all discussion that follows, N denotes the number of points in the frequency grid.

Basic Syntax

Suppose `usys` is a `ufrd` or `uss` with M uncertain elements. The results of

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

are such that `perfmarg` is a structure with the following fields:

Field	Description
`LowerBound`	Lower bound on robust performance margin, positive scalar.
`UpperBound`	Upper bound on robust performance margin, positive scalar.
`CriticalFrequency`	The value of frequency at which the performance degradation curve crosses the $y = 1/x$ curve. See "Generalized Robustness Analysis".

`perfmargunc` is a `struct` of values of uncertain elements associated with the intersection of the performance degradation curve and the $y = 1/x$ curve. See "Generalized Robustness Analysis". There are M field names, which are the names of uncertain elements of `usys`.

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

If `usys` is an array of uncertain models, the outputs are struct arrays whose entries correspond to each model in the array.

Examples

Create a plant with a nominal model of an integrator, and include additive unmodeled dynamics uncertainty of a level of 0.4 (this corresponds to 100% model uncertainty at 2.5 rads/s).

```
P = tf(1,[1 0]) + ultidyn('delta',[1 1],'bound',0.4);
```

Design a "proportional" controller K that puts the nominal closed-loop bandwidth at 0.8 rad/s. Roll off K at a frequency 25 times the nominal closed-loop bandwidth. Form the closed-loop sensitivity function.

```
BW = 0.8;
K = tf(BW,[1/(25*BW) 1]);
S = feedback(1,P*K);
```

Assess the performance margin of the closed-loop sensitivity function. Because the nominal gain of the sensitivity function is 1, and the performance degradation curve is monotonically increasing (see

"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.

```
[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 "Generalized 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(usys)
```

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

Field	Description
`Sensitivity`	A `struct` with *M* fields, field names are names of uncertain elements of `usys`. Values of fields are positive and contain the local sensitivity of the overall Stability Margin to that element's uncertainty range. For instance, a value of 25 indicates that if the uncertainty range is enlarged by 8%, then the stability margin should drop by about 2% (25% of 8). If the `Sensitivity` property of the `robustperfOptions` object is `'off'`, the values are set to `NaN`.
`Frequency`	*N*-by-1 frequency vector associated with analysis.
`BadUncertainValues`	*N*-by-1 struct array containing the worst uncertain element values at each frequency.

Field	Description
`MussvBnds`	A 1-by-2 `frd`, with upper and lower bounds from `mussv`. The (1,1) entry is the μ-upper bound (corresponds to `perfmarg.LowerBound`) and the (1,2) entry is the μ-lower bound (for `perfmarg.UpperBound`).
`MussvInfo`	Structure of compressed data from `mussv`.

Specifying Additional Options

Use `robustperfOptions` to specify additional options for the `robustperf` computation. For example, you can control what is displayed during the computation, turn the sensitivity computation on or off, set the step size in the sensitivity computation, or control the option argument used in the underlying call to `mussv`. For example, you can turn the display on and turn off the sensitivity by executing

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

See the `robustperfOptions` reference page for more information about available options.

Algorithms

A rigorous robust performance analysis consists of two steps:

1. Verify that the nominal system is stable.
2. Robust performance analysis on an augmented system.

The algorithm in `robustperf` follows this in spirit, with the following limitations:

- If `usys` is a `uss` object, then `robustperf` explicitly checks the stability of the nominal value. However, if `usys` is a `ufrd` model, `robustperf` instead assumes that the nominal value is stable, and does not perform this check.

- The exact performance margin is guaranteed to be no larger than `UpperBound` (some uncertain elements associated with this magnitude cause instability – one instance is returned in the structure `perfmargunc`). The instability created by `perfmargunc` occurs at the frequency value in `CriticalFrequency`.
- Similarly, the exact performance margin is guaranteed to be no smaller than `LowerBound`.

Limitations

Because the calculation is carried out with a frequency gridding, it is possible (likely) that the true critical frequency is missing from the frequency vector used in the analysis. This is similar to the problem in `robuststab`. However, in comparing to `robuststab`, the problem in `robustperf` is less acute. The robust performance margin, considered a function of problem data and frequency, is typically a continuous function (unlike the robust stability margin, described in Getting Reliable Estimates of Robustness Margins). Hence, in robust performance margin calculations, increasing the density of the frequency grid will always increase the accuracy of the answers, and in the limit, answers arbitrarily close to the actual answers are obtainable with finite frequency grids.

See Also

`mussv` | `norm` | `robustperfOptions` | `robuststab` | `actual2normalized` | `wcgain` | `wcsens` | `wcmargin`

robustperfOptions

Purpose

Option set for `robustperf`

Syntax

```
options = robustperfOptions
options = robustperfOptions(Name,Value,...)
```

Description

`options = robustperfOptions` returns the default option set for the `robustperf` command.

`options = robustperfOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

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

`robustperfOptions` takes the following `Name` arguments:

'Display'

String specifying whether `robustperf` displays progress of `mussv` computations.

- `'off'` — Do not display progress.
- `'on'` — Display progress. This setting overrides the silent (`'s'`) option in the `Mussv` string.

Default: `'off'`

'Sensitivity'

String specifying whether `robustperf` computes the sensitivity of the performance margin with respect to each individual uncertain element. This element-by-element sensitivity provides an indication of which elements the performance margin is most sensitive to. Turning off the element-by-element sensitivity calculation speeds up `robustperf`.

- `'on'` — Compute the sensitivity for each uncertain element.
- `'off'` — Do not compute the sensitivity for each uncertain element.

Default: `'on'`

'VaryUncertainty'

Percentage variation of uncertainty for computing sensitivity. The sensitivity estimate uses a finite difference calculation.

Default: 25

'Mussv'

Option string for the `mussv` calculation that `robustperf` performs. See `mussv` for the available options.

Default: `''` (default behavior of `mussv`)

Output Arguments

options

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

Examples

Create an options set for a `robustperf` calculation that displays the progress of the `mussv` calculation. Also, turn off the element-by-element sensitivity calculation.

```
 options = robustperfOptions('Display','on','Sensitivity','off');
```

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

```
options = robustperfOptions;
options.Display = 'on';
options.Sensitivity = 'off';
```

See Also `robustperf`

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 both equal 0.
UpperBound	Upper bound on stability margin, positive scalar. If less than 1, the uncertain system is not stable for all values of the modeled uncertainty.
DestabilizingFrequency	The critical value of frequency at which instability occurs, with uncertain elements closest to their nominal values. At a particular value of uncertain elements (see destabunc below), the poles migrate across the stability boundary (imaginary-axis in continuous-time systems, unit-disk in discrete-time systems) at the frequency given by DestabilizingFrequency.

destabunc is a structure of values of uncertain elements, closest to nominal, that cause instability. There are *M* field names, which are the names of uncertain elements of sys. The value of each field is the corresponding value of the uncertain element, such that when jointly combined, lead to instability. The command pole(usubs(sys,destabunc)) shows the instability. If A is an uncertain element of sys, then

```
actual2normalized(destabunc.A,sys.Uncertainty.A)
```

will be less than or equal to UpperBound, and for at least one uncertain element of sys, this normalized distance will be equal to UpperBound, proving that UpperBound is indeed an upper bound on the robust stability margin.

Report is a text description of the arguments returned by robuststab.

If sys is an array of uncertain models, the outputs are struct arrays whose entries correspond to each model in the array.

Examples

Construct a feedback loop with a second-order plant and a PID controller with approximate differentiation. The second-order plant has frequency-dependent uncertainty, in the form of additive unmodeled dynamics, introduced with an ultidyn object and a shaping filter.

robuststab is used to compute the stability margins of the closed-loop system with respect to the plant model uncertainty.

```
P = tf(4,[1 .8 4]);
delta = ultidyn('delta',[1 1],'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 `robuststabOptions` object is `'off'`, the values are set to `NaN`.
`Frequency`	*N*-by-1 frequency vector associated with analysis.
`BadUncertainValues`	*N*-by-1 struct array containing the destabilizing uncertain element values at each frequency.
`MussvBnds`	A 1-by-2 `frd`, with upper and lower bounds from `mussv`. The (1,1) entry is the μ-upper bound (corresponds to `stabmarg.LowerBound`) and the (1,2) entry is the μ-lower bound (for `stabmarg.UpperBound`).
`MussvInfo`	Structure of compressed data from `mussv`.

Specifying Additional Options

Use `robuststabOptions` to specify additional options for the `robuststab` computation. For example, you can control what is displayed during the computation, turning the sensitivity computation on or off, set the step-size in the sensitivity computation, or control the option argument used in the underlying call to `mussv`. For instance, you can turn the display on, and the sensitivity calculation off by executing

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

See the `robuststabOptions` reference page for more information about available options.

Algorithms

A rigorous robust stability analysis consists of two steps:

1. Verify that the nominal system is stable;
2. Verify that no poles cross the stability boundary as the uncertain elements vary within their ranges.

Because the stability boundary is also associated with the frequency response, the second step can be interpreted (and carried out) as a frequency domain calculation. This amounts to a classical μ-analysis problem.

The algorithm in `robuststab` follows this in spirit, with the following limitations.

- If `sys` is a `uss` object, then the first requirement of stability of nominal value is explicitly checked within `robuststab`. However, if `sys` is an `ufrd`, then the verification of nominal stability from the nominal frequency response data is not performed, and is instead assumed.
- In the second step (monitoring the stability boundary for the migration of poles), rather than check all points on stability boundary, the algorithm only detects migration of poles across the stability boundary at the frequencies in `info.Frequency`.

See "Limitations" on page 2-362 for information about issues related to migration detection.

The exact stability margin is guaranteed to be no larger than `UpperBound` (some uncertain elements associated with this magnitude cause instability – one instance is returned in the structure `destabunc`). The instability created by `destabunc` occurs at the frequency value in `DestabilizingFrequency`.

Similarly, the exact stability margin is guaranteed to be no smaller than `LowerBound`. In other words, for all modeled uncertainty with magnitude up to `LowerBound`, the system is guaranteed stable. These bounds are derived using the upper bound for the structured singular value, which is essentially optimally-scaled, small-gain theorem analysis.

Limitations

Under most conditions, the robust stability margin at each frequency is a continuous function of the problem data at that frequency. Because the problem data, in turn, is a continuous function of frequency, it follows that finite frequency grids are usually adequate in correctly assessing robust stability bounds, assuming the frequency grid is dense enough.

Nevertheless, there are simple examples that violate this. In some problems, the migration of poles from stable to unstable *only* occurs at a finite collection of specific frequencies (generally unknown to you). Any frequency grid that excludes these critical frequencies (and almost every grid will exclude them) will result in undetected migration and misleading results, namely stability margins of ∞.

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

See Also

`loopmargin` | `mussv` | `robuststabOptions` | `robustperf` | `wcgain` | `wcsens` | `wcmargin`

Purpose

Option set for `robuststab`

Syntax

```
options = robuststabOptions
options = robuststabOptions(Name,Value,...)
```

Description

`options = robuststabOptions` returns the default option set for the `robuststab` command.

`options = robuststabOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

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

`robuststabOptions` takes the following `Name` arguments:

'Display'

String specifying whether `robuststab` displays progress of `mussv` computations.

- `'off'` — Do not display progress.
- `'on'` — Display progress. This setting overrides the silent (`'s'`) option in the `Mussv` string.

Default: `'off'`

'Sensitivity'

String specifying whether `robuststab` computes the sensitivity of the stability margin with respect to each individual uncertain element. This element-by-element sensitivity provides an indication of which elements the stability margin is most sensitive to. Turning off the element-by-element sensitivity calculation speeds up `robuststab`.

- `'on'` — Compute the sensitivity for each uncertain element.
- `'off'` — Do not compute the sensitivity for each uncertain element.

  Default: `'on'`

'VaryUncertainty'

Percentage variation of uncertainty for computing sensitivity. The sensitivity estimate uses a finite difference calculation.

Default: 25

'Mussv'

Option string for the `mussv` calculation that `robustperf` performs. See `mussv` for the available options.

Default: `''` (default behavior of `mussv`)

Output Arguments

options

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

Examples

Create an options set for a `robuststab` calculation that displays the progress of the `mussv` calculation. Also, turn off the element-by-element sensitivity calculation.

```
 options = robuststabOptions('Display','on','Sensitivity','off');
```

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

```
options = robuststabOptions;
options.Display = 'on';
options.Sensitivity = 'off';
```

See Also `robuststab`

schurmr

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, $\sigma\iota$.

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

This method guarantees an error bound on the infinity norm of the *additive error* $\|$ `G-GRED` $\|_\infty$ for well-conditioned model reduced problems [1]:

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

This table describes input arguments for `schurmr`.

Argument	Description
`G`	LTI model to be reduced (without any other inputs will plot its Hankel singular values and prompt for reduced order).
`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.

Algorithms

Given a state space (*A,B,C,D*) of a system and *k,* the desired reduced order, the following steps will produce a similarity transformation to truncate the original state-space system to the k^{th} order reduced model [16].

1 Find the controllability and observability grammians *P* and *Q*.

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

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

$$V_D^T PQV_D = \begin{bmatrix} \lambda n & \dots & \dots \\ 0 & \dots & \dots \\ 0 & 0 & \lambda_1 \end{bmatrix}$$

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

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

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

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

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

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:

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

```
end
```

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

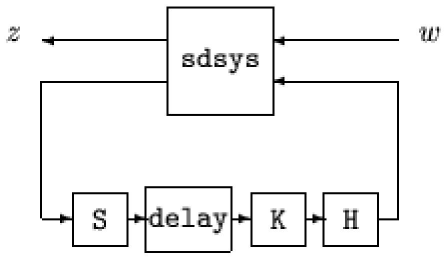
Compute L_2 norm of continuous-time system in feedback with discrete-time system

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
```

Algorithms

`sdhinfnorm` uses variations of the formulas described in the Bamieh and Pearson paper to obtain an equivalent discrete-time system. (These variations are done to improve the numerical conditioning of the algorithms.) A preliminary step is to determine whether the norm of the continuous-time system over one sampling period without control is less than the given value. This requires a search and is, computationally, a relatively expensive step.

References

Bamieh, B.A., and J.B. Pearson, “A General Framework for Linear Periodic Systems with Applications to Sampled-Data Control,” *IEEE Transactions on Automatic Control,* Vol. AC–37, 1992, pp. 418-435.

See Also

`gapmetric` | `hinfsyn` | `norm` | `sdhinfsyn` | `sdlsim`

sdhinfsyn

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,...)
```

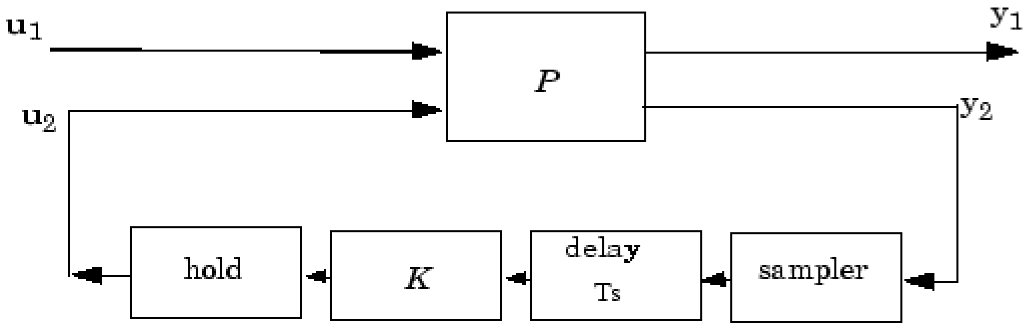
Description

`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

Algorithms

`sdhinfsyn` uses a variation of the formulas described in the Bamieh and Pearson paper [1] to obtain an equivalent discrete-time system. (This is done to improve the numerical conditioning of the algorithms.) A preliminary step is to determine whether the norm of the continuous-time system over one sampling period without control is less than the given γ-value. This requires a search and is computationally a relatively expensive step.

References

[1] Bamieh, B.A., and J.B. Pearson, "A General Framework for Linear Periodic Systems with Applications to Sampled-Data Control," *IEEE Transactions on Automatic Control,* Vol. AC–37, 1992, pp. 418-435.

See Also

`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]';
```

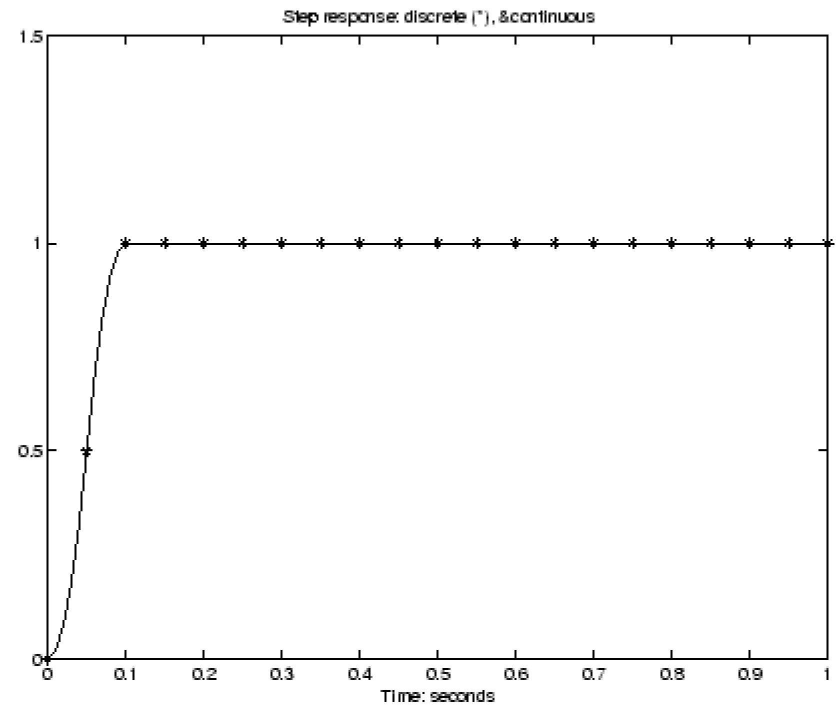
```
sysoutname = 'dclp';
cleanupsysic = 'yes';
sysic;
```

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

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

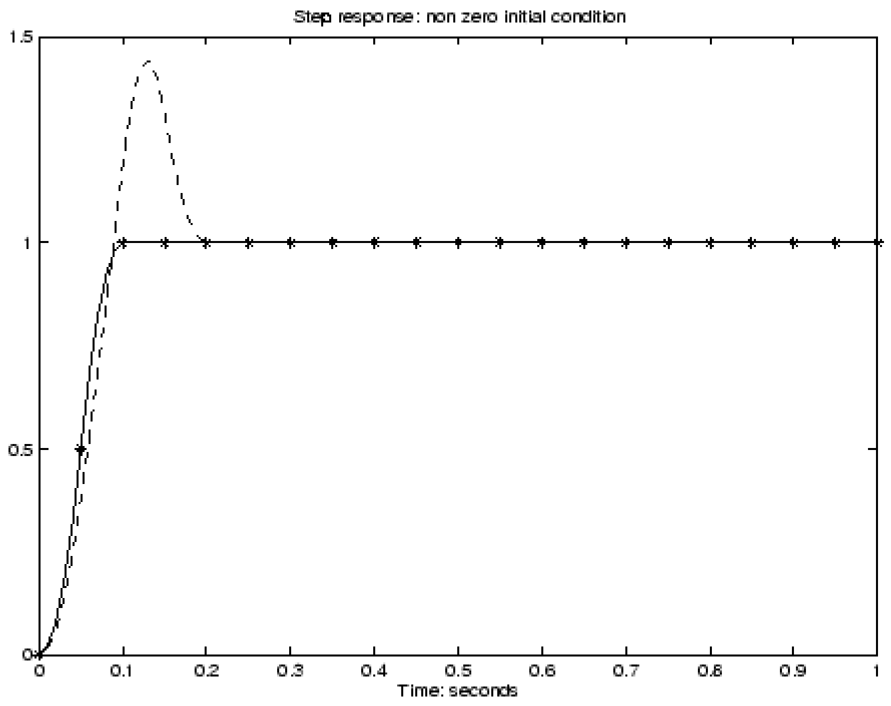
The continuous interconnection is set up and the sampled data response is calculated with `sdlsim`.

```
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')
```



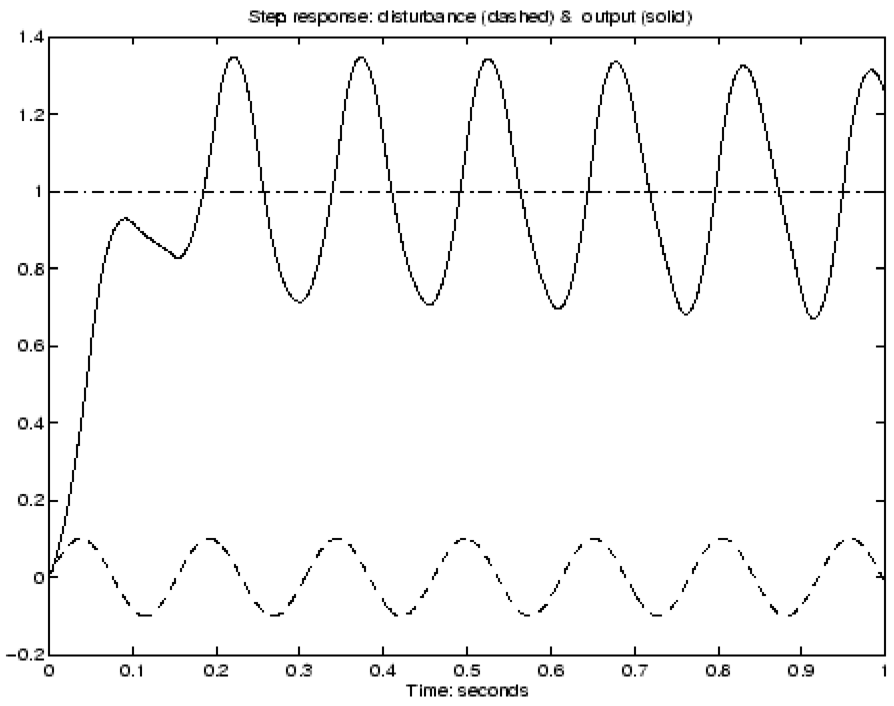
You can see the effect of a nonzero initial condition in the continuous-time system. Note how examining the system at only the sample points will underestimate the amplitude of the overshoot.

```
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)')
```



Algorithms

`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

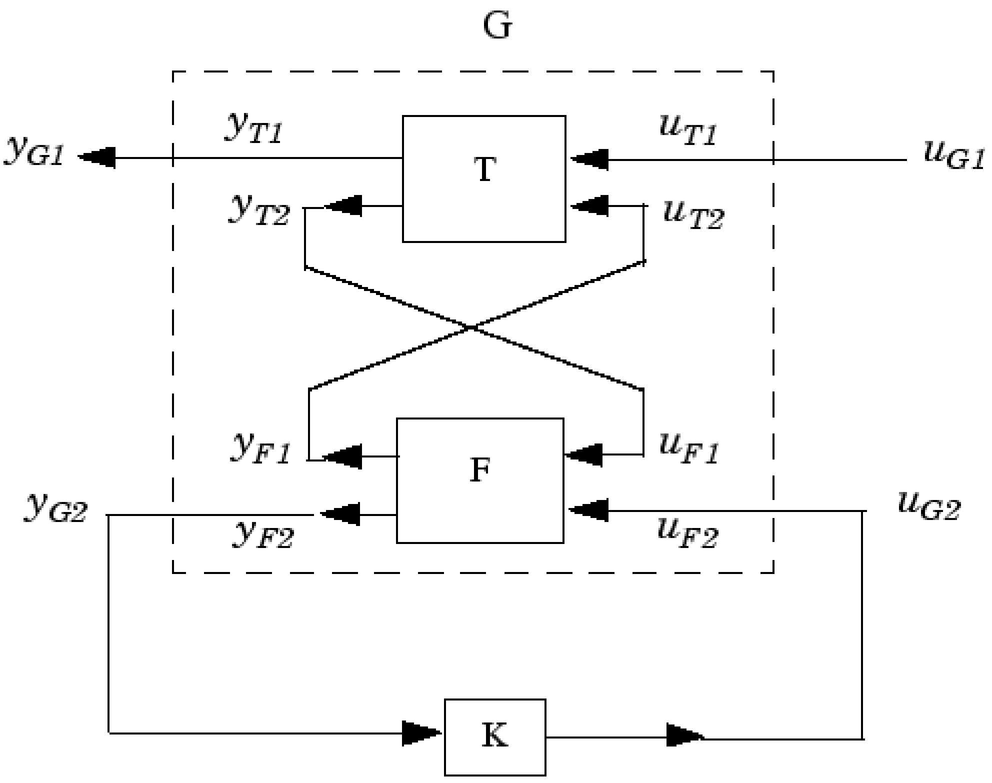
```
[G,T] = sectf(F,SECF,SECG)
```

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

where `A`,`B` are scalars in $[-_\infty, {}_\infty]$ or square matrices; `a,b` are vectors; `S=[S11 S12;S21,S22]` is a square matrix whose blocks `S11,S12,S21,S22` are either scalars or square matrices; `S` is a two-port system `S=mksys`(a,b1,b2,`...`,`'tss'`) with transfer function

$$S(s) = \begin{bmatrix} S_{11}(s) & S_{12}(s) \\ S_{21}(s) & S_{22}(s) \end{bmatrix}$$

Output Arguments	Description
`G`	Transformed plant $G(s)$=`lftf(T,F)`
`T`	LFT sector transform, maps conic sector `SECF` into conic sector `SECG`

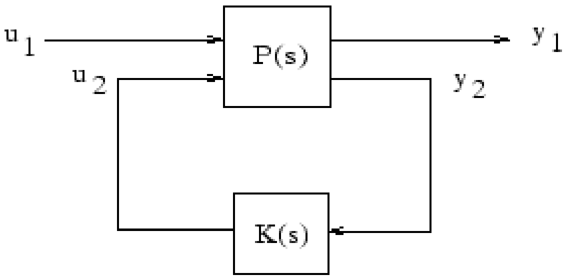
Output Variables	
`G`	The transformed plant $G(s)$ = `lftf(T,F)`:
`T`	The linear fractional transformation $T(s)$ = `T`

Examples

The statement $G(j\omega)$ inside sector[–1, 1] is equivalent to the H_∞ inequality

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

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, $_\infty$]. 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 2-387.



Sector Transform Block Diagram

Here is a simple example of the sector transform.

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

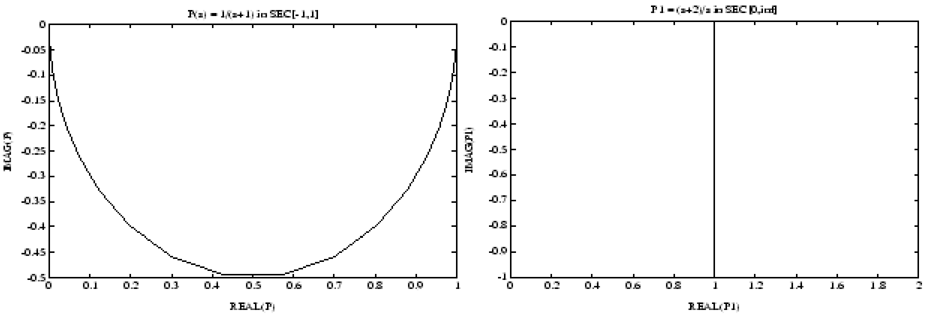
You can compute this by simply executing the following commands:

```
P = ss(tf(1,[1 1]));
```

```
P1 = sectf(P,[-1,1],[0,Inf]);
```

The Nyquist plots for this transformation are depicted in Example of Sector Transform on page 2-387. The condition $P_1(s)$ inside $[0, _\infty]$ implies that $P_1(s)$ is stable and $P_1(j\omega)$ is *positive real*, i.e.,

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



Example of Sector Transform

Algorithms

`sectf` uses the generalization of the sector concept of [3] described by [1]. First the sector input data `Sf= SECF` and `Sg=SECG` is converted to two-port state-space form; non-dynamical sectors are handled with empty *a*, *b1*, *b2*, *c1*, *c2* matrices. Next the equation

$$S_{g(}s)\begin{bmatrix} u_{g_1} \\ y_{g_1} \end{bmatrix} = S_f(s)\begin{bmatrix} u_{f_1} \\ y_{f_1} \end{bmatrix}$$

is solved for the two-port transfer function $T(s)$ from $u_{g_1}y_{f_1}$ to $u_{f_1}y_{g_1}$. Finally, the function `lftf` is used to compute $G(s)$ as `G = lftf(T,F)`.

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 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`

setmvar

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

$$x^{\cdot} = Ax + Bu$$

and the problem of finding a stabilizing state-feedback law $u = Kx$ where K is an unknown matrix.

By the Lyapunov Theorem, this is equivalent to finding $P > 0$ and K such that

$$(A + BK)P + P(A + BK^T) + I < 0.$$

With the change of variable $Y := KP$, this condition reduces to the LMI

$$AP + PA^T + BY + Y^TB^T + I < 0.$$

This LMI is entered by the commands

```
n = size(A,1)                    % number of states
```

```
ncon = size(B,2)              % number of inputs

setlmis([])
P = lmivar(1,[n 1])           % P full symmetric
Y = lmivar(2,[ncon n])        % Y rectangular

lmiterm([1 1 1 P],A,1,'s')    % AP+PA'
lmiterm([1 1 1 Y],B,1,'s')    % BY+Y'B'
lmiterm([1 1 1 0],1)          % I
lmis = getlmis
```

To find out whether this problem has a solution K for the particular Lyapunov matrix $P = I$, set P to I by typing

```
news = setmvar(lmis,P,1)
```

The resulting LMI system `news` has only one variable $Y = K$. Its feasibility is assessed by calling `feasp`:

```
[tmin,xfeas] = feasp(news)
Y = dec2mat(news,xfeas,Y)
```

The computed Y is feasible whenever `tmin` < 0.

See Also

`evallmi | delmvar`

showlmi

Purpose

Return left and right sides of LMI after evaluation of all variable terms

Syntax

```
[lhs,rhs] = showlmi(evalsys,n)
```

Description

For given values of the decision variables, the function `evallmi` evaluates all variable terms in a system of LMIs. The left and right sides of the *n*-th LMI are then constant matrices that can be displayed with `showlmi`. If `evalsys` is the output of `evallmi`, the values `lhs` and `rhs` of these left and right sides are given by

```
[lhs,rhs] = showlmi(evalsys,n)
```

An error is issued if `evalsys` still contains variable terms.

Examples

See the description of `evallmi`.

See Also

`evallmi` | `setmvar`

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
```

Algorithms

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

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`

skewdec

Purpose Form skew-symmetric matrix

Syntax x = skewdec(m,n)

Description `skewdec(m,n)` forms the m-by-m skew-symmetric matrix

$$\begin{bmatrix} 0 & -(n-1) & -(n-2) & \ldots \\ (n+1) & 0 & -(n-3) & \ldots \\ (n+2) & (n+3) & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{bmatrix}$$

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`

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`.

$[G_1(s)] := (A_{11}, B_1, C_1, D_1)$ denotes the slow part of $G(s)$. The slow poles have low frequency and magnitude values.

$[G_2(s)] := (A_{22}, B_2, C_2, D_2)$ 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 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.

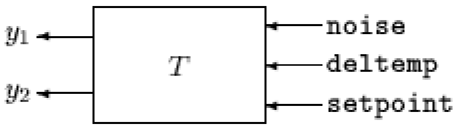
`sysic` also requires that for every subsystem name listed in `systemnames`, a corresponding variable, `input_to_ListedSubSystemName` must exist in the calling workspace. This variable is similar to `outputvar` – it defines the input signals to this particular subsystem as linear combinations of individual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

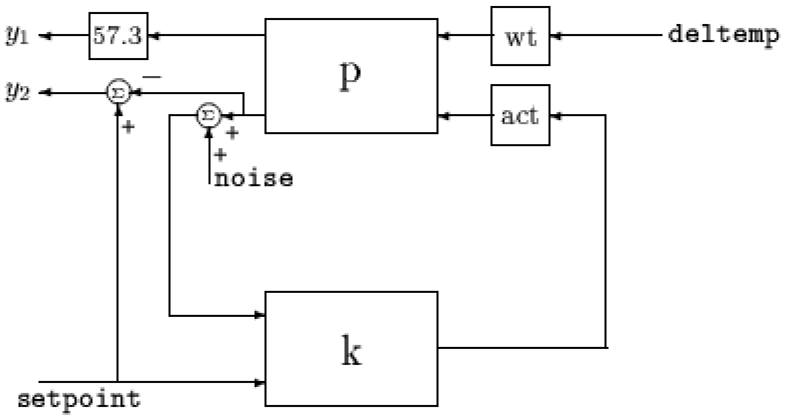
After running `sysic` the variables `systemnames, inputvar, outputvar` and all of the `input_to_ListedSubSystemName` will exist in the workspace. Setting the optional variable `cleanupsysic` to `'yes'` will cause these variables to be removed from the workspace after `sysic` has formed the interconnection.

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`

systune

Purpose

Tune fixed-structure control systems

Syntax

```
[CL,fSoft] = systune(CL0,SoftReqs)
[CL,fSoft,gHard] = systune(CL0,SoftReqs,HardReqs)
[CL,fSoft,gHard] = systune(CL0,SoftReqs,HardReqs,options)
[CL,fSoft,gHard,info] = systune( ___ )
```

Description

`[CL,fSoft] = systune(CL0,SoftReqs)` tunes the free parameters of the control system model, `CL0`, to best meet the soft tuning requirements. The best achieved soft constraint values are returned as `fSoft`.

Note For tuning Simulink models with `systune`, see use `slTunable` to create an interface to your Simulink model. You can then tune the control system with `slTunable.systune` (requires Simulink Control Design).

`[CL,fSoft,gHard] = systune(CL0,SoftReqs,HardReqs)` tunes the control system to best meet the soft tuning requirements subject to satisfying the hard tuning requirements (constraints). It returns the best achieved values for the soft and hard constraints.

`[CL,fSoft,gHard] = systune(CL0,SoftReqs,HardReqs,options)` specifies options for the optimization.

`[CL,fSoft,gHard,info] = systune( ___ )` also returns detailed information about each optimization run. All input arguments described for the previous syntaxes also apply here.

Input Arguments

CL0 - Control system to tune

generalized state-space model | model array

Control system to tune, specified as a generalized state-space (`genss`) model or array of models with tunable parameters. To construct `CL0`:

1. Parameterize the tunable elements of your control system. You can use predefined structures, such as `ltiblock.pid`, `ltiblock.gain`, and `ltiblock.tf`. Alternatively, you can create your own structure from elementary tunable parameters (`realp`).
2. Build a closed-loop model of the overall control system as an interconnection of fixed and tunable components. To do sou Use model interconnection commands such as `feedback` and `connect`. Use `loopswitch` blocks to mark optional loop-opening locations for specifying and assessing open-loop requirements.

Specify an array of tunable `genss` models that have the same tunable parameters for robust tuning of a controller against a set of plant models.

SoftReqs - Soft tuning requirements (objectives)

vector of `TuningGoal` requirement objects

Soft tuning requirements (objectives) for tuning the control system, specified as a vector of `TuningGoal` requirement objects. These requirement objects can include `TuningGoal.Tracking`, `TuningGoal.Gain`, or `TuningGoal.Margins` (See "Specifying Design Requirements for systune" for a complete list.).

`systune` tunes the tunable parameters of the control system to minimize the soft tuning requirements. This tuning is subject to satisfying the hard tuning requirements (if any).

HardReqs - Hard tuning requirements (constraints)

`[]` (default) | vector of `TuningGoal` requirement objects

Hard tuning requirements (constraints) for tuning the control system, specified as a vector of `TuningGoal` requirement objects.

These requirement objects can include `TuningGoal.Tracking`, `TuningGoal.Gain`, or `TuningGoal.Margins`. (See "Specifying Design Requirements for systune" for a complete list.)

`systune` converts each hard tuning requirement to a normalized scalar value. `systune` then optimizes the free parameters minimize those normalized values. A hard requirement is satisfied if the normalized value is less than 1.

options - Options for tuning algorithm

`systuneOptions` object

Options for the tuning algorithm, specified as an options set you create with `systuneOptions`. Available options include:

- Number of additional optimizations to run. Each optimization starts from random initial values of the free parameters.
- Tolerance for terminating the optimization.
- Flag for using parallel processing.

Output Arguments

CL - Tuned control system

generalized state-space model

Tuned control system, returned as a generalized state-space (`genss`) model. This model has the same number and type of tunable elements (Control Design Blocks) as `CL0`. The current values of these elements are the tuned parameters. Use `getBlockValue` or `showTunable` to access values of the tuned elements.

If you provide an array of control system models to tune as the input argument, `CL0`, `systune` tunes the parameters of all the models simultaneously. In this case, `CL` is an array of tuned `genss` models. For more information, see "Tune Controller Against Set of Plant Models".

fSoft - Best achieved soft constraint values

vector

Best achieved soft constraint values, returned as a vector. `systune` converts the soft requirements to a function of the free parameters of the control system. The command then tunes the parameters to minimize that function subject to the hard constraints. (See “Algorithms” on page 2-427.) `fSoft` contains the best achieved value for each of the soft constraints. These values appear in `fSoft` in the same order that the constraints are specified in `SoftReqs`. `fSoft` values are meaningful only when the hard constraints are satisfied.

gHard - Best achieved hard constraint values

vector

Best achieved hard constraint values, returned as a vector. `systune` converts the hard requirements to a function of the free parameters of the control system. The command then tunes the parameters to drive those values below 1. (See “Algorithms” on page 2-427.) `gHard` contains the best achieved value for each of the hard constraints. These values appear in `gHard` in the same order that the constraints are specified in `HardReqs`. If all values are less than 1, then the hard constraints are satisfied.

info - Detailed information about optimization runs

structure

Detailed information about each optimization run, returned as a data structure. In addition to examining detailed results of the optimization, you can use `info` as an input to `viewSpec` when validating a tuned MIMO system. `info` contains scaling data that `viewSpec` needs for correct evaluation of MIMO open-loop requirements such as loop shapes and stability margins.

The fields of `info` are:

Run - Run number

scalar

Run number, returned as a scalar. If you use the `RandomStart` option of `systuneOptions` to perform multiple optimization runs, `info` is a struct array, and `info.Run` is the index.

Iterations - Total number of iterations

scalar

Total number of iterations performed during run, returned as a scalar. This value is the number of iterations performed in each run before the optimization terminates.

fBest - Best overall soft constraint value

scalar

Best overall soft constraint value, returned as a scalar. `systune` converts the soft requirements to a function of the free parameters of the control system. The command then tunes the parameters to minimize that function subject to the hard constraints. (See "Algorithms" on page 2-427.) `info.fBest` is the maximum soft constraint value at the final iteration. This value is meaningful only when the hard constraints are satisfied.

gBest - Best overall hard constraint value

scalar

Best overall hard constraint value, returned as a scalar. `systune` converts the hard requirements to a function of the free parameters of the control system. The command then tunes the parameters to drive those values below 1. (See "Algorithms" on page 2-427.) `info.gBest` is the maximum hard constraint value at the final iteration. This value must be less than 1 for the hard constraints to be satisfied.

fSoft - Individual soft constraint values

vector

Individual soft constraint values, returned as a vector. `systune` converts each soft requirement to a normalized value that is a function of the free parameters of the control system. The command then tunes the parameters to minimize that value subject to the hard constraints. (See "Algorithms" on page 2-427.) `info.fSoft` contains the individual values of the soft constraints at the end of each run. These values appear in `fSoft` in the same order that the constraints are specified in `SoftReqs`.

gHard - Individual hard constraint values

vector

Individual hard constraint values, returned as a vector. `systune` converts each hard requirement to a normalized value that is a function of the free parameters of the control system. The command then tunes the parameters to minimize those values. A hard requirement is satisfied if its value is less than 1. (See "Algorithms" on page 2-427.) `info.gHard` contains the individual values of the hard constraints at the end of each run. These values appear in `gHard` in the same order that the constraints are specified in `HardReqs`.

MinDecay - Minimum decay rate of closed-loop poles

vector

Minimum decay rate of closed-loop poles, returned as a vector.

By default, closed-loop pole locations of the tuned system are constrained to satisfy $\mathrm{Re}(p) < -10^{-7}$. Use the `MinDecay` option of `systuneOptions` to change this constraint.

Blocks - Tuned values of tunable blocks and parameters

structure

Tuned values of tunable blocks and parameters in the tuned control system, `CL`, returned as a structure. You can also use `getBlockValue` or `showBlockValue` to access the tuned parameter values.

LoopScaling - Optimal diagonal scaling for MIMO tuning requirements

state-space model

Optimal diagonal scaling for evaluating MIMO tuning requirements, returned as a state-space model.

When applied to multiloop control systems, `TuningGoal.LoopShape` and `TuningGoal.Margins` can be sensitive to the scaling of the loop transfer functions to which they apply. This sensitivity can lead to poor optimization results. `systune` automatically corrects scaling issues and returns the optimal diagonal scaling matrix `d` as a state-space model in `info.LoopScaling`.

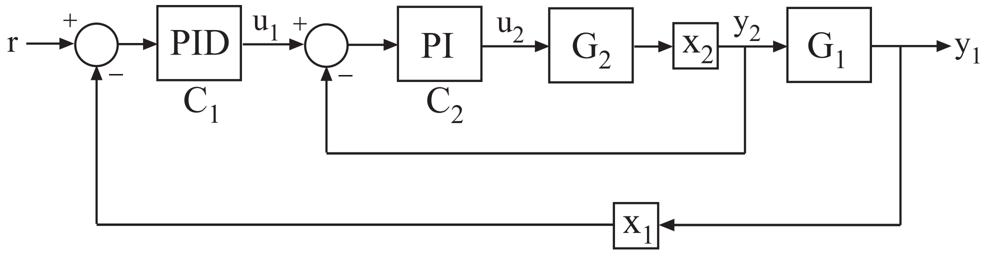
The loop channels associated with each diagonal entry of `D` are listed in `info.LoopScaling.InputName`. The scaled loop transfer is `D\L*D`, where `L` is the open-loop transfer measured at the locations `info.LoopScaling.InputName`.

Examples

Tune Control System to Soft Requirements

Tune a cascaded control system to meet requirements of reference tracking and disturbance rejection.

The cascaded control system of the following illustration includes two tunable controllers, the PI controller for the inner loop, C_2, and the PID controller for the outer loop, C_1.



x_1 and x_2 mark optional loop opening locations, where loops can be opened or signals injected for the purpose of specifying requirements for tuning the system.

Tune the free parameters of this control system to meet the following requirements:

- The output signal, y_1, tracks the reference signal, r, with a response time of 10 seconds and a steady-state error of 1%.
- A disturbance injected at x_2 is suppressed at y_1 by a factor of 10.

Create tunable Control Design Blocks to represent the controllers, and numeric LTI models to represent the plants. Also, create switch blocks to represent the loop opening locations.

```
G2 = zpk([],-2,3);
G1 = zpk([],[-1 -1 -1],10);
```

```
C20 = ltiblock.pid('C2','pi');
C10 = ltiblock.pid('C1','pid');

X1 = loopswitch('X1');
X2 = loopswitch('X2');
```

The `loopswitch` blocks represent optional loop-opening locations that are closed by default. You can also use these locations to specify input or output signals for tuning requirements.

Connect these components to build a model of the entire closed-loop control system.

```
InnerLoop = feedback(X2*G2*C20,1);
CL0 = feedback(G1*InnerLoop*C10,X1);
CL0.InputName = 'r';
CL0.OutputName = 'y';
```

`CL0` is a tunable `genss` model. Specifying names for the input and output channels allows you to identify them when you specify tuning requirements for the system.

Specify tuning requirements for reference tracking and disturbance rejection.

```
Rtrack = TuningGoal.Tracking('r','y',10,0.01);
Rreject = TuningGoal.Gain('X2','y',0.1);
```

The `TuningGoal.Tracking` requirement specifies that the signal at `'y'` track the signal at `'r'` with a response time of 10 seconds and a tracking error of 1%.

The `TuningGoal.Gain` requirement limits the gain from the implicit input associated with the `loopswitch` block, `X2`, to `'y'`. Limiting this gain to a value less than 1 ensures that a disturbance injected at `X2` is suppressed at the output.

Tune the control system.

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

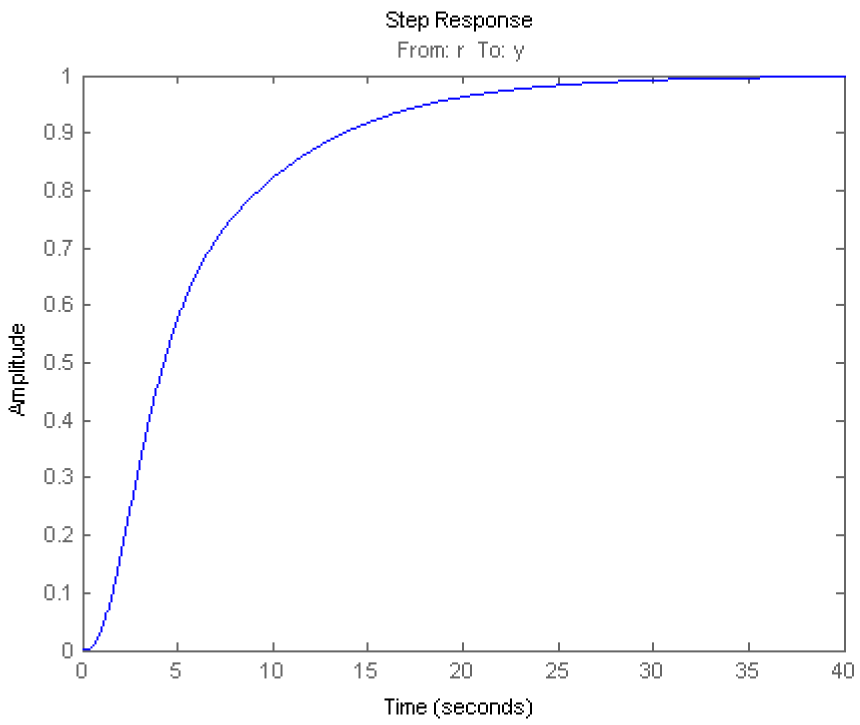
```
Final: Soft = 1.24, Hard = -Inf, Iterations = 109
```

`systune` converts each tuning requirement into a normalized scalar value, *f*. The command adjusts the tunable parameters of `CL0` to minimize the *f* values. For each requirement, the requirement is satisfied if $f < 1$ and violated if $f > 1$. `fSoft` is the vector of minimized *f* values. The largest of the minimized *f* values is displayed as `Soft`.

The output model `CL` is the tuned version of `CL0`. `CL` contains the same Control Design Blocks as `CL0`, with current values equal to the tuned parameter values.

Validate that the tuned control system meets the tracking requirement by examining the step response from `'r'` to `'y'`.

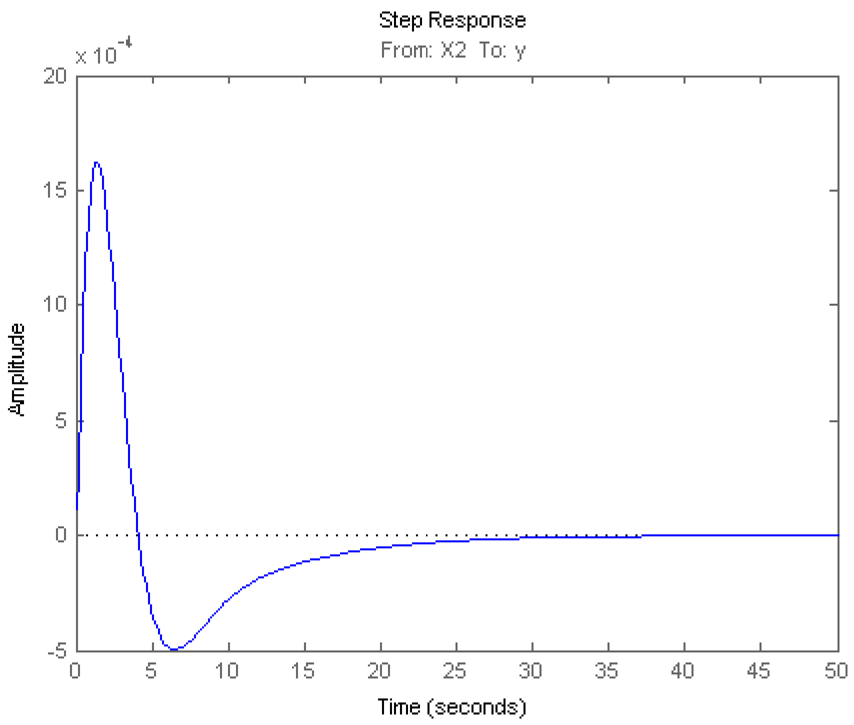
```
stepplot(CL)
```



The step plot shows that in the tuned control system, CL, the output tracks the input with approximately the desired response time.

Validate the tuned system against the disturbance rejection requirement by examining the closed-loop response to a signal injected at X2.

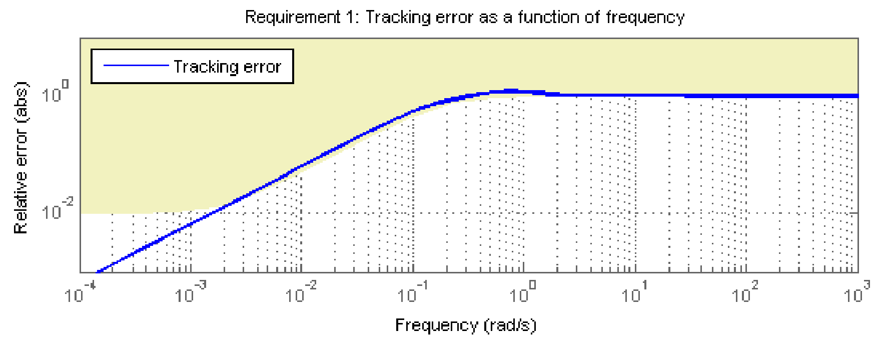
```
CLdist = getIOTransfer(CL,'X2','y');
stepplot(CLdist);
```

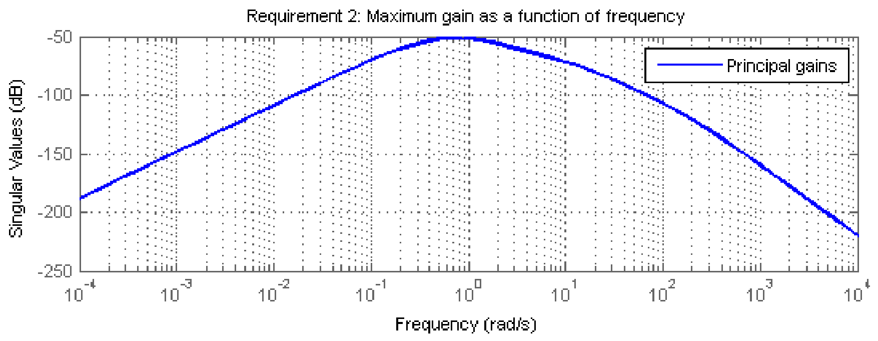


`getIOTransfer` extracts the closed-loop response from the specified inputs to outputs. In general, `getIOTransfer` and `getLoopTransfer` are useful for validating a control system tuned with `systune`.

You can also use `viewSpec` to compare the responses of the tuned control system directly against the tuning requirements, `Rtrack` and `Rreject`.

```
viewSpec([Rtrack,Rreject],CL)
```

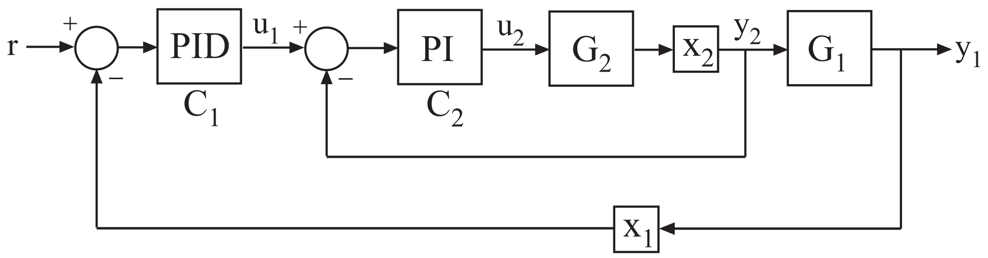




Tune Control System to Both Hard and Soft Requirements

Tune a cascaded control system to meet requirements of reference tracking and disturbance rejection. These requirements are subject to a hard constraint on the stability margins of the inner and outer loops.

The cascaded control system of the following illustration includes two tunable controllers, the PI controller for the inner loop, C_2, and the PID controller for the outer loop, C_1.



x_1 and x_2 mark optional loop opening locations. These locations indicate where you can open loops or inject signals for the purpose of specifying requirements for tuning the system.

Tune the free parameters of this control system to meet the following requirements:

- The output signal, y_1, tracks the reference signal at r with a response time of 10 seconds and a steady-state error of 1%.
- A disturbance injected at x_2 is suppressed at the output, y_1, by a factor of 10.

Impose these tuning requirements subject to hard constraints on the stability margins of both loops.

Create tunable Control Design Blocks to represent the controllers and numeric LTI models to represent the plants. Also, create switch blocks to represent the loop opening locations.

```
G2 = zpk([],-2,3);
G1 = zpk([],[-1 -1 -1],10);

C20 = ltiblock.pid('C2','pi');
C10 = ltiblock.pid('C1','pid');

X1 = loopswitch('X1');
X2 = loopswitch('X2');
```

The `loopswitch` blocks represent optional loop-opening locations that are closed by default. You can also use these locations to specify input or output signals for tuning requirements.

Connect these components to build a model of the entire closed-loop control system.

```
InnerLoop = feedback(X2*G2*C20,1);
CL0 = feedback(G1*InnerLoop*C10,X1);
CL0.InputName = 'r';
CL0.OutputName = 'y';
```

`CL0` is a tunable `genss` model. Specifying names for the input and output channels allows you to identify them when you specify tuning requirements for the system.

Specify tuning requirements for reference tracking and disturbance rejection.

```
Rtrack = TuningGoal.Tracking('r','y',10,0.01);
Rreject = TuningGoal.Gain('X2','y',0.1);
```

The `TuningGoal.Tracking` requirement specifies that the signal at `'y'` tracks the signal at `'r'` with a response time of 10 seconds and a tracking error of 1%.

The `TuningGoal.Gain` requirement limits the gain from the implicit input associated with the `loopswitch` block `X2` to the output, `'y'`. Limiting this gain to a value less than 1 ensures that a disturbance injected at `X2` is suppressed at the output.

Specify tuning requirements for the gain and phase margins.

```
RmargOut = TuningGoal.Margins('X1',18,60);
RmargIn = TuningGoal.Margins('X2',18,60);
RmargIn.Openings = 'X1';
```

`RmargOut` imposes a minimum gain margin of 18 dB and a minimum phase margin of 60 degrees. Specifying `X1` imposes that requirement on the outer loop. Similarly, `RmargIn` imposes the same requirements on the inner loop, identified by `X2`. To ensure that the inner loop margins are evaluated with the outer loop open, include the outer loop-opening location, `X1`, in `RmargIn.Openings`.

Tune the control system to meet the soft requirements of tracking and disturbance rejection, subject to the hard constraints of the stability margins.

```
SoftReqs = [Rtrack,Rreject];
HardReqs = [RmargIn,RmargOut];
[CL,fSoft,gHard] = systune(CL0,SoftReqs,HardReqs);
```

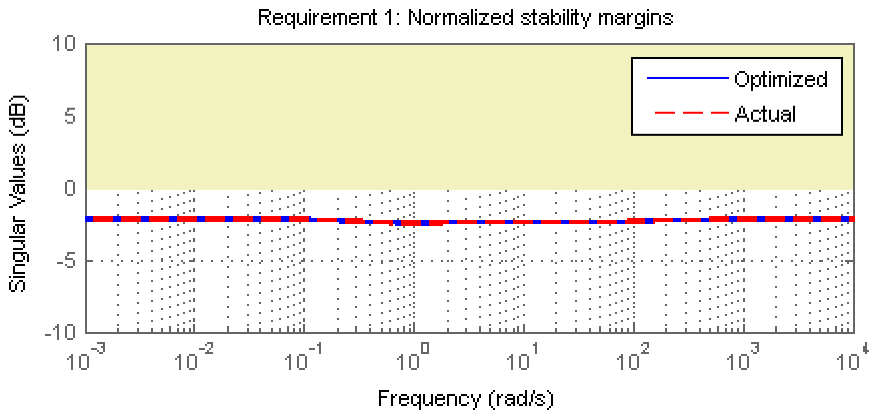
```
Final: Soft = 1.71, Hard = 0.9998, Iterations = 208
```

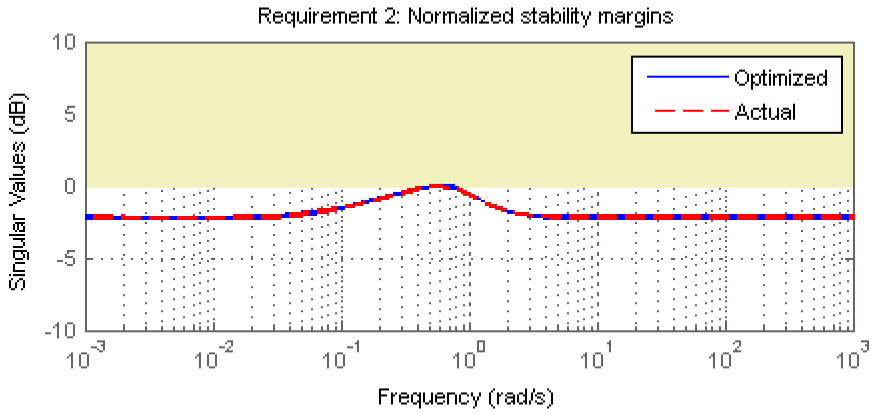
`systune` converts each tuning requirement into a normalized scalar value, *f* for the soft constraints and *g* for the hard constraints. The command adjusts the tunable parameters of `CL0` to minimize the *f* values, subject to the constraint that each $h < 1$.

The displayed value `Hard` is the largest of the minimized *h* values in `gHard`. This value is less than 1, indicating that both the hard constraints are satisfied.

Validate the tuned control system against the stability margin requirements.

```
viewSpec(HardReqs,CL)
```

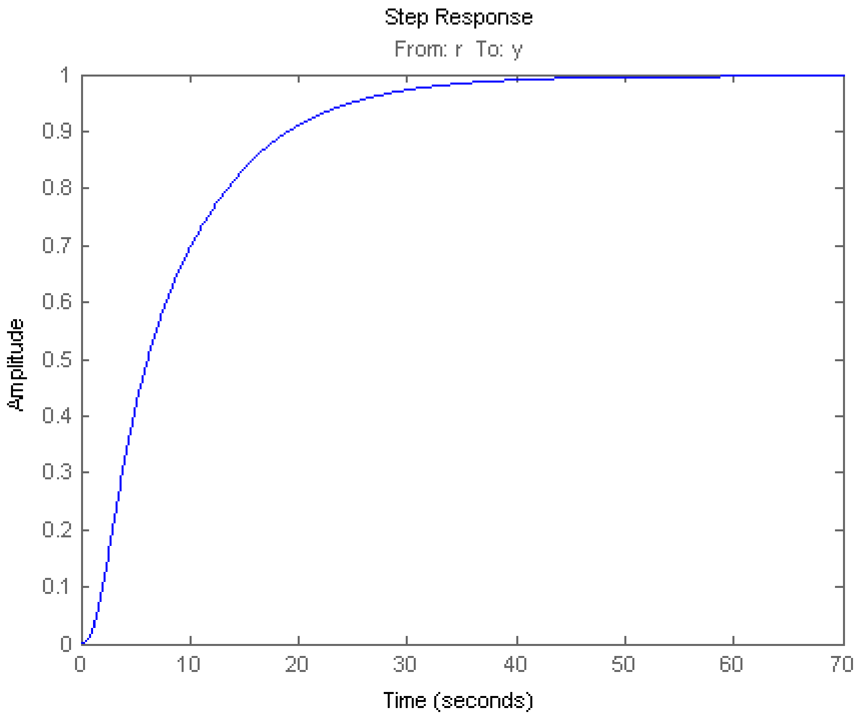




The `viewSpec` plot confirms that the stability margin requirements for both loops are satisfied by the tuned control system at all frequencies. The red liness represent the actual stability margins of the tuned system. The blue lines represent the margin used in the optimization calculation, which is an upper bound on the actual margin.

Examine whether the tuned control system meets the tracking requirement by examining the step response from `'r'` to `'y'`.

```
stepplot(CL)
```



The step plot shows that in the tuned control system, `CL`, the output tracks the input but the response is somewhat slower than desired. Thus, the tuned control system cannot meet the soft requirement of a fast response time subject to the hard constraints of the stability margins. To achieve the desired performance, you may need to relax one of your requirements or convert one or more hard constraints to soft constraints.

Algorithms

x is the vector of tunable parameters in the control system to tune. `systune` converts each soft and hard tuning requirement `SoftReqs(i)` and `HardReqs(j)` into normalized values $f_i(x)$ and $g_j(x)$, respectively. `systune` then solves the minimization problem:

Minimize $\max_i f_i(x)$ subject to $\max_j g_j(x) < 1$, for $x_{\min} < x < x_{\max}$.

x_{min} and x_{max} are the minimum and maximum values of the free parameters of the control system.

`systune` returns the control system with parameters tuned to the values that best solve the minimization problem. `systune` also returns the best achieved values of $f_i(x)$ and $g_j(x)$, as `fSoft` and `gHard` respectively.

For information about the functions $f_i(x)$ and $g_j(x)$ for each type of constraint, see the reference pages for each `TuningGoal` requirement object.

`systune` uses the nonsmooth optimization algorithms described in [1].

`systune` computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] Apkarian, P. and D. Noll, "Nonsmooth H-infinity Synthesis." *IEEE Transactions on Automatic Control*, Vol. 51, No. 1, (2006), pp. 71–86.

[2] Bruisma, N.A. and M. Steinbuch, "A Fast Algorithm to Compute the H_∞-Norm of a Transfer Function Matrix," *System Control Letters*, Vol. 14, No, 4 (1990), pp. 287–293.

See Also

`systuneOptions` | `viewSpec` | `genssslTunable.systune` | | `looptune` | `loopswitchslTunable` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Margins` |

Related Examples

- “Tuning Control Systems with SYSTUNE”
- “Building Tunable Models”

Concepts

- “Set Up Your Control System for Tuning with systune”
- “Specifying Design Requirements for systune”
- “Generalized Models”

Purpose

Set options for systune

Syntax

```
options = systuneOptions
options = systuneOptions(Name,Value)
```

Description

`options = systuneOptions` returns the default option set for the `systune` command.

`options = systuneOptions(Name,Value)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

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

`systuneOptions` takes the following `Name` arguments:

'Display'

Amount of information to display during `systune` runs.

`Display` takes the following values:

- `'final'` — Display a one-line summary at the end of each optimization run. The display includes the best achieved values for the soft and hard constraints, `fSoft` and `gHard`. The display also includes the number of iterations for each run.
- `'sub'` — Display the result of each optimization subproblem.
- `'iter'` — Display optimization progress after each iteration. The display includes the value after each iteration of the objective parameter being minimized. The objective parameter is whichever is larger of the hard constraints or the scaled soft constraints. The display also includes a progress value that indicates the percent change in the constraints from the previous iteration.

- `'off'` — Run in silent mode, displaying no information during or after the run.

  Default: `'final'`

'MaxIter'

Maximum number of iterations in each optimization run.

Default: 300

'RandomStart'

Number of additional optimizations starting from random values of the free parameters in the controller.

If `RandomStart = 0`, `systune` performs a single optimization run starting from the initial values of the tunable parameters. Setting `RandomStart = N > 0` runs *N* additional optimizations starting from *N* randomly generated parameter values.

`systune` tunes by finding a local minimum of a gain minimization problem. To increase the likelihood of finding parameter values that meet your design requirements, set `RandomStart > 0`. You can then use the best design that results from the multiple optimization runs.

Use with `UseParallel = true` to distribute independent optimization runs among MATLAB workers (requires Parallel Computing Toolbox software).

Default: 0

'UseParallel'

Parallel processing flag.

Set to `true` to enable parallel processing by distributing randomized starts among workers in a parallel pool. If there is an available parallel pool, then the software performs independent optimization runs concurrently among workers in that pool. If no parallel pool is available, one of the following occurs:

- If **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, then the software starts a parallel pool using the settings in those preferences.
- If **Automatically create a parallel pool** is not selected in your preferences, then the software performs the optimization runs successively, without parallel processing.

If **Automatically create a parallel pool** is not selected in your preferences, you can manually start a parallel pool using `parpool` before running the tuning command.

Using parallel processing requires Parallel Computing Toolbox software.

Default: `false`

'SoftTarget'

Target value for soft constraints.

The optimization stops when the largest soft constraint value falls below the specified `SoftTarget` value. The default value `SoftTarget = 0` minimizes the soft constrains subject to satisfying the hard constraints.

Default: 0

'SoftTol'

Relative tolerance for termination.

The optimization terminates when the relative decrease in the soft constraint value decreases by less than `SoftTol` over 10 consecutive iterations. Increasing `SoftTol` speeds up termination, and decreasing `SoftTol` yields tighter final values.

Default: 0.001

'SoftScale'

A priori estimate of best soft constraint value.

For problems that mix soft and hard constraints, providing a rough estimate of the optimal value of the soft constraints (subject to the hard constraints) helps to speed up the optimization.

Default: 1

'ScalingOrder'

D-scaling order.

The D-scaling order is the number of states in the diagonal scalings involved in computing MIMO stability margins and loop shapes. Increasing this order can improve results at the expense of additional computations.

When tuning to stability margin requirements, use `viewspec` to assess the gap between the optimized margins and the actual margins. Try increasing the scaling order if this gap is too large.

Default: 0

'MinDecay'

Minimum decay rate for closed-loop poles.

Constrains all closed-loop pole locations `|p|` to satisfy `Re(p) < -MinDecay`. Adjust the minimum value if the optimization cannot meet the default minimum value, or if the default minimum value conflicts with other requirements. For specifying other constraints on the closed-loop pole locations, use `TuningGoal.Poles`.

Default: `1e-7`

Output Arguments

options

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

Examples

Create Options Set for `systune`

Create an options set for a `systune` run using five random restarts. Also, set the display level to show the progress of each iteration, and increase the relative tolerance of the soft constraint value to 0.01.

```
options = systuneOptions('RandomStart',5,'Display','iter',...
                         'SoftTol',0.01);
```

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

```
options = systuneOptions;
options.RandomStart = 5;
options.Display = 'iter';
options.SoftTol = 0.01;
```

Configure Option Set for Parallel Optimization Runs

Configure an option set for a `systune` run using 20 random restarts. Execute these independent optimization runs concurrently on multiple workers in a parallel pool.

If you have the Parallel Computing Toolbox software installed, you can use parallel computing to speed up `systune` tuning of fixed-structure control systems. When you run multiple randomized `systune` optimization starts, parallel computing speeds up tuning by distributing the optimization runs among workers.

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences, manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a `systuneOptions` set that specifies 20 random restarts to run in parallel.

```
options = systuneOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the `systuneOptions` set when you call `systune`. For example, suppose you have already created a tunable control system model, `CL0`. For tuning this system, you have created vectors `SoftReqs` and `HardReqs` of `TuningGoal` requirements objects. These vectors represent your soft and hard constraints, respectively. In that case, the following command uses parallel computing to tune the control system of `CL0`.

```
[CL,fSoft,gHard] = systune(CL0,SoftReqs,HardReqs,options);
```

See Also

| `systune`

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

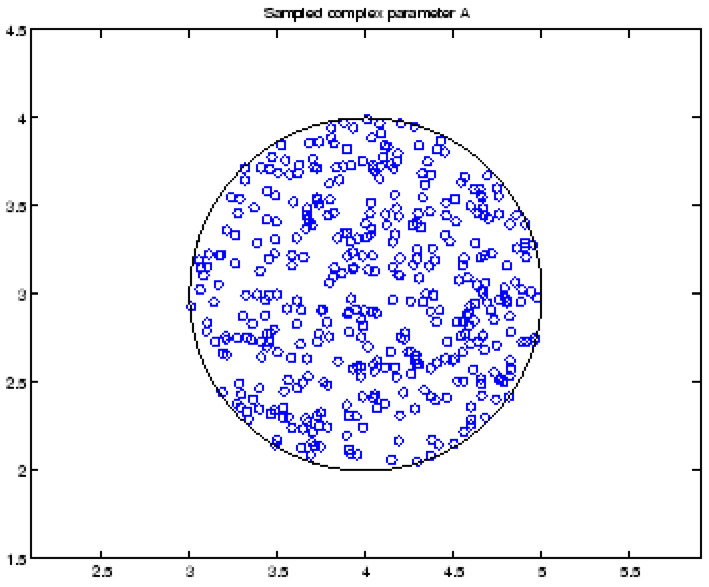
Create an uncertain complex parameter with internal name `A`. The uncertain parameter's possible values are a complex disc of radius `1`, centered at `4+3j`. The value of `A.percentage` is `20` (radius is 1/5 of the magnitude of the nominal value).

```
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

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]));
```

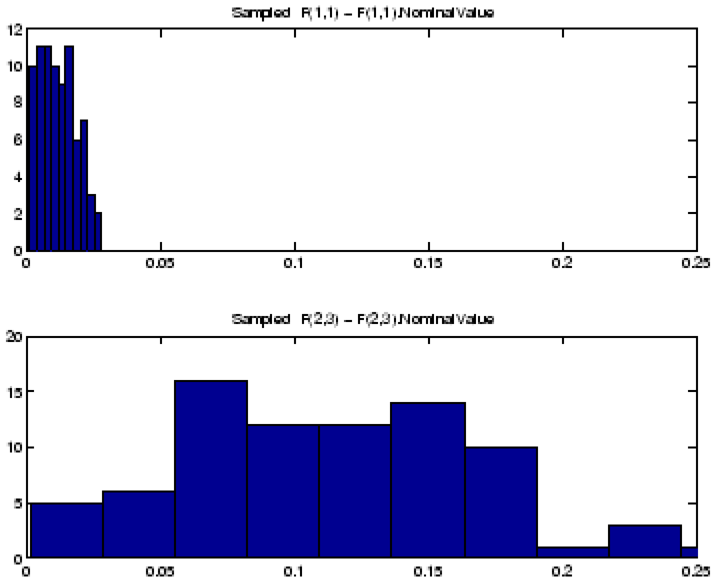
Sample the difference between the uncertain matrix and its nominal value at 80 points, yielding a 2-by-3-by-80 matrix `typicaldev`.

```
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 2-441, 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 $W2$.

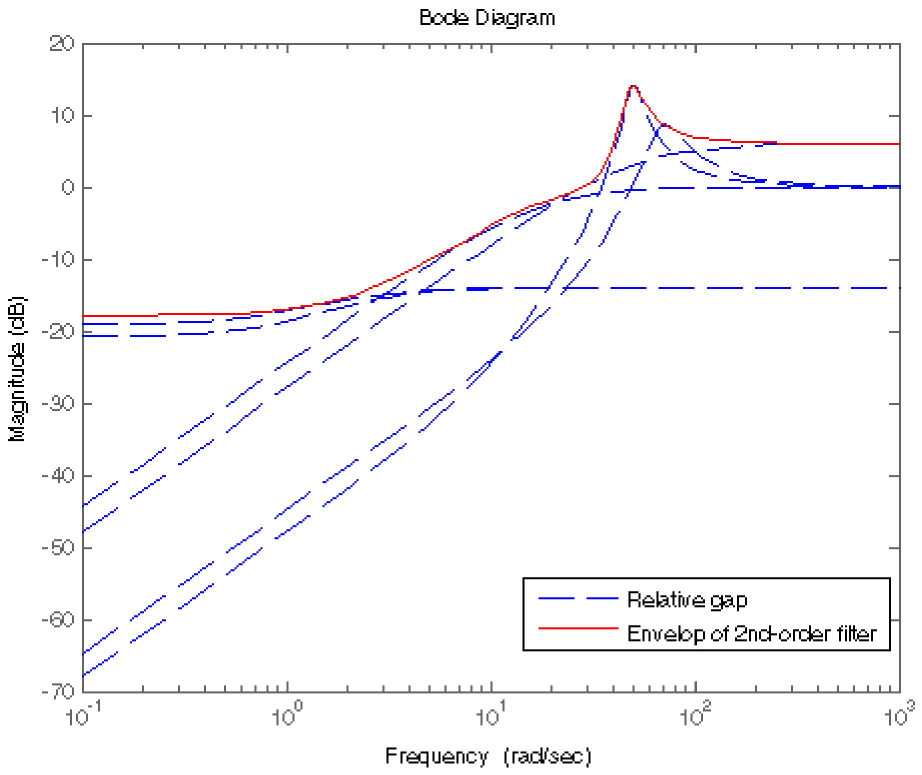
`[usys,info] = (Parray,...)` returns a structure `info` that contains optimization information. `info.W1opt` and `Info.W2opt` contain the values of $W1$ and $W2$ computed on a frequency grid and `info.W1` and `info.W2` contain the fitted shaping filters W1 and W2.

`[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 *W*1 and *W*2. This syntax speeds up command execution by reusing previously computed information. Use this syntax when you are changing filter orders in an iterative workflow.

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 $usys = Pnom \times (I + W_1 \times \Delta \times W_2)$.
- Output Multiplicative of the form $usys = (I + W1 \times \Delta \times W2) \times Pnom$.
- Additive of the form $usys = Pnom + W1 \times \Delta \times W_2$.

Use additive uncertainty to model the absolute gaps between `Pnom` and `Parray`, and multiplicative uncertainty to model relative gaps.

Note For SISO models, input and output multiplicative uncertainty are equivalent. For MIMO systems with more outputs than inputs, the input multiplicative structure may be too restrictive and not adequately cover the range of models.

The model structure $usys = Pnom \times (I + W \times \Delta)$ that you obtain using `usys = ucover(Parray,Pnom,ord)`, corresponds to $W_1 = W \times I$ and $W_1 = 1$.

Examples

1 Fit an uncertain model to multiple LTI responses:

Create the nominal plant.

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

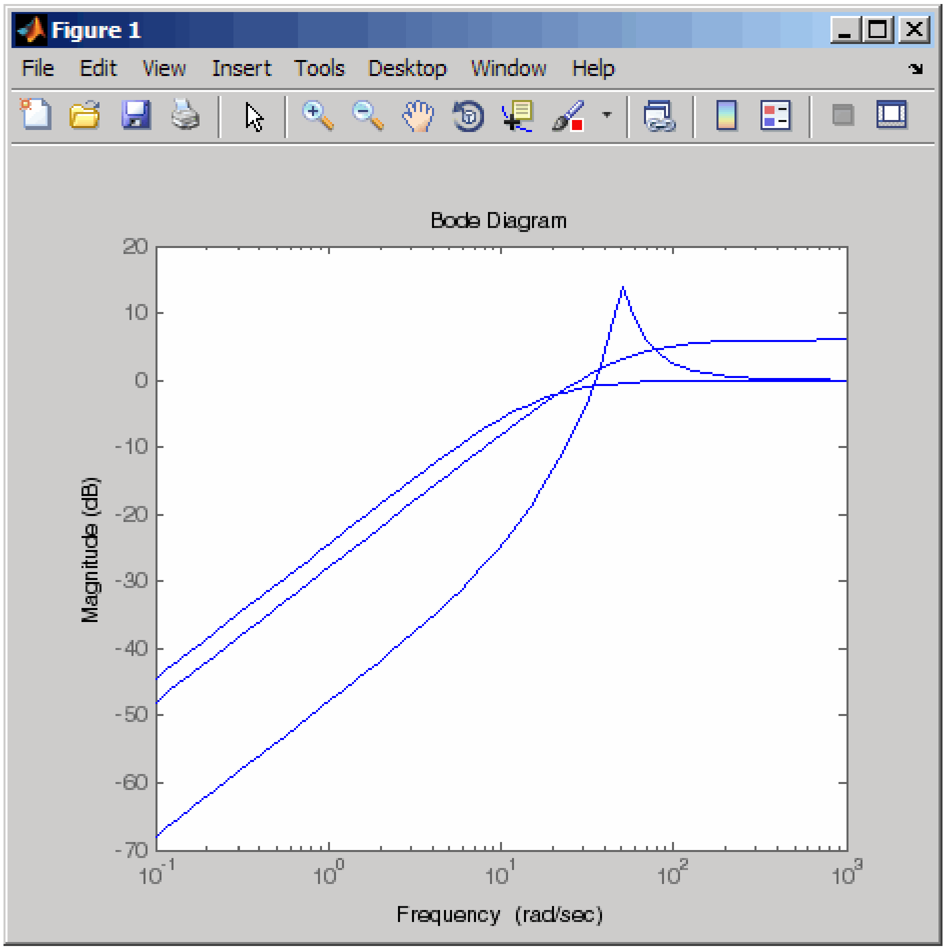
2 Create an LTI array whose responses the uncertain model should fit.

```
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.



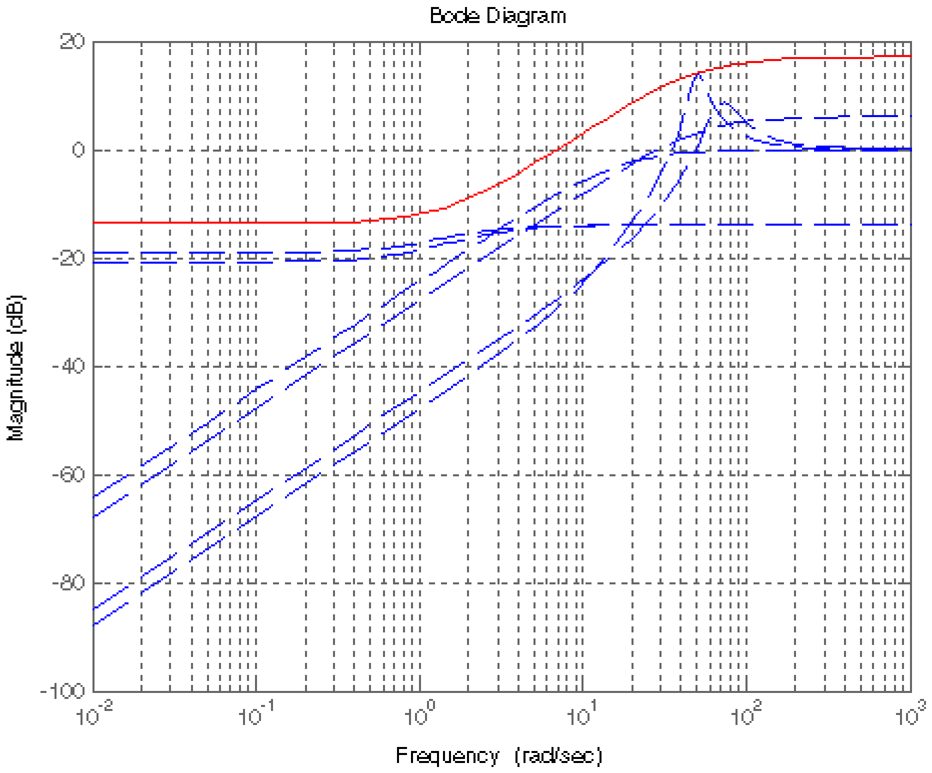
4 Use a 1st-order shaping filter to fit the maximum relative errors.

```
[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 W_1 is too conservative and exceeds the maximum relative error at most frequencies.



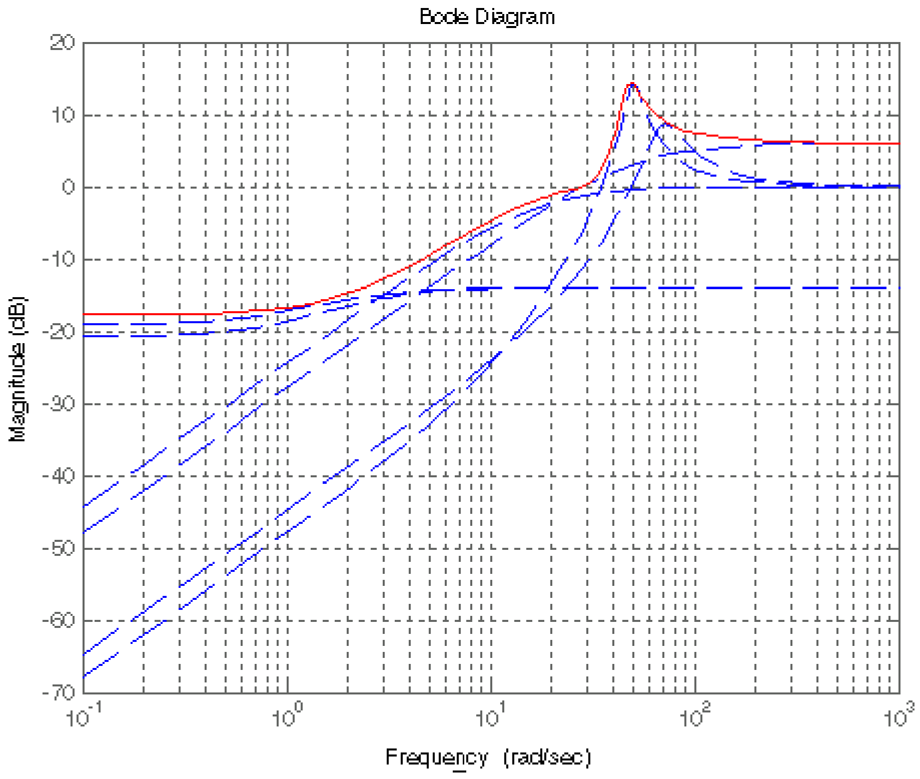
6 To obtain a tighter fit, rerun the function using a 4th-order filter.

```
[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 W_1 closely matches the minimum uncertainty amount.



Algorithms

The `ucover` command designs the minimum-phase shaping filters W_1 and W_2 in two steps:

1. Computes the optimal values of W_1 and W_2 on a frequency grid.
2. Fits W_1 and W_2 values with the dynamic filters of the specified orders using the `fitmagfrd` command.

Tutorials

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

See Also

`ss` | `tf` | `zpk` | `frd` | `usample`

udyn

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`

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.

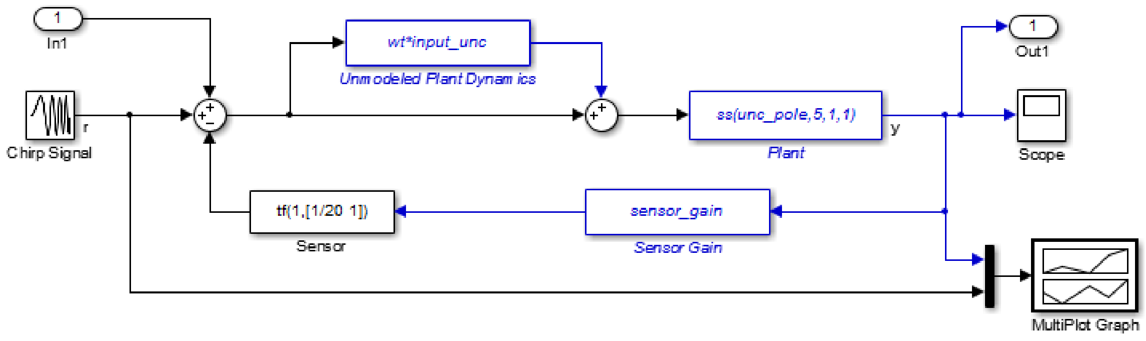
Examples

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'
```

Tutorials

"Vary Uncertainty Values Using Individual Uncertain State Space Blocks"

"Vary Uncertainty Values Across Multiple Uncertain State Space Blocks"

Robustness Analysis in Simulink

How To

"Simulate Uncertainty Effects"

See Also

`usample` | Uncertain State Space

Purpose

Uncertain frequency response data 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)
ufrd_sys = ufrd(M,freqs)
ufrd_sys = ufrd(M,freqs,frequnits)
ufrd_sys = ufrd(M,freqs,frequnits,timeunits)
usysfrd = ufrd(usys,frequency,'Units',units)
usysfrd = ufrd(usys,frequency,'Units',units,'P1',V1,'P2',V2,
   ...)
usys = ufrd(response,frequency)
usysfrd = ufrd(sysfrd)
```

Description

Uncertain frequency response data models (`ufrd`) arise when combining numeric `frd` models with uncertain models such as `ureal`, `ultidyn`, or `uss`. A `ufrd` model keeps track of how the uncertain elements affect the frequency response. Use `ufrd` for robust stability and worst-case performance analysis.

There are three ways to construct a `ufrd` model:

1 Combine numeric `frd` models with uncertain models using model arithmetic. For example:

```
sys = frd(rand(100,1),logspace(-2,2,100));
k = ureal('k',1);
D = ultidyn('Delta',[1 1]);
ufrd_sys = k*sys*(1+0.1*D)
```

`ufrd_sys` is a `ufrd` model with uncertain elements `k` and `D`.

2 `ufrd_sys = ufrd(M,freqs)` converts the dynamic system model or static model `M` to `ufrd`. If `M` contains Control Design Blocks that do not represent uncertainty, these blocks are replaced by their current value. (To preserve both tunable and uncertain Control Design Blocks, use `genfrd` instead.)

Use `ufrd_sys = ufrd(M,freqs,frequnits)` to specify the frequency units of the frequencies in `freqs` with the string `frequnits`. Use `ufrd_sys = ufrd(M,freqs,frequnits,timeunits)` to specify the time unit of `ufrd_sys` when `M` is a static model.

3 Use `frd` to construct a `ufrd` model from an uncertain matrix (`umat`) representing uncertain frequency response data. For example:

```
a = ureal('delta',1,'percent',50);
freq = logspace(-2,2,100);
RespData = rand(1,1,100) * a;
usys = frd(RespData,freq,0.1)
```

Examples

Compute the uncertain frequency response of an uncertain system (`uss` model) with both parametric uncertainty (`ureal`) and unmodeled dynamics uncertainty (`ultidyn`).

```
p1 = ureal('p1',5,'Range',[2 6]);
p2 = ureal('p2',3,'Plusminus',0.4);
p3 = ultidyn('p3',[1 1]);
Wt = makeweight(.15,30,10);
A = [-p1 0;p2 -p1];
B = [0;p2];
C = [1 1];
usys = uss(A,B,C,0)*(1+Wt*p3);

usysfrd = ufrd(usys,logspace(-2,2,60));
```

Plot 20 random samples and the nominal value of the uncertain frequency response.

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

See Also `frd` | `ss` | `uss` | `genfrd`

How To

- “Control Design Blocks”

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 `linearizeOptions` to create `options`.

`ulin=ulinearize('sys',op)` linearizes the model `sys` at the operating point specified in the operating point object `op`. The software uses root-level inport and outport blocks in `sys` as I/O points for linearization.

`ulin_block=ulinearize('sys',op,'blockname',...)` takes the name of a block `blockname` in the model `sys` and returns a linear time-invariant uncertain system `ulin_block.` You can also specify a fourth argument `options` to provide options for the linearization.

`[ulin,op] = ulinearize('sys',snapshottimes,...)` creates operating points for linearization by simulating the model and taking snapshots of the system's states and inputs at times specified in the vector `snapshottimes`. `ulin` is a set of linear time-invariant uncertain systems and `op` is the set of operating point objects used in linearization. You can also specify I/O object for linearization, or a block name. If you do not specify an I/O object or block name, the linearization uses root-level inport and outport blocks in the model. You can also supply an additional argument, `options`, to provide options for linearization.

`ulin = ulinearize('sys','StateOrder',stateorder)` creates a linear-time-invariant uncertain system `ulin`, whose states are in a specified order. Specify the state order in the cell array `stateorder` by entering the names of the blocks containing states in the model. For all blocks, you can enter block names as the full block path. For continuous blocks, you can alternatively enter block names as the user-defined unique state name.

Examples

Compute uncertain linearization of a Simulink model containing Uncertain State Space blocks:

```
% Define uncertain variables and uncertain system variables
% to use in Uncertain State Space blocks.
unc_pole = ureal('unc_pole',-5,'Range',[-10 -4]);
plant = ss(unc_pole,5,1,0);
wt = makeweight(0.25,130,2.5);
input_unc = ultidyn('input_unc',[1 1]);
sensor_pole = ureal('sensor_pole',-20,'Range',[-30 -10]);
sensor = tf(1,[1/(-sensor_pole) 1]);

% Open Simulink model. The model contains three Uncertain State
% Space blocks named Unmodeled Plant Dynamics, Uncertain Plant and
% Uncertain Sensor, and linearization I/O points.
open_system('rct_ulinearize_uss')

% Obtain linearization I/O points.
mdl = 'rct_ulinearize_uss';
io = getlinio(mdl);

% Compute the uncertain linearization of the model.
```

```
ulin = ulinearize(mdl,io)
% MATLAB returns an uss object with 5 states.
```

Tutorials

"Linearize Block to Uncertain Model"

Linearization of Simulink Models with Uncertainty

How To

"Obtain Uncertain State-Space Model from Simulink Model"

See Also

`ureal` | `udyn` | `ultidyn` | `uss`

ultidyn

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

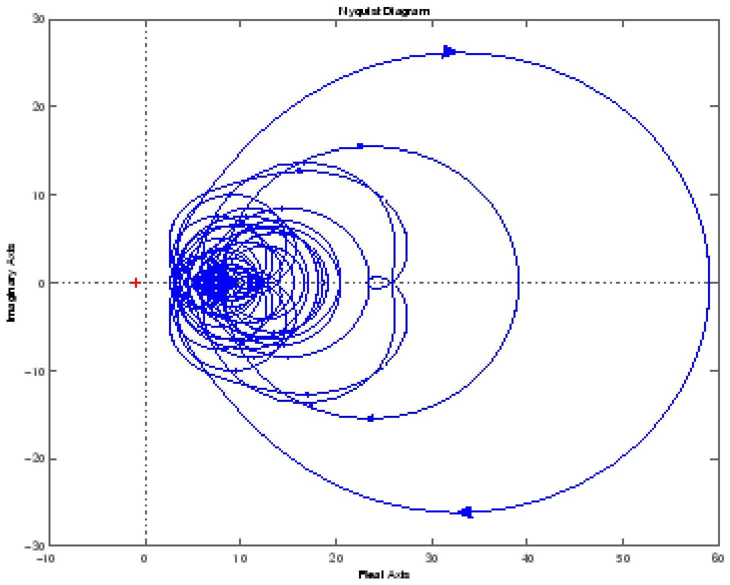
Create an `ultidyn` object with internal name `'H'`, dimensions `2-by-3`, norm bounded by `7`.

```
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)
M = umat(A)
```

Description

Uncertain matrices are rational expressions involving uncertain elements of type `ureal`, `ucomplex`, or `ucomplexm`. Use uncertain matrices for worst-case gain analysis and for building uncertain state-space (`uss`) models.

Create uncertain matrices by creating uncertain elements and combining them using arithmetic and matrix operations. For example:

```
 p = ureal('p',1);
 M = [0 p; 1 p^2]
```

creates a 2-by-2 uncertain matrix (a `umat` object) with the uncertain parameter `p`.

The syntax `M = umat(A)` converts the double array `A` to a `umat` object with no uncertainty.

Most standard matrix manipulations are valid on uncertain matrices, including addition, multiplication, inverse, horizontal and vertical concatenation. Specific rows/columns of an uncertain matrix can be referenced and assigned also.

If `M` is a `umat`, then `M.NominalValue` is the result obtained by replacing each uncertain element in `M` with its own nominal value.

If `M` is a `umat`, then `M.Uncertainty` is an object describing all the uncertain elements in `M`. All element can be referenced and their properties modified with this `Uncertainty` gateway. For instance, if `B` is an uncertain real parameter in `M`, then `M.Uncertainty.B` accesses the uncertain element `B` in `M`.

Examples

Create 3 uncertain elements and then a `3-by-2 umat`.

```
a = ureal('a',5,'Range',[2 6]);
b = ucomplex('b',1+j,'Radius',0.5);
```

```
c = ureal('c',3,'Plusminus',0.4);
M = [a b;b*a 7;c-a b^2]
```

`M` is an uncertain matrix (`umat` object) with the uncertain parameters `a`, `b`, and `c`.

View the properties of `M` with `get`

```
get(M)
```

The nominal value of `M` is the result when all atoms are replaced by their nominal values.

```
M.NominalValue
ans =
   5.0000             1.0000 + 1.0000i
   5.0000 + 5.0000i   7.0000
  -2.0000                  0 + 2.0000i
```

Change the nominal value of `a` within `M` to 4. The nominal value of `M` reflects this change.

```
M.Uncertainty.a.NominalValue = 4;
M.NominalValue
ans =
   4.0000             1.0000 + 1.0000i
   4.0000 + 4.0000i   7.0000
  -1.0000                  0 + 2.0000i
```

Get a random sample of `M`, obtained by taking random samples of the uncertain atoms within `M`.

```
usample(M)
ans =
   2.0072             0.8647 + 1.3854i
   1.7358 + 2.7808i   7.0000
   1.3829              -1.1715 + 2.3960i
```

Select the 1st and 3rd rows, and the 2nd column of `M`. The result is a 2-by-1 `umat`, whose dependence is only on `b`.

```
M([1 3],2)
```

See Also

`ureal` | `ultidyn` | `ucomplex` | `ucomplexm` | `usample`

uplot

Purpose

Plot multiple frequency response objects and `double`s on same graph

Syntax

```
uplot(G1)
uplot(G1,G2)
uplot(G1,Xdata,Ydata)
uplot(G1,Xdata,Ydata,...)
uplot(G1,linetype)
uplot(G1,linetype,G2,...)
uplot(G1,linetype,Xdata,Ydata,linetype)
uplot(type,G1,linetype,Xdata,Ydata,linetype)
H = uplot(G1)
H = uplot(G1,G2)
H = uplot(G1,Xdata,Ydata)
H = uplot(G1,Xdata,Ydata,...)
H = uplot(G1,linetype)
H = uplot(G1,linetype,G2,...)
H = uplot(G1,linetype,Xdata,Ydata,linetype)
```

Description

`uplot` plots `double` and `frd` objects. The syntax is the same as the MATLAB `plot` command except that all data is contained in `frd` objects, and the axes are specified by `type`.

The (optional) `type` argument must be one of

Type	Description
`'iv,d'`	Data versus independent variable (default)
`'iv,m'`	Magnitude versus independent variable
`'iv,lm'`	`log`(magnitude) versus independent variable
`'iv,p'`	Phase versus independent variable
`'liv,m'`	Magnitude versus `log`(independent variable)
`'liv,d'`	Data versus `log`(independent variable)
`'liv,m'`	Magnitude versus `log`(independent variable)
`'liv,lm'`	`log`(magnitude) versus `log`(independent variable)

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.

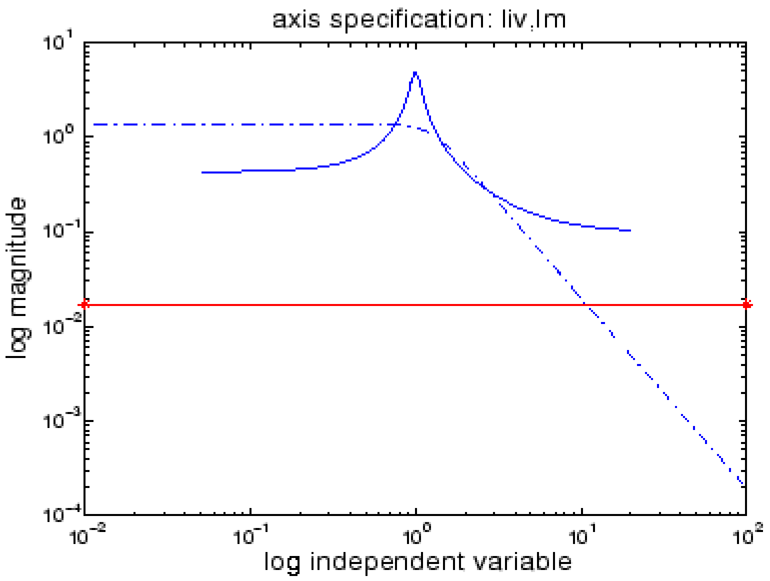
```
a1 = [-1,1;-1,-0.5];
b1 = [0;2]; c1 = [1,0]; d1 = 0;
sys1 = ss(a1,b1,c1,d1);
a2 = [-.1,1;-1,-0.05];
b2 = [1;1]; c2 = [-0.5,0]; d2 = 0.1;
sys2 = ss(a2,b2,c2,d2);
omega = logspace(-2,2,100);
sys1g = frd(sys1,omega);
omega2 = [ [0.05:0.1:1.5] [1.6:.5:20] [0.9:0.01:1.1] ];
omega2 = sort(omega2);
sys2g = 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`

uss/usample

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)
```

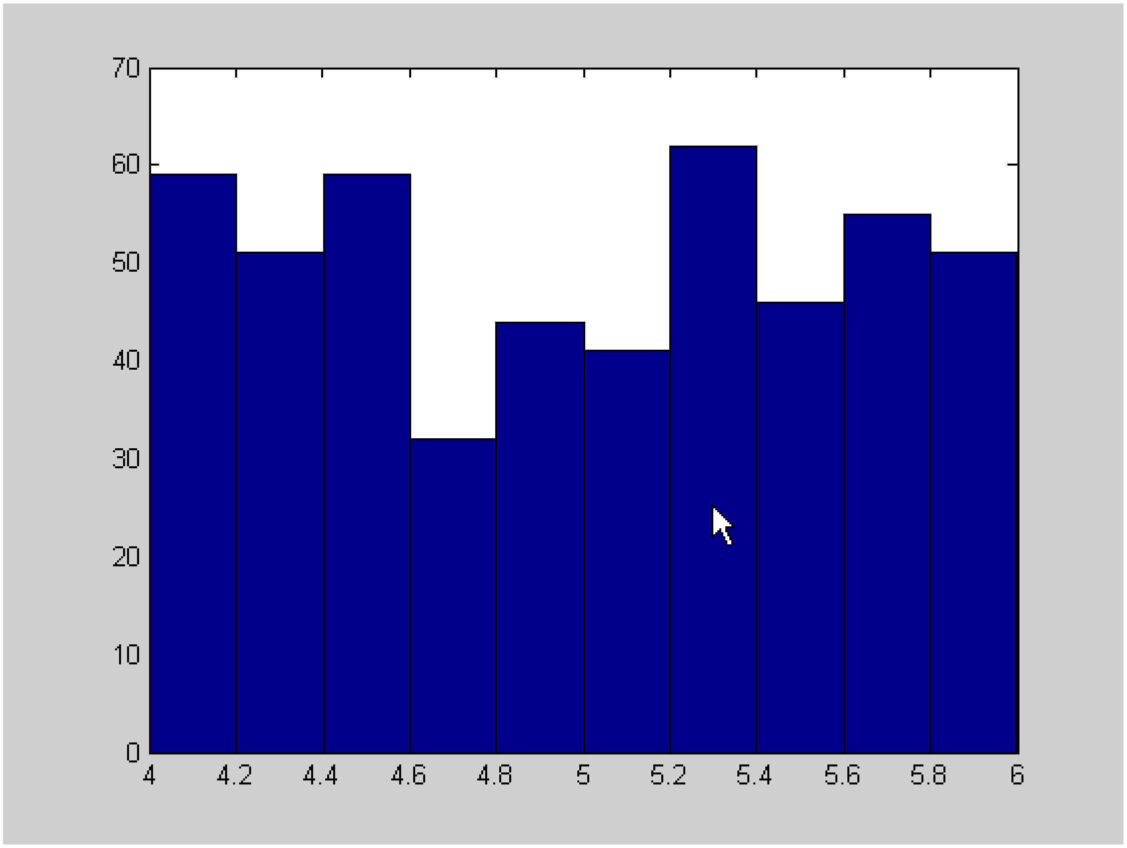
affects how ultidyn elements within A are sampled, restricting the poles of the samples. If A is a continuous-time uss or ufrd, then the poles of sampled GainBounded ultidyn elements in SampleValues will each have magnitude <= BW. If A is a discrete-time, then sampled GainBounded ultidyn elements are 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 Elements") 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).

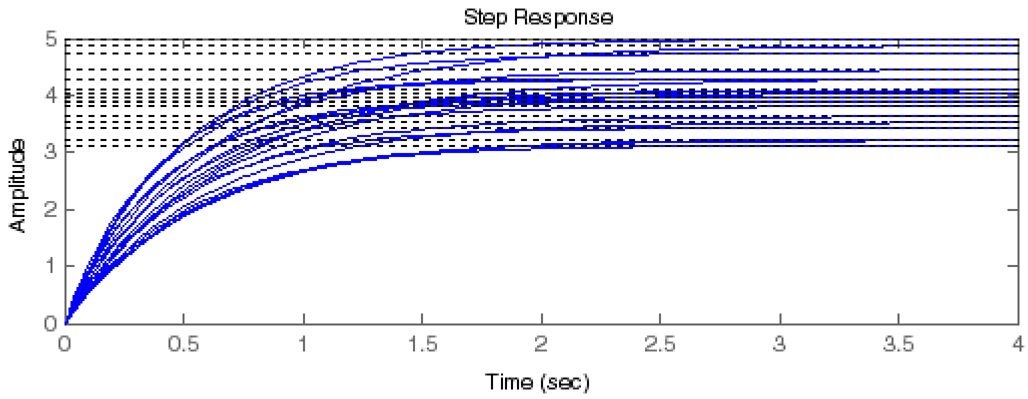
```
[Psample1D,Values1D] = usample(P,20);
size(Psample1D)
20x1 array of state-space models
Each model has 1 output, 1 input, and 1 state.
```

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.
```

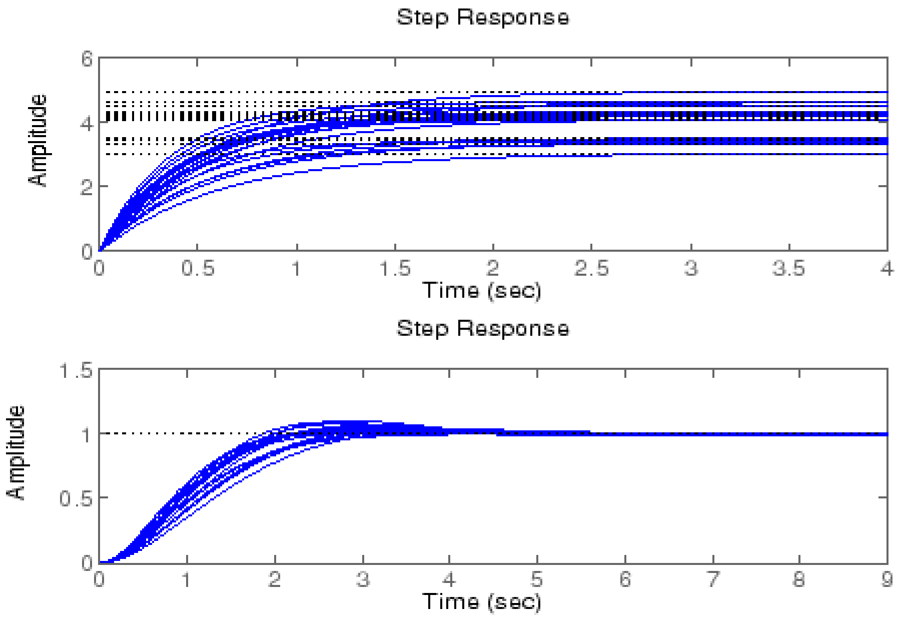
You can plot the 1-D sampled plant step responses

```
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

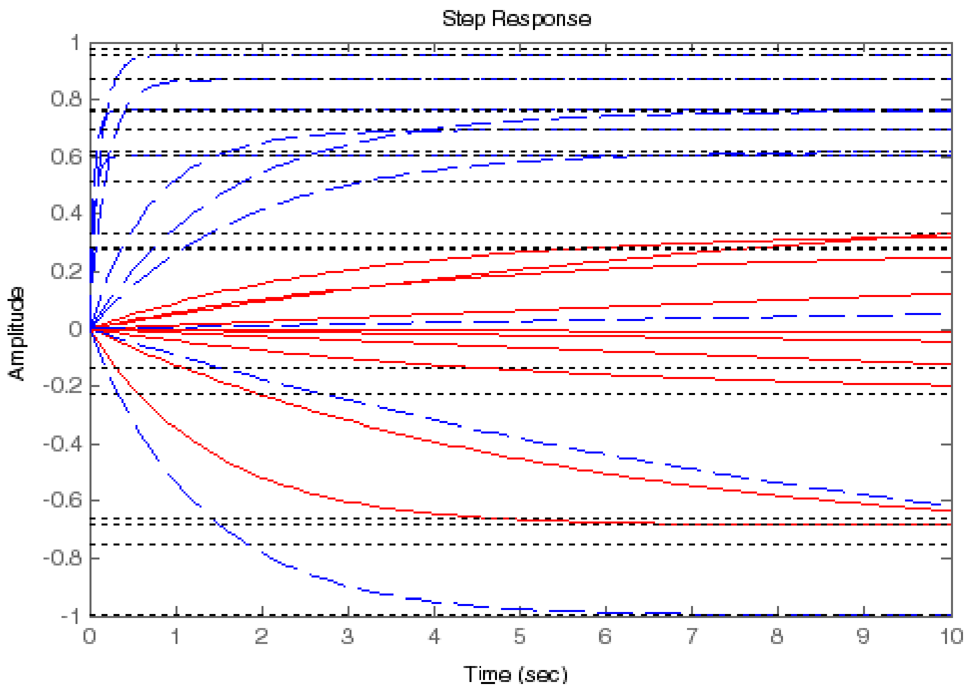
```
A = ultidyn('A',[1 1]);
B = ultidyn('B',[1 1]);
```

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)
```

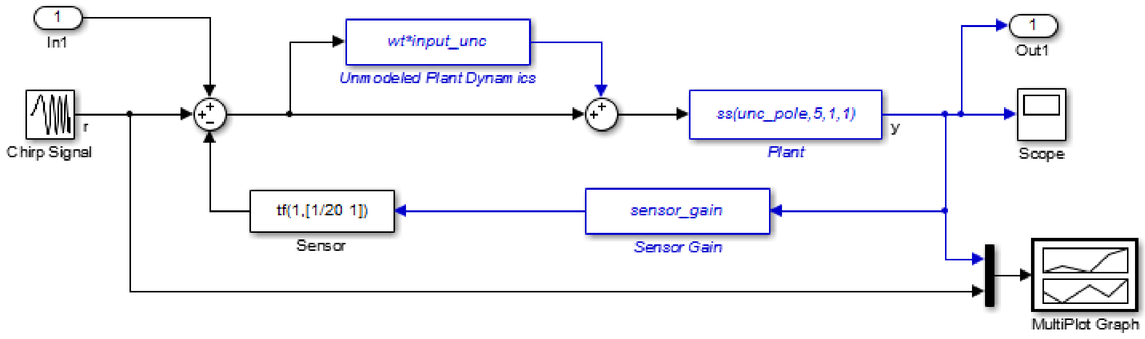
Example 2

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`.



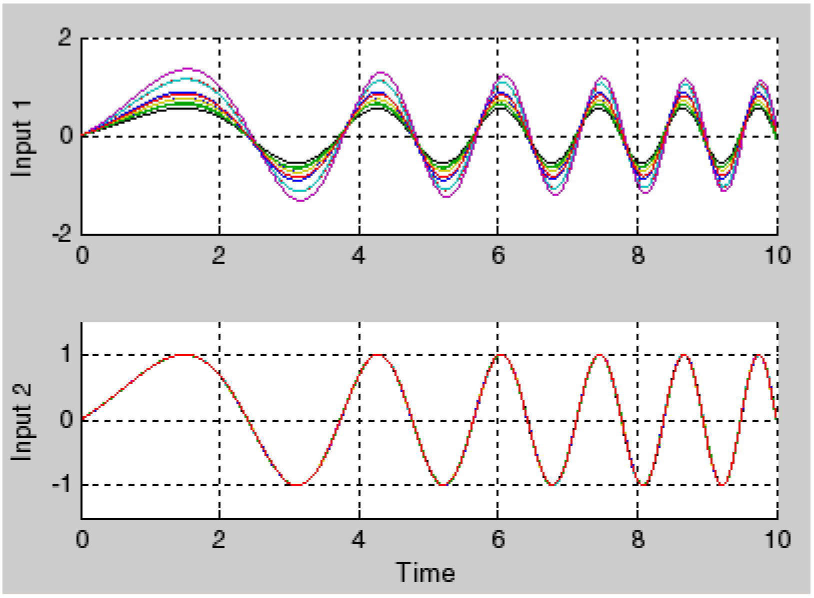
2 Use `ufind` to find all Uncertain State Space blocks and uncertain variables in the model.

```
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.



Tutorials

"Vary Uncertainty Values Using Individual Uncertain State Space Blocks"

"Vary Uncertainty Values Across Multiple Uncertain State Space Blocks"

Robustness Analysis in Simulink

How To

"Simulate Uncertainty Effects"

See Also

`ufind` | `usubs` | `ureal` | `ucomplex` | `ultidyn` | `umat` | `ufrd` | `uss`

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 Robustness Analysis in Simulink for a more detailed example of how to use `usimfill`.

Open the model file associated with the example.

```
open_system('usim_model');
unc_pole = ureal('unc_pole',-5,'Range',[-10 -4]);
plant = ss(unc_pole,5,1,1);
input_unc = ultidyn('input_unc',[1 1]);
```

```
wt = makeweight(0.25,130,2.5);
sensor_gain = ureal('sensor_gain',1,'Range',[0.1 2]);
```

This has three USS System blocks. They are plant with a `ureal` atom named `unc_pole`; `input_unc` which is a `ultidyn` object, and `sensor_gain` which is a `ureal` atom.

Run `usimfill` on the model, filling in the field with the 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	Description
`sname`	Simulink diagram name
`silent`	Display inconsistencies between uncertain atoms, when not empty. Default is empty.

Output Arguments	Description
`cflag`	Compatibility flag set to 1 if all uncertainties are consistent, set to 0 if an uncertainty definition(s) is consistent and set to –1 if common uncertainties in different blocks have different `Uncertainty value`.
`allupaths`	Path names of USS System blocks in the model (cell).

Output Arguments	Description
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

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 Robustness Analysis in Simulink for a more detailed example of how to use `usimsamp`.

Open the model file associated with the example.

```
open_system('usim_model');
```

This has 3 USS System blocks. They are `plant` with a `ureal` atom named `unc_pole`; `input_unc` which is a `ultidyn` object, and `sensor_gain` which is a `ureal` atom.

Run `usimsamp` on the model, yielding a structure as described above.

```
unc_pole = ureal('unc_pole',-5,'Range',[-10 -4]);
plant = ss(unc_pole,5,1,1);
input_unc = ultidyn('input_unc',[1 1]);
```

```
wt = makeweight(0.25,130,2.5);
sensor_gain = ureal('sensor_gain',1,'Range',[0.1 2]);
data = usimsamp('usim_model')
data =
      input_unc: [1x1 ss]
    sensor_gain: 0.9935
       unc_pole: -4.1308
```

See Also

`usample` | `usimfill` | `usiminfo` | `usubs`

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`

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)
B = usubs(M,atomname1,value1,atomname2,value2,...)
B = usubs(M,ElementName1,value1,ElementName2,value2,...)
B = usubs(M,S)
B = usubs(M,...,'-once')
B = usubs(M,...,'-batch')
```

Description

Use `usubs` to substitute a specific value for an uncertain element of an uncertain model object. The value can itself be uncertain. It needs to be the correct size, but otherwise can be of any class, and can be an array. Hence, the result can be of any class. In this manner, uncertain elements act as symbolic placeholders, for which specific values (which can also contain other placeholders too) can be substituted.

`B = usubs(M,ElementName1,value1,ElementName2,value2,...)` sets the elements in `M`, identified by `ElementName1`, `ElementName2`, etc., to the values in `value1`, `value2`, etc. respectively.

Any `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.

`B = usubs(M,S)` instantiates the uncertain elements of `M` to the values specified in the structure `S`. The field names of `S` are the names of the uncertain elements to replace. The values are the corresponding replacement values. To provide several replacement values, make `S` a struct array, where each struct contains one set of replacement values. A structure such as `S` typically comes from robustness analysis commands such as `robuststab`, `usample`, or `wcgain`.

`B = usubs(M,...,'-once')` performs vectorized substitution in the uncertain model array `M`. Each uncertain element is replaced by a single value, but this value may change across the model array. To specify different substitute values for each model in the array `M`, use:

- A cell array for each `valueN` that causes the uncertain element `ElementNameN` in `M(:,:,k)` to be replaced by `valueN(k)`. For example, if `M` is a 2-by-3 array, then a 2-by-3 cell array `value1` replaces `ElementName1` of the model `M(:,:,k)` with the corresponding `value1(k)`.
- A struct array `S` that specifies one set of substitute values `S(k)` for each model `M(:,:,k)`.

Numeric array formats are also accepted for `value1,value2,...`. For example, `value1` can be a 2-by-3 array of LTI models, a numeric array of size `[size(name1) 2 3]`, or a 2-by-3 matrix when the uncertain element `name1` is scalar-valued. The array sizes of `M`, `S`, `value1,value2,...` must agree along non-singleton dimensions. Scalar expansion takes place along singleton dimensions.

Vectorized substitution (`'-once'`) is the default for model arrays when no substitution method is specified.

`B = usubs(M,...,'-batch')` performs batch substitution in the uncertain model array `M`. Each uncertain element is replaced by an array of values, and the same values are used for all models in `M`. In batch substitution, `B` is a model array of size `[size(M) VS]`, where `VS` is the size of the array of substitute values.

Examples

Evaluate Uncertain Matrix for Multiple Values of Uncertain Parameters

Evaluate an uncertain matrix at several different values of the uncertain parameters of the matrix.

Create an uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b];
```

Evaluate the matrix at four different combinations of values for the uncertain parameters `a` and `b`.

```
B = usubs(M,'a',[1;2;3;4],'b',[10;11;12;13]);
```

This command evaluates `M` for the four different (`a`,`b`) combinations (1,10), (2,11), and so on. Therefore, `B` is a 1-by-2-by-4 array of numeric values containing the four evaluated values of `M`.

Evaluate Uncertain Matrix over Grid of Uncertain Parameters

Evaluate an uncertain matrix over a 3-by-4 grid of values of the uncertain parameters of the matrix.

Create a 2-by-2 uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b;0 a*b];
```

Build arrays of values for the uncertain parameters.

```
aval = [1;2;3;4];
bval = [10;20;30];
[as,bs] = ndgrid(aval,bval);
```

This command builds two 4-by-3 grids of values.

Evaluate M over the parameter grids `A` and `B`.

```
B = usubs(M,'a',as,'b',bs);
```

This command evaluates `M` for each four different combination of values `(A(k),B(k))` `B` is a 2-by-2-by-4-by-3 array of numeric values, which is a 4-by-3 array of values of `M`, i.e., `M` evaluated over the parameter grids.

Instantiate Uncertain Parameter by Batch Substitution of Parameter for Array of Values

Evaluate an array of uncertain models, substituting an array of values for an uncertain parameter.

Create a 1-by-2 uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b];
```

Replace a by each of the values 1, 2, 3, and 4.

```
Ma = usubs(M,'a',[1;2;3;4]);
```

This command returns a 4-by-1 array of 1-by-2 uncertain matrices that contain one uncertain parameter b.

For each model in the array Ma, evaluate b at 10, 20, and 30.

```
B = usubs(Ma,'b',[10;20;30],'-batch');
```

The '-batch' flag causes usubs to evaluate each model in the array at all three values of b. Thus B is a 4-by-3 array of M values.

The '-batch' syntax here yields the same result as the parameter grid approach used in the previous example:

```
aval = [1;2;3;4];
bval = [10;20;30];
[as,bs] = ndgrid(aval,bval);
B = usubs(M,'a',as,'b',bs);
```

Instantiate Uncertain Parameter Using Different Value for Each Entry in Array

Evaluate an array of uncertain models, substituting a different value for the uncertain parameter in each entry in the array.

Create a 1-by-2 uncertain matrix with two uncertain parameters.

```
a = ureal('a',5);
b = ureal('b',-3);
M = [a b];
```

Replace a by each of the values 1, 2, 3, and 4.

```
Ma = usubs(M,'a',[1;2;3;4]);
```

This command returns a 4-by-1 array of 1-by-2 uncertain matrices that contain one uncertain parameter `b`.

For each model in the array `Ma`, evaluate `b`. Use `b = 10` for the first entry in the array, `b = 20` for the second entry, and so on.

```
B = usubs(Ma,'b',{10;20;30;40},'-once');
```

The `'-once'` flag causes `usubs` to evaluate the first model in the array using the first specified value for `b`, the second model for the second specified value, etc.

Replace Uncertain Parameters with Values Returned by usample

Replace the uncertain parameters in an uncertain models by values specified in struct array form, as returned by `usample`.

This is useful, for example, when you have multiple uncertain models that use the same set of parameters, and you want to evaluate all models at the same parameter values.

Create two uncertain matrices that have the same uncertain parameters, `a` and `b`.

```
a = ureal('a',5);
b = ureal('b',-3);
M1 = [a b];
M2 = [a b;0 a*b];
```

Generate some random samples of `M1`.

```
[M1rand,samples] = usample(M1,5);
```

`M1rand` is an array of five values of `M1`, evaluated at randomly generated values of `a` and `b`. These `a` and `b` values are returned in the struct array `samples`.

Examine the struct array `samples`.

```
samples
```

```
samples =

5x1 struct array with fields:
    a
    b
```

The field names of `samples` correspond to the uncertain parameters of `M1`. The values are the parameter values used to generate `M1rand`. Because `M2` has the same parameters, you can use this structure to evaluate `M2` at the same set of values.

```
M2rand = usubs(M2,samples);
```

This command returns a 1-by-5 array of instantiations of `M2`.

See Also

`gridureal` | `usample` | `simplify`

viewSpec

Purpose

View tuning requirements; validate design against requirements

Syntax

```
viewSpec(Req)
viewSpec(Req,T)
viewSpec(Req,T,Info)
```

Description

`viewSpec(Req)` displays a graphical view of a `TuningGoal` tuning requirement or vector of tuning requirements.

`viewSpec(Req,T)` plots the performance of a tuned control system against the tuning requirement.

`viewSpec(Req,T,Info)` uses the `Info` structure returned by `systune` for correct scaling of MIMO open-loop requirements, such as loop shapes and stability margins.

Input Arguments

Req - Tuning requirement to view or validate

`TuningGoal` requirement object | vector of `TuningGoal` objects

Tuning requirement to view or validate, specified as a `TuningGoal` requirement object or vector of `TuningGoal` objects. `TuningGoal` requirement objects include:

- `TuningGoal.Tracking`
- `TuningGoal.Gain`
- `TuningGoal.WeightedGain`
- `TuningGoal.Variance`
- `TuningGoal.WeightedVariance`
- `TuningGoal.LoopShape`
- `TuningGoal.Margins`
- `TuningGoal.Poles`
- `TuningGoal.StableController`

T - Tuned control system

generalized state-space model | `slTunable` interface object

Tuned control system, specified as a generalized state-space (`genss`) model or an `slTunable` interface to a Simulink model.

The control system, `T`, is typically the result of using the tuning requirement to tune control system parameters with `systune`.

Example: `[T,fSoft,gHard,Info] = systune(T0,SoftReq,HardReq)`, where `T0` is a tunable `genss` model

Example: `[T,fSoft,gHard,Info] = systune(ST0,SoftReq,HardReq)`, where `ST0` is a `slTunable` interface object

Info - System information

data structure returned by `systune`

System information, specified as the data structure returned by `systune` when you use that command to tune a control system. Use `Info` when validating tuned MIMO systems, to ensure that `viewSpec` correctly scales open-loop requirements such as loop shapes and stability margins.

Examples

Visualize Tuning Requirement as Function of Frequency

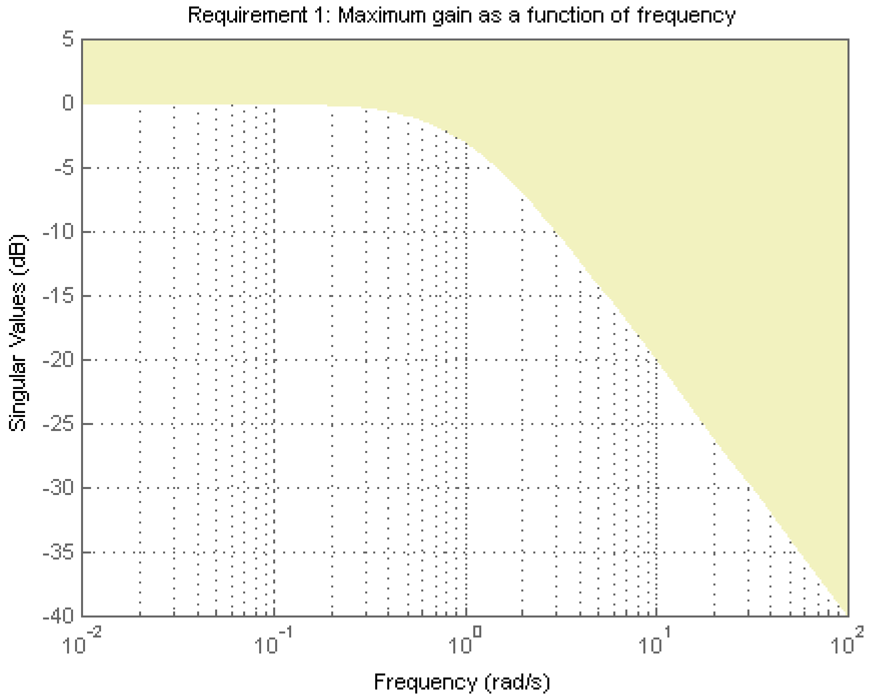
Create a tuning requirement that constrains the response from a signal, `'d'`, to another signal, `'y'`, to roll off at 20 dB/decade at frequencies greater than 1. The requirement also imposes disturbance rejection (maximum gain of 1) in the frequency range [0,1].

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

When you use a frequency response data (`frd`) model to sketch the bounds of a gain constraint or loop shape, the tuning requirement interpolates the constraint. This interpolation coverts the constraint to a smooth function of frequency.

Examine the interpolated gain constraint using `viewSpec`.

```
viewSpec(Req)
```



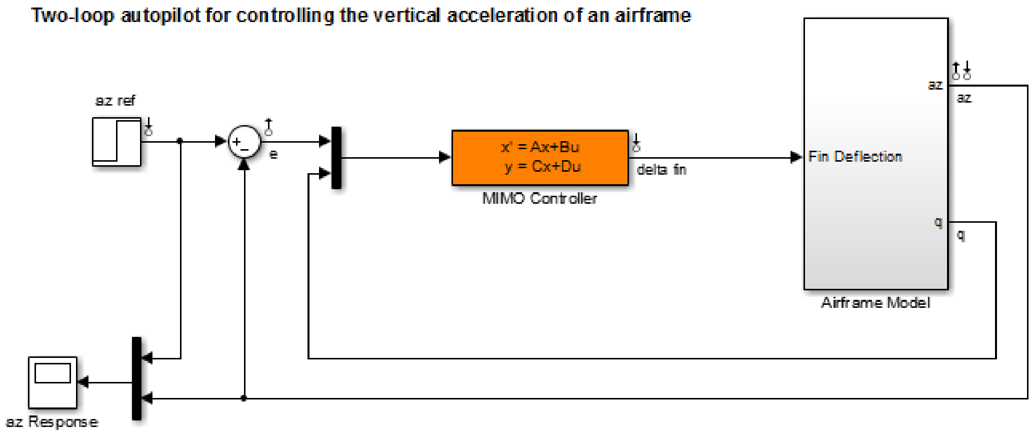
The yellow region represents gain values that violate the tuning requirement.

Validate Tuning Result Against Requirements

Validate a control system tuned with `systune` to determine whether small violations of tuning requirements are acceptable.

When you tune a control system using tuning commands such as `systune`, use `viewSpec` to compare the tuned result against the tuning requirements. Doing so can help you determine whether the tuned system comes sufficiently close to meeting your soft requirements.

Create tracking, roll-off, stability margin, and disturbance rejection requirements for tuning the following control system.



```
Req1 = TuningGoal.Tracking('az ref','az',1);
Req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0]));
Req3 = TuningGoal.Margins('delta fin',7,45);
MaxGain = frd([2 200 200],[0.02 2 200]);
Req4 = TuningGoal.Gain('delta fin','az',MaxGain);
```

Tune the model using these tuning requirements.

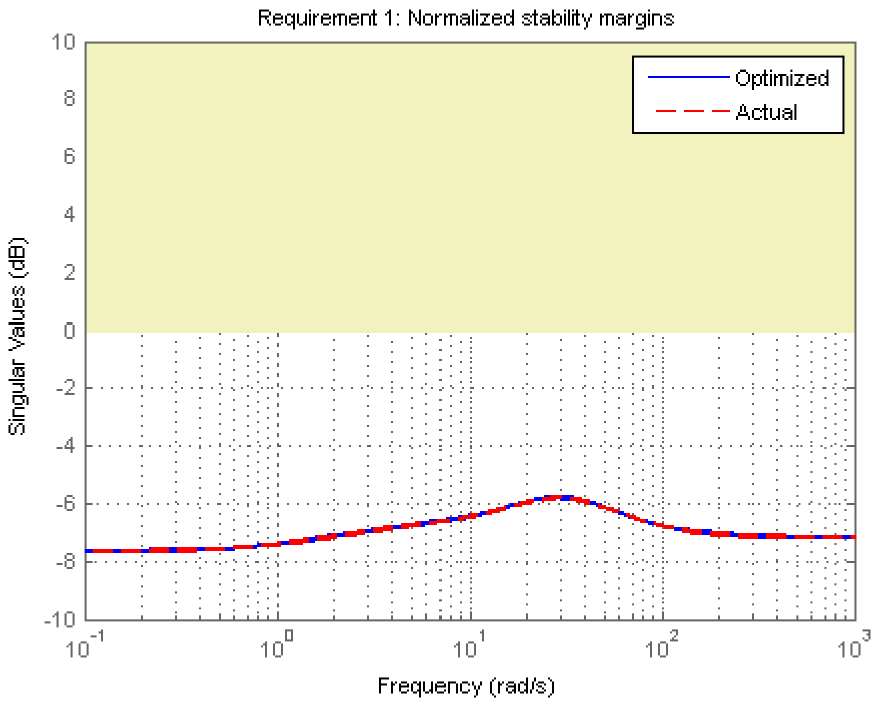
```
open_system('rct_airframe2')

ST0 = slTunable('rct_airframe2','MIMO Controller');
addControl(ST0,'delta fin');

rng('default');
[ST1,fSoft,~,Info] = systune(ST0,[Req1,Req2,Req3,Req4]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 55
```

ST1 is a tuned version of the slTunable interface to the control system. ST1 contains the tuned values of the tunable parameters of the MIMO controller in the model.

Verify that the tuned system satisfies the margin requirement.

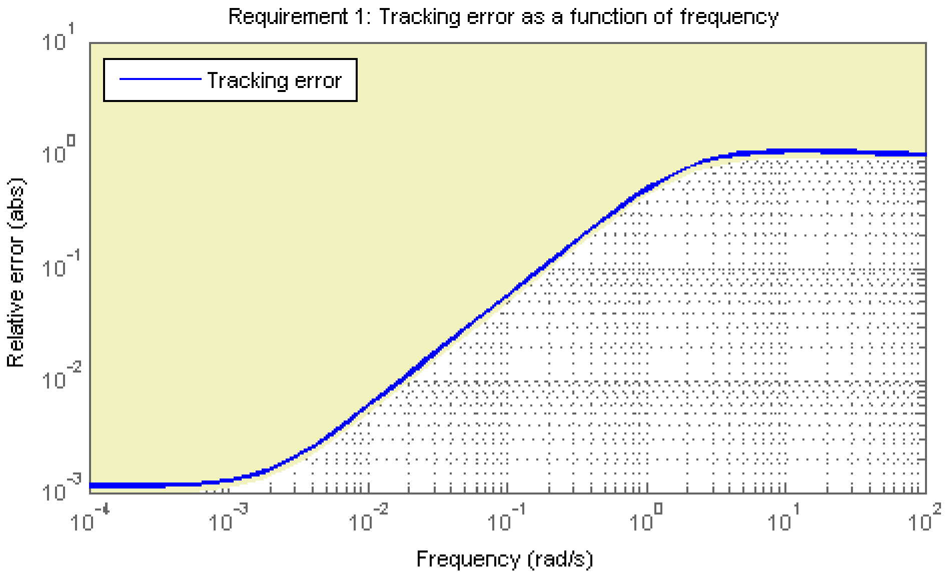
```
viewSpec(Req3,ST1,Info)
```

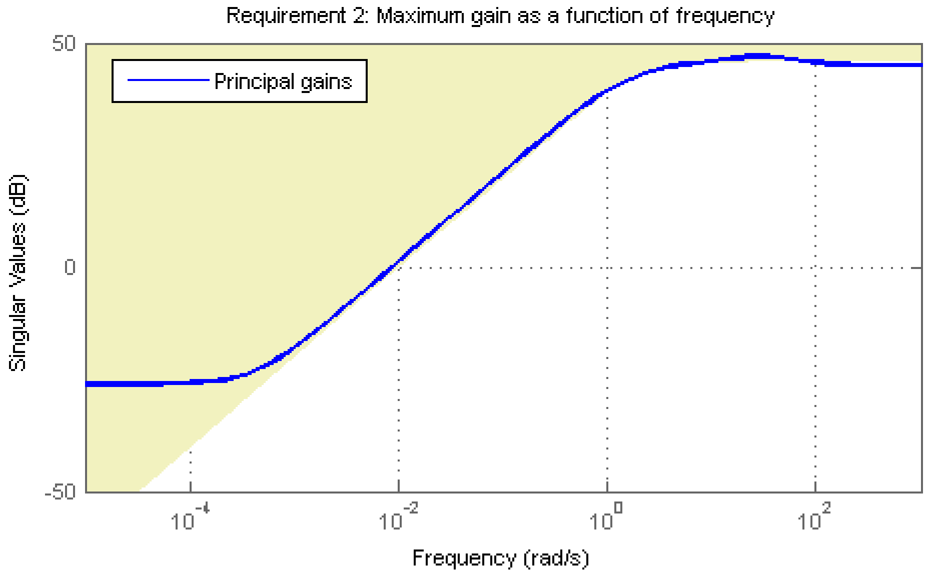


The yellow region denotes margins that do not satisfy the requirement. The red plot represents the actual stability margin of the tuned system, ST1. The blue plot represents the margin used in the optimization calculation, which is an upper bound on the actual margin. For ST1, the plot indicates that the margin requirement is satisfied at all frequencies.

Validate the tracking and disturbance rejection requirements in the frequency domain.

```
viewSpec([Req1,Req4],ST1,Info)
```


Requirement 1: Tracking error as a function of frequency
Tracking error
Relative error (abs)
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
Frequency (rad/s)


Requirement 2: Maximum gain as a function of frequency
Principal gains
Singular Values (dB)
50
0
-50
10^{-4}
10^{-2}
10^{0}
10^{2}
Frequency (rad/s)

When you provide a vector of requirements, `viewSpec` puts all the requirements into a single figure window.

The first plot shows that the tuned system very nearly meets the tracking requirement. The slight violation suggests that setpoint tracking will perform close to expectations.

The second plot shows that the disturbance rejection levels off in violation of the requirement at very low frequencies. A small bump near 35 rad/s suggests possible damped oscillations at this frequency.

Use `step` and `getIOTransfer` to examine setpoint tracking and disturbance rejection in the time domain.

See Also

`systune` | `genss` | `evalSpecslTunable.systune` | `slTunable` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Sensitivity` | `TuningGoal.Overshoot` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain` | `TuningGoal.Margins` | `TuningGoal.WeightedGain` | `TuningGoal.Variance` | `TuningGoal.WeightedVariance` | `TuningGoal.LoopShape` | `TuningGoal.Poles` | `TuningGoal.StableController` |

Concepts

- “Generalized Models”
- “Performance and Robustness Specifications for looptune”

Purpose

Calculate bounds on worst-case gain of uncertain system

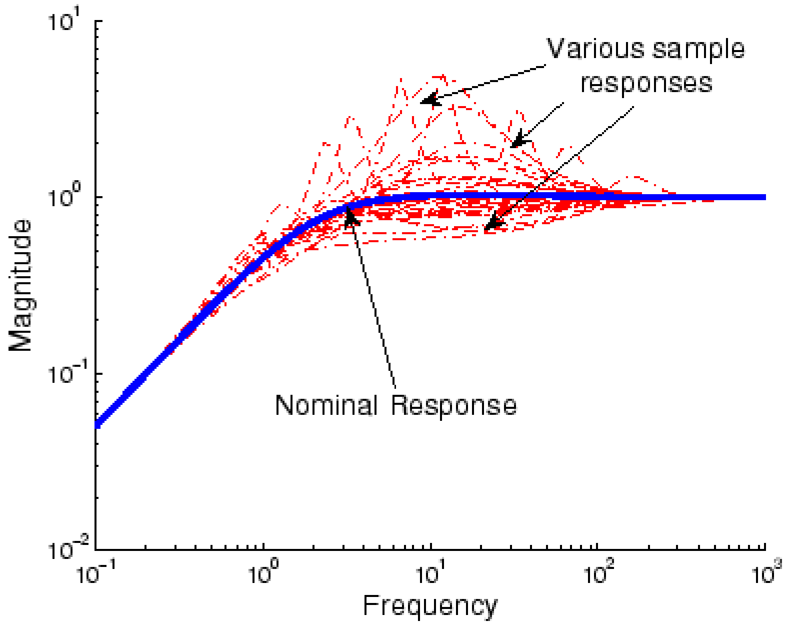
Syntax

```
[wcg,wcu,info] = wcgain(sys)
[wcg,wcu,info] = wcgain(sys,opts)
```

Description

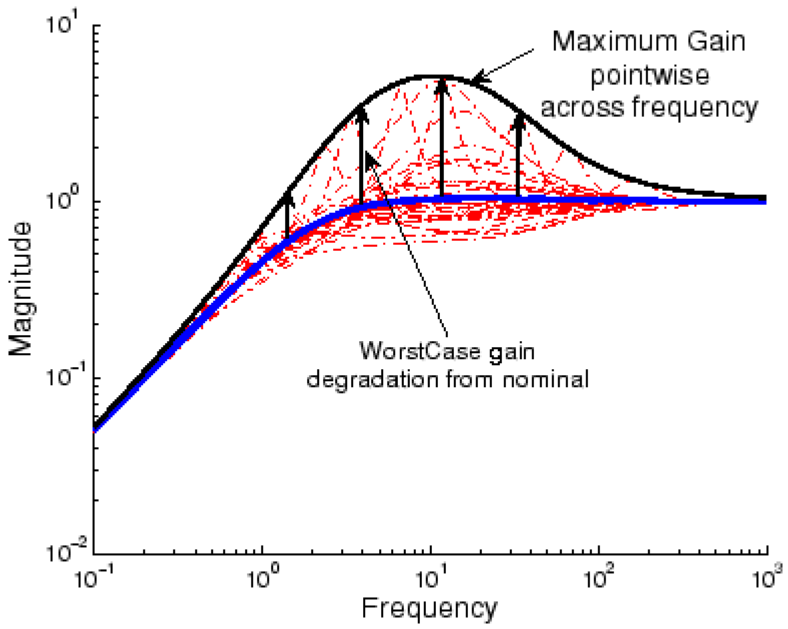
The gain of an uncertain system generally depends on the values of its uncertain elements. Here “gain” refers to the frequency response magnitude. (For multi-input, multi-output systems, the “gain” refers to the maximum singular value of the frequency response matrix.) Determining the maximum gain over 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.



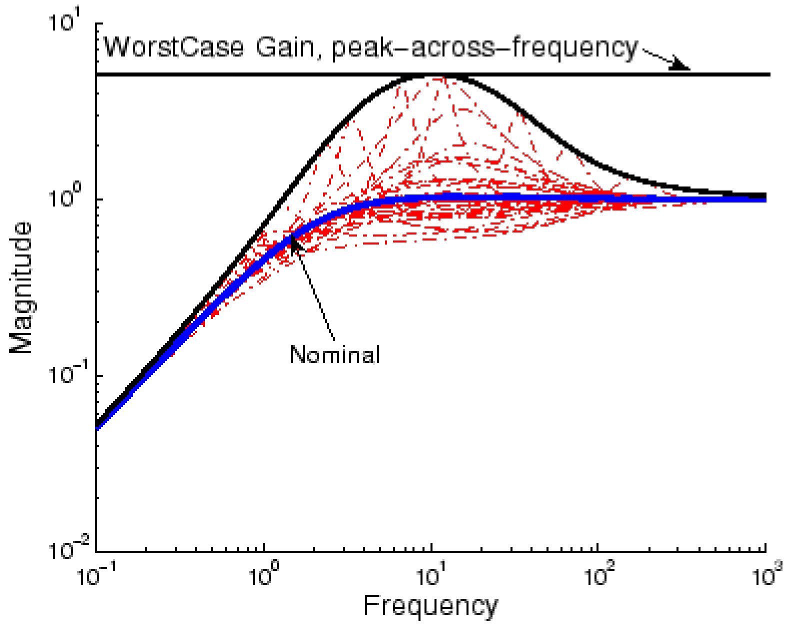
`wcgain` can perform two types of analysis on uncertain systems.

- A *max-over-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 *max-over-frequency*. You can control which analysis is performed by using the `MaxOverFrequency` option in the `wcgainOptions` options set.

Likewise, for arrays of uncertain models, the default `wcgain` analysis is *max-over-array*. This means that `wcgain` computes the worst-case gain over all models in the array. To compute the worst-case gain for each model separately, set the `MaxOverArray` option in the `wcgainOptions` options set to `'off'`.

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 `sys`.

```
[wcg,wcu] = wcgain(sys)
```

`wcg` 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`).

`wcu` is a structure containing values of uncertain elements that yield the worst-case uncertainty. 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.

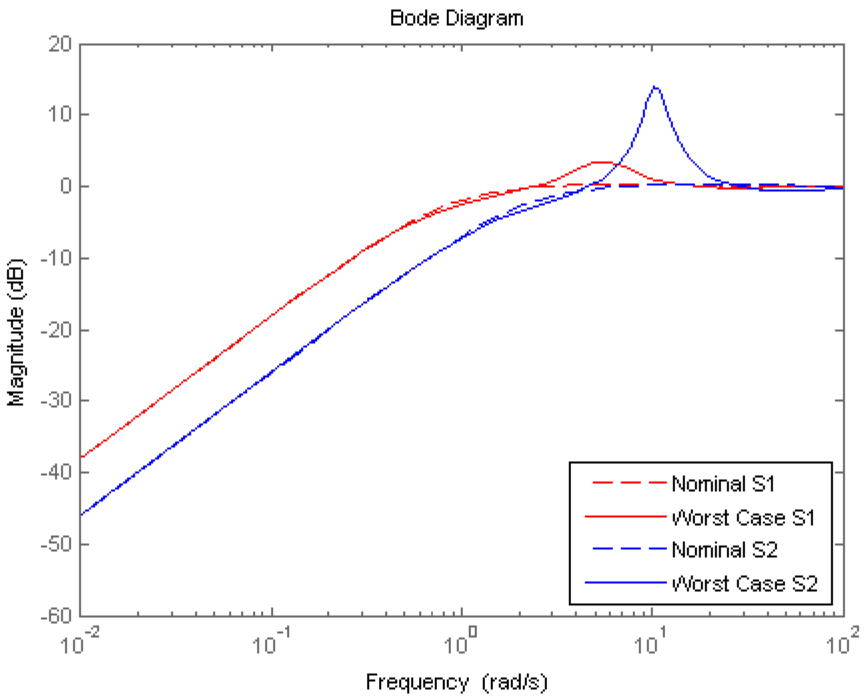
```
P = tf(1,[1 0]) + ultidyn('delta',[1 1],'bound',0.4);
BW1 = 0.8;
K1 = tf(BW1,[1/(25*BW1) 1]);
S1 = feedback(1,P*K1);
BW2 = 2.0;
K2 = tf(BW2,[1/(25*BW2) 1]);
S2 = feedback(1,P*K2);
```

Assess the worst-case gain of the closed-loop sensitivity function.

```
[maxgain1,wcunc1] = wcgain(S1);
[maxgain2,wcunc2] = wcgain(S2);

maxgain1, maxgain2

maxgain1 =

           LowerBound: 1.5067
           UpperBound: 1.5069
    CriticalFrequency: 5.4742


maxgain2 =

           LowerBound: 5.1037
           UpperBound: 5.1045
    CriticalFrequency: 10.3694
```

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')
legend('Nominal S1','Worst Case S1','Nominal S2','Worst Case S2',...
        'Location','SouthEast')
```



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.

```
[wcg,wcu,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 `wcgainOptions` object is `'off'`, the values are `NaN`.
`Frequency`	*N*-by-1 frequency vector associated with analysis.
`BadUncertainValues`	Structure of worst-case uncertainty values.
`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.

Specifying Additional Options

Use `wcgainOptions` to specify additional options for the worst-case gain computation. For example, you can turn the sensitivity computation on or off, set the step-size in the sensitivity computation, adjust the stopping criteria, or control behavior across frequency and array dimensions. For instance, you can turn the sensitivity calculation off as follows:

```
opt = wcgainOptions('Sensitivity','off');
[maxgain,maxgainunc,info] = wcgain(sys,opt)
```

To compute the worst-case gain as a function of frequency, set the `MaxOverFrequency` option to `'off'`.

For a model array `sys`, set the `MaxOverFrequency` option to `'off'` to compute the worst-case gain for each individual model in the array.

See the `wcgainOptions` reference page for more information about available options for `wcgain`.

Behavior on Notn-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 “max-over” can be used to customize the computation.

Algorithms

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 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.

Alternatives

Use wcgainplot to plot the worst-case gain of an uncertain system.

See Also

mussv | norm | robuststab | wcgainOptions | wcsens | wcmargin | wcgainplot | robustperf

Purpose

Option set for `wcgain`, `wcgainplot`, `wcnorm`, or `wcsens`

Syntax

```
opt = wcgainOptions
opt = wcgainOptions(Name,Value,...)
```

Description

`opt = wcgainOptions` returns the default option set for a `wcgain` calculation. The commands `wcgainplot`, `wcnorm`, and `wcsens` also use `wcgain` to compute their results. Use a `wcgainOptions` options set to control options for those calculations.

`opt = wcgainOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

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

'Sensitivity'

Determines whether to compute the sensitivity of worst-case gain with respect to each individual uncertain element.

`Sensitivity` is a string that takes the following values:

- `'on'` — `wcgain` computes the sensitivity of the worst-case gain with respect to each individual uncertain element. This provides an indication of which elements are most problematic.
- `'off'` — `wcgain` does not compute the sensitivity of the worst-case gain with respect to each individual uncertain element.

Default: `'on'`

'VaryUncertainty'

Percentage variation of uncertainty for sensitivity calculations. The sensitivity estimate uses a finite difference calculation.

Default: 25

'LowerBoundOnly'

Determines whether only the lower bound is computed.

`LowerBoundOnly` is a string that takes the following values:

- `'on'` — `wcgain` only computes a lower bound on the worst-case gain
- `'off'` — `wcgain` computes lower and upper bounds on the worst-case gain

Default: `'off'`

'MaxOverFrequency'

`MaxOverFrequency` is a string that takes the following values:

- `'on'` — `wcgain` computes the worst-case H_∞ norm (peak gain over frequency)
- `'off'` — `wcgain` computes the worst-case gain at each frequency point

Default: `'on'`

'MaxOverArray'

For uncertain model arrays, determines if worst-case gain is calculated over entire array or individually for all models in array.

`MaxOverArray` is a string that takes the following values:

- `'on'` — `wcgain` computes the worst-case gain over all models
- `'off'` — `wcgain` computes the worst-case gain for each model individually

Default: `'on'`

'AbsTol'

Absolute tolerance on computed bound.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`.

Relaxing tolerance speeds up the computation.

Default: 0.02

'RelTol'

Relative tolerance on computed bound.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`.

Default: 0.05

'AbsMax'

Absolute threshold for lower bound.

The algorithm terminates if `LowerBound >= AbsMax + RelMax * NominalGain`.

Specify `AbsMax` and `RelMax` to terminate when the lower bound is large enough compared to the nominal gain.

Default: 5

'RelMax'

Relative threshold for lower bound.

The algorithm terminates if `LowerBound >= AbsMax + RelMax * NominalGain`.

Specify AbsMax and RelMax to terminate when the lower bound is large enough compared to the nominal gain.

Default: 20

'NSearch'

Number of lower bound searches at each frequency

Default: 2

Output Arguments

opt

Option set containing the specified options for wcgain.

Examples

Create an options set for wcgain with only the lower bound being calculated and 5 lower bound searches at each frequency.

```
opt = wcgainOptions('LowerBoundOnly','on','Nsearch',5)
```

Alternatively, use dot notation to set the values of opt.

```
opt = wcgainOptions;
opt.LowerBound = 'on';
opt.NSearch = 5;
```

See Also

wcgain | wcgainplot | wcnorm | wcsens

Purpose

Graphical worst-case gain analysis

Syntax

```
wcgainplot(sys)
wcgainplot(sys,w)
wcgainplot(sys,...,options)
```

Description

`wcgainplot(sys)` plots the nominal and worst-case gains of the uncertain system `sys` as a function of frequency. For multi-input, multi-output (MIMO) systems, gain refers to the largest singular value of the frequency response matrix. (See `sigma` for more information about singular values.) The plot includes:

- Nominal — nominal gain of `sys`
- Worst — the response falling within the uncertainty of `sys` that has the highest peak gain
- Worst-case gain (lower bound) — the lowest worst-case gain at each frequency
- Worst-case gain (upper bound) — the highest gain within the uncertainty at each frequency
- Sampled Uncertainty — 20 responses randomly sampled from `sys`

`wcgainplot(sys,w)` focuses the plot on the frequencies specified by `w`.

- If `w` is a cell array `{wmin,wmax}`, `wcgainplot` plots the worst-case gains in the range `{wmin,wmax}`.
- If `w` is an array of frequencies, `wcgainplot` plots the worst-case gains at each frequency in the array.

`wcgainplot(sys,...,options)` uses the options set `options` to specify additional options for the computation of the worst-case gains. Use `wcgainOptions` to create the options set.

Input Arguments

sys

Uncertain dynamic system.

w

Frequencies of worst-case gain plots. Specify frequencies in radians/`TimeUnit`, where `TimeUnit` is the time unit of `sys`.

- If `w` is a cell array `{wmin,wmax}`, `wcgainplot` plots the worst-case gains in the range `{wmin,wmax}`.
- If `w` is an array of frequencies, `wcgainplot` plots the worst-case gains at each frequency in the array.

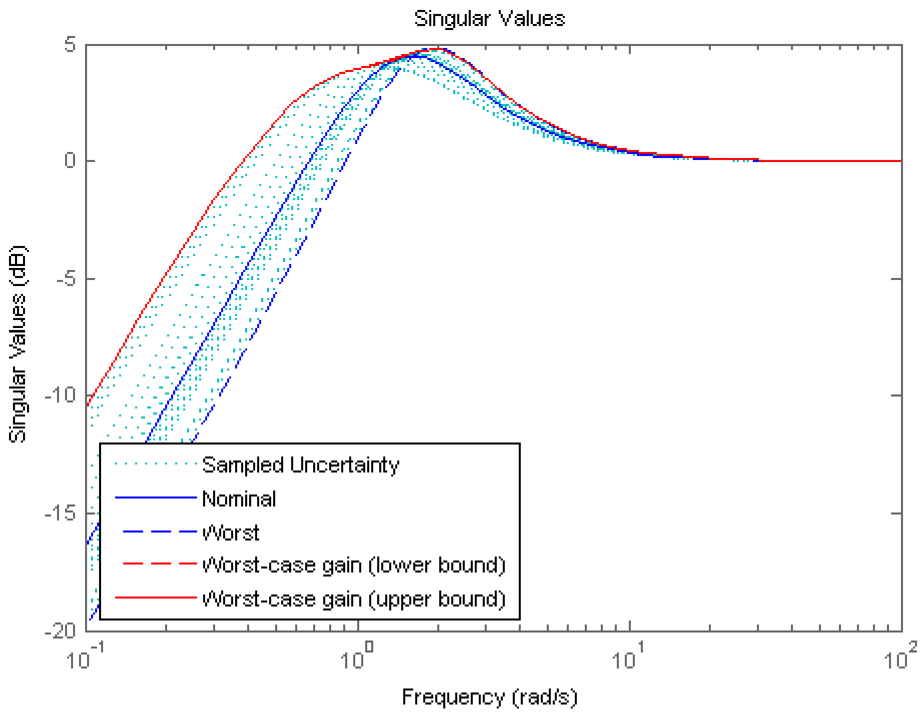
options

Options set specifying additional options for the computation of the worst-case gains. Use `wcgainOptions` to create the options set.

Examples

Plot the worst-case gain of the system $sys = \frac{s^2+3s}{s^2+2s+a}$, where the uncertain parameter `a = 2 +/- 1`. Plot the worst-case gain between 0.1 and 100 rad/s.

```
a = ureal('a',2)
sys = tf([1 3 0],[1 2 a]);
wcgainplot(sys,{.1 100})
```



The `Worst` curve identifies the single response within the uncertainty that yields the highest gain at any frequency. The `Worst-case gain (upper bound)` curve is the envelope produced by finding the highest gain within the uncertainty at each frequency.

Algorithms

`wcgainplot` uses `wcgain` to compute the worst-case gains. Use the `options` argument to `wcgainplot` to set options for the `wcgain` algorithm.

`wcgainplot` uses `usample` to compute the `Sampled Uncertainty` curves.

See Also

`wcgain` | `wcgainOptions` | `usample` | `sigma`

wcgopt

Purpose Options object for use with `wcgain`, `wcsens`, and `wcmargin`

Note `wcgopt` will be removed in a future version. Use `wcgainOptions` or `wcmarginOptions` instead.

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 allowable loop-at-a-time variations in the nominal system gain and phase for which the feedback loop retains stability.

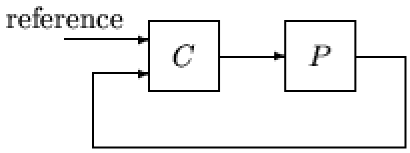
An alternative to classical gain and phase margins is the disk margin. The *disk margin* is the largest region for each channel such that for all gain and phase variations inside the region the nominal closed-loop system is stable. See the `dmplot` and `loopmargin` reference pages to learn more about the algorithm.

Consider a system with uncertain elements. It is of interest to determine the margin of each individual channel in the presence of uncertainty. These margins are called worst-case margins. Worst-case margin, `wcmargin` calculates the largest disk margin such that for values of the uncertainty and all gain and phase variations inside the disk, the closed-loop system is stable. The worst-case gain and phase margin bounds are defined based on the balanced sensitivity function. Hence, results from the worst-case margin calculation imply that the closed-loop system is stable for a given uncertainty set and would remain stable in the presence of an additional gain and phase margin variation in the specified input/output channel.

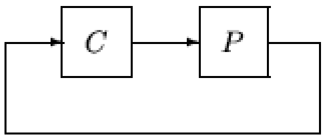
`wcmargL = wcmargin(L)` calculates the combined worst-case input and output loop-at-a-time gain/phase margins of the feedback loop consisting of the loop transfer matrix `L` in negative feedback with an identity matrix. `L` must be an uncertain system, `uss` or `ufrd` object. If `L` is a `uss` object, the frequency range and number of points used to calculate `wcmargL` are chosen automatically. Note that in this case, the worst-case margins at the input and output are equal because an identity matrix is used in feedback. wcmarg is a NU-by-1 structure with the following fields:

Field	Description
GainMargin	Guaranteed bound on worst-case, single-loop gain margin at plant inputs. Loop-at-a-time analysis.
PhaseMargin	Loop-at-a-time worst-case phase margin at plant inputs. Units are degrees.
Frequency	Frequency associated with the worst-case margin (rad/s).
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 wcmarginOptions 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 N_Y outputs and N_U inputs and hence the controller `C` must have N_U outputs and N_Y inputs. `wcmargi` is a `NU`-by-`1` structure with the following fields:

Field	Description
`GainMargin`	Guaranteed bound on worst-case, single-loop gain margin at plant inputs. Loop-at-a-time analysis.
`PhaseMargin`	Loop-at-a-time worst-case phase margin at plant inputs. Units are degrees.

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 `wcmarginOptions` object is `'off'`, the values are `NaN`.

`wcmargo` is an N-by-1 structure for the single loop transfer matrix input and `wcmargo` is an N_Y-by-1 structure when the plant and controller are input. In both these cases, `wcmargo` has the same fields as `wcmargi`. The worst-case bound on the gain and phase margins are calculated based on a balanced sensitivity function.

`[wcmargi,wcmargo] = wcmargin(L,opt)` and

`[wcmargi,wcmargo] = wcmargin(p,c,opt)` specify options described in `opt`. (See `wcmarginOptions` for more details on the options for `wcmargin`.)

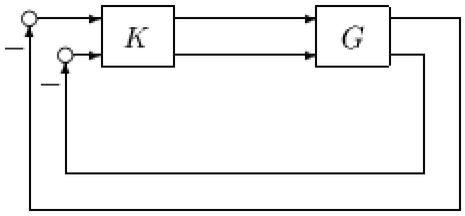
The sensitivity of the worst-case margin calculations to the individual uncertain elements is selected using the options object `opt`. To compute sensitivities, create a `wcmarginOptions` options object, and set the `Sensitivity` property to `'on'`.

Examples

MIMO Loop-at-a-Time Margins

This example is designed to illustrate that loop-at-a-time margins (gain, phase, and/or distance to –1) can be inaccurate measures of multivariable robustness margins. Margins of the individual loops can be very sensitive to small perturbations within other loops.

The nominal closed-loop system considered here is shown as follows.



G and K are 2-by-2 multi-input/multi-output (MIMO) systems, defined as

$$G := \frac{1}{s^2+\alpha^2}\begin{bmatrix} s-\alpha^2 & \alpha(s+1) \\ -\alpha(s+1) & s-\alpha^2 \end{bmatrix}, K = I_2$$

Set α := 10, construct the nominal model G in state-space form, and compute its frequency response.

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

The nominal plant was analyzed previously using the command. Based on experimental data, the gain of the first input channel, `b(1,1)`, is found to vary between 0.97 and 1.06. The following statement generates the updated uncertain model.

```
ingain1 = ureal('ingain1',1,'Range',[0.97 1.06]);
b = [ingain1 0;0 1];
Gunc = ss(a,b,c,d);
```

Because of differences between measured data and the plant model an 8% unmodeled dynamic uncertainty is added to the plant outputs.

```
unmod = ultidyn('unmod',[2 2],'Bound',0.08);
Gmod = (eye(2)+unmod)*Gunc;
Gmodg = ufrd(Gmod,logspace(-1,3,60));
```

You can use the command `wcmargin` to determine the worst-case gain and phase margins in the presences of the uncertainty.

```
[wcmi,wcmo] = wcmargin(Gmodg,K);
```

The worst-case analysis corresponds to maximum allowable disk margin for all possible defined uncertainty ranges. The worst-case single-loop margin analysis performed using `wcmargin` results in a maximum allowable gain margin variation of 1.31 and phase margin variations of ± 15.6 degs in the second input channel in the presence of the uncertainties `'unmod'` and `'ingain1'`. `wcmi(1)`

```
ans =
     GainMargin: [0.3613 2.7681]
    PhaseMargin: [-50.2745 50.2745]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
wcmi(2)
ans =
     GainMargin: [0.7585 1.3185]
    PhaseMargin: [-15.6426 15.6426]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
```

Hence even though the second channel had infinite gain margin and 90 degrees of phase margin, allowing variation in both uncertainties, `'unmod'` and `'ingain1'` leads to a dramatic reduction in the gain and phase margin.

You can display the sensitivity of the worst-case margin in the second input channel to `'unmod'` and `'ingain1'` as follows:

```
wcmi(2).Sensitivity
ans =
    ingain1: 12.1865
      unmod: 290.4557
```

The results indicate that the worst-case margins are not very sensitive to the gain variation in the first input channel, `'ingain1'`, but very sensitive to the LTI dynamic uncertainty at the output of the plant.

The worst-case single-loop margin at the output results in a maximum allowable gain margin variation of 1.46 and phase margin variation of ± 21.3 degs in the second output channel in the presence of the uncertainties `'unmod'` and `'ingain1'`.

```
wcmo(1)
ans =
     GainMargin: [0.2521 3.9664]
    PhaseMargin: [-61.6995 61.6995]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
wcmo(2)
ans =
     GainMargin: [0.6835 1.4632]
    PhaseMargin: [-21.2984 21.2984]
      Frequency: 0.1000
     Sensitivity: [1x1 struct]
```

You can display the sensitivity of the worst-case margin in the second output channel to `'unmod'` and `'ingain1'` as follows:

```
wcmo(2).Sensitivity
ans =
    ingain1: 16.3435
      unmod: 392.1320
```

The results are similar to the worst-case margins at the input. However, the worst-case margins at the second output channel are even more sensitive to the LTI dynamic uncertainty than the input channel margins.

See Also `dmplot` | `loopsens` | `robuststab` | `usubs` | `wcgain` | `wcmarginOptions` | `wcsens`

Purpose

Option set for `wcmargin`

Syntax

```
opt = wcmarginOptions
opt = wcmarginOptions(Name,Value,...)
```

Description

`opt = wcmarginOptions` returns the default option set for `wcmargin`.

`opt = wcmarginOptions(Name,Value,...)` creates an option set with the options specified by one or more `Name,Value` pair arguments.

Input Arguments

Name-Value Pair Arguments

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

'Sensitivity'

Determines whether to compute the sensitivity of worst-case gain with respect to each individual uncertain element.

`Sensitivity` is a string that takes the following values:

- `'on'` — Sensitivity of the worst-case gain is computed with respect to each individual uncertain element. This provides an indication of which elements are most problematic.
- `'off'` — `wcmargin` does not compute the sensitivity of the worst-case gain with respect to each individual uncertain element.

Default: `'off'`

'AbsTol'

Absolute tolerance on computed worst-case margin bounds.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`

Default: 0.02

'RelTol'

Relative tolerance on computed worst-case margin bounds.

The algorithm terminates if `UpperBound-LowerBound <= max(AbsTol, Reltol*UpperBound)`

Default: 0.05

Output Arguments

opt

Option set containing the specified options for `wcmargin`.

Examples

Create an options set for `wcmargin` with an 0.01 and 0.03 as the absolute and relative tolerances on the worst-case margin bounds, respectively.

```
opt = wcmarginOptions('AbsTol',0.01,'RelTol',0.03);
```

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

```
opt = wcmarginOptions;
opt.AbsTol = 0.01;
opt.RelTol = 0.03;
```

See Also

`wcmargin` | `wcgainOptions`

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 field names, which are the names of uncertain elements of `mat`. The value of each field is the corresponding value of the uncertain element, such that when jointly combined, lead to the norm value in `maxnorm.LowerBound`. The following command shows the norm:

```
 norm(usubs(mat,maxnormunc))
```

Basic syntax with third output argument

A third output argument provides information about sensitivities of the worst-case norm to the uncertain elements ranges.

```
[maxnorm,maxnormunc,info] = 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 `wcgainOptions` 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.

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}

$$\kappa(M) = \max_{\substack{\Delta \in C^{m\times m} \\ \sigma_{\max}(\Delta) \le 1}} \left(\sigma_{\max}(M\Delta M^{-1})\right)$$

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(N.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
```

Algorithms

See `wcgain`

See Also

`norm` | `wcgain` | `wcgainOptions`

Purpose

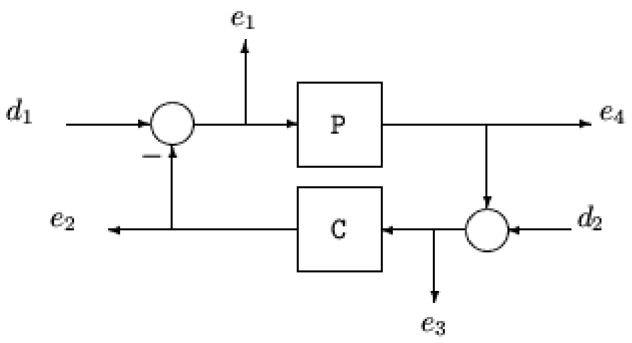
Calculate worst-case sensitivity and complementary sensitivity functions of plant-controller feedback loop

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 ($TF_{e3\leftarrow d2}$)	$(I + CP)$–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
`To`	Worst-case output-to-plant complementary sensitivity function

Fields of `wcst` (Continued)

Field	Description
`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

`loopsens` | `robuststab` | `usubs` | `wcgain` | `wcgopt` | `wcmargin`

Block Reference

MultiPlot Graph

Purpose

Plot multiple signals

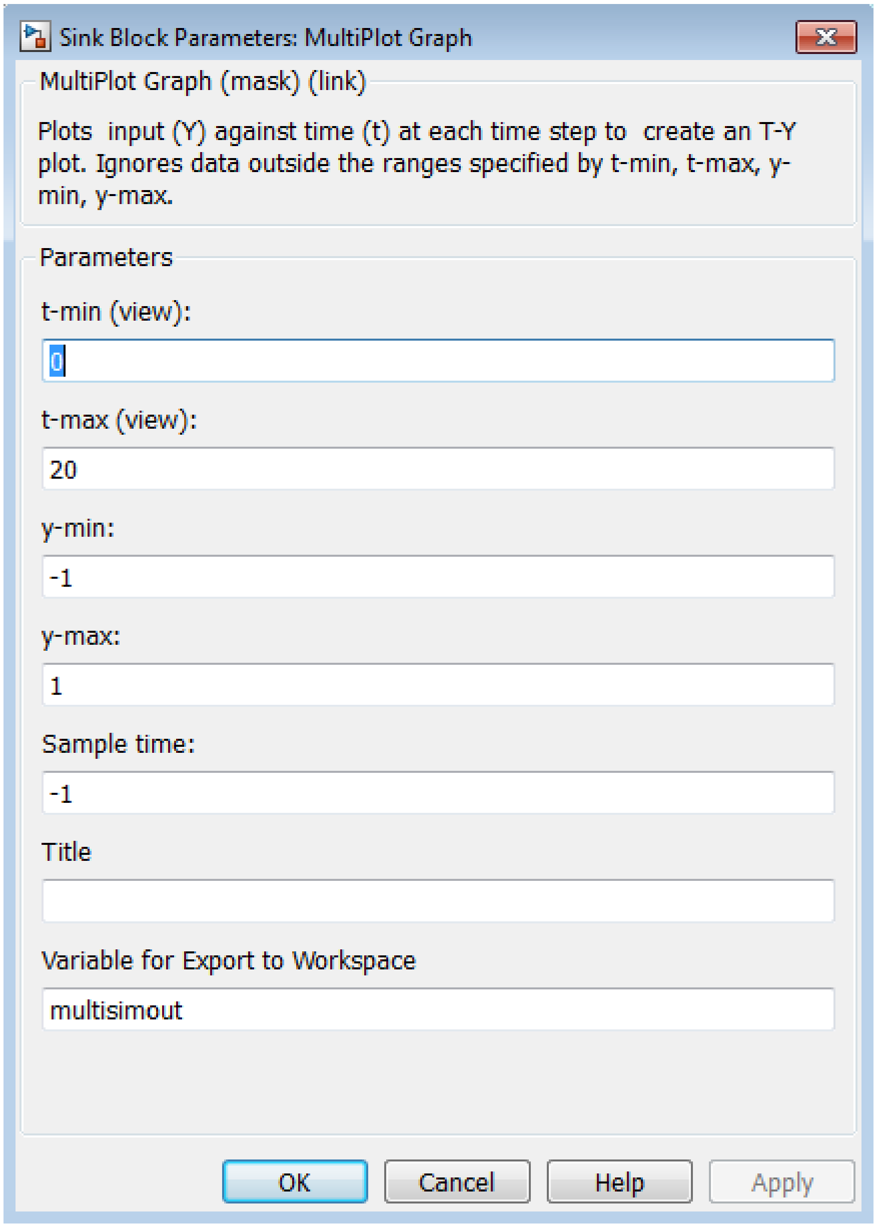
Description

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.


Sink Block Parameters: MultiPlot Graph

MultiPlot Graph (mask) (link)

Plots input (Y) against time (t) at each time step to create an T-Y plot. Ignores data outside the ranges specified by t-min, t-max, y-min, y-max.

Parameters

t-min (view): 0

t-max (view): 20

y-min: -1

y-max: 1

Sample time: -1

Title

Variable for Export to Workspace: multisimout

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.

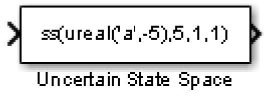
Variable for Export to Workspace

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

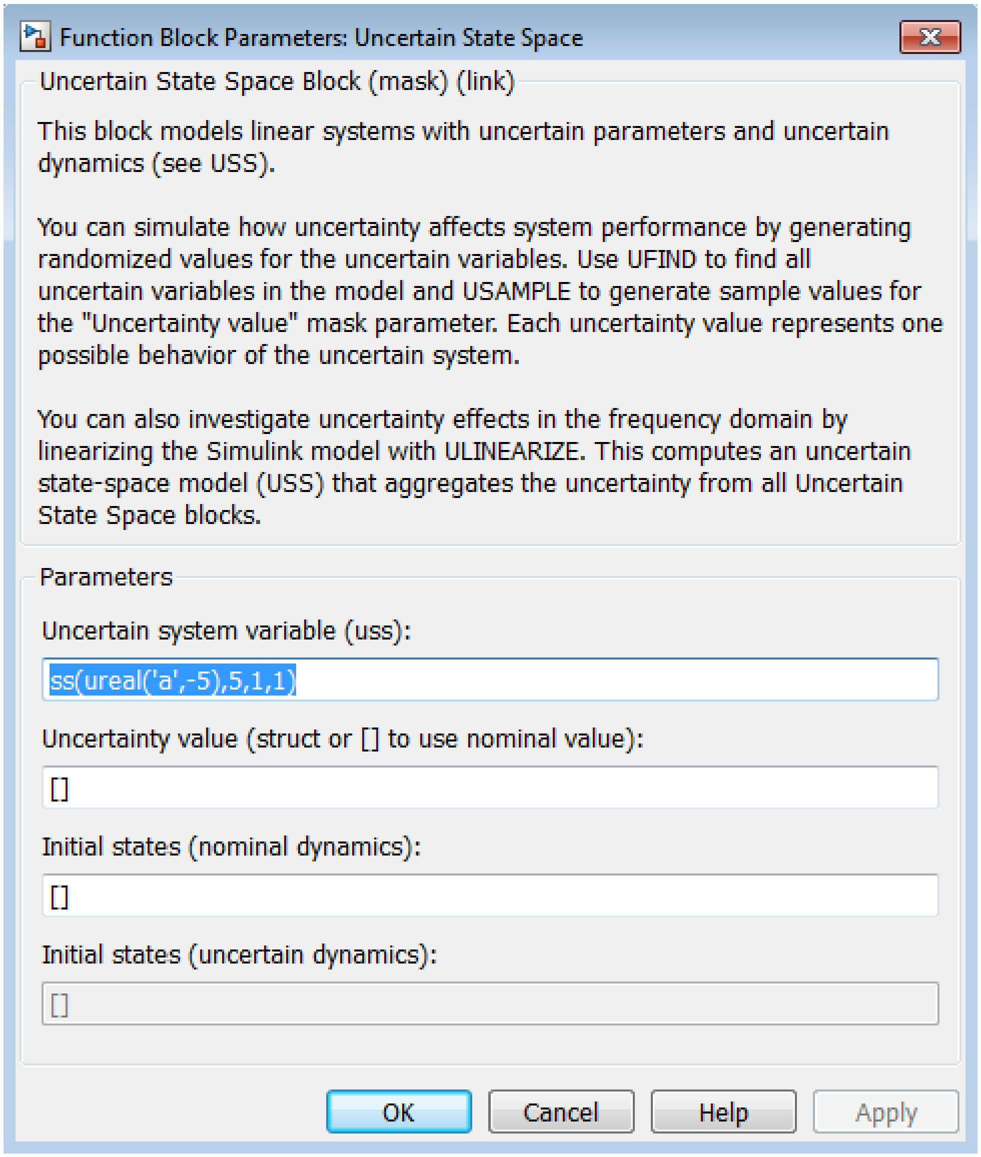
Specify uncertain system in Simulink

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


Function Block Parameters: Uncertain State Space

Uncertain State Space Block (mask) (link)

This block models linear systems with uncertain parameters and uncertain dynamics (see USS).

You can simulate how uncertainty affects system performance by generating randomized values for the uncertain variables. Use UFIND to find all uncertain variables in the model and USAMPLE to generate sample values for the "Uncertainty value" mask parameter. Each uncertainty value represents one possible behavior of the uncertain system.

You can also investigate uncertainty effects in the frequency domain by linearizing the Simulink model with ULINEARIZE. This computes an uncertain state-space model (USS) that aggregates the uncertainty from all Uncertain State Space blocks.

Parameters

Uncertain system variable (uss):

ss(ureal('a',-5),5,1,1)

Uncertainty value (struct or [] to use nominal value):

[]

Initial states (nominal dynamics):

[]

Initial states (uncertain dynamics):

[]

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 "Vary Uncertainty Values Using Individual Uncertain State Space Blocks" and "Vary Uncertainty Values Across Multiple Uncertain State Space Blocks" in the *Robust Control Toolbox User's Guide*.

Initial states (nominal dynamics)

If the nominal value of the uncertain state variable, `unc_sys.NominalValue where unc_sys` is the uncertain system variable specified in the **Uncertain system variable** field, has states, specify the initial condition for these states. The value defaults to zero.

Initial states (uncertain dynamics)

If the uncertain system contains some dynamic uncertainty (`ultidyn`), specify the initial state of these dynamics. The value defaults to zero.

See Also

`ufind`, `usample`, `ulinearize`, `uss`, `umat`, `ureal`, `ultidyn`

Tutorials

Robustness Analysis in Simulink

Linearization of Simulink Models with Uncertainty

How To

"Simulate 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.

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